PokeDX token smart contract internal audit

This contract is inspired by and in part a rewrite of reflect.finance and Safemoon, that aims to:

* - fix a majority of the issues reported in the Certik Safemoon audit (e.g., SSL-03)

* https://www.certik.org/projects/safemoon

* - Improve the reflection mechanism

* - exclude burn wallet from reflection

* - time-lock lp tokens from auto lp

* - Add flexibility to the contract for future governance compatibility

* - make it easier to maintain the code and develop it further

* - remove redundant code

* - optimize gas

PokeDX token is split into 4 contracts to make it easier to maintain the code and further develop it.

Contracts: 1) **Tokenomics** – to make it easier to change the tokenomics

2) **BaseRfiToken** - the logic of reflective mechanism

3) **Liquifier –** Logic for auto liquidity mechanism

4) **PokeDX** – Logic of PDX token

Function overview:

Function visibility meaning:

Internal - 💡 Those functions and state variables can only be accessed internally (i.e., from within the current contract or contracts deriving from it), without using this.

Private - 💡 is only visible for the contract they are defined in.

Tokenomics Contract:

Constructor - Run function _addFees(), which sets the fees only once during deployment.

```
constructor()
{ _addFees();
}
```

_addFee() - Add fee into array.

```
  function _addFee(
    uint256 position,
    FeeType name,
    uint256 value,
    address recipient
  ) private {
    fees.push(Fee(position, name, value, recipient, 0
)); sumOfFees += value;
  }
```

_addFees() - Used in constructor to add all fees only once during deployment of contract.
Note: The RFI recipient is ignored, but we need to give a valid address value.

```
function _addFees() private {
  /**
   * The value of fees is given in part per 1000 (based on the
 value of FEES_DIVISOR),
   * e.g. for 5% use 50, for 3.5% use 35, etc.
   */
  _addFee(1, FeeType.Rfi, 20, address(this));
  _addFee(2, FeeType.Liquidity, 20, address(this));
  _addFee(3, FeeType.Burn, 0, burnAddress);
}
```

_getFeesCount() - Returns number of fees

```
function _getFeesCount() internal view returns (uint256) {
  return fees.length;
}
```

_getFeeStruct() - Returns Fee by index. Used by _getFee().

```
function _getFeeStruct(uint256 index) private view returns
 (Fee storage) {
  require(index >= 0 && index < fees.length,
"FeesSettings._getFeeStruct: Fee index out of bounds");
  return fees[index];
}
```

_getFee() - Returns fee details(position, name, value, recipient, total)

```
function _getFee(uint256 index)
  internal
  view
  returns (
    uint256,
    FeeType,
    uint256,
    address,
    uint256
  )
{
  Fee memory fee = _getFeeStruct(index);
  return
 (fee.position, fee.name, fee.value, fee.recipient, fee.total);
}
```

_addFeeCollectedAmount() - assign value to fee variable and updating fee.total.

```
function _addFeeCollectedAmount(uint256 index, uint256 amount)
internal {
  Fee storage fee = _getFeeStruct(index);
  fee.total = fee.total + amount;
}
```

getCollectedFeeTotal() - Returns total amount of fee.

```
function getCollectedFeeTotal(uint256 index) internal view
returns (uint256) {
   Fee memory fee = _getFeeStruct(index);
   return fee.total;
 }
```

BaseRfiToken Contract:

Constructor - assign reflected supply to owner account
-exclude owner and this contract from fee
-exclude the owner and this contract from rewards

```
constructor() {
  _reflectedBalances[owner()] = _reflectedSupply;

  // exclude owner and this contract from fee
  _isExcludedFromFee[owner()] = true;
  _isExcludedFromFee[address(this)] = true;


// exclude the owner and this contract from rewards
  _exclude(owner());
  _exclude(address(this));

  emit Transfer(address(0), owner(), TOTAL_SUPPLY);
}
```

Functions required by IERC20Metadata

```
function name() external pure override returns (string memory)
{
  return NAME;
}

function symbol() external pure override returns (string memory
) {
  return SYMBOL;
}

function decimals() external pure override returns (uint8) {
  return DECIMALS;
}
```

Functions required by IERC20

```
function totalSupply() external pure override returns (
uint256) {
   return TOTAL_SUPPLY;
 }

 function balanceOf(address account) public view override
returns (uint256) {
   if (_isExcludedFromRewards[account]) return
 _balances[account];
   return tokenFromReflection(_reflectedBalances[account]);
 }

 function transfer(address recipient, uint256 amount)
external override returns (bool) {
   _transfer(_msgSender(), recipient, amount);
   return true;
 }

 function allowance(address owner, address spender) external
view override returns (uint256) {
   return _allowances[owner][spender];
 }

 function approve(address spender, uint256 amount) external
 override returns (bool) {
   _approve(_msgSender(), spender, amount);
   return true;
 }

 function transferFrom(
   address sender,
   address recipient,
   uint256 amount
 ) external override returns (bool) {
   _transfer(sender, recipient, amount);
   _approve(sender, _msgSender(), _allowances[sender][
_msgSender()] - amount);
   return true;
 }
```

burn() - This is a "soft" burn (total supply is not reduced). RFI holders get two benefits from burning tokens:

1) Tokens in the burn address increase the % of tokens held by holders not excluded from rewards (assuming the burn address is excluded)

2) Tokens in the burn address cannot be sold (which in turn drains the liquidity pool)

In RFI, holders already get % of each transaction, hence the value of their tokens increases (in a way). Therefore there is no need to do a "hard" burn (reducing the total supply). What matters (in RFI) is to make sure that a large amount of tokens cannot be sold = draining the liquidity pool = lowering the value of tokens holders own. For this purpose, transferring tokens to a (vanity) burn address is the most appropriate way to "burn". An extra check is placed into the `transfer` function to make sure the burn address cannot withdraw the tokens it has (although the chance of someone having/finding the private key is virtually zero).

```
function burn(uint256 amount) external {
  address sender = _msgSender();
  require(sender != address(0),
"BaseRfiToken: burn from the zero address");
  require(sender != address(burnAddress),
"BaseRfiToken: burn from the burn address");

  uint256 balance = balanceOf(sender);
  require(balance >= amount,
"BaseRfiToken: burn amount exceeds balance");

  uint256 reflectedAmount = amount * _getCurrentRate();

  // remove the amount from the sender's balance first
  _reflectedBalances[sender] = _reflectedBalances[sender] -
 reflectedAmount;
  if (_isExcludedFromRewards[sender]) _balances[sender] =
 _balances[sender] - amount;

  _burnTokens(sender, amount, reflectedAmount);
}
```

_burnTokens() - "Soft" burns the specified amount of tokens by sending them to the burn address

```
function _burnTokens(
  address sender,
  uint256 tBurn,
  uint256 rBurn
) internal {
  /**
   * @dev
 Do not reduce _totalSupply and/or _reflectedSupply. (soft) bur
ning by sending
   * tokens to the burn address (which should be excluded from
 rewards) is sufficient
   * in RFI
   */
  _reflectedBalances[burnAddress] =
 _reflectedBalances[burnAddress] + rBurn;
  if
 (_isExcludedFromRewards[burnAddress]) _balances[burnAddress] =
 _balances[burnAddress] + tBurn;

  /**
   * @dev
 Emit the event so that the burn address balance is updated (on
bscscan)
   */
  emit Transfer(sender, burnAddress, tBurn);
}
```

Functions required by ERC20

```
function _approve(
  address owner,
  address spender,
  uint256 amount
) internal {
  require(owner != address(0),
"BaseRfiToken: approve from the zero address");
  require(spender != address(0),
"BaseRfiToken: approve to the zero address");

  _allowances[owner][spender] = amount;
  emit Approval(owner, spender, amount);
}
```

```
function increaseAllowance(address spender, uint256
 addedValue) public virtual returns (bool) {
  _approve(_msgSender(), spender, _allowances[_msgSender
()][spender] + addedValue);
  return true;
}

function decreaseAllowance(address spender, uint256
 subtractedValue) public virtual returns (bool) {
  _approve(_msgSender(), spender, _allowances[_msgSender
()][spender] - subtractedValue);
  return true;
}
```

isExcludedFromReward() - Checks if the address is excluded from reflection reward.

```
function isExcludedFromReward(address account) external view
returns (bool) {
  return _isExcludedFromRewards[account];
}
```

reflectionFromToken() - Calculates and returns the reflected amount for the given amount with or without the transfer fees (deductTransferFee true/false)

```
function reflectionFromToken(uint256 tAmount, bool
 deductTransferFee) external view returns (uint256) {
  require(tAmount <= TOTAL_SUPPLY,
"Amount must be less than supply");
  if (!deductTransferFee) {
    (uint256 rAmount, , , , ) = _getValues(tAmount, 0);
    return rAmount;
  } else {
    (, uint256 rTransferAmount, , , ) = _getValues(tAmount,
_getSumOfFees());
    return rTransferAmount;
  }
}
```

tokenFromReflection() - Calculates and returns the amount of tokens corresponding to the given reflected amount.

```
function tokenFromReflection(uint256 rAmount) internal view
returns (uint256) {
  require(rAmount <= _reflectedSupply,
"Amount must be less than total reflections");
  uint256 currentRate = _getCurrentRate();
  return rAmount / currentRate;
}
```

excludeFromReward() - exclude address from reflection reward.

```
function excludeFromReward(address account) external onlyOwner
{
  require(!_isExcludedFromRewards[account],
"Account is not included");
  _exclude(account);
}
```

_exclude() - Is used by **excludeFromReward()** to exclude address from reflection reward.

```
function _exclude(address account) internal {
  if (_reflectedBalances[account] > 0) {
    _balances[account] = tokenFromReflection
(_reflectedBalances[account]);
  }
  _isExcludedFromRewards[account] = true;
  _excluded.push(account);
}
```

includeInReward() - Include address back in reflection rewards.

```
function includeInReward(address account) external onlyOwner {
  require(_isExcludedFromRewards[account],
"Account is not excluded");
  for (uint256 i = 0; i < _excluded.length; i++) {
    if (_excluded[i] == account) {
      _excluded[i] = _excluded[_excluded.length - 1];
      _balances[account] = 0;
      _isExcludedFromRewards[account] = false;
      _excluded.pop();
      break;
    }
  }
}
```

setExcludedFromFee() - Exclude address form fee.

```
function setExcludedFromFee(address account, bool value)
external onlyOwner {
  _isExcludedFromFee[account] = value;
}
```

isExcludedFromFee() - Checks if address is excluded from fee.

```
function isExcludedFromFee(address account) public view returns
 (bool) {
  return _isExcludedFromFee[account];
}
```

_isUnlimitedSender() and **isUnlimitedRecipient() -** The owner needs to be excluded from the whale mechanism(max tx limit) to receive initially mined tokens and be able to distribute them. Therefore this check is done by a transfer function.

```
function _isUnlimitedSender(address account) internal view returns (bool) {
  // the owner should be the only whitelisted sender
  return (account == owner());
}


function _isUnlimitedRecipient(address account) internal view returns (bool) {
  // the owner should be a white-listed recipient
  // and anyone should be able to burn as many tokens as
  // he/she wants

  return (account == owner() || account == burnAddress);
}
```

_transfer() - Take care of transferring pdx tokens and implement all necessary checks, hooks and functions.

```
function _transfer(
  address sender,
  address recipient,
  uint256 amount
) private {
  require(sender != address(0), "BaseRfiToken: transfer from the zero address");
  require(recipient != address(0), "BaseRfiToken: transfer to the zero address");
  require(sender != address(burnAddress),
"BaseRfiToken: transfer from the burn address");
  require(amount > 0, "Transfer amount must be greater than zero");

  // indicates whether or not feee should be deducted from the transfer
  bool takeFee = true;

  if (paused()) {
    takeFee = false;
  } else {
    /**
     * Check the amount is within the max allowed limit as long as a
     * unlimited sender/recepient is not involved in the transaction
     */
    if (amount > maxTransactionAmount && !_isUnlimitedSender(sender) && !
_isUnlimitedRecipient(recipient)) {
      revert("Transfer amount exceeds the maxTxAmount.");
    }
  }

  // if any account belongs to _isExcludedFromFee account then remove the fee
  if (_isExcludedFromFee[sender] || _isExcludedFromFee[recipient]) {
    takeFee = false;
  }

  _beforeTokenTransfer(sender, recipient, amount, takeFee);
  _transferTokens(sender, recipient, amount, takeFee);
}
```

_transferTokens() - is called by **_transfer** function. And make sure that fees/rewards get reflected correctly.

```
function _transferTokens(
  address sender,
  address recipient,
  uint256 amount,
  bool takeFee
) private {
  /**
   * We don't need to know anything about the individual fees here
   * (like Safemoon does with `_getValues`). All that is required
   * for the transfer is the sum of all fees to calculate the % of the total
   * transaction amount which should be transferred to the recipient.
   *
   * The `_takeFees` call will/should take care of the individual fees
   */
  uint256 sumOfFees = _getSumOfFees();
  if (!takeFee) {
    sumOfFees = 0;
  }

  (
    uint256 rAmount,
    uint256 rTransferAmount,
    uint256 tAmount,
    uint256 tTransferAmount,
    uint256 currentRate
  ) = _getValues(amount, sumOfFees);

  /**
   * Sender's and Recipient's reflected balances must be always updated regardless
of
   * whether they are excluded from rewards or not.
   */
  _reflectedBalances[sender] = _reflectedBalances[sender] - rAmount;
  _reflectedBalances[recipient] = _reflectedBalances[recipient] + rTransferAmount;

  /**
   * Update the true/nominal balances for excluded accounts
   */
  if (_isExcludedFromRewards[sender]) {
    _balances[sender] = _balances[sender] - tAmount;
  }
  if (_isExcludedFromRewards[recipient]) {
    _balances[recipient] = _balances[recipient] + tTransferAmount;
  }

  _takeFees(amount, currentRate, sumOfFees);
  emit Transfer(sender, recipient, tTransferAmount);
}
```

_takeFees() - is called by **transferTokens** function and applies set fees.

```
function _takeFees(
  uint256 amount,
  uint256 currentRate,
  uint256 sumOfFees
) private {
  if (sumOfFees > 0 && !paused()) {
    _takeTransactionFees(amount, currentRate);
  }
}
```

_getValues() - called by other functions to get right current values.

```
function _getValues(uint256 tAmount, uint256 feesSum)
  internal
  view
  returns (
    uint256,
    uint256,
    uint256,
    uint256,
    uint256
  )
{
  uint256 tTotalFees = (tAmount * feesSum) / FEES_DIVISOR;
  uint256 tTransferAmount = tAmount - tTotalFees;
  uint256 currentRate = _getCurrentRate();
  uint256 rAmount = tAmount * currentRate;
  uint256 rTotalFees = tTotalFees * currentRate;
  uint256 rTransferAmount = rAmount - rTotalFees;

  return (rAmount, rTransferAmount, tAmount, tTransferAmount, currentRate);
}
```

_getCurrentRate() - called by other functions to get the right current rate.

```
function _getCurrentRate() internal view returns (uint256) {
  (uint256 rSupply, uint256 tSupply) = _getCurrentSupply();
  return rSupply / tSupply;
}
```

**** _getCurrentSupply() -** called by other functions to get the right current supply.

Note: This function uses the for-loop for evaluating total supply, which can cause out_of_gas issues if the list of addresses is too long. This is also pointed out in the Techrate audit.

Reason for using the for loop: The excluded address list it's not meant to be long (no need to exclude many addresses from the reward). Therefore its serves the purpose as a "transparency mechanism/tool" where the owner of the contract cannot exclude many addresses e.g., for a bribe etc.

```
function _getCurrentSupply() internal view returns (uint256, uint256) {
  uint256 rSupply = _reflectedSupply;
  uint256 tSupply = TOTAL_SUPPLY;

  /**
   * The code below removes balances of addresses excluded from rewards from
   * rSupply and tSupply, which effectively increases the % of transaction fees
   * delivered to non-excluded holders
   */
  for (uint256 i = 0; i < _excluded.length; i++) {
    if (_reflectedBalances[_excluded[i]] > rSupply || _balances[_excluded[i]] >
 tSupply)
      return (_reflectedSupply, TOTAL_SUPPLY);
    rSupply = rSupply - _reflectedBalances[_excluded[i]];
    tSupply = tSupply - _balances[_excluded[i]];
  }
  if (tSupply == 0 || rSupply < _reflectedSupply / TOTAL_SUPPLY) return
 (_reflectedSupply, TOTAL_SUPPLY);
  return (rSupply, tSupply);
}
```

__redistribute() - Redistributes the specified amount among the current holders via the **reflect.finance** algorithm, i.e. by updating the _reflectedSupply (_rSupply) which ultimately adjusts the current rate used by **tokenFromReflection** function and, in turn, the value returns from **balanceOf** function. This is the bit of clever math which allows **RFI** to redistribute the fee without having to iterate through all holders.

```
function _redistribute(
  uint256 amount,
  uint256 currentRate,
  uint256 fee,
  uint256 index
) internal {
  uint256 tFee = (amount * fee) / FEES_DIVISOR;
  uint256 rFee = tFee * currentRate;

  _reflectedSupply = _reflectedSupply - rFee;
  _addFeeCollectedAmount(index, tFee);
}
```

Pause() and **unpause() -** These functions are intended to pause and unpause fees(reflection mechanism in a way) in special occasions like incompatible evm updates, pre-sale, or to resolve issues regarding interaction with other contracts like pancakeswap's router, etc.

Basically, this function (and features of the code) allows for this token to be 2in1.

- If a paused contract behaves like a regular ERC20/BEP20 token.
- If unpaused - the contract behaves like a reflection token.

```
 function pause() public onlyOwner {
    _pause();
  }

  function unpause() public onlyOwner {
    _unpause();
  }
}
```

Liquifier Contract:

Liquifier is an upgraded(bug fixed according to Certik audit) version of safemoon's auto liquidity AKA **autoLP feature.

****_setNumberOfTokensToSwapToLiquidity() -** Enable to set a number of tokens which should be swapped for liquidity. This helps fix the issue where the token price grows to the point where autoLP dumping the same number of tokens big with much higher value now and can badly impact the price.

```
function _setNumberOfTokensToSwapToLiquidity(uint256 tokenAmount) external
 onlyOwner {
  numberOfTokensToSwapToLiquidity = tokenAmount;
  emit NumberOfTokensToSwapToLiquidityChanged(tokenAmount);
}
```

showNumberOfTokensToSwapToLiquidity() - Returns number of tokens to swap to Liquidity...

```
function showNumberOfTokensToSwapToLiquidity() external view returns (uint256) {
  return numberOfTokensToSwapToLiquidity;
}
```

initializeLiquiditySwapper() - Set router address, max tx amount and number of tokens to swap to liquidity.

```
function initializeLiquiditySwapper(
  address env,
  uint256 maxTx,
  uint256 liquifyAmount
) internal {
  _setRouterAddress(env);

  maxTransactionAmount = maxTx;
  numberOfTokensToSwapToLiquidity = liquifyAmount;
}
```

liquify() - As described in the comment in the code:

1) Check if the contract has collected enough tokens to swap and liquify
2) Check if swap and liquify is enabled
3) Make sure not to get caught in a circular liquidity event
4) Finally, don't swap and liquify if the sender is the uniswap pair

```
function liquify(uint256 contractTokenBalance, address sender) internal {
  if
 (contractTokenBalance >= maxTransactionAmount) contractTokenBalance = maxTransacti
onAmount;

  bool
 isOverRequiredTokenBalance = (contractTokenBalance >= numberOfTokensToSwapToLiquid
ity);

  /**
   * - first check if the contract has collected enough tokens to swap and liquify
   * - then check swap and liquify is enabled
   * - then make sure not to get caught in a circular liquidity event
   * - finally, don't swap & liquify if the sender is the uniswap pair
   */
  if
 (isOverRequiredTokenBalance && swapAndLiquifyEnabled && !inSwapAndLiquify && (send
er != _pair))
{
    // TODO check if the `(sender != _pair)` is necessary because that basically
    // stops swap and liquify for all "buy" transactions
    _swapAndLiquify(contractTokenBalance);
  }
}
```

_setRouterAddress() - sets the router address, creates the router and factory pair to enable swapping and liquifying tokens.

```
function _setRouterAddress(address router) private {
  IPancakeV2Router _newPancakeRouter = IPancakeV2Router(router);
  _pair = IPancakeV2Factory(_newPancakeRouter.factory()).createPair(address(this
), _newPancakeRouter.WETH());
  _router = _newPancakeRouter;
  emit RouterSet(router);
}
```

_swapAndLiquify() - This function performs the swap and liquefaction.
For more info check comments in the code preview below.

```
function _swapAndLiquify(uint256 amount) private lockTheSwap {
  // split the contract balance into halves
  uint256 half = amount / 2;
  uint256 otherHalf = amount - 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);
// <- this breaks the ETH -> HATE swap when swap+liquify is triggered

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

  // add liquidity to uniswap
  _addLiquidity(otherHalf, newBalance);

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

_swapTokensForEth() - This function performs the swap of half tokens for ETH/BNB(depends on environment), so function **_addLiquidity** can use it to add liquidity.
For more info, check comments in the code preview below.

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

  _approveDelegate(address(this), address(_router), tokenAmount);

  // make the swap
  _router.swapExactTokensForETHSupportingFeeOnTransferTokens(
    tokenAmount,

// The minimum amount of output tokens that must be received for the transaction no
t to revert.
    // 0 = accept any amount (slippage is inevitable)
    0,
    path,
    address(this),
    block.timestamp
  );
}
```

setLPReceiver() and **showLPReceiver() -** These two functions are used to set LP token receiver from autoLP and show LP token receiver from autoLP.

```
function setLPReceiver(address receiver) external onlyOwner {
  LPReceiver = receiver;
  emit LPReceiverChanged(LPReceiver);
}

function showLPReceiver() external view returns (address) {
  return LPReceiver;
}
```

addLiquidity() - This function is responsible for adding liquidity (two pairs of token) created by lp fee rfi mechanism.
For more info, check comments in the code preview below.

```
function _addLiquidity(uint256 tokenAmount, uint256 ethAmount) private {
  // approve token transfer to cover all possible scenarios
  _approveDelegate(address(this), address(_router), tokenAmount);

  // add the liquidity
  (uint256 tokenAmountSent, uint256 ethAmountSent, uint256 liquidity) =
 _router.addLiquidityETH{value: ethAmount}(
    address(this),
    tokenAmount,

// Bounds the extent to which the WETH/token price can go up before the transactio
n reverts.

// Must be <= amountTokenDesired; 0 = accept any amount (slippage is inevitable)
    0,

// Bounds the extent to which the token/WETH price can go up before the transactio
n reverts.
    // 0 = accept any amount (slippage is inevitable)
    0,

// this is a centralized risk if the owner's account is ever compromised (see Cert
ik SSL-04)
    // owner(),
    LPReceiver,
    block.timestamp
  );

  // fix the forever locked BNBs as per the certik's audit
  /**
   * The swapAndLiquify function converts half of the contractTokenBalance SafeMoo
n tokens to BNB.
   * For every swapAndLiquify function call, a small amount of BNB remains in the
 contract.
   * This amount grows over time with the swapAndLiquify function being called thr
oughout the life
   * of the contract. The Safemoon contract does not contain a method to withdraw
 these funds,
   * and the BNB will be locked in the Safemoon contract forever.
   */
  withdrawableBalance = address(this).balance;
  emit LiquidityAdded(tokenAmountSent, ethAmountSent, liquidity);
}
```

setRouterAddress() - Enables the owner to call the **_setRouterAddress** function externally to change the router address in the future.

```
function setRouterAddress(address router) external onlyOwner {
  _setRouterAddress(router);
}
```

setSwapAndLiquifyEnabled() - Sends the swap and liquify flag to the provided value. If set to "false" tokens collected in the contract will NOT be converted into liquidity.

```
function setSwapAndLiquifyEnabled(bool enabled) external onlyOwner
{  swapAndLiquifyEnabled = enabled;
   emit SwapAndLiquifyEnabledUpdated(swapAndLiquifyEnabled);
 }
```

withdrawLockedBNB() - With this function, the owner can withdraw ETH(BNB) collected in the contract from "autoLP" or if someone (accidentally) sends ETH/BNB directly to the contract.
Note: This addresses the contract flaw pointed out in the Certik Audit of Safemoon (SSL-03): The swapAndLiquify function converts half of the contractTokenBalance SafeMoon tokens to BNB. For every swapAndLiquify function call, a small amount of BNB remains in the contract. This amount grows over time with the swapAndLiquify function being called throughout the life of the contract. The Safemoon contract does not contain a method to withdraw these funds, and the BNB will be locked in the Safemoon contract forever.

```
function withdrawLockedBNB(address payable recipient) external onlyOwner {
  require(recipient != address(0),
"Cannot withdraw the BNB balance to the zero address");
  require(withdrawableBalance > 0, "The BNB balance must be greater than 0");

  // prevent re-entrancy attacks
  uint256 amount = withdrawableBalance;
  withdrawableBalance = 0;
  recipient.transfer(amount);
}
```

PokeDX Contract:

Constructor: Run **initializeLiquiditySwapper** function which sets the correct values for autoLP. Then **_exclude** function exclude the pair and burn address from rewards.
Lastly, constructors pre-approve the initial liquidity supply (to save a bit of time).

```
constructor(address _env) {
  initializeLiquiditySwapper
(_env, maxTransactionAmount, numberOfTokensToSwapToLiquidity);

  // exclude the pair address from rewards - we don't want to redistribute
  // tx fees to these two; redistribution is only for holders, dah!
  _exclude(_pair);
  _exclude(burnAddress);
  _approve(owner(), address(_router), ~uint256(0));
}
```

_isV2Pair() - A delegate which should return true if the given address is the V2 Pair and false otherwise.

```
function _isV2Pair(address account) internal view override returns (bool) {
  return (account == _pair);
}
```

_getSumOfFees() - Returns total amount of fees

```
function _getSumOfFees() internal view override returns (uint256) {
  return sumOfFees;
}
```

_beforeTokenTransfer() - This is a hook which if paused, is false and eventually updates token balance in the contract and calls **liquify** function with updated token balance.

```
function _beforeTokenTransfer(
  address sender,
  address,
  uint256,
  bool
) internal override {
  if (!paused()) {
    uint256 contractTokenBalance = balanceOf(address(this));
    liquify(contractTokenBalance, sender);
  }
}
```

_takeTransactionFees() - This function first checks if the fees are enabled. Then loops through the Fee struct array. Each Fee struct represents each fee type with its values (1. position in the array (to be able to later change it with **increaseFee** and **decreaseFee** function), 2. name, 3. value, 4. recipient, 5. Total (counts the total amount of reflection accumulated by that fee)). This needs to happen to be able to deal with different kinds of fees, and this way, we can have different mechanisms for each fee if needed. E.g., burn and RFI have different mechanisms...

```
function _takeTransactionFees(uint256 amount, uint256 currentRate) internal
 override {
  if (paused()) {
    return;
  }
  uint256 feesCount = _getFeesCount();
  for (uint256 index = 0; index < feesCount; index++) {
    (, FeeType name, uint256 value, address recipient, ) = _getFee(index);
    // no need to check value < 0 as the value is uint (i.e. from 0 to 2^256-1)
    if (value == 0) continue;

    if (name == FeeType.Rfi) {
      _redistribute(amount, currentRate, value, index);
    } else if (name == FeeType.Burn) {
      _burn(amount, currentRate, value, index);
    } else {
      _takeFee(amount, currentRate, value, recipient, index);
    }
  }
}
```

_burn() - This function is used internally to perform burn (if burn fee is set).

```
function _burn(
  uint256 amount,
  uint256 currentRate,
  uint256 fee,
  uint256 index
) private {
  uint256 tBurn = (amount * fee) / FEES_DIVISOR;
  uint256 rBurn = tBurn * currentRate;

  _burnTokens(address(this), tBurn, rBurn);
  _addFeeCollectedAmount(index, tBurn);
}
```

_takeFee() - This function is called by **_takeTransactionFees** function and does it's part in the RFI mechanism.

```
function _takeFee(
  uint256 amount,
  uint256 currentRate,
  uint256 fee,
  address recipient,
  uint256 index
) private {
  uint256 tAmount = (amount * fee) / FEES_DIVISOR;
  uint256 rAmount = tAmount * currentRate;

  _reflectedBalances[recipient] = _reflectedBalances[recipient] + rAmount;
  if (_isExcludedFromRewards[recipient]) _balances[recipient] =
 _balances[recipient] + tAmount;

  _addFeeCollectedAmount(index, tAmount);
}
```

_approveDelegate() - This function is called by **_addLiquidity** and **swapTokensForEth** function to approve router spending on contract's behalf.

```
function _approveDelegate(
    address owner,
    address spender,
    uint256 amount
  ) internal override {
    _approve(owner, spender, amount);
  }
```

showFee() - This function returns the fee type, and its' values by calling the it's position in the array.
E.g.,

11. showFee

index (uint256)

0

Query

uint256, uint8, uint256, address, uint256

[**showFee** method Response]

» uint256 : 1 **Position**

» uint8 : 1 **Name(bscscan cant interpret srtuct(FeeType))**

» uint256 : 20 **20 = 2%**

» address : 0x43a0C5EB1763A211Aa3c05849A617f2eE0452767 ***Recipient**

» uint256 : 2116851893760506 **Total reflected until now**

*In the RFI fee recipient is ignored, but we need to give a valid address value

```
function showFee(uint256 index)
  external
  view
  returns (
    uint256,
    FeeType,
    uint256,
    address,
    uint256
  )
{
  return _getFee(index);
}
```

increaseFee() and **decreaseFee() -** These functions increase and decrease the chosen fee. Not possible to set fees to more than 8% combined. For transparency both functions emit events when the fee is changed.

```
  function increaseFee(uint256 index, uint256 addedValue) external onlyOwner {
    require((_getSumOfFees() + addedValue) <= 80, "Maximum 8% fee is allowed!");
    uint256 _sumOfFees = sumOfFees;
    uint256 updatedSumOfFees = _sumOfFees + addedValue;
    sumOfFees = updatedSumOfFees;
    fees[index].value += addedValue;
    emit FeeIncreased(index, addedValue, sumOfFees);
  }

  function decreaseFee(uint256 index, uint256 subtractedValue) external onlyOwner {
    require((_getSumOfFees() - subtractedValue) >= 0,
"Can't really go negative in there...");
    uint256 _sumOfFees = sumOfFees;
    uint256 updatedSumOfFees = _sumOfFees - subtractedValue;
    sumOfFees = updatedSumOfFees;
    fees[index].value -= subtractedValue;
    emit FeeDecreased(index, subtractedValue, sumOfFees);
  }
}
```