Smart Contract Security Audit Report

The SlowMist Security Team received the DollarProtocol team's application for smart contract security audit of the Shares and Dollars on Aug. 25, 2020. The following are the details and results of this smart contract security audit:

Project name :

DollarProtocol

The Contract address: (The scope of the audit is limited to the following two files)

Dollars.sol:

https://github.com/Dollar-Protocol/Core-Contracts/blob/master/dollars.sol

commit: 9241e4f2a4535a23149d455b42fd08b7d26335c5

Shares.sol:

https://github.com/Dollar-Protocol/Core-Contracts/blob/master/seigniorageShares.sol

commit: 2f41f3f1222a90aa3dac93eb5c5880c34b361f49

The audit items and results :

(Other unknown security vulnerabilities are not included in the audit responsibility scope)

No.	Audit Items	Audit Subclass	Audit Subclass Result
1	Overflow Audit	-	Passed
2	Race Conditions Audit	-	Passed
3	Authority Control Audit	Permission vulnerability audit	Passed
		Excessive auditing authority	Passed
4	Safety Design Audit	Zeppelin module safe use	Passed
		Compiler version security	Passed
		Hard-coded address security	Passed
		Fallback function safe use	Passed
		Show coding security	Passed
		Function return value security	Passed
		Call function security	Passed
5	Denial of Service Audit	-	Passed

6	Gas Optimization Audit	-	Passed
7	Design Logic Audit	-	Passed
8	“False Deposit” vulnerability Audit	-	Passed
9	Malicious Event Log Audit	-	Passed
10	Scoping and Declarations Audit	-	Passed
11	Replay Attack Audit	ECDSA's Signature Replay Audit	Passed
12	Uninitialized Storage Pointers Audit	-	Passed
13	Arithmetic Accuracy Deviation Audit	-	Passed

Audit Result : **Passed**

Audit Number : 0X002008310002

Audit Date : Aug. 31, 2020

Audit Team : SlowMist Security Team

Summary: This is DollarProtocol project, and the scope of audit is limited to the Dollars contract and SeigniorageShares contract in the DollarProtocol project. The protocol wishes to be a two token elastic monetary supply. The share tokens pay out dividends in dollars when there is a positive rebase. Negative rebased open up an auction to burn dollars for shares on chain. SafeMath security module is used, which is a commendable approach. The contract does not have the Overflow and the Race Conditions issue.

In the Shares contract, the minter role can mint unlimited tokens through the mintShares function, and the owner can change minter role through the changeMinter function.

In the Dollars contract the owner can set the size of `_maxDiscount` at will through the setMaxDiscount function. And the owner can set the contract monetary policy address through the setMonetaryPolicy function, but the project monetary policy logic is not within the scope of this audit.

The source code:

Shares.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity >=0.4.24;

import "./interface/IDollars.sol";
import "openzeppelin-eth/contracts/math/SafeMath.sol";
import "openzeppelin-eth/contracts/ownership/Ownable.sol";
import "openzeppelin-eth/contracts/token/ERC20/ERC20Detailed.sol";

import "./lib/SafeMathInt.sol";

/*
 *  SeigniorageShares ERC20
 */


contract SeigniorageShares is ERC20Detailed, Ownable {
    address private _minter;

    modifier onlyMinter() {
        require(msg.sender == _minter, "DOES_NOT_HAVE_MINTER_ROLE");
        _;
    }

    using SafeMath for uint256;
    using SafeMathInt for int256;

    uint256 private constant DECIMALS = 9;
    uint256 private constant MAX_UINT256 = ~uint256(0);
    uint256 private constant INITIAL_SHARE_SUPPLY = 21 * 10**6 * 10**DECIMALS;

    uint256 private constant MAX_SUPPLY = ~uint128(0);   // (2^128) - 1

    uint256 private _totalSupply;
```

```
struct Account {
    uint256 balance;
    uint256 lastDividendPoints;
}

bool private _initializedDollar;
// eslint-ignore
IDollars Dollars;

mapping(address=>Account) private _shareBalances;
mapping (address => mapping (address => uint256)) private _allowedShares;

function setDividendPoints(address who, uint256 amount) external onlyMinter returns (bool) {
    _shareBalances[who].lastDividendPoints = amount;
    return true;
}

function changeMinter(address minter) external onlyOwner {
    _minter = minter;
}
```

//SlowMist// Minter can mint unlimited tokens through the mintShares function

```
function mintShares(address who, uint256 amount) external onlyMinter returns (bool) {
    _shareBalances[who].balance = _shareBalances[who].balance.add(amount);
    _totalSupply = _totalSupply.add(amount);
    return true;
}

function externalTotalSupply()
    external
    view
    returns (uint256)
{
    return _totalSupply;
}

function externalRawBalanceOf(address who)
    external
    view
    returns (uint256)
{
        return _shareBalances[who].balance;
    }

    function lastDividendPoints(address who)
        external
        view
        returns (uint256)
    {
        return _shareBalances[who].lastDividendPoints;
    }
```

//SlowMist// The initialize function can be called by any user for initialization

```
    function initialize(address owner_)
        public
        initializer
    {
        ERC20Detailed.initialize("Seigniorage Shares", "SHARE", uint8(DECIMALS));
        Ownable.initialize(owner_);

        _initializedDollar = false;

        _totalSupply = INITIAL_SHARE_SUPPLY;
        _shareBalances[owner_].balance = _totalSupply;

        emit Transfer(address(0x0), owner_, _totalSupply);
    }

    // instantiate dollar
    function initializeDollar(address dollarAddress) public onlyOwner {
        require(_initializedDollar == false, "ALREADY_INITIALIZED");
        Dollars = IDollars(dollarAddress);
        _initializedDollar = true;
    }

     /**
      * @return The total number of Dollars.
      */
    function totalSupply()
        public
        view
        returns (uint256)
    {
        return _totalSupply;
    }

    function balanceOf(address who)
        public
        view
        returns (uint256)
    {
        return _shareBalances[who].balance;
    }

    /**
     * @dev Transfer tokens to a specified address.
     * @param to The address to transfer to.
     * @param value The amount to be transferred.
     * @return True on success, false otherwise.
     */
    function transfer(address to, uint256 value)
        public
        updateAccount(msg.sender)
        updateAccount(to)
        validRecipient(to)
        returns (bool)
    {
        _shareBalances[msg.sender].balance = _shareBalances[msg.sender].balance.sub(value);
        _shareBalances[to].balance = _shareBalances[to].balance.add(value);
        emit Transfer(msg.sender, to, value);
        return true; //SlowMist// The return value conforms to the EIP20 specification
    }

    /**
     * @dev Function to check the amount of tokens that an owner has allowed to a spender.
     * @param owner_ The address which owns the funds.
     * @param spender The address which will spend the funds.
     * @return The number of tokens still available for the spender.
     */
    function allowance(address owner_, address spender)
        public
        view
        returns (uint256)
    {
        return _allowedShares[owner_][spender];
    }

    /**
     * @dev Transfer tokens from one address to another.
     * @param from The address you want to send tokens from.
     * @param to The address you want to transfer to.
     * @param value The amount of tokens to be transferred.
     */
    function transferFrom(address from, address to, uint256 value)
        public
        validRecipient(to)
        updateAccount(from)
        updateAccount(msg.sender)
        updateAccount(to)
        returns (bool)
    {
        _allowedShares[from][msg.sender] = _allowedShares[from][msg.sender].sub(value);

        _shareBalances[from].balance = _shareBalances[from].balance.sub(value);
        _shareBalances[to].balance = _shareBalances[to].balance.add(value);
        emit Transfer(from, to, value);


        return true; //SlowMist// The return value conforms to the EIP20 specification

    }

    /**
     * @dev Approve the passed address to spend the specified amount of tokens on behalf of
     * msg.sender. This method is included for ERC20 compatibility.
     * increaseAllowance and decreaseAllowance should be used instead.
     * Changing an allowance with this method brings the risk that someone may transfer both
     * the old and the new allowance - if they are both greater than zero - if a transfer
     * transaction is mined before the later approve() call is mined.
     *
     * @param spender The address which will spend the funds.
     * @param value The amount of tokens to be spent.
     */
    function approve(address spender, uint256 value)
        public
        validRecipient(spender)
        updateAccount(msg.sender)
        updateAccount(spender)
        returns (bool)
    {
        _allowedShares[msg.sender][spender] = value;
        emit Approval(msg.sender, spender, value);
        return true; //SlowMist// The return value conforms to the EIP20 specification
    }

    modifier validRecipient(address to) {
        require(to != address(0x0));
        require(to != address(this));
        _;
    }

    /**
     * @dev Increase the amount of tokens that an owner has allowed to a spender.
     * This method should be used instead of approve() to avoid the double approval vulnerability
     * described above.
     * @param spender The address which will spend the funds.
     * @param addedValue The amount of tokens to increase the allowance by.
     */
    function increaseAllowance(address spender, uint256 addedValue)
        public
        updateAccount(msg.sender)
        updateAccount(spender)
        returns (bool)
    {
        _allowedShares[msg.sender][spender] =
            _allowedShares[msg.sender][spender].add(addedValue);
        emit Approval(msg.sender, spender, _allowedShares[msg.sender][spender]);
        return true;
    }

    /**
     * @dev Decrease the amount of tokens that an owner has allowed to a spender.
     *
     * @param spender The address which will spend the funds.
     * @param subtractedValue The amount of tokens to decrease the allowance by.
     */
    function decreaseAllowance(address spender, uint256 subtractedValue)
        public
        updateAccount(msg.sender)
        updateAccount(spender)
        returns (bool)
    {
        uint256 oldValue = _allowedShares[msg.sender][spender];
        if (subtractedValue >= oldValue) {
            _allowedShares[msg.sender][spender] = 0;
        } else {
            _allowedShares[msg.sender][spender] = oldValue.sub(subtractedValue);
        }
        emit Approval(msg.sender, spender, _allowedShares[msg.sender][spender]);
        return true;
    }

    // attach this to any action to auto claim
    modifier updateAccount(address account) {
        require(_initializedDollar == true, "DOLLAR_NEEDS_INITIALIZATION");

        Dollars.claimDividends(account);
        _;
    }
}
```

Dollars.sol:

```
//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity >=0.4.24;

import "./lib/UInt256Lib.sol";
import "./lib/SafeMathInt.sol";
import "./interface/ISeignorageShares.sol";
import "openzeppelin-eth/contracts/math/SafeMath.sol";
import "openzeppelin-eth/contracts/token/ERC20/ERC20Detailed.sol";
import "openzeppelin-eth/contracts/ownership/Ownable.sol";
import "openzeppelin-eth/contracts/utils/ReentrancyGuard.sol";

interface IDollarPolicy {
    function getUsdSharePrice() external view returns (uint256 price);
}

/*
```

```
 *  Dollar ERC20
 */

contract Dollars is ERC20Detailed, Ownable, ReentrancyGuard {
    using SafeMath for uint256;
    using SafeMathInt for int256;

    event LogRebase(uint256 indexed epoch, uint256 totalSupply);
    event LogContraction(uint256 indexed epoch, uint256 dollarsToBurn);
    event LogRebasePaused(bool paused);
    event LogBurn(address indexed from, uint256 value);
    event LogClaim(address indexed from, uint256 value);
    event LogMonetaryPolicyUpdated(address monetaryPolicy);

    // Used for authentication
    address public monetaryPolicy;
    address public sharesAddress;

    modifier onlyMonetaryPolicy() {
        require(msg.sender == monetaryPolicy);
        _;
    }

    // Precautionary emergency controls.
    bool public rebasePaused;

    modifier whenRebaseNotPaused() {
        require(!rebasePaused);
        _;
    }

    // coins needing to be burned (9 decimals)
    uint256 private _remainingDollarsToBeBurned;

    modifier validRecipient(address to) {
        require(to != address(0x0));
        require(to != address(this));
        _;
    }

    uint256 private constant DECIMALS = 9;
    uint256 private constant MAX_UINT256 = ~uint256(0);
```

```
uint256 private constant INITIAL_DOLLAR_SUPPLY = 1 * 10**6 * 10**DECIMALS;
uint256 private _maxDiscount;

modifier validDiscount(uint256 discount) {
    require(discount >= 0, 'POSITIVE_DISCOUNT');                // 0%
    require(discount <= _maxDiscount, 'DISCOUNT_TOO_HIGH');
    _;
}

uint256 private constant MAX_SUPPLY = ~uint128(0);   // (2^128) - 1

uint256 private _totalSupply;

uint256 private constant POINT_MULTIPLIER = 10 ** 9;

uint256 private _totalDividendPoints;
uint256 private _unclaimedDividends;

ISeigniorageShares Shares;

mapping(address => uint256) private _dollarBalances;

// This is denominated in Dollars, because the cents-dollars conversion might change before
// it's fully paid.
mapping (address => mapping (address => uint256)) private _allowedDollars;

IDollarPolicy DollarPolicy;
uint256 public burningDiscount; // percentage (10 ** 9 Decimals)
uint256 public defaultDiscount; // discount on first negative rebase
uint256 public defaultDailyBonusDiscount; // how much the discount increases per day for consecutive contractions

uint256 public minimumBonusThreshold;

bool reEntrancyMutex; // unsused

/**
 * @param monetaryPolicy_ The address of the monetary policy contract to use for authentication.
 */

//SlowMist// The project monetary policy logic is not within the scope of this audit

function setMonetaryPolicy(address monetaryPolicy_)
    external
```

```
        onlyOwner
    {
        monetaryPolicy = monetaryPolicy_;
        DollarPolicy = IDollarPolicy(monetaryPolicy_);
        emit LogMonetaryPolicyUpdated(monetaryPolicy_);
    }

    function setBurningDiscount(uint256 discount)
        external
        onlyOwner
        validDiscount(discount)
    {
        burningDiscount = discount;
    }

    // amount in is 10 ** 9 decimals
    function burn(uint256 amount)
        external
        updateAccount(msg.sender)
        nonReentrant
    {
        require(amount > 0, 'AMOUNT_MUST_BE_POSITIVE');
        require(burningDiscount >= 0, 'DISCOUNT_NOT_VALID');
        require(_remainingDollarsToBeBurned > 0, 'COIN_BURN_MUST_BE_GREATER_THAN_ZERO');
        require(amount <= _dollarBalances[msg.sender], 'INSUFFICIENT_DOLLAR_BALANCE');
        require(amount <= _remainingDollarsToBeBurned,
'AMOUNT_MUST_BE_LESS_THAN_OR_EQUAL_TO_REMAINING_COINS');

        _burn(msg.sender, amount);
    }

    function setDefaultDiscount(uint256 discount)
        external
        onlyOwner
        validDiscount(discount)
    {
        defaultDiscount = discount;
    }
    //SlowMist// The owner can set the size of '_maxDiscount' at will through the setMaxDiscount
function
```

```
    function setMaxDiscount(uint256 discount)
        external
        onlyOwner
    {
        _maxDiscount = discount;
    }

    function setDefaultDailyBonusDiscount(uint256 discount)
        external
        onlyOwner
        validDiscount(discount)
    {
        defaultDailyBonusDiscount = discount;
    }

    /**
     * @dev Pauses or unpauses the execution of rebase operations.
     * @param paused Pauses rebase operations if this is true.
     */
    //SlowMist// Suspending rebase operations upon major abnormalities is a recommended
approach
    function setRebasePaused(bool paused)
        external
        onlyOwner
    {
        rebasePaused = paused;
        emit LogRebasePaused(paused);
    }

    // action of claiming funds
    function claimDividends(address account) external updateAccount(account) returns (uint256) {
        uint256 owing = dividendsOwing(account);
        return owing;
    }

    function setMinimumBonusThreshold(uint256 minimum)
        external
        onlyOwner
    {
        require(minimum >= 0, 'POSITIVE_MINIMUM');
        require(minimum < _totalSupply, 'MINIMUM_TOO_HIGH');
        minimumBonusThreshold = minimum;
    }

    /**
     * @dev Notifies Dollars contract about a new rebase cycle.
     * @param supplyDelta The number of new dollar tokens to add into circulation via expansion.
     * @return The total number of dollars after the supply adjustment.
     */
    function rebase(uint256 epoch, int256 supplyDelta)
        external
        onlyMonetaryPolicy
        whenRebaseNotPaused
        returns (uint256)
    {
        if (supplyDelta == 0) {
            emit LogRebase(epoch, _totalSupply);
            return _totalSupply;
        }

        if (supplyDelta < 0) {
            uint256 dollarsToBurn = uint256(supplyDelta.abs());
            if (dollarsToBurn > _totalSupply.div(10)) { // maximum contraction is 10% of the total USD Supply
                dollarsToBurn = _totalSupply.div(10);
            }

            if (dollarsToBurn.add(_remainingDollarsToBeBurned) > _totalSupply) {
                dollarsToBurn = _totalSupply.sub(_remainingDollarsToBeBurned);
            }

            if (_remainingDollarsToBeBurned > minimumBonusThreshold) {
                burningDiscount = burningDiscount.add(defaultDailyBonusDiscount) > _maxDiscount ?
                    _maxDiscount : burningDiscount.add(defaultDailyBonusDiscount);
            } else {
                burningDiscount = defaultDiscount; // default 1%
            }

            _remainingDollarsToBeBurned = _remainingDollarsToBeBurned.add(dollarsToBurn);
            emit LogContraction(epoch, dollarsToBurn);
        } else {
            disburse(uint256(supplyDelta));
            emit LogRebase(epoch, _totalSupply);
            if (_totalSupply > MAX_SUPPLY) {
                _totalSupply = MAX_SUPPLY;
            }
        }

        return _totalSupply;
    }

    //SlowMist// The initialize function can be called by any user for initialization

    function initialize(address owner_, address seigniorageAddress)
        public
        initializer
    {
        ERC20Detailed.initialize("Dollars", "USD", uint8(DECIMALS));
        Ownable.initialize(owner_);

        rebasePaused = false;
        _totalSupply = INITIAL_DOLLAR_SUPPLY;

        sharesAddress = seigniorageAddress;
        Shares = ISeigniorageShares(seigniorageAddress);

        _dollarBalances[owner_] = _totalSupply;
        _maxDiscount = 50 * 10 ** 9; // 50%
        defaultDiscount = 1 * 10 ** 9;                // 1%
        burningDiscount = defaultDiscount;
        defaultDailyBonusDiscount = 1 * 10 ** 9;      // 1%
        minimumBonusThreshold = 100 * 10 ** 9;      // 100 dollars is the minimum threshold. Anything above warrants
increased discount

        emit Transfer(address(0x0), owner_, _totalSupply);
    }

    function dividendsOwing(address account) public view returns (uint256) {
        if (_totalDividendPoints > Shares.lastDividendPoints(account)) {
            uint256 newDividendPoints = _totalDividendPoints.sub(Shares.lastDividendPoints(account));
            uint256 sharesBalance = Shares.externalRawBalanceOf(account);
            return sharesBalance.mul(newDividendPoints).div(POINT_MULTIPLIER);
        } else {
            return 0;
        }
    }

    // auto claim modifier
    // if user is owned, we pay out immediatly
    // if user is not owned, we prevent them from claiming until the next rebase
    modifier updateAccount(address account) {
        uint256 owing = dividendsOwing(account);

        if (owing > 0) {
            _unclaimedDividends = _unclaimedDividends.sub(owing);
            _dollarBalances[account] += owing;
        }

        Shares.setDividendPoints(account, _totalDividendPoints);

        emit LogClaim(account, owing);
        _;
    }

    /**
     * @return The total number of dollars.
     */
    function totalSupply()
        public
        view
        returns (uint256)
    {
        return _totalSupply;
    }

    /**
     * @param who The address to query.
     * @return The balance of the specified address.
     */
    function balanceOf(address who)
        public
        view
        returns (uint256)
    {
        return _dollarBalances[who].add(dividendsOwing(who));
    }
```

```
function getRemainingDollarsToBeBurned()
    public
    view
    returns (uint256)
{
    return _remainingDollarsToBeBurned;
}

/**
 * @dev Transfer tokens to a specified address.
 * @param to The address to transfer to.
 * @param value The amount to be transferred.
 * @return True on success, false otherwise.
 */
function transfer(address to, uint256 value)
    public
    validRecipient(to)
    updateAccount(msg.sender)
    updateAccount(to)
    returns (bool)
{
    _dollarBalances[msg.sender] = _dollarBalances[msg.sender].sub(value);
    _dollarBalances[to] = _dollarBalances[to].add(value);
    emit Transfer(msg.sender, to, value);

    return true; //SlowMist// The return value conforms to the EIP20 specification

}

/**
 * @dev Function to check the amount of tokens that an owner has allowed to a spender.
 * @param owner_ The address which owns the funds.
 * @param spender The address which will spend the funds.
 * @return The number of tokens still available for the spender.
 */
function allowance(address owner_, address spender)
    public
    view
    returns (uint256)
{
    return _allowedDollars[owner_][spender];
}
```

```
/**
 * @dev Transfer tokens from one address to another.
 * @param from The address you want to send tokens from.
 * @param to The address you want to transfer to.
 * @param value The amount of tokens to be transferred.
 */
function transferFrom(address from, address to, uint256 value)
    public
    validRecipient(to)
    updateAccount(from)
    updateAccount(msg.sender)
    updateAccount(to)
    returns (bool)
{
    _allowedDollars[from][msg.sender] = _allowedDollars[from][msg.sender].sub(value);

    _dollarBalances[from] = _dollarBalances[from].sub(value);
    _dollarBalances[to] = _dollarBalances[to].add(value);
    emit Transfer(from, to, value);

    return true; //SlowMist// The return value conforms to the EIP20 specification
}

/**
 * @dev Approve the passed address to spend the specified amount of tokens on behalf of
 * msg.sender. This method is included for ERC20 compatibility.
 * increaseAllowance and decreaseAllowance should be used instead.
 * Changing an allowance with this method brings the risk that someone may transfer both
 * the old and the new allowance - if they are both greater than zero - if a transfer
 * transaction is mined before the later approve() call is mined.
 *
 * @param spender The address which will spend the funds.
 * @param value The amount of tokens to be spent.
 */
function approve(address spender, uint256 value)
    public
    validRecipient(spender)
    updateAccount(msg.sender)
    updateAccount(spender)
    returns (bool)
```

```
{
    _allowedDollars[msg.sender][spender] = value;
    emit Approval(msg.sender, spender, value);
    return true; //SlowMist// The return value conforms to the EIP20 specification
}

/**
 * @dev Increase the amount of tokens that an owner has allowed to a spender.
 * This method should be used instead of approve() to avoid the double approval vulnerability
 * described above.
 * @param spender The address which will spend the funds.
 * @param addedValue The amount of tokens to increase the allowance by.
 */
function increaseAllowance(address spender, uint256 addedValue)
    public
    updateAccount(msg.sender)
    updateAccount(spender)
    returns (bool)
{
    _allowedDollars[msg.sender][spender] =
        _allowedDollars[msg.sender][spender].add(addedValue);
    emit Approval(msg.sender, spender, _allowedDollars[msg.sender][spender]);
    return true;
}

/**
 * @dev Decrease the amount of tokens that an owner has allowed to a spender.
 *
 * @param spender The address which will spend the funds.
 * @param subtractedValue The amount of tokens to decrease the allowance by.
 */
function decreaseAllowance(address spender, uint256 subtractedValue)
    public
    updateAccount(spender)
    updateAccount(msg.sender)
    returns (bool)
{
    uint256 oldValue = _allowedDollars[msg.sender][spender];
    if (subtractedValue >= oldValue) {
        _allowedDollars[msg.sender][spender] = 0;
    } else {
            _allowedDollars[msg.sender][spender] = oldValue.sub(subtractedValue);
        }
        emit Approval(msg.sender, spender, _allowedDollars[msg.sender][spender]);
        return true;
    }

    function consultBurn(uint256 amount)
        public
        returns (uint256)
    {
        require(amount > 0, 'AMOUNT_MUST_BE_POSITIVE');
        require(burningDiscount >= 0, 'DISCOUNT_NOT_VALID');
        require(_remainingDollarsToBeBurned > 0, 'COIN_BURN_MUST_BE_GREATER_THAN_ZERO');
        require(amount <= _dollarBalances[msg.sender].add(dividendsOwing(msg.sender)),
'INSUFFICIENT_DOLLAR_BALANCE');
        require(amount <= _remainingDollarsToBeBurned,
'AMOUNT_MUST_BE_LESS_THAN_OR_EQUAL_TO_REMAINING_COINS');

        uint256 usdPerShare = DollarPolicy.getUsdSharePrice(); // 1 share = x dollars
        usdPerShare = usdPerShare.sub(usdPerShare.mul(burningDiscount).div(100 * 10 ** 9)); // 10^9
        uint256 sharesToMint = amount.mul(10 ** 9).div(usdPerShare); // 10^9

        return sharesToMint;
    }

    function unclaimedDividends()
        public
        view
        returns (uint256)
    {
        return _unclaimedDividends;
    }

    function totalDividendPoints()
        public
        view
        returns (uint256)
    {
        return _totalDividendPoints;
    }

    function disburse(uint256 amount) internal returns (bool) {
        _totalDividendPoints =
_totalDividendPoints.add(amount.mul(POINT_MULTIPLIER).div(Shares.externalTotalSupply()));
        _totalSupply = _totalSupply.add(amount);
        _unclaimedDividends = _unclaimedDividends.add(amount);
        return true;
    }

    function _burn(address account, uint256 amount)
        internal
    {
        _totalSupply = _totalSupply.sub(amount);
        _dollarBalances[account] = _dollarBalances[account].sub(amount);

        uint256 usdPerShare = DollarPolicy.getUsdSharePrice(); // 1 share = x dollars
        usdPerShare = usdPerShare.sub(usdPerShare.mul(burningDiscount).div(100 * 10 ** 9)); // 10^9
        uint256 sharesToMint = amount.mul(10 ** 9).div(usdPerShare); // 10^9
        _remainingDollarsToBeBurned = _remainingDollarsToBeBurned.sub(amount);

        Shares.mintShares(account, sharesToMint);

        emit Transfer(account, address(0), amount);
        emit LogBurn(account, amount);
    }
}
```