Black Paper

Security audit

# Doors3

First security audit
Date : May 31th, 2023

# Summary

# I. Introduction

## 1. About Black Paper

Black Paper has been created to help developer teams. Our goal is to help you to make your smart contract safer.
Cybersecurity requires specific expertise which is very different from smart contract development logic. To ensure everything is well fixed, we stay available to help you.

## 2. Methodology

### a. Preparation

This smart contract is a custom ERC20 token.
No first technical meeting was done before code review.

### b. Audit

Before manually auditing, we pass contracts into automatic tools. This allows us to find some easy-to-find vulnerabilities.
Afterward, we manually go deeper. Every variable and function in the scope are analyzed.

You can find many articles on the lesson website. Here is a snippet list of what we test :

- Constructor Mismatch
- Ownership Takeover
- Redundant Fallback Function
- Overflows & Underflows
- Reentrancy
- Money-Giving Bug
- Blackhole
- Unauthorized Self-Destruct
- Revert DoS
- Unchecked External Call
- Gasless Send
- Send Instead Of Transfer
- Costly Loop
- Use Of Untrusted Libraries
- Use Of Predictable Variables
- Transaction Ordering Dependence
- Deprecated Uses

This is not an exhaustive list since we also focus on logic execution exploits, and help optimizing gas price.

## c. Reporting

Every point in the code is subject to internal discussions with the team. At this stage, a majority of the probable issues have already been identified and documented.

Post the completion of the code review, analysis, and testing, we prepare a report which contains for each vulnerability :

- Explanation
- Severity score
- How to fix it / Recommendation

Here are severity score definitions.

| | |
|---|---|
| Critical | A critical vulnerability is a severe issue that can cause significant damage to the contract and its users. These vulnerabilities are easy to exploit and can result in the loss of funds, theft of sensitive data, or other serious consequences. Immediate attention is required to address these vulnerabilities. |
| Major | A major vulnerability is an issue that can cause significant problems for the contract and its users, but not to the same extent as a critical vulnerability. These vulnerabilities are also easy to exploit and may result in the loss of funds or other negative consequences, but they can be mitigated with timely action. |
| Medium | A medium vulnerability is an issue that could potentially cause problems for the contract and its users, but the difficulty to exploit is higher than major or critical vulnerabilities. These vulnerabilities may pose a risk to the contract's functionality or security, but they can be addressed without causing significant disruption. |
| Low | A low vulnerability is a minor issue that does not pose a significant risk to the contract or its users. These vulnerabilities are difficult to exploit and may be cosmetic or technical in nature, but they do not compromise the contract's security or functionality. |
| Informational | An informational finding is not a vulnerability but rather a suggestion or recommendation for improvement. These findings may include best practices for contract design, suggestions for improving code readability, or other non-critical issues. While not urgent, addressing these findings can help to optimize the contract's performance and reduce the risk of future vulnerabilities. |

## 3. Disclaimer

In this audit, we sent all vulnerabilties found by our team. **We can't guarantee all vulnerabilities have been found.**

## 4. Scope

The scope of the audit is :
- *d3v3 (1).sol* which was send by email

We assume that the following smart contracts don't need to be audited and come from OpenZeppelin libraries :

- ERC20.sol
- SafeMath.sol

## II. Vulnerabilities

### Critical

2 critical severity issues were found :

- CRIT-1 Denial of service due to out of gas
- CRIT-2 Missing verifications

### Major

3 major severity issues were found :

- MAJ-1 User funds loss
- MAJ-2 Division by zero
- MAJ-3 Locked stacking

### Medium

4 medium severity issues were found :

- MED-1 Centralization risks in owner role
- MED-2 Unfair MATIC distribution
- MED-3 Unfair reward distribution
- MED-4 Front running benefits
- MED-5 Stacked token freezing

### Low

4 low severity issues were found :

- LOW-1 Owner should be set with OpenZeppelin standard
- LOW-2 Useless complexity in maximum supply
- LOW-3 Unused state
- LOW-4 Bad use of mapping variable

### Informational

10 informational severity issues were found :

- INF-1 Too many digit
- INF-2 Use ERC20 standard could improve readability
- INF-3 Useless informations in event
- INF-4 Useless library
- INF-5 Inconsistent parameter name

- INF-6 Some variables should be constant
- INF-7 Owner address can be immutable
- INF-8 Unoptimized verification
- INF-9 buyTokens could revert due to overflow
- INF-10 Comments should follow NatSpec format

## CRIT-1 Denial of service due to out of gas

*Impact: Critical*

### Description:

In `deposit` and `removeAuthorizedAddress` functions, for loop are used. The number of iterations depends on `employeeAddresses` length. It can lead to DoS due to gas limitation.

To fix the `deposit` function DoS, the only option would be to call `removeAuthorizedAddress`. Unfortunately, in this case `removeAuthorizedAddress` would also be DoS.

This issue could appear when `employeeAddresses` length hit a few thousand.

### Recommendation:

We highly recommend not using loops when they are not necessary. In addition to this vulnerability, gas costs will increase too much.

1/ To resolve `deposit` DoS, we recommend adding a batch function that allows the `owner` to deposit by batch. Warning : it is important to ensure that all deposits have been done before being able to do anything else on the contract.

2/ To resolve `removeAuthorizedAddress` DoS, we recommend adding a mapping variable employeeIndex storing the index of each employee in `employeeAddresses` array. Some changes also need to be done in other functions to ensure `employeeIndex` is well saved.

Then, for example, the following code:

```
        for (uint256 i = 0; i < employeeAddresses.length; i++) {
            if (employeeAddresses[i] == addr) {
                employeeAddresses[i] = employeeAddresses[employeeAddresses.length - 1]; // Replace the
address to be removed with the last address in the array
                employeeAddresses.pop(); // Remove the last address (duplicate) from the array
                break;
            }
        }
```

can be replaced by:

```
        employeeAddresses[employeeIndex[addr]] = employeeAddresses[employeeAddresses.length - 1];
        employeeAddresses.pop();
```

Moreover, we recommend removing the first loop:

```
        for (uint256 i = 0; i < employeeAddresses.length; i++) {
            if (employeeAddresses[i] != addr) {
                employeeMATICShare[employeeAddresses[i]] += additionalSharePerEmployee; // Add the
additional share to each remaining employee
            }
        }
```

Because the owner is able to dispatch shares in the `deposit` function in another transaction, this avoids out of gas vulnerability.

## CRIT-2 Missing verifications

*Impact: Critical*

### Description:

Owner can add the same employee address multiple times.

```
function addAuthorizedAddress(address addr) public onlyOwner {
    authorizedAddresses[addr] = true;
    totalEmployees += 1;
    employeeAddresses.push(addr);
    emit AuthorizedAddressAdded(addr);
}
```

It can be done by mistake. If this happens, there is no way to revert back. It can lead to many unwanted behaviors (multiple shares when calling the deposit, wrong `totalemployees`, …).

### Recommendation:

We recommend adding a condition at the beginning of the function.

```
 function addAuthorizedAddress(address addr) public onlyOwner {
    require(!authorizedAddresses[addr])
    authorizedAddresses[addr] = true;
    totalEmployees += 1;
    employeeAddresses.push(addr);
    emit AuthorizedAddressAdded(addr);
}
```

## MAJ-1 User funds loss

*Impact: Critical*

### Description:

In `buyTokens` function, token equivalent is calculated with the following code:

```
uint256 tokenAmount = ethAmount.mul(conversionRate).div(1 ether);
```

The result is not necessarily exact because this is an integer division.
Users can lose MATIC by sending more tokens than needed.

### Recommendation:

We recommend modifying how conversion is calculated. What can be done is to avoid division by 1 ether and replace it with a smaller value.
The calculation depends on `conversionRate` and ERC20 tokens decimals. At least, the function should be reverted when: `(ethAmount * conversionRate) % (1 ether) != 0`.

## MAJ-2 Division by zero

*Impact: Major*

### Description:

If there is one employee, authorization can’t be removed.

```
        uint256 additionalSharePerEmployee = removedEmployeeShare / (totalEmployees - 1); // Distribute
the MATIC share of the removed employee among the remaining employees
```

### Recommendation:

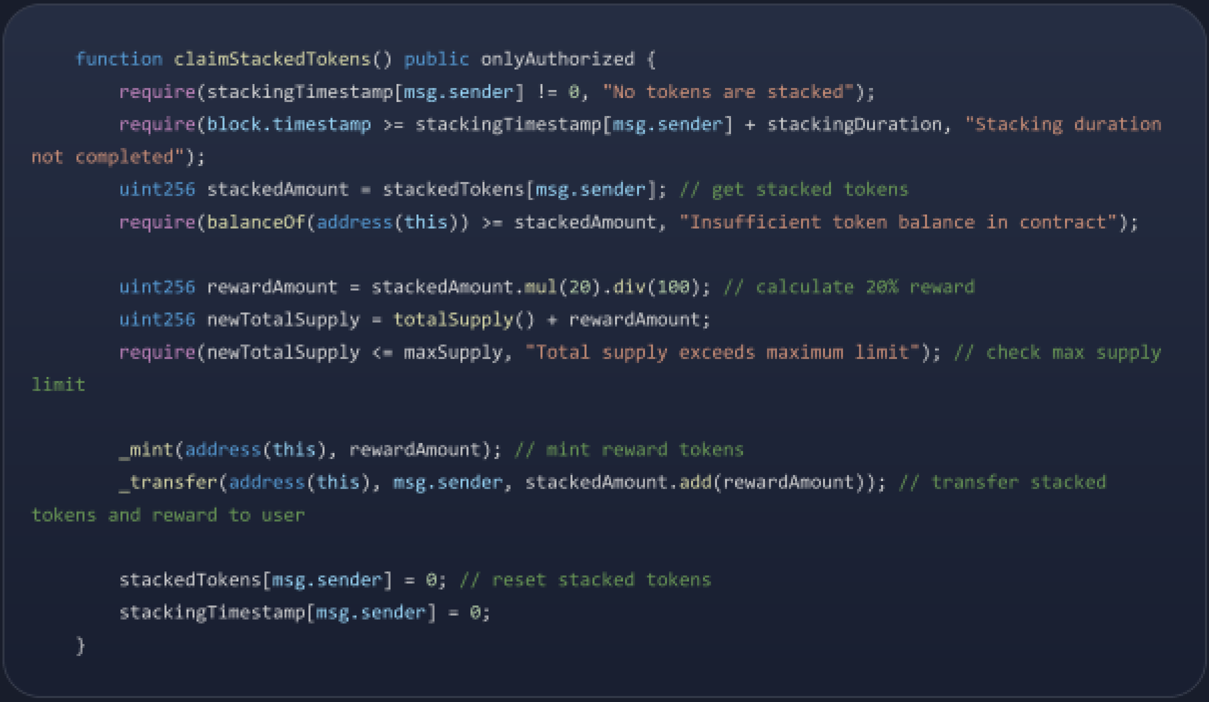
We recommend adding a condition at the beginning of the function and adding a workflow for cases when there is 0 or 1 employee.

## MAJ-3 Locked stacking

*Impact: Major*

### Description:

In the case maximum supply is reached, there is no way for employees to claim stacked tokens.



```
    function claimStackedTokens() public onlyAuthorized {
        require(stackingTimestamp[msg.sender] != 0, "No tokens are stacked");
        require(block.timestamp >= stackingTimestamp[msg.sender] + stackingDuration, "Stacking duration
not completed");
        uint256 stackedAmount = stackedTokens[msg.sender]; // get stacked tokens
        require(balanceOf(address(this)) >= stackedAmount, "Insufficient token balance in contract");

        uint256 rewardAmount = stackedAmount.mul(20).div(100); // calculate 20% reward
        uint256 newTotalSupply = totalSupply() + rewardAmount;
        require(newTotalSupply <= maxSupply, "Total supply exceeds maximum limit"); // check max supply
limit

        _mint(address(this), rewardAmount); // mint reward tokens
        _transfer(address(this), msg.sender, stackedAmount.add(rewardAmount)); // transfer stacked
tokens and reward to user

        stackedTokens[msg.sender] = 0; // reset stacked tokens
        stackingTimestamp[msg.sender] = 0;
    }
```

### Recommendation:

This vulnerability needs to change smart contract design. Here are a two proposals:

- not use a capped supply and allow minting only with stacking
- add an emergency function to allow users to unstack tokens without rewards

## MED-1 Centralization risks in owner role

*Impact: Medium*

### Description:

The owner role can be assigned to a single externally owned account (EOA). It can lead to centralization and an increased risk of private key leaks.

### Recommendation:

To mitigate this risk, we recommend using a multisignature wallet that is jointly owned by multiple individuals. This would distribute control and reduce the likelihood of a single point of failure.

## MED-2 Unfair MATIC distribution

*Impact: Medium*

### Description:

In `exchangeTokensForETH` function, token equivalent is calculated with the following code:

```
        uint256 ethAmount = amount.mul(1 ether).div(conversionRate); // Calculate the equivalent ETH
amount
```

The result is not necessarily exact because this is an integer division. A user can have less than expected MATIC distribution.

### Recommendation:

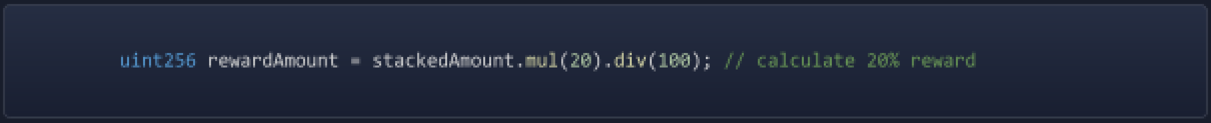
We recommend to revert call when: `(amount * 1 ether) % conversionRate != 0`.

## MED-3 Unfair reward distribution

*Impact: Medium*

### Description:

In `claimStackedTokens` function, token equivalent is calculated with the following code:



```
uint256 rewardAmount = stackedAmount.mul(20).div(100); // calculate 20% reward
```

The result is not necessarily exact because this is an integer division. A user can have less than expected reward distribution.

### Recommendation:

We recommend reverting the call in `stackTokens` when : `amount % 5!= 0`.

## MED-4 Front running benefits

*Impact: Medium*

### Description:

The owner can change the conversion rate. Seeing this transaction, a malicious user could for example exchange tokens for MATIC in a previous transaction with higher gas.

### Recommendation:

We recommend adding a maximum gap in `setConversionRate` function to mitigate malicious user benefits.

## MED-5 Stacked token freezing

*Impact: Medium*

### Description:

The owner can call the `removeAuthorizedAddress` function. In the case the user is staking tokens, he will not be authorized to unstake because of `onlyAuthorized` modifier.

### Recommendation:

We recommend removing the `onlyAuthorized` modifier on the `claimStackedTokens` function.

## LOW-1 Owner should be set with OpenZeppelin standard

*Impact: Low*

### Description:

OpenZeppelin contracts are used in this contract. There is no reason to don't use the Ownable OpenZeppelin smart contract : *https://github.com/OpenZeppelin/*
Because this smart contract is used by most solidity projects, it decreases the probability of vulnerability. Moreover, it adds readability.

### Recommendation:

We recommend using Ownable.sol OpenZeppelin smart contract.

```
import "@openzeppelin/contracts/access/Ownable.sol";
```

Afterwards, make labTOKEN2 inherit from Ownable.

## LOW-2 Useless complexity in maximum supply

*Impact: Low*

### Description:

OpenZeppelin contracts are used in this contract. There is no reason to don't use the ERC20capped OpenZeppelin extension : *https://github.com/OpenZeppelin/*
Because this smart contract is used by most solidity projects, it decreases the probability of vulnerability. Moreover, it adds a lot of readability.

### Recommendation:

We recommend using Ownable.sol OpenZeppelin smart contract.

```
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Capped.sol";
```

Then, many parts of the code can be simplified. For example, the following code:

```
        uint256 newTotalSupply = totalSupply() + _amount;
        require(newTotalSupply <= maxSupply, "Total supply exceeds maximum limit");
        _mint(_to, _amount);
```

Will be replaced by:

```
        _mint(_to, _amount);
```

## LOW-3 Unused state

*Impact: Low*

### Description:

`stackingRewardPercentage` is not used.

### Recommendation:

It seems that it can be used in line 190.

```
uint256 rewardAmount = stackedAmount.mul(20).div(100); // calculate 20% reward
```

## LOW-4 Bad use of mapping variable

*Impact: Low*

### Description:

`availableTokens` is only used to store `availableTokens[address(this)]` value. This is highly confusing. Moreover, even to add a uint256 variable adds too much complexity.

### Recommendation:

The best option should be to use `balanceOf(address(this))` to know available balance. The only reason to not do so is because of how staking works. A user could use staked tokens in `exchangeTokensForETH` and `buyTokens` which can lead to missing funds when claiming.

We recommend changing the way stacking works by using `_burn` and `_mint`.

```
function stackTokens(uint256 amount) public onlyAuthorized onlyNotStacked {
        require(balanceOf(msg.sender) >= amount, "Insufficient token balance");
        _burn(msg.sender, amount);
        stackedTokens[msg.sender] += amount; // update stacked tokens
        stackingTimestamp[msg.sender] = block.timestamp;
    }


function claimStackedTokens() public onlyAuthorized {
        require(stackingTimestamp[msg.sender] != 0, "No tokens are stacked");
        require(block.timestamp >= stackingTimestamp[msg.sender] + stackingDuration, "Stacking duration
not completed");
        uint256 stackedAmount = stackedTokens[msg.sender]; // get stacked tokens

        uint256 rewardAmount = stackedAmount.mul(20).div(100); // calculate 20% reward
        uint256 newTotalSupply = totalSupply() + rewardAmount;
        require(newTotalSupply <= maxSupply, "Total supply exceeds maximum limit"); // check max supply
limit

        _mint(msg.sender, rewardAmount + stackedAmount); // mint reward tokens

        stackedTokens[msg.sender] = 0; // reset stacked tokens
        stackingTimestamp[msg.sender] = 0;
    }
```

## INF-1 Too many digit

*Impact: Informational*

### Description:

`maxSupply` and `initialSupply` are hard to read because of too many digits. It's easy to misread the number of zeros in big numbers.

```
uint256 public maxSupply = 21000000 * 10 ** 18;
```

```
uint256 initialSupply = 11000000 * 10 ** 18;
```

### Recommendation:

We recommend using scientific notation.

```
uint256 public maxSupply = 21e6 * 10 ** 18;
```

```
uint256 initialSupply = 11e6 * 10 ** 18;
```

## INF-2 Use ERC20 standard could improve readability

*Impact: Informational*

### Description:

`decimals()` can be used to help external users to have a better understanding.

```
uint256 public maxSupply = 21000000 * 10 ** 18;
```

```
uint256 initialSupply = 11000000 * 10 ** 18;
```

### Recommendation:

We recommend using scientific notation.

```
uint256 public maxSupply = 21000000 * 10 ** decimals();
```

```
uint256 initialSupply = 11000000 * 10 ** decimals();
```

## INF-3 Useless informations in event

*Impact: Informational*

### Description:

`Deposit` and `Withdrawal` events can only be triggered with `owner` as `user`. Because the owner can't be changed, there is no reason to add it, and even more to add the `indexed` keyword.

```
event Deposit(address indexed user, uint256 amount);
event Withdrawal(address indexed user, uint256 amount);
```

### Recommendation:

LOW-1 vulnerability recommendations also allow the owner to change. If this is done, then no other change needs to be done.
Else, `user` parameter can be removed from those two events.

## INF-4 Useless library

*Impact: Informational*

### Description:

Since the compiler version is ^0.8.0. There is no need to use the SafeMath library.

```
using SafeMath for uint256;
```

### Recommendation:

We recommend not using SafeMath for basic mathematical operations in order to win readability and save gas.

## INF-5 Inconsistent parameter name

*Impact: Informational*

### Description:

In `buyTokens` function, `ethAmount` parameter can lead to misunderstanding.

```
function buyTokens(uint256 ethAmount) public payable onlyAuthorized {
```

### Recommendation:

We recommend to use `weiAmount` name or to directly use `msg.value` as a unique input.

## INF-6 Some variables should be constant

*Impact: Informational*

### Description:

Those variables can't be changed:

- `stackingDuration`
- `maxSupply`
- `stackingRewardPercentage`

They should be constant in order to win readability and save gas.

### Recommendation:

We recommend using constant keywords.

## INF-7 Owner address can be immutable

*Impact: Informational*

### Description:

Because the owner address is set on the constructor, and can’t be changed, it could be immutable.

### Recommendation:

LOW-1 recommendations fix this optimization. Else, we recommend using the immutable keyword.

## INF-8 Unoptimized verification

*Impact: Informational*

### Description:

In `setConversionRate` function, code can be optimized to save gas.

```
function setConversionRate(uint256 newRate) public onlyOwner {
    require(newRate > 0, "Conversion rate must be greater than 0");
    conversionRate = newRate;
    emit NewRate(conversionRate);
}
```

### Recommendation:

We recommend to replace by the following code:

```
function setConversionRate(uint256 newRate) public onlyOwner {
    require(newRate != 0, "Conversion rate must be different from 0");
    conversionRate = newRate;
    emit NewRate(conversionRate);
}
```

## INF-9 `buyTokens` could revert due to overflow

*Impact: Informational*

### Description:

`tokenAmount` could overflow when being calculated.

```
        uint256 tokenAmount = ethAmount.mul(conversionRate).div(1 ether);
```

### Recommendation:

If this happens, this means it would else be revert because maximum supply is 21e6*10**18.
But the error “insufficient tokens available” would not be triggered.
We don’t recommend doing anything on this point. Just keep it in mind for next updates or if maximum supply will not be the same in production.

## INF-10 Comments should follow NatSpec format

*Impact: Informational*

### Description:

Comments don’t follow NatSpec format.

### Recommendation:

Change comments to respect Solidity documentation recommendations :
https://docs.soliditylang.org/en/v0.8.17/natspec-format.html