MythX

REPORT 60F9083AFDA7880018695000

Created	Thu Jul 22 2021 05:55:06 GMT+0000 (Coordinated Universal Time)
Number of analyses	1
User	60f906b2a6e184dcafc6e947

REPORT SUMMARY

Analyses ID	Main source file	Detected vulnerabilities
440eab94-f70a-4d02-8f6c-f5424c80c3c7	BaronToken.sol	22

Analysis `440eab94-f70a-4d02-8f6c-f5424c80c3c7`

MythX

Started	Thu Jul 22 2021 05:55:17 GMT+0000 (Coordinated Universal Time)
Finished	Thu Jul 22 2021 06:41:06 GMT+0000 (Coordinated Universal Time)
Mode	Deep
Client Tool	Remythx
Main Source File	`BaronToken.Sol`

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	19	3

ISSUES

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "renounceOwnership" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

`BaronToken.sol`

Locations

```
* thereby removing any functionality that is only available to the owner.
*/
function renounceOwnership() public virtual onlyOwner {
emit OwnershipTransferred(_owner, address(0));
_owner = address(0);
}

/**
```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "transferOwnership" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

BaronToken.sol

Locations

```
* Can only be called by the current owner.
*/
function transferOwnership(address newOwner) public virtual onlyOwner {
require(newOwner != address(0), "Ownable: new owner is the zero address");
emit OwnershipTransferred(_owner, newOwner);
_owner = newOwner;
}
}

```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "decimals" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

BaronToken.sol

Locations

```
* @dev Returns the token decimals.
*/
function decimals() public override view returns (uint8) {
return _decimals;
}

/**
```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "symbol" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

BaronToken.sol

Locations

```
* @dev Returns the token symbol.
*/
function symbol() public override view returns (string memory) {
return _symbol;
}

/**
```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "totalSupply" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

BaronToken.sol

Locations

```
* @dev See {BEP20-totalSupply}.
*/
function totalSupply() public override view returns (uint256) {
return _totalSupply;
}

/**
```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "transfer" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

BaronToken.sol

Locations

```
* - the caller must have a balance of at least `amount`.
*/
function transfer(address recipient, uint256 amount) public override returns (bool) {
_transfer(_msgSender(), recipient, amount);
return true;
}

/**
```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "allowance" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

BaronToken.sol

Locations

```
* @dev See {BEP20-allowance}.
*/
function allowance(address owner, address spender) public override view returns (uint256) {
return _allowances[owner][spender];
}

/**
```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "approve" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

BaronToken.sol

Locations

```
* - `spender` cannot be the zero address.
*/
function approve(address spender, uint256 amount) public override returns (bool) {
_approve(_msgSender(), spender, amount);
return true;
}

/**
```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "transferFrom" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

BaronToken.sol

Locations

```
* `amount`.
*/
function transferFrom(
address sender,
address recipient,
uint256 amount
) public override returns (bool) {
_transfer(sender, recipient, amount);
_approve(
sender,
_msgSender(),
_allowances[sender][_msgSender()].sub(amount, 'BEP20: transfer amount exceeds allowance')
);
return true;
}

/**
```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "increaseAllowance" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

BaronToken.sol

Locations

```
* - `spender` cannot be the zero address.
*/
function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
_approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
return true;
}

/**
```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "decreaseAllowance" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

BaronToken.sol

Locations

```
* `subtractedValue`.
*/
function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool) {
_approve(
_msgSender(),
spender,
_allowances[_msgSender()][spender].sub(subtractedValue, 'BEP20: decreased allowance below zero')
);
return true;
}

/**
```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "mint" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

BaronToken.sol

Locations

```
* - `msg.sender` must be the token owner
*/
function mint(uint256 amount) public onlyOwner returns (bool) {
_mint(_msgSender(), amount);
return true;
}

/**
```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "mint" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

BaronToken.sol

Locations

```

/// @notice Creates `_amount` token to `_to`. Must only be called by the owner (MasterChef).
function mint(address _to, uint256 _amount) public onlyOwner {
_mint(_to, _amount);
_moveDelegates(address(0), _delegates[_to], _amount);
}

/// @dev overrides transfer function to meet tokenomics of BARON
```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "updateTransferTaxRate" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

BaronToken.sol

Locations

```
* Can only be called by the current operator.
*/
function updateTransferTaxRate(uint16 _transferTaxRate) public onlyOperator {
require(_transferTaxRate <= MAXIMUM_TRANSFER_TAX_RATE, "BARON::updateTransferTaxRate: Transfer tax rate must not exceed the maximum rate.");
emit TransferTaxRateUpdated(msg.sender, transferTaxRate, _transferTaxRate);
transferTaxRate = _transferTaxRate;
}

/**
```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "updateBurnRate" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

BaronToken.sol

Locations

```
* Can only be called by the current operator.
*/
function updateBurnRate(uint16 _burnRate) public onlyOperator {
require(_burnRate <= 100, "BARON::updateBurnRate: Burn rate must not exceed the maximum rate.");
emit BurnRateUpdated(msg.sender, burnRate, _burnRate);
burnRate = _burnRate;
}


```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "updateMinAmountToLiquify" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

BaronToken.sol

Locations

```
* Can only be called by the current operator.
*/
function updateMinAmountToLiquify(uint256 _minAmount) public onlyOperator {
emit MinAmountToLiquifyUpdated(msg.sender, minAmountToLiquify, _minAmount);
minAmountToLiquify = _minAmount;
}

/**
```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "updateSwapAndLiquifyEnabled" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

BaronToken.sol

Locations

```
* Can only be called by the current operator.
*/
function updateSwapAndLiquifyEnabled(bool _enabled) public onlyOperator {
emit SwapAndLiquifyEnabledUpdated(msg.sender, _enabled);
swapAndLiquifyEnabled = _enabled;
}

/**
```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "updateBaronFarmRouter" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

BaronToken.sol

Locations

```
* Can only be called by the current operator.
*/
function updateBaronFarmRouter(address _router) public onlyOperator {
baronFarmRouter = IUniswapV2Router02(_router);
baronFarmPair = IUniswapV2Factory(baronFarmRouter.factory()).getPair(address(this), baronFarmRouter.WETH());
require(baronFarmPair != address(0), "BARON::updateBaronFarmRouter: Invalid pair address.");
emit BaronFarmRouterUpdated(msg.sender, address(baronFarmRouter), baronFarmPair);
}

/**
```

MEDIUM

SWC-000

Function could be marked as external.

The function definition of "transferOperator" is marked "public". However, it is never directly called by another function in the same contract or in any of its descendants. Consider to mark it as "external" instead.

Source file

BaronToken.sol

Locations

```
* Can only be called by the current operator.
*/
function transferOperator(address newOperator) public onlyOperator {
require(newOperator != address(0), "BARON::transferOperator: new operator is the zero address");
emit OperatorTransferred(_operator, newOperator);
_operator = newOperator;
}

// Copied and modified from YAM code:
```

LOW

SWC-120

Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Source file

BaronToken.sol

Locations

```
returns (uint256)
{
require(blockNumber < block.number, "BARON::getPriorVotes: not yet determined");

uint32 nCheckpoints = numCheckpoints[account];
```

LOW

SWC-120

Potential use of "block.number" as source of randonmness.

The environment variable "block.number" looks like it might be used as a source of randomness. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Source file

BaronToken.sol

Locations

```
internal
{
uint32 blockNumber = safe32(block.number, "BARON::_writeCheckpoint: block number exceeds 32 bits");

if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber) {
```

LOW

SWC-123

Requirement violation.

A requirement was violated in a nested call and the call was reverted as a result. Make sure valid inputs are provided to the nested call (for instance, via passed arguments).

Source file

`BaronToken.sol`

Locations

```
function updateBaronFarmRouter(address _router) public onlyOperator {
baronFarmRouter = IUniswapV2Router02(_router);
baronFarmPair = IUniswapV2Factory(baronFarmRouter.factory()).getPair(address(this), baronFarmRouter.WETH());
require(baronFarmPair != address(0), "BARON::updateBaronFarmRouter: Invalid pair address.");
emit BaronFarmRouterUpdated(msg.sender, address(baronFarmRouter), baronFarmPair);
```

Source file

`BaronToken.sol`

Locations

```

// BaronToken with Governance.
contract BaronToken is BEP20 {
// Transfer tax rate in basis points. (default 2%)
uint16 public transferTaxRate = 200;
// Burn rate % of transfer tax. (default 50% x 2% = 1% of total amount).
uint16 public burnRate = 50;
// Max transfer tax rate: 10%.
uint16 public constant MAXIMUM_TRANSFER_TAX_RATE = 1000;
// Burn address
address public constant BURN_ADDRESS = 0x000000000000000000000000000000000000dEaD;

// Addresses that excluded from antiWhale
mapping(address => bool) private _excludedFromAntiWhale;
// Automatic swap and liquify enabled
bool public swapAndLiquifyEnabled = false;
// Min amount to liquify. (default 500 BARON)
uint256 public minAmountToLiquify = 500 ether;
// The swap router, modifiable. Will be changed to BaronFarm's router when our own AMM release
IUniswapV2Router02 public baronFarmRouter;
// The trading pair
address public baronFarmPair;
// In swap and liquify
bool private _inSwapAndLiquify;

// The operator can only update the transfer tax rate
address private _operator;

// Events
event OperatorTransferred(address indexed previousOperator, address indexed newOperator);
event TransferTaxRateUpdated(address indexed operator, uint256 previousRate, uint256 newRate);
event BurnRateUpdated(address indexed operator, uint256 previousRate, uint256 newRate);
event SwapAndLiquifyEnabledUpdated(address indexed operator, bool enabled);
event MinAmountToLiquifyUpdated(address indexed operator, uint256 previousAmount, uint256 newAmount);
event BaronFarmRouterUpdated(address indexed operator, address indexed router, address indexed pair);
event SwapAndLiquify(uint256 tokensSwapped, uint256 ethReceived, uint256 tokensIntoLiqudity);

modifier onlyOperator() {
require(_operator == msg.sender, "operator: caller is not the operator");
_;
}


modifier lockTheSwap {
_inSwapAndLiquify = true;
```

```
_;
_inSwapAndLiquify = false;
}

modifier transferTaxFree {
uint16 _transferTaxRate = transferTaxRate;
transferTaxRate = 0;
_;
transferTaxRate = _transferTaxRate;
}

/**
* @notice Constructs the BaronToken contract.
*/
constructor() public BEP20("BaronFarm Token", "BARON") {
_operator = _msgSender();
emit OperatorTransferred(address(0), _operator);
}

/// @notice Creates `_amount` token to `_to`. Must only be called by the owner (MasterChef).
function mint(address _to, uint256 _amount) public onlyOwner {
_mint(_to, _amount);
_moveDelegates(address(0), _delegates[_to], _amount);
}

/// @dev overrides transfer function to meet tokenomics of BARON
function _transfer(address sender, address recipient, uint256 amount) internal virtual override{
// swap and liquify
if (
swapAndLiquifyEnabled == true
&& _inSwapAndLiquify == false
&& address(baronFarmRouter) != address(0)
&& baronFarmPair != address(0)
&& sender != baronFarmPair
&& sender != owner()
) {
swapAndLiquify();
}

if (recipient == BURN_ADDRESS || transferTaxRate == 0) {
super._transfer(sender, recipient, amount);
} else {
// default tax is 2% of every transfer
uint256 taxAmount = amount.mul(transferTaxRate).div(10000);
uint256 burnAmount = taxAmount.mul(burnRate).div(100);
uint256 liquidityAmount = taxAmount.sub(burnAmount);
require(taxAmount == burnAmount + liquidityAmount, "BARON::transfer: Burn value invalid");

// default 98% of transfer sent to recipient
uint256 sendAmount = amount.sub(taxAmount);
require(amount == sendAmount + taxAmount, "BARON::transfer: Tax value invalid");

super._transfer(sender, BURN_ADDRESS, burnAmount);
super._transfer(sender, address(this), liquidityAmount);
super._transfer(sender, recipient, sendAmount);
amount = sendAmount;
}
}

/// @dev Swap and liquify
function swapAndLiquify() private lockTheSwap transferTaxFree {
uint256 contractTokenBalance = balanceOf(address(this));
if (contractTokenBalance >= minAmountToLiquify) {
```

```
// only min amount to liquify
uint256 liquifyAmount = minAmountToLiquify;

// split the liquify amount into halves
uint256 half = liquifyAmount.div(2);
uint256 otherHalf = liquifyAmount.sub(half);

// capture the contract's current ETH balance.
// this is so that we can capture exactly the amount of ETH that the
// swap creates, and not make the liquidity event include any ETH that
// has been manually sent to the contract
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);

// add liquidity
addLiquidity(otherHalf, newBalance);

emit SwapAndLiquify(half, newBalance, otherHalf);
}
}

/// @dev Swap tokens for eth
function swapTokensForEth(uint256 tokenAmount) private {
// generate the baronFarm pair path of token -> weth
address[] memory path = new address[](2);
path[0] = address(this);
path[1] = baronFarmRouter.WETH();

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

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

/// @dev Add liquidity
function addLiquidity(uint256 tokenAmount, uint256 ethAmount) private {
// approve token transfer to cover all possible scenarios
_approve(address(this), address(baronFarmRouter), tokenAmount);

// add the liquidity
baronFarmRouter.addLiquidityETH{value: ethAmount}(
address(this),
tokenAmount,
0, // slippage is unavoidable
0, // slippage is unavoidable
operator(),
block.timestamp
);
}

// To receive BNB from baronFarmRouter when swapping
receive() external payable {}
```

```

/**
* @dev Update the transfer tax rate.
* Can only be called by the current operator.
*/
function updateTransferTaxRate(uint16 _transferTaxRate) public onlyOperator {
require(_transferTaxRate <= MAXIMUM_TRANSFER_TAX_RATE, "BARON::updateTransferTaxRate: Transfer tax rate must not exceed the maximum rate.");
emit TransferTaxRateUpdated(msg.sender, transferTaxRate, _transferTaxRate);
transferTaxRate = _transferTaxRate;
}

/**
* @dev Update the burn rate.
* Can only be called by the current operator.
*/
function updateBurnRate(uint16 _burnRate) public onlyOperator {
require(_burnRate <= 100, "BARON::updateBurnRate: Burn rate must not exceed the maximum rate.");
emit BurnRateUpdated(msg.sender, burnRate, _burnRate);
burnRate = _burnRate;
}


/**
* @dev Update the min amount to liquify.
* Can only be called by the current operator.
*/
function updateMinAmountToLiquify(uint256 _minAmount) public onlyOperator {
emit MinAmountToLiquifyUpdated(msg.sender, minAmountToLiquify, _minAmount);
minAmountToLiquify = _minAmount;
}

/**
* @dev Update the swapAndLiquifyEnabled.
* Can only be called by the current operator.
*/
function updateSwapAndLiquifyEnabled(bool _enabled) public onlyOperator {
emit SwapAndLiquifyEnabledUpdated(msg.sender, _enabled);
swapAndLiquifyEnabled = _enabled;
}

/**
* @dev Update the swap router.
* Can only be called by the current operator.
*/
function updateBaronFarmRouter(address _router) public onlyOperator {
baronFarmRouter = IUniswapV2Router02(_router);
baronFarmPair = IUniswapV2Factory(baronFarmRouter.factory()).getPair(address(this), baronFarmRouter.WETH());
require(baronFarmPair != address(0), "BARON::updateBaronFarmRouter: Invalid pair address.");
emit BaronFarmRouterUpdated(msg.sender, address(baronFarmRouter), baronFarmPair);
}

/**
* @dev Returns the address of the current operator.
*/
function operator() public view returns (address) {
return _operator;
}

/**
* @dev Transfers operator of the contract to a new account (`newOperator`).
* Can only be called by the current operator.
*/
function transferOperator(address newOperator) public onlyOperator {
```

```
require(newOperator != address(0), "BARON::transferOperator: new operator is the zero address");
emit OperatorTransferred(_operator, newOperator);
_operator = newOperator;
}

// Copied and modified from YAM code:
// https://github.com/yam-finance/yam-protocol/blob/master/contracts/token/YAMGovernanceStorage.sol
// https://github.com/yam-finance/yam-protocol/blob/master/contracts/token/YAMGovernance.sol
// Which is copied and modified from COMPOUND:
// https://github.com/compound-finance/compound-protocol/blob/master/contracts/Governance/Comp.sol

/// @dev A record of each accounts delegate
mapping (address => address) internal _delegates;

/// @notice A checkpoint for marking number of votes from a given block
struct Checkpoint {
uint32 fromBlock;
uint256 votes;
}

/// @notice A record of votes checkpoints for each account, by index
mapping (address => mapping (uint32 => Checkpoint)) public checkpoints;

/// @notice The number of checkpoints for each account
mapping (address => uint32) public numCheckpoints;

/// @notice The EIP-712 typehash for the contract's domain
bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domain(string name,uint256 chainId,address verifyingContract)");

/// @notice The EIP-712 typehash for the delegation struct used by the contract
bytes32 public constant DELEGATION_TYPEHASH = keccak256("Delegation(address delegatee,uint256 nonce,uint256 expiry)");

/// @notice A record of states for signing / validating signatures
mapping (address => uint) public nonces;

/// @notice An event thats emitted when an account changes its delegate
event DelegateChanged(address indexed delegator, address indexed fromDelegate, address indexed toDelegate);

/// @notice An event thats emitted when a delegate account's vote balance changes
event DelegateVotesChanged(address indexed delegate, uint previousBalance, uint newBalance);

/**
* @notice Delegate votes from `msg.sender` to `delegatee`
* @param delegator The address to get delegatee for
*/
function delegates(address delegator)
external
view
returns (address)
{
return _delegates[delegator];
}

/**
* @notice Delegate votes from `msg.sender` to `delegatee`
* @param delegatee The address to delegate votes to
*/
function delegate(address delegatee) external {
return _delegate(msg.sender, delegatee);
}

/**
* @notice Delegates votes from signatory to `delegatee`
* @param delegatee The address to delegate votes to
* @param nonce The contract state required to match the signature
* @param expiry The time at which to expire the signature
* @param v The recovery byte of the signature
* @param r Half of the ECDSA signature pair
* @param s Half of the ECDSA signature pair
*/
function delegateBySig(
address delegatee,
uint nonce,
uint expiry,
uint8 v,
bytes32 r,
bytes32 s
)
external
{
bytes32 domainSeparator = keccak256(
abi.encode(
DOMAIN_TYPEHASH,
keccak256(bytes(name())),
getChainId(),
address(this)
)
);

bytes32 structHash = keccak256(
abi.encode(
DELEGATION_TYPEHASH,
delegatee,
nonce,
expiry
)
);

bytes32 digest = keccak256(
abi.encodePacked(
"\x19\x01",
domainSeparator,
structHash
)
);

address signatory = ecrecover(digest, v, r, s);
require(signatory != address(0), "BARON::delegateBySig: invalid signature");
require(nonce == nonces[signatory]++, "BARON::delegateBySig: invalid nonce");
require(now <= expiry, "BARON::delegateBySig: signature expired");
return _delegate(signatory, delegatee);
}

/**
* @notice Gets the current votes balance for `account`
* @param account The address to get votes balance
* @return The number of current votes for `account`
*/
function getCurrentVotes(address account)
external
view
returns (uint256)
{
uint32 nCheckpoints = numCheckpoints[account];
return nCheckpoints > 0 ? checkpoints[account][nCheckpoints - 1].votes : 0;
}
```

```

/**
* @notice Determine the prior number of votes for an account as of a block number
* @dev Block number must be a finalized block or else this function will revert to prevent misinformation.
* @param account The address of the account to check
* @param blockNumber The block number to get the vote balance at
* @return The number of votes the account had as of the given block
*/
function getPriorVotes(address account, uint blockNumber)
external
view
returns (uint256)
{
require(blockNumber < block.number, "BARON::getPriorVotes: not yet determined");

uint32 nCheckpoints = numCheckpoints[account];
if (nCheckpoints == 0) {
return 0;
}

// First check most recent balance
if (checkpoints[account][nCheckpoints - 1].fromBlock <= blockNumber) {
return checkpoints[account][nCheckpoints - 1].votes;
}

// Next check implicit zero balance
if (checkpoints[account][0].fromBlock > blockNumber) {
return 0;
}

uint32 lower = 0;
uint32 upper = nCheckpoints - 1;
while (upper > lower) {
uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
Checkpoint memory cp = checkpoints[account][center];
if (cp.fromBlock == blockNumber) {
return cp.votes;
} else if (cp.fromBlock < blockNumber) {
lower = center;
} else {
upper = center - 1;
}
}
return checkpoints[account][lower].votes;
}

function _delegate(address delegator, address delegatee)
internal
{
address currentDelegate = _delegates[delegator];
uint256 delegatorBalance = balanceOf(delegator); // balance of underlying BARON (not scaled);
_delegates[delegator] = delegatee;

emit DelegateChanged(delegator, currentDelegate, delegatee);

_moveDelegates(currentDelegate, delegatee, delegatorBalance);
}

function _moveDelegates(address srcRep, address dstRep, uint256 amount) internal {
if (srcRep != dstRep && amount > 0) {
if (srcRep != address(0)) {
// decrease old representative
uint32 srcRepNum = numCheckpoints[srcRep];
uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1].votes : 0;
uint256 srcRepNew = srcRepOld.sub(amount);
_writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
}

if (dstRep != address(0)) {
// increase new representative
uint32 dstRepNum = numCheckpoints[dstRep];
uint256 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].votes : 0;
uint256 dstRepNew = dstRepOld.add(amount);
_writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
}
}
}

function _writeCheckpoint(
address delegatee,
uint32 nCheckpoints,
uint256 oldVotes,
uint256 newVotes
)
internal
{
uint32 blockNumber = safe32(block.number, "BARON::_writeCheckpoint: block number exceeds 32 bits");

if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber) {
checkpoints[delegatee][nCheckpoints - 1].votes = newVotes;
} else {
checkpoints[delegatee][nCheckpoints] = Checkpoint(blockNumber, newVotes);
numCheckpoints[delegatee] = nCheckpoints + 1;
}

emit DelegateVotesChanged(delegatee, oldVotes, newVotes);
}

function safe32(uint n, string memory errorMessage) internal pure returns (uint32) {
require(n < 2**32, errorMessage);
return uint32(n);
}

function getChainId() internal pure returns (uint) {
uint256 chainId;
assembly { chainId := chainid() }
return chainId;
}
}
```