Test Audit Report

Prepared by: L.Petroulakis

Table of Contents

Protocol Summary

Our Protocol is a core ERC20 contract with additional functionality including utilization of UniswapV2 pairing and a staking/reward mechanism

Disclaimer

All effort was made to find as many vulnerabilities in the code in the given time period, with additional provision of changes to be considered.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

The CodeHawks severity matrix was used to determine severity. See the documentation for more details.

Audit Details

Scope

The scope of the audit includes the following files:

- [src/contracts/MyContract.sol]

Roles

Executive Summary

Issues found

Findings

High

[H-1] Critical errors in `MyContract::burn` break protocol functionality

Description There are critical errors within the implementation of the `burn` function in the contract. There is no access control and anyone can call the function to reduce any user's tokens by any amount. Secondly, there is no check for the amount of tokens to be burned. Finally, there seems to be no incentive for an external user to burn their tokens.

Impact The protocol's functionality is severely impaired as anyone can call the function to maliciously impact the amount of tokens of another user.

Recommended mitigation Consider revamping the `burn` function to restrict burning to `msg.sender` and check for the amount of tokens to be burned.

```
- function burn(address target, uint256 amount) external {
-        _balances[target] = _balances[target] - amount;
+ function burn(uint256 amount) external {
+   require(balanceOf(msg.sender) >= amount, "Burn amount exceeds
balance");
+   _burn(msg.sender, amount); // Only burns tokens owned by msg.sender
}
```

Otherwise, consider adding access control and restricting burn function to the owner. Furthermore, consider adding an incentive for users to burn their tokens, since there seems to be no reason for an external user to burn their tokens currently.

[H-2] `addLiquidity` sends tokens to the zero address, effectively burning them.

Description By sending LP tokens to the zero address, you effectively burn them. This means that the added liquidity cannot be withdrawn or managed in any way in the future, making it permanently locked in the liquidity pool.

Impact Locking liquidity without the possibility of retrieval means the project loses the ability to control or adjust its liquidity pools, and losing control over those tokens could affect liquidity rewards and other functionalities of the protocol working with the liquidity pool.

Recommended mitigation Consider sending the tokens to the owner or another address that can be controlled by the project, like a multi-sig wallet. This way they can be managed and withdrawn in the future as needed.

```
    function addLiquidity(uint256 tokenAmount, uint256 ethAmount) private
{
        // add the liquidity
        uniswapV2Router.addLiquidityETH{value: ethAmount}(
            address(this),
            tokenAmount,
            0, // slippage is unavoidable
            0, // slippage is unavoidable
-            address(0),
+            owner(),
            block.timestamp
        );
    }
```

[H-3] Critical errors in `stake` function break staking functionality

Description In the `stake` function there are several errors that completely break the intended functionality. The function does not check if the user has enough tokens to stake, nor does it add the staked amount to potential previously staked amounts. This means that whatever value is passed to the function will just overwrite the previous value instead of being added to it. Additionally, there is no check against the zero value, and there is no actual transfer of tokens from the user to the contract.

Impact The staking function is completely broken, absolutely eliminating the purpose of its existence.

Recommended mitigation Consider updating the `stake` function to something similar to the following:

```
function stake(uint256 amount) external {
    require(amount > 0, "Cannot stake zero amount");
    super._transfer(msg.sender, address(this), amount); // transfers tokens to the contract
    stackedAmounts[msg.sender] += amount; // accumulates the staked tokens
    stackedTimes[msg.sender] = block.timestamp; // updates the stake time
}
```

[H-4] Incorrect math and logic in `MyContract::harvest` leads to broken reward calculation and introduces potential reentrancy risk

Description The `harvest` function has an incorrect order of operations in its `rewards` calculation. The following line is flawed and additonal logic is missing:

```
uint256 rewards = block.timestamp - stackedTimes[msg.sender] / balanceOf(address(this));
```

Impact `lock.timestamp - stackedTimes[msg.sender] / balanceOf(address(this))` does not follow the intended order of operations. This will perform `stackedTimes[msg.sender] / balanceOf(address(this))` first, which is likely not the intended calculation and leads to unpredictable results.

Additionally, there is no apparent multiplier or APY functionality, as is typical of staking operations in protocols, leading to potentially incorrect calculation of rewards. Moreover, no check is made to ensure that `balanceOf(address(this))` is not zero, which could lead to a division by zero error. Finally, there is no update of the `stackedTimes` variable, which could lead to reentrancy being used to repeatedly call harvest and extract rewards far exceeding the intended amount.

Proof of Code Consider a following scenario where an attacker exploits the reentrancy vulnerability in the `harvest` function through the lack of `stackedTimes` update:

1. the attacker deploys their AttackContract with the address of MyContract.
2. they call attack() to stake an amount in MyContract and make themselves eligible to harvest rewards.
3. the attack() function calls harvest() on MyContract.
4. during the transfer in harvest, the attacking contract's fallback() function is triggered.
5. inside fallback(), harvest is called again, repeating the `super._transfer` process before the state can be updated in MyContract.
6. the reentry continues as long as the target contract still has balance.
7. each reentrant call effectively transfers additional rewards to the attacker without updating stackedTimes[msg.sender].
8. after the repeated calls, the attacker can call withdraw() to transfer the accumulated funds to their address.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

interface IMyContract {
    function stake(uint256 amount) external;
    function harvest() external;
}

contract AttackContract {
    IMyContract public targetContract;
    address public owner;

    constructor(address _targetContract) {
        targetContract = IMyContract(_targetContract);
        owner = msg.sender;
    }

    fallback() external payable {
        if (address(targetContract).balance >= 1 ether) {  // just an
example condition
            targetContract.harvest();
        }
    }

    function attack() external payable {
        require(msg.sender == owner, "Not the owner");

        // we have to assume that the attacker stakes some initial amount
to be eligible for rewards
        targetContract.stake{value: msg.value}(msg.value);

        // First harvest call, triggers reentrancy
        targetContract.harvest();
    }

    function withdraw() external {
        require(msg.sender == owner, "Not the owner");
        payable(owner).transfer(address(this).balance);
    }
}
```

Recommended mitigation A properly functioning `harvest` function should calculate rewards more accurately and update stackedTimes accordingly. Consider adding OZ's ReentrancyGuard and updating the function to something similar to the following:

```
function harvest() external nonReentrant {
    require(stackedAmounts[msg.sender] > 0, "No staked amount");

    // Time elapsed since last stake or last harvest
    uint256 elapsed = block.timestamp - stackedTimes[msg.sender];

    // Calculate reward based on staked amount and an APY multiplier
    uint256 rewardRate = 10; // Example reward rate (10%)
    uint256 rewards = (stackedAmounts[msg.sender] * rewardRate * elapsed)
/ (365 days * 100);

    // Transfer rewards to user
    super._transfer(address(this), msg.sender, rewards);

    // Update last harvest time
    stackedTimes[msg.sender] = block.timestamp;
}
```

[H-5] Problematic logic in `MyContract::unstake` leaves the function redundant and in need of a separate call to `harvest`.

Description For the unstake function, users generally expect to receive both their staked amount and any accumulated rewards upon unstaking. The current design doesn’t transfer rewards when users unstake, meaning users would need to call harvest separately to collect rewards.

Recommended mitigation The unstake function should transfer both the user’s staked amount and any accumulated rewards. To achieve this, `harvest` could be called within `unstake` to consolidate rewards and the principal amount in one transaction:

```
function unstake() external nonReentrant {
    require(stackedAmounts[msg.sender] > 0, "No staked amount");

    harvest();

    uint256 stakedAmount = stackedAmounts[msg.sender];
    stackedAmounts[msg.sender] = 0;
    super._transfer(address(this), msg.sender, stakedAmount);
}
```

[H-6] Incorrect order of subtraction in `swapAndSendToFee` does not properly calculate `newBalance` and leads to reverts

Description In the `swapAndSendToFee` function, the order of subtraction is incorrect, leading to an incorrect calculation of `newBalance`. This leads to no tokens being sent to the `_marketingWalletAddress` and the function reverting due to solidity version 0.8.0's underflow protection, resulting to failed transactions.

Proof of Concept This is the incorrect line of code as it subtracts the updated balance from the original balance of tokens:

```
uint256 newBalance =
initialCAKEBalance.sub(IERC20(rewardToken).balanceOf(address(this)));
```

Recommended mitigation Consider updating the function to properly calculate the new balance and add a check to ensure that the new balance is greater than zero before proceeding with the transfer:

```
-  function swapAndSendToFee(uint256 tokens) private {
-        uint256 initialCAKEBalance =
IERC20(rewardToken).balanceOf(address(this));

-        swapTokensForCake(tokens);
-        uint256 newBalance =
initialCAKEBalance.sub(IERC20(rewardToken).balanceOf(address(this)));
-        IERC20(rewardToken).transfer(_marketingWalletAddress,
newBalance);
-    }
+   function swapAndSendToFee(uint256 tokens) private {
+    uint256 initialCAKEBalance =
IERC20(rewardToken).balanceOf(address(this));

+    swapTokensForCake(tokens);

+    uint256 newBalance =
IERC20(rewardToken).balanceOf(address(this)).sub(initialCAKEBalance);
// the following could be redundant if the newBalance is always positive,
but still a good practice
+    if (newBalance > 0) {
+        IERC20(rewardToken).transfer(_marketingWalletAddress,
newBalance);
+    }
+ }
```

[H-7] Lack of Slippage Protection in critical instances of the contract's functionality can lead to considerable loss of funds during operations

Description In `swapTokensForEth`, `swapTokensForCake` and `addLiquidity` there is no slippage protection implemented, and the contract simply considers the loss of funds to slippage unavoidable but acceptable to all extents. This can lead to considerable loss of funds during the swapping process, especially in volatile markets or when dealing with large amounts of tokens.

Recommended mitigation Consider implementing slippage protection in all three of these instances similar to the following where the slippage is set to 2%:

```
function addLiquidity(uint256 tokenAmount, uint256 ethAmount) private {
    uint256 amountTokenMin = tokenAmount.mul(98).div(100);
    uint256 amountETHMin = ethAmount.mul(98).div(100);

    uniswapV2Router.addLiquidityETH{value: ethAmount}(
        address(this),
        tokenAmount,
        amountTokenMin,
        amountETHMin,
        owner(),
        block.timestamp
    );
}
```

[H-8] Absence of the actual initial transfer of 1e6 tokens to the `_marketingWalletAddress` causes a fundamental misallocation of tokens as the wallet might not be able to fulfill its purpose, impacting transparency and trust and breaking the protocol's functionality.

Description During deployment and within the constructor, after the owner is minted the totalSupply of 1e9 tokens, the balance of the marketing wallet is set without actual transfer of token from the owner to it.

Proof of Concept In the contract's constructor:

```
    _mint(owner(), totalSupply_);
    _balances[_marketingWalletAddress] = 1000000;
```

Recommended mitigation Consider making the following changes to the contract's constructor:

```diff
_mint(owner(), totalSupply_);
- _balances[_marketingWalletAddress] = 1000000;
+ _transfer(owner(), _marketingWalletAddress, 1000000);
```

[H-9] Inadequate check for blacklisted addresses in `_transfer` could lead to transferring funds to blacklisted addresses

Description The `_transfer` function does check if the sending address is address(0) or a blacklisted address but does not check if the recipient address is blacklisted before transferring funds. This could lead to transferring funds to blacklisted addresses.

Recommended mitigation Consider making the following change to the check for blacklisted addresses in the `_transfer` function before transferring funds:

```
+ require(!blacklistAddresses[to], "Bot detected");
```

[H-10] Incorrect setting of `totalFees` in the constructor affects the contract's fee structure from the start, creating inconsistencies in the protocol's functionality

Description In the contract's constructor, `totalFees` is set to `tokenRewardsFee.add(liquidityFee)`, which omits the `marketingFee`. This means that `totalFees` will initially only reflect the `tokenRewardsFee` and `liquidityFee` components, but not the `marketingFee`.

Recommended mitigation Consider updating the `totalFees` calculation in the contract's constructor to include all fee components:

```
- totalFees = tokenRewardsFee.add(liquidityFee);
+ totalFees = tokenRewardsFee.add(liquidityFee).add(marketingFee);
```

[H-11] Bad management and design when it comes to `rebase` and `mint` functions, along with a lack of max supply can lead to uncontrolled iflation

Description The contract lacks a max supply, something that the `rebase` and `mint` functions should be designed to respect when executing their operations. Additionally, there is no flexibility for the owner to mint new tokens to any address rather than their own, and the rebase function affects new tokens directly, without any proportional scaling of balances or adjustments based on a rebase factor, an approach that is fundamentally flawed for a typical rebase mechanism.

Impact Without a maximum supply limit, the owner could mint or rebase an unlimited number of tokens, which could cause severe inflation, drastically reducing the token’s value. Additionally, the lack of proportional scaling in the rebase function could lead to an unfair distribution of tokens, as the rebase would affect all token holders equally, regardless of their initial holdings.

Recommended mitigation

1. Define a maximum supply cap and initialize it in the constructor before minting the totalSupply to the owner

```
uint256 public immutable maxSupply;
```

```
constructor() ERC20("Contract", "CTR") {
    // .....
    uint256 totalSupply_ = 1000000000;
    maxSupply = 2000000000; // example: max supply of 2 billion tokens
    _mint(owner(), totalSupply_);
    // .......
}
```

2. Alter the `mint` function to enforce a maximum supply

```
function mint(address to, uint256 amount) external onlyOwner {
    require(totalSupply().add(amount) <= maxSupply, "max supply
exceeded");
    _mint(to, amount);
}
```

3. Alter the rebase function to respect maximum supply and adjust balances proportionally with something like a rebase factor

```
uint256 public rebaseFactor = 1e18;
```

```
function rebase(uint256 newSupply) external onlyOwner {
    require(newSupply <= maxSupply, "max supply exceeded");
    require(newSupply != totalSupply, "new supply is same as the current
one");

    rebaseFactor = rebaseFactor.mul(newSupply).div(totalSupply);

    totalSupply = newSupply;

    emit Rebase(newSupply);
}

// just sample code, typically you have to adjust all balances and other
variables accordingly
function balanceOf(address account) public view override returns (uint256)
{
    return _scaledBalanceOf(account);
}

// internal function to calculate scaled balance for an account
function _scaledBalanceOf(address account) internal view returns (uint256)
{
    // we retrieve the original balance and then scale it by the
rebaseFactor
```

```
        return super.balanceOf(account).mul(rebaseFactor).div(1e18);
    }
```

[H-12] Hardcoded fee value in `_transfer` leads to disproportionate impact on small transfers and ca cause a zero transfer issue

Proof of Concept Within the `_transfer` function, the hardcode value mentioned isused as follows:

```
super._transfer(from, 0xBdf1a2e17DECb2aAC725F0A1C8C4E2205E70719C, 1);
```

This however, arises several concerns about its use.

1. Unclear purpose and lack of natspec to explain the hardcoded value. this ambiguity can lead to misunderstandings among developers, auditors and users, and it creates an impression of potential unintended functionality or an error/bug.
2. unintended token draining.
3. Disproportional impact on small transfers. The hardcoded value of 1 token is likely to have a much larger impact on small transfers than on larger ones, potentially leading to zero transfers in some cases.

Recommended mitigation Consider adding a natspec comment to explain the purpose of the hardcoded value, or replacing it with a dynamic value that can be set by the owner or passed as a parameter to the function. This would make the function more flexible and transparent, and reduce the risk of unintended behavior or errors.

[H-13] Reverse path in `swapTokensForEth` causes swaps to fail

Description This function's purpose is to swap tokens for ETH, but the path is set in reverse order instead of the correct one. That means that every swap called through `swapTokensForEth` will likely fail, as the path is invalid. **Impact** Every call to `swapTokensForEth` will fail, as the path is set in reverse order. **Proof of Concept** In the `swapTokensForEth` function, the order of addresses in the path is crucial because it tells Uniswap how to route the swap. The path array specifies the sequence of token conversions, and guides the router through the process. Here's how the path order impacts the execution and why an incorrect path will cause the swap to fail.

Uniswap Path Mechanics 1. Path Array Structure: Each element in the path array represents a token in the swap sequence.

```
Starting Token (path[0]): This should be the token being swapped, which in
this case is the contract's token (address(this)).
Ending Token (path[path.length - 1]): This should be the token you want to
end up with after the swap, which in this case is WETH (wrapped ETH), so
you can receive ETH.

2. Order of Execution in the Path:
```

```
Uniswap uses this path to determine the pools it will access for the swap.
For example, a path of `[TokenA, WETH]` tells Uniswap to swap TokenA for
WETH.
The router needs a clear path from TokenA to WETH in the liquidity pools
to perform the swap. If the path doesn’t start with TokenA, Uniswap will
not find the correct route and will throw an error.
```

Why the Incorrect Path Fails In the original swapTokensForEth function, the path was set as:

```
path[0] = uniswapV2Router.WETH();
path[1] = address(this);
```

This tells Uniswap that it should be swapping from WETH rather than the contract’s token, which is not the intended operation, potentially leading to uniswap not finding a valid path and a failed tx. **Recommended mitigation** Swap the order of the addresses in the path array.

```diff
- path[0] = uniswapV2Router.WETH();
- path[1] = address(this);
+ path[0] = address(this);
+ path[1] = uniswapV2Router.WETH();
```

[H-14] Incorrect path in `swapTokensForCake` leads to failed swaps (Root + Impact)

Description Similar as above, the path set in the `swapTokensForCake` function is incorrect, leading to failed swaps. The function is meant to use the swap to first swap the contract’s token for WETH, and then WETH for CAKE. However, the path is set in incorrect order, breaking the intended functionality.

Recommended mitigation Make the following change.

```diff
- path[0] = address(this);
- path[1] = rewardToken;
- path[2] = uniswapV2Router.WETH();
+ path[0] = address(this);
+ path[1] = uniswapV2Router.WETH();
+ path[2] = rewardToken;
```

Medium

[M-1] Undeclared `TokenCreated` event in the constructor leads to loss of trust and readability

Description The creation of the token is declared in the end of the constructor, but the event `TokenCreated` is not declared in the contract. This leads to a loss of trust and readability of the contract.

Proof of Concept The event emitted in the end of the constructor is not declared along with the other events in the contract.

```
emit TokenCreated(owner(), address(this), TokenType.antiBotBaby, VERSION);
```

Recommended mitigation Consider adding the following event declaration in the contract:

```diff
+ event TokenCreated(address indexed owner, address indexed token,
TokenType tokenType, uint256 version);
```

[M-2] Lack of a setter function for `rewardToken` impairs the flexibility and adaptability of the contract

Description While not directly risking the security of the protocol, it is advisable to follow best practices and have a setter function for `rewardToken`. This would allow the owner to change the reward token in the future if needed, without having to deploy a new contract.

Recommended mitigation Consider making the following addition to the contract:

```
// event declaration
event RewardTokenUpdated(address indexed oldRewardToken, address indexed
newRewardToken);

function setRewardToken(address newRewardToken) external onlyOwner {
    require(newRewardToken != address(0), "no address 0");

    emit RewardTokenUpdated(rewardToken, newRewardToken);
    rewardToken = newRewardToken;
}
```

Low

Informational

[I-1] Missing events declaration and events emissions impairs transparency

Description Throughout the contract there are critical events that are neither declared nor emitted. This impairs transparency and makes it difficult to track the contract's behavior.

Recommended mitigation Consider tracking the functions missing the emission of events like `setMarketingWallet` or `setBlacklist` and add both the declaration and the emission of the events.

[I-2] Hardcoded address of the PancakeSwap Router in the constructor must be dealt with caution immediately after the deployment.

Description The contract has a hardcoded address of the PancakeSwap Router in the constructor. This address must be updated immediately after the deployment to avoid any potential security risks.

Recommended mitigation When deploying to any other network such as the mainnet or a different testnet, you should:

1. Call updateUniswapV2Router immediately after deployment to set the correct router address for the target network.
2. Ensure that any liquidity or swap functions are only called after updating the router address to avoid issues with an invalid or incorrect router.

Gas

[G-1] Redundant path of 3 addresses in `swapTokensForEth`

Description The path in `swapTokensForEth` is redundant and can be simplified to a two addresses, from the token to Weth, to consume less gas.

```
address[] memory path = new address[](3);
```

Recommended mitigation Consinder the following change

```diff
- address[] memory path = new address[](3);
+ address[] memory path = new address[](2);
```

[G-2] Redundant transfer within the `_transfer` function when `amount == 0` leads to unecessary consumption of gas instead of exiting early with a require statement.

Description the way the code works now means that when the amount is 0, the function will deduct 0 token from the sender's balance and then add 0 token to the receipient's balance, consuming uneccessary gas.

Recommended mitigation Add a require statement to check if the amount is greater than 0 and exit early if it is not.

```diff
+ require(amount > 0, "Amount must be greater than 0");
```