

---

*BRISDividendTracker.sol*

---

## Medium severity issues

### • Owner Role / Centralized Risk

Only the owner role has authority over the functions shown below. Any compromise to the `_owner` account may allow the hacker to take advantage of this authority. And in the constructor, the deployer wallet is excluded from paying any fee.

- ✓ `excludeFromDividends`
- ✓ `updateClaimWait`
- ✓ `setBalance`
- ✓ `processAccount`

#### ✓ Recommendation

The risk describes the current project design and potentially makes iterations to improve the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multi-signature wallets.

## Low severity issues

### • Usage Of `block.timestamp`

`Block.timestamp` is used in the contract. The variable `block` is a set of variables. The timestamp does not always reflect the current time and may be inaccurate. The value of a block can be influenced by miners. Maximal Extractable Value attacks require a timestamp of up to 900 seconds. There is no guarantee that the value is right, all that is guaranteed is that it is higher than the timestamp of the previous block.

- ✓
- ✓ Recommendation

You can use an Oracle to get the exact time.

## Medium severity issues

### • Reentrancy Guard Not Used

A reentrancy attack can occur when the contract creates a function that makes an external call to another untrusted contract before resolving any effects. If the attacker can control the untrusted contract, they can make a recursive call back to the original function, repeating interactions that would have otherwise not run after the external call resolved the effects.

In the `_withdrawDividendOfUser` function, we are using `.call()` to send the native token to another address. We hardcoded the gas value to 3000 but because this is an external call and the destination address is not verified, we have to care about this.



```
1 function _withdrawDividendOfUser(address payable user)
2 {
3     internal
4     returns (uint256)
5 {
6     uint256 _withdrawableDividend = withdrawableDividendOf(user);
7     if (_withdrawableDividend > 0) {
8         withdrawDividends[user] = withdrawDividends[user].add(
9             _withdrawableDividend
10        );
11        emit DividendWithdrawn(user, _withdrawableDividend);
12        (bool success, ) = user.call{
13            value: _withdrawableDividend,
14            gas: 3000
15        }("");
16
17        if (!success) {
18            withdrawDividends[user] = withdrawDividends[user].sub(
19                _withdrawableDividend
20            );
21            return 0;
22        }
23
24        return _withdrawableDividend;
25    }
26    return 0;
27 }
```

### ✓ Recommendation

We recommend using the Checks-Effects-Interactions Pattern to avoid the risk of calling unknown contracts or applying OpenZeppelin ReentrancyGuard library - `nonReentrant` modifier for the aforementioned functions to prevent reentrancy attack.

## Low severity issues

### • Missing Emit Events

There should always be events emitted in the sensitive functions.

- ✓ processAccount
- ✓ process
- ✓ setBalance

### ✓ Recommendation


It is recommended emitting events for sensitive functions.

## Medium severity issues

### • Reentrancy Guard Not Used

A reentrancy attack can occur when the contract creates a function that makes an external call to another untrusted contract before resolving any effects. If the attacker can control the untrusted contract, they can make a recursive call back to the original function, repeating interactions that would have otherwise not run after the external call resolved the effects.

In the swapAndSendDividends function, we are using .call() to send the native token to another contract. If the owner's wallet gets hacked, the attacker can deploy a new dividendTracker contract containing Malicious code and use a Reentrancy attack.



```
1  function swapAndSendDividends(uint256 tokens) private lockTheSwap {
2      // we get eth bbalance of contract;
3      uint256 initialBalance = address(this).balance;
4      swapTokensForEth(tokens);
5      uint256 dividends = address(this).balance.sub(initialBalance);
6
7      /////////////////////////////////////////////////// ***** EXTERNAL CALL / STAY SAFE *****
8      * ///////////////////////////////////////////////////
9      (bool success, ) = address(dividendTracker).call{value: dividends}("");
10
11     if (success) {
12         emit SendDividends(tokens, dividends);
13     }
14 }
```

### ✓ Recommendation

We recommend using the Checks-Effects-Interactions Pattern to avoid the risk of calling unknown contracts or applying OpenZeppelin ReentrancyGuard library - nonReentrant modifier for the aforementioned functions to prevent reentrancy attack.

## • Liquidity Wallet / Centralized Risk

In the constructor function, the liquidityWallet address is set to the deployer address. Any compromise to the \_owner account may allow the hacker to take advantage of this authority. In the following, In the constructor function, owner wallet and liquidityWallet are excluded from fees and dividends.

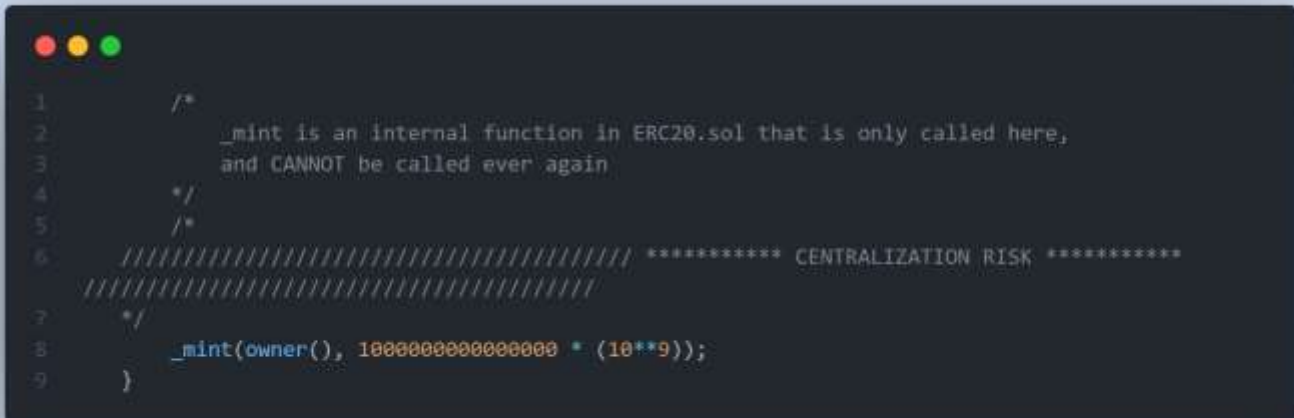


### ✓ Recommendation

The risk describes the current project design and potentially makes iterations to improve the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multi-signature wallets.

## • Initial Token Distribution / Centralization

In the constructor function, the deployer is minting tokens for his wallet address. All the initial supply are sent to the contract deployer when deploying the contract. This could be a centralization risk, the deployer can distribute those tokens without obtaining the consensus of the community.



```
1      /*
2      _mint is an internal function in ERC20.sol that is only called here,
3      and CANNOT be called ever again
4      */
5      /*
6      /////////////////////////////////// ***** CENTRALIZATION RISK *****
7      ///////////////////////////////////
8      */
9      _mint(owner(), 1000000000000000 * (10**9));
10 }
```

### ✓ Recommendation

We recommend the team to be transparent regarding the initial token distribution process, and the team shall make enough efforts to restrict the access of the private key, We also advise the client to adopt Multisig, Timelock, and/or DAO in the project to manage the specific account in this case.

## • Owner Role / Centralized Risk

Only the owner role has authority over the functions shown below. Any compromise to the \_owner account may allow the hacker to take advantage of this authority. And in the constructor, the deployer wallet is excluded from paying any fee.

- ✓ updateDividendTracker
- ✓ updateUniswapV2Router
- ✓ excludeFromFees
- ✓ excludeMultipleAccountsFromFees
- ✓ setAutomatedMarketMakerPair
- ✓ updateLiquidityWallet
- ✓ updateGasForProcessing
- ✓ updateClaimWait
- ✓ withdraw
- ✓ setMaxSellTxAMount
- ✓ setSwapTokensAmt
- ✓ setBNBRewardsFee
- ✓ setMarketingFee
- ✓ setMarketingWallet
- ✓ addToBlackList
- ✓ removeFromBlackList
- ✓ setSwapEnabled
- ✓ setBuyBackFee
- ✓ setBuyBackEnabled
- ✓ setBuybackUpperLimit

### ✓ Recommendation

The risk describes the current project design and potentially makes iterations to improve the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multi-signature wallets.

## Low severity issues

- **Unnecessary code**

In the function `_transfer`, line 360, we doubled checked balance more than `buyBackUpperLimit`.



```
1  if (buyBackEnabled && balance > uint256(1 * 10**15)) {
2      // why we check again balance should be more than 1* 10**15
3      if (balance > buyBackUpperLimit) balance = buyBackUpperLimit;
4
5      // call buy back function;
6      buyBackTokens(balance.div(100));
7  }
```

- ✓ **Recommendation**

We can remove the second if condition.

- **Third Party Dependency**

The contract is serving as the underlying entity to interact with one or more third party protocols. The scope of the audit treats third party entities as black boxes and assume their functional correctness. However, in the real world, third parties can be compromised and this may lead to lost or stolen assets. In addition, upgrades of third parties can possibly create severe impacts, such as increasing fees of third parties, migrating to new LP pools, etc.

- ✓ uniswapV2Router

- ✓ **Recommendation**

We understand that the business logic requires interaction with the third parties. We encourage the team to constantly monitor the statuses of third parties to mitigate the side effects when unexpected activities are observed.



## ● Potential Sandwich Attacks

A sandwich attack might happen when an attacker observes a transaction swapping tokens or adding liquidity without setting restrictions on slippage or minimum output amount. The attacker can manipulate the exchange rate by frontrunning (before the transaction is attacked) a transaction to purchase one of the assets and make profits by back running (after the transaction is attacked) a transaction to sell the asset after the transaction is attacked.

The following functions are called without setting restrictions on slippage or minimum output amount, so transactions triggering these functions are vulnerable to sandwich attacks, especially when the input amount is large:

- ✓ `uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens()`
- ✓ `uniswapV2Router.addLiquidityETH()`

### ✓ Recommendation

We recommend setting reasonable minimum output amounts, instead of 0, based on token prices when calling the aforementioned functions.

## ● Missing Emit Events

The function `updateUniswapV2Router ()` updates the new router without updating the following related states:

- ✓ `dividendTracker.excludeFromDividends(address(_uniswapV2Router));`

### ✓ Recommendation

We advise the client to recheck the function.

## ● Missing Zero Address Validation

Addresses should be checked before assignment or external call to make sure they are not zero addresses.

`updateDividendTracker`, `updateUniswapV2Router`, `excludeFromFees`, `excludeMultipleAccountsFromFees`, `setAutomatedMarketMakerPair`, `updateLiquidityWallet`,

### ✓ Recommendation

We advise adding a zero-check for the passed-in address value to prevent unexpected errors.

## • Missing Emit Events

There should always be events emitted in the sensitive functions.

- ✓ swapTokensForEth
- ✓ swapAndSendToMarketing
- ✓ updateClaimWait
- ✓ withdraw
- ✓ setMaxSellTxAMount
- ✓ setSwapTokensAmt
- ✓ setBNBRewardsFee
- ✓ setMarketingFee
- ✓ setMarketingWallet
- ✓ addToBlackList
- ✓ removeFromBlackList
- ✓ setSwapEnabled
- ✓ setBuyBackFee
- ✓ setBuyBackEnabled
- ✓ setBuybackUpperLimit

### ✓ Recommendation

It is recommended emitting events for sensitive functions.

## • Unlocked Pragma Used

Contracts should be deployed with the same compiler version and flags that they have been tested the most with. Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, the latest compiler which may have higher risks of undiscovered bugs. Contracts may also be deployed by others and the pragma indicates the compiler version intended by the original authors.

```
// bad
pragma solidity ^6.4.4;

// good
pragma solidity 6.4.4;
```

### ✓ Recommendation

Should lock pragmas to a specific compiler version.