REDSF

Smart Contract Audit

Website: https://redsf.ai

November 21, 2024

REDSF

Smart Contract Audit

Preface

This audit is of the REDSF smart contract that was provided for detailed analysis on November 18, 2024. The entire solidity smart contract code is listed at the end of the report.

SMART CONTRACT MAINET PROVIDED FOR REVIEW:
https://polygonscan.com/address/0x203e278C51a519f06494e667eef409dea7ae5b6d#code

DISCLAIMER:

Disclaimer:

This audit report is based on a professional review of the provided smart contract deployed on the Polygon Mainnet. It is important to note that this assessment represents our expert opinion and analysis of the code at the time of the evaluation. The findings and recommendations presented herein are not intended to serve as warranties, guarantees, or assurances of the contract's performance, security, or functionality on any live network, including the Ethereum / Polygon mainnet.

We expressly disclaim any responsibility for errors, omissions, or inaccuracies in this report, as the assessment is conducted on a non-exhaustive basis and may not cover all possible scenarios or future developments. The audit is conducted in accordance with industry best practices and standards at the time of evaluation.

Furthermore, we are unable to confirm the deployment of this specific contract on the Ethereum / Polygon mainnet. This report is solely based on the provided code and does not verify the actual deployment status on any live blockchain. It is the responsibility of the contract deployer to ensure the accurate deployment of the contract and adhere to security best practices when deploying to production environments.

Users, developers, and stakeholders are advised to perform additional due diligence and testing before deploying or interacting with the contract on any live network. This report should be considered as a tool for risk assessment rather than a guarantee of the contract's security or performance. In the dynamic and rapidly evolving field of blockchain technology, risks and vulnerabilities may emerge over time, and it is crucial to stay vigilant and up-to-date on security best practices.

By relying on this audit report, the reader acknowledges and accepts that the audit is based on the provided information and that no warranties, guarantees, or assurances are expressed or implied. While this is done to our best effort and knowledge, please notice that no tool can guarantee complete accuracy or comprehensiveness in detecting all possible vulnerabilities.

Smart Contract Audit Report: REDSF

Overview:

REDSF is an ERC-20 compliant token contract with extended functionalities including transaction taxes for buys, sells, and transfers, controlled by an owner with two-step ownership transfer capabilities, and equipped with ReentrancyGuard for enhanced security against reentrant attacks. The contract allows dynamic adjustment of tax rates and recipients, promoting flexibility in tokenomics management.

Key Features:

- **Tokenomics**:
  - Initial supply: 1 billion tokens with a decimal of 18.
  - Transaction taxes for buying, selling, and other transfers, which can be adjusted by the owner.

- **Ownership Management**:
  - Implements `Ownable2Step`, allowing safer ownership transitions.

- **Tax Mechanisms**:
  - Adjustable taxes for buy, sell, and transfer events.
  - A tax receiver address receives the collected taxes, which can be updated by the owner.

- **Security Measures**:
  - Utilizes `ReentrancyGuard` to protect against potential reentrant attacks.

- **Flexibility and Control**:
  - Owner can exclude or include addresses from tax.
  - Owner can manage known decentralized exchange (DEX) addresses to differentiate between buy and sell transactions.

Security Assessment:

1. Code Correctness and Best Practices:

- The contract follows the ERC-20 standard.
- Functions are clearly separated based on functionality, and visibility is correctly set to avoid unintended exposure.
- The use of custom errors enhances gas efficiency and readability of error handling.

2. Owner Controls:

- The contract gives extensive control to the owner, including tax management and the ability to exclude addresses from fees. While this offers flexibility, it also centralizes power significantly, which could be a risk if not managed transparently.

3. Reentrancy Protection:

- Functions that transfer tokens use the `nonReentrant` modifier, mitigating risks of reentrancy attacks effectively.

4. Tax Logic:

- Tax rates are validated upon updates, ensuring they remain within reasonable bounds (0-100%).
- Proper checks ensure that taxes are only applied to transfers involving buys and sells, with an exemption for internal transfers or wallet-to-wallet transactions that do not interact with DEXs.

Potential Vulnerabilities:

1. Centralization Risks:

- Centralized control in tax settings and ownership transitions could lead to potential misuse if the owner account is compromised.

2. Gas Usage:

- The implementation of taxes and checks in transfer functions could lead to higher gas costs, particularly when multiple state updates and conditions are evaluated.

3. Upgradability and Maintenance:

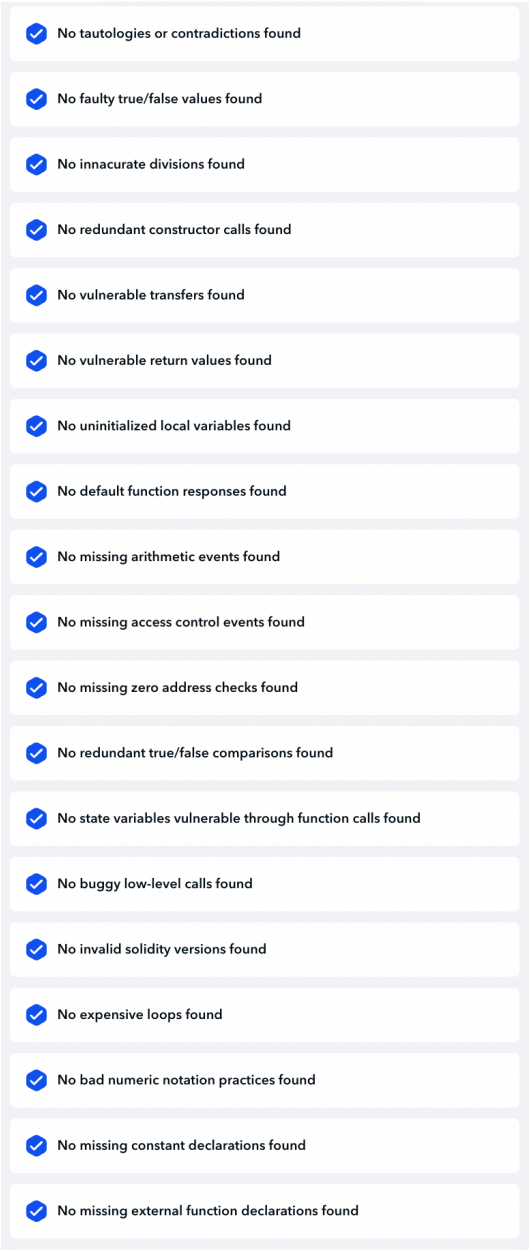
- The contract does not support upgradability. This could be a limitation if there's a need to adapt to future business requirements or to address issues not currently foreseeable.

Recommendations:

1. **Decentralization of Control:**
    - Consider implementing governance mechanisms for critical operations like tax rate changes or ownership transfers to distribute control among trusted parties or stakeholders.
2. **Enhanced Tax Management:**
    - Introduce caps or more nuanced rules on how taxes are applied, potentially based on transaction size or other factors, to prevent overly punitive rates on smaller token holders.
3. **Audit and Monitoring:**
    - Regular audits and continuous monitoring of contract interactions are recommended to ensure security and proper functioning. Monitoring tools could help detect abnormal behavior early.
4. **Transparency and Communication:**
    - Given the significant owner privileges, maintaining transparency about changes and providing detailed logs or justifications for changes can help build and maintain trust with the community.

Conclusion:

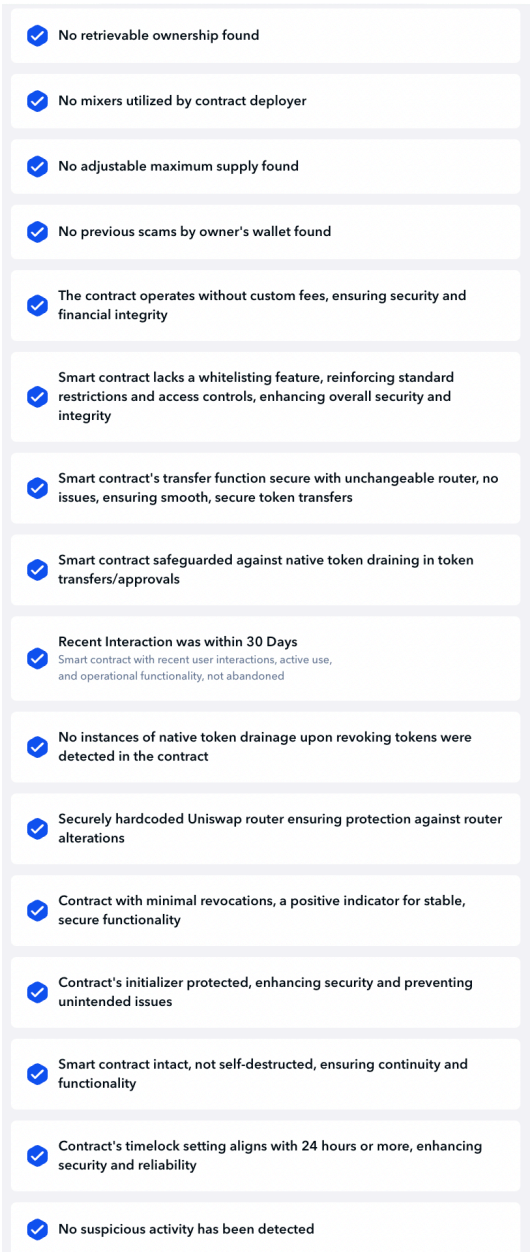
The REDSF smart contract incorporates robust features for transaction tax management, ownership control, and security against reentrant attacks. While its functionalities align with its intended use case, the centralization of control points warrants careful governance and operational security to mitigate potential risks. Regular audits and adherence to best practices in smart contract development and management are crucial for its long-term success and security.



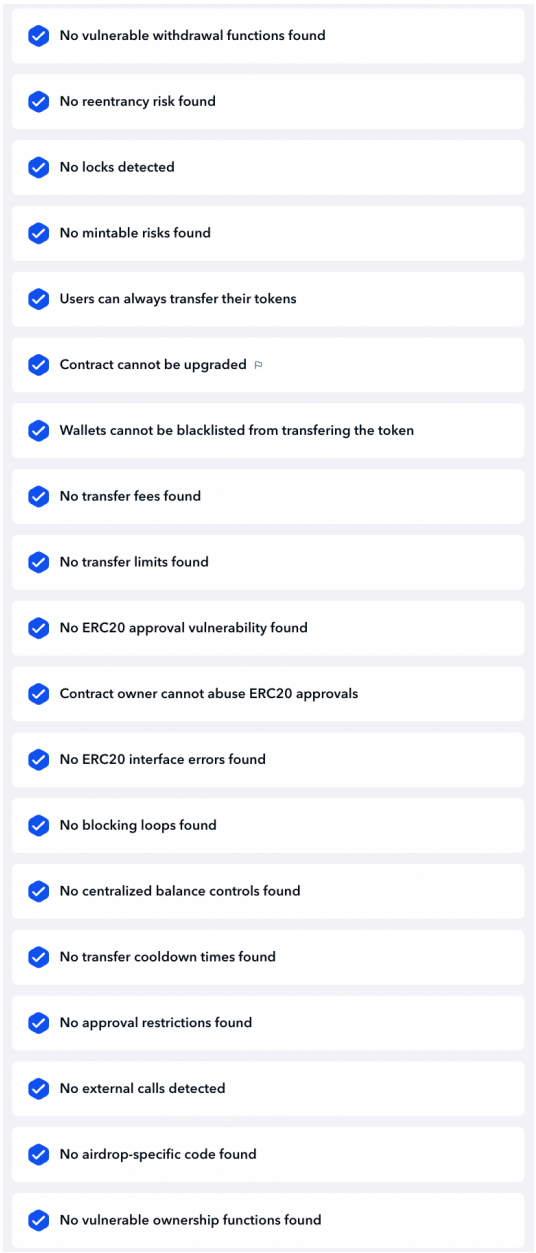

```
/**
 *Submitted for verification at polygonscan.com on 2024-11-18
*/

/**
 *Submitted for verification at Etherscan.io on 2024-10-15
*/

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.22;


// File: @openzeppelin/contracts/utils/Context.sol


// OpenZeppelin Contracts (last updated v5.0.1) (utils/Context.sol)

pragma solidity ^0.8.19;

/**
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
*/
abstract contract Context {
    function _msgSender() internal view virtual returns (address) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes calldata) {
        return msg.data;
    }
```

- No compiler version inconsistencies found
- No unchecked call responses found
- No vulnerable self-destruct functions found
- No assertion vulnerabilities found
- No old solidity code found
- No external delegated calls found
- No external call dependency found
- No vulnerable authentication calls found
- No invalid character typos found
- No RTL characters found
- No dead code found
- No risky data allocation found
- No uninitialized state variables found
- No uninitialized storage variables found
- No vulnerable initialization functions found
- No risky data handling found
- No number accuracy bug found
- No out-of-range number vulnerability found
- No map data deletion vulnerabilities found

```
    function _contextSuffixLength() internal view virtual returns (uint256) {
        return 0;
    }
}


/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * The initial owner is set to the address provided by the deployer. This can
 * later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
abstract contract Ownable is Context {
    address private _owner;

    /**
     * @dev The caller account is not authorized to perform an operation.
     */
    error OwnableUnauthorizedAccount(address account);

    /**
     * @dev The owner is not a valid owner account. (eg. `address(0)`)
     */
    error OwnableInvalidOwner(address owner);

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Initializes the contract setting the address provided by the deployer as the initial owner.
     */
    constructor(address initialOwner) {
        if (initialOwner == address(0)) {
            revert OwnableInvalidOwner(address(0));
        }
        _transferOwnership(initialOwner);
```




```
}

/**
 * @dev Throws if called by any account other than the owner.
 */
modifier onlyOwner() {
    _checkOwner();
    _;
}

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

/**
 * @dev Throws if the sender is not the owner.
 */
function _checkOwner() internal view virtual {
    if (owner() != _msgSender()) {
        revert OwnableUnauthorizedAccount(_msgSender());
    }
}

/**
 * @dev Leaves the contract without owner. It will not be possible to call
 * `onlyOwner` functions. Can only be called by the current owner.
 *
 * NOTE: Renouncing ownership will leave the contract without an owner,
 * thereby disabling any functionality that is only available to the owner.
 */
function renounceOwnership() public virtual onlyOwner {
    _transferOwnership(address(0));
}

/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`).
 * Can only be called by the current owner.
 */
function transferOwnership(address newOwner) public virtual onlyOwner {
```




```
        if (newOwner == address(0)) {
            revert OwnableInvalidOwner(address(0));
        }
        _transferOwnership(newOwner);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`).
     * Internal function without access restriction.
     */
    function _transferOwnership(address newOwner) internal virtual {
        address oldOwner = _owner;
        _owner = newOwner;
        emit OwnershipTransferred(oldOwner, newOwner);
    }
}

// File: @openzeppelin/contracts/access/Ownable2Step.sol


// OpenZeppelin Contracts (last updated v5.0.0) (access/Ownable2Step.sol)

pragma solidity ^0.8.19;


/**
 * @dev Contract module which provides access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * The initial owner is specified at deployment time in the constructor for `Ownable`. This
 * can later be changed with {transferOwnership} and {acceptOwnership}.
 *
 * This module is used through inheritance. It will make available all functions
 * from parent (Ownable).
 */
abstract contract Ownable2Step is Ownable {
    address private _pendingOwner;

    event OwnershipTransferStarted(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Returns the address of the pending owner.
     */
    function pendingOwner() public view virtual returns (address) {
        return _pendingOwner;
    }

    /**
     * @dev Starts the ownership transfer of the contract to a new account. Replaces the pending transfer if there is one.
     * Can only be called by the current owner.
     */
    function transferOwnership(address newOwner) public virtual override onlyOwner {
```

```
        _pendingOwner = newOwner;
        emit OwnershipTransferStarted(owner(), newOwner);
    }

    /**
     * @dev Transfers ownership of the contract to a new account (`newOwner`) and deletes any pending owner.
     * Internal function without access restriction.
     */
    function _transferOwnership(address newOwner) internal virtual override {
        delete _pendingOwner;
        super._transferOwnership(newOwner);
    }

    /**
     * @dev The new owner accepts the ownership transfer.
     */
    function acceptOwnership() public virtual {
        address sender = _msgSender();
        if (pendingOwner() != sender) {
            revert OwnableUnauthorizedAccount(sender);
        }
        _transferOwnership(sender);
    }
}

// File: @openzeppelin/contracts/token/ERC20/IERC20.sol


// OpenZeppelin Contracts (last updated v5.0.0) (token/ERC20/IERC20.sol)

pragma solidity ^0.8.19;

/**
 * @dev Interface of the ERC20 standard as defined in the EIP.
 */
interface IERC20 {
    /**
     * @dev Emitted when `value` tokens are moved from one account (`from`) to
     * another (`to`).
     *
     * Note that `value` may be zero.
     */
    event Transfer(address indexed from, address indexed to, uint256 value);

    /**
     * @dev Emitted when the allowance of a `spender` for an `owner` is set by
     * a call to {approve}. `value` is the new allowance.
     */
    event Approval(address indexed owner, address indexed spender, uint256 value);

    /**
     * @dev Returns the value of tokens in existence.
     */
    function totalSupply() external view returns (uint256);
```

```
    /**
     * @dev Returns the value of tokens owned by `account`.
     */
    function balanceOf(address account) external view returns (uint256);

    /**
     * @dev Moves a `value` amount of tokens from the caller's account to `to`.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address to, uint256 value) external returns (bool);

    /**
     * @dev Returns the remaining number of tokens that `spender` will be
     * allowed to spend on behalf of `owner` through {transferFrom}. This is
     * zero by default.
     *
     * This value changes when {approve} or {transferFrom} are called.
     */
    function allowance(address owner, address spender) external view returns (uint256);

    /**
     * @dev Sets a `value` amount of tokens as the allowance of `spender` over the
     * caller's tokens.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * IMPORTANT: Beware that changing an allowance with this method brings the risk
     * that someone may use both the old and the new allowance by unfortunate
     * transaction ordering. One possible solution to mitigate this race
     * condition is to first reduce the spender's allowance to 0 and set the
     * desired value afterwards:
     * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
     *
     * Emits an {Approval} event.
     */
    function approve(address spender, uint256 value) external returns (bool);

    /**
     * @dev Moves a `value` amount of tokens from `from` to `to` using the
     * allowance mechanism. `value` is then deducted from the caller's
     * allowance.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transferFrom(address from, address to, uint256 value) external returns (bool);
}

// File: @openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol
```

```
// OpenZeppelin Contracts (last updated v5.0.0) (token/ERC20/extensions/IERC20Metadata.sol)

pragma solidity ^0.8.19;


/**
 * @dev Interface for the optional metadata functions from the ERC20 standard.
 */
interface IERC20Metadata is IERC20 {
    /**
     * @dev Returns the name of the token.
     */
    function name() external view returns (string memory);

    /**
     * @dev Returns the symbol of the token.
     */
    function symbol() external view returns (string memory);

    /**
     * @dev Returns the decimals places of the token.
     */
    function decimals() external view returns (uint8);
}


// OpenZeppelin Contracts (last updated v5.0.0) (interfaces/draft-IERC6093.sol)
pragma solidity ^0.8.19;

/**
 * @dev Standard ERC20 Errors
 * Interface of the https://eips.ethereum.org/EIPS/eip-6093[ERC-6093] custom errors for ERC20 tokens.
 */
interface IERC20Errors {
    /**
     * @dev Indicates an error related to the current `balance` of a `sender`. Used in transfers.
     * @param sender Address whose tokens are being transferred.
     * @param balance Current balance for the interacting account.
     * @param needed Minimum amount required to perform a transfer.
     */
    error ERC20InsufficientBalance(address sender, uint256 balance, uint256 needed);

    /**
     * @dev Indicates a failure with the token `sender`. Used in transfers.
     * @param sender Address whose tokens are being transferred.
     */
    error ERC20InvalidSender(address sender);

    /**
     * @dev Indicates a failure with the token `receiver`. Used in transfers.
     * @param receiver Address to which tokens are being transferred.
     */
```

```
    error ERC20InvalidReceiver(address receiver);

    /**
     * @dev Indicates a failure with the `spender`'s `allowance`. Used in transfers.
     * @param spender Address that may be allowed to operate on tokens without being their owner.
     * @param allowance Amount of tokens a `spender` is allowed to operate with.
     * @param needed Minimum amount required to perform a transfer.
     */
    error ERC20InsufficientAllowance(address spender, uint256 allowance, uint256 needed);

    /**
     * @dev Indicates a failure with the `approver` of a token to be approved. Used in approvals.
     * @param approver Address initiating an approval operation.
     */
    error ERC20InvalidApprover(address approver);

    /**
     * @dev Indicates a failure with the `spender` to be approved. Used in approvals.
     * @param spender Address that may be allowed to operate on tokens without being their owner.
     */
    error ERC20InvalidSpender(address spender);
}

/**
 * @dev Standard ERC721 Errors
 * Interface of the https://eips.ethereum.org/EIPS/eip-6093[ERC-6093] custom errors for ERC721 tokens.
 */
interface IERC721Errors {
    /**
     * @dev Indicates that an address can't be an owner. For example, `address(0)` is a forbidden owner in EIP-20.
     * Used in balance queries.
     * @param owner Address of the current owner of a token.
     */
    error ERC721InvalidOwner(address owner);

    /**
     * @dev Indicates a `tokenId` whose `owner` is the zero address.
     * @param tokenId Identifier number of a token.
     */
    error ERC721NonexistentToken(uint256 tokenId);

    /**
     * @dev Indicates an error related to the ownership over a particular token. Used in transfers.
     * @param sender Address whose tokens are being transferred.
     * @param tokenId Identifier number of a token.
     * @param owner Address of the current owner of a token.
     */
    error ERC721IncorrectOwner(address sender, uint256 tokenId, address owner);

    /**
     * @dev Indicates a failure with the token `sender`. Used in transfers.
     * @param sender Address whose tokens are being transferred.
     */
    error ERC721InvalidSender(address sender);
```

```
    /**
     * @dev Indicates a failure with the token `receiver`. Used in transfers.
     * @param receiver Address to which tokens are being transferred.
     */
    error ERC721InvalidReceiver(address receiver);

    /**
     * @dev Indicates a failure with the `operator`'s approval. Used in transfers.
     * @param operator Address that may be allowed to operate on tokens without being their owner.
     * @param tokenId Identifier number of a token.
     */
    error ERC721InsufficientApproval(address operator, uint256 tokenId);

    /**
     * @dev Indicates a failure with the `approver` of a token to be approved. Used in approvals.
     * @param approver Address initiating an approval operation.
     */
    error ERC721InvalidApprover(address approver);

    /**
     * @dev Indicates a failure with the `operator` to be approved. Used in approvals.
     * @param operator Address that may be allowed to operate on tokens without being their owner.
     */
    error ERC721InvalidOperator(address operator);
}

/**
 * @dev Standard ERC1155 Errors
 * Interface of the https://eips.ethereum.org/EIPS/eip-6093[ERC-6093] custom errors for ERC1155 tokens.
 */
interface IERC1155Errors {
    /**
     * @dev Indicates an error related to the current `balance` of a `sender`. Used in transfers.
     * @param sender Address whose tokens are being transferred.
     * @param balance Current balance for the interacting account.
     * @param needed Minimum amount required to perform a transfer.
     * @param tokenId Identifier number of a token.
     */
    error ERC1155InsufficientBalance(address sender, uint256 balance, uint256 needed, uint256 tokenId);

    /**
     * @dev Indicates a failure with the token `sender`. Used in transfers.
     * @param sender Address whose tokens are being transferred.
     */
    error ERC1155InvalidSender(address sender);

    /**
     * @dev Indicates a failure with the token `receiver`. Used in transfers.
     * @param receiver Address to which tokens are being transferred.
     */
    error ERC1155InvalidReceiver(address receiver);

    /**
```

```
     * @dev Indicates a failure with the `operator`s approval. Used in transfers.
     * @param operator Address that may be allowed to operate on tokens without being their owner.
     * @param owner Address of the current owner of a token.
     */
    error ERC1155MissingApprovalForAll(address operator, address owner);

    /**
     * @dev Indicates a failure with the `approver` of a token to be approved. Used in approvals.
     * @param approver Address initiating an approval operation.
     */
    error ERC1155InvalidApprover(address approver);

    /**
     * @dev Indicates a failure with the `operator` to be approved. Used in approvals.
     * @param operator Address that may be allowed to operate on tokens without being their owner.
     */
    error ERC1155InvalidOperator(address operator);

    /**
     * @dev Indicates an array length mismatch between ids and values in a safeBatchTransferFrom operation.
     * Used in batch transfers.
     * @param idsLength Length of the array of token identifiers
     * @param valuesLength Length of the array of token amounts
     */
    error ERC1155InvalidArrayLength(uint256 idsLength, uint256 valuesLength);
}


// File: @openzeppelin/contracts/token/ERC20/ERC20.sol


// OpenZeppelin Contracts (last updated v5.0.0) (token/ERC20/ERC20.sol)

pragma solidity ^0.8.19;




/**
 * @dev Implementation of the {IERC20} interface.
 *
 * This implementation is agnostic to the way tokens are created. This means
 * that a supply mechanism has to be added in a derived contract using {_mint}.
 *
 * TIP: For a detailed writeup see our guide
 * https://forum.openzeppelin.com/t/how-to-implement-erc20-supply-mechanisms/226[How
 * to implement supply mechanisms].
 *
 * The default value of {decimals} is 18. To change this, you should override
 * this function so it returns a different value.
 *
 * We have followed general OpenZeppelin Contracts guidelines: functions revert
 * instead returning `false` on failure. This behavior is nonetheless
 * conventional and does not conflict with the expectations of ERC20
```

```
 * applications.
 *
 * Additionally, an {Approval} event is emitted on calls to {transferFrom}.
 * This allows applications to reconstruct the allowance for all accounts just
 * by listening to said events. Other implementations of the EIP may not emit
 * these events, as it isn't required by the specification.
 */
abstract contract ERC20 is Context, IERC20, IERC20Metadata, IERC20Errors {
    mapping(address account => uint256) private _balances;

    mapping(address account => mapping(address spender => uint256)) private _allowances;

    uint256 private _totalSupply;

    string private _name;
    string private _symbol;

    /**
     * @dev Sets the values for {name} and {symbol}.
     *
     * All two of these values are immutable: they can only be set once during
     * construction.
     */
    constructor(string memory name_, string memory symbol_) {
        _name = name_;
        _symbol = symbol_;
    }

    /**
     * @dev Returns the name of the token.
     */
    function name() public view virtual returns (string memory) {
        return _name;
    }

    /**
     * @dev Returns the symbol of the token, usually a shorter version of the
     * name.
     */
    function symbol() public view virtual returns (string memory) {
        return _symbol;
    }

    /**
     * @dev Returns the number of decimals used to get its user representation.
     * For example, if `decimals` equals `2`, a balance of `505` tokens should
     * be displayed to a user as `5.05` (`505 / 10 ** 2`).
     *
     * Tokens usually opt for a value of 18, imitating the relationship between
     * Ether and Wei. This is the default value returned by this function, unless
     * it's overridden.
     *
     * NOTE: This information is only used for _display_ purposes: it in
     * no way affects any of the arithmetic of the contract, including
```

```
     * {IERC20-balanceOf} and {IERC20-transfer}.
     */
    function decimals() public view virtual returns (uint8) {
        return 18;
    }

    /**
     * @dev See {IERC20-totalSupply}.
     */
    function totalSupply() public view virtual returns (uint256) {
        return _totalSupply;
    }

    /**
     * @dev See {IERC20-balanceOf}.
     */
    function balanceOf(address account) public view virtual returns (uint256) {
        return _balances[account];
    }

    /**
     * @dev See {IERC20-transfer}.
     *
     * Requirements:
     *
     * - `to` cannot be the zero address.
     * - the caller must have a balance of at least `value`.
     */
    function transfer(address to, uint256 value) public virtual returns (bool) {
        address owner = _msgSender();
        _transfer(owner, to, value);
        return true;
    }

    /**
     * @dev See {IERC20-allowance}.
     */
    function allowance(address owner, address spender) public view virtual returns (uint256) {
        return _allowances[owner][spender];
    }

    /**
     * @dev See {IERC20-approve}.
     *
     * NOTE: If `value` is the maximum `uint256`, the allowance is not updated on
     * `transferFrom`. This is semantically equivalent to an infinite approval.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function approve(address spender, uint256 value) public virtual returns (bool) {
        address owner = _msgSender();
        _approve(owner, spender, value);
```

```
    return true;
}

/**
 * @dev See {IERC20-transferFrom}.
 *
 * Emits an {Approval} event indicating the updated allowance. This is not
 * required by the EIP. See the note at the beginning of {ERC20}.
 *
 * NOTE: Does not update the allowance if the current allowance
 * is the maximum `uint256`.
 *
 * Requirements:
 *
 * - `from` and `to` cannot be the zero address.
 * - `from` must have a balance of at least `value`.
 * - the caller must have allowance for ``from``'s tokens of at least
 * `value`.
 */
function transferFrom(address from, address to, uint256 value) public virtual returns (bool) {
    address spender = _msgSender();
    _spendAllowance(from, spender, value);
    _transfer(from, to, value);
    return true;
}

/**
 * @dev Moves a `value` amount of tokens from `from` to `to`.
 *
 * This internal function is equivalent to {transfer}, and can be used to
 * e.g. implement automatic token fees, slashing mechanisms, etc.
 *
 * Emits a {Transfer} event.
 *
 * NOTE: This function is not virtual, {_update} should be overridden instead.
 */
function _transfer(address from, address to, uint256 amount) internal {
    if (from == address(0)) {
        revert ERC20InvalidSender(address(0));
    }
    if (to == address(0)) {
        revert ERC20InvalidReceiver(address(0));
    }
    _update(from, to, amount);
}

/**
 * @dev Transfers a `value` amount of tokens from `from` to `to`, or alternatively mints (or burns) if `from`
 * (or `to`) is the zero address. All customizations to transfers, mints, and burns should be done by overriding
 * this function.
 *
 * Emits a {Transfer} event.
 */
function _update(address from, address to, uint256 amount) internal virtual {
```

```
        if (from == address(0)) {
            // Overflow check required: The rest of the code assumes that totalSupply never overflows
            _totalSupply += amount;
        } else {
            uint256 fromBalance = _balances[from];
            if (fromBalance < amount) {
                revert ERC20InsufficientBalance(from, fromBalance, amount);
            }
            unchecked {
                // Overflow not possible: amount <= fromBalance <= totalSupply.
                _balances[from] = fromBalance - amount;
            }
        }

        if (to == address(0)) {
            unchecked {
                // Overflow not possible: amount <= totalSupply or amount <= fromBalance <= totalSupply.
                _totalSupply -= amount;
            }
        } else {
            unchecked {
                // Overflow not possible: balance + amount is at most totalSupply, which we know fits into a uint256.
                _balances[to] += amount;
            }
        }

        emit Transfer(from, to, amount);
    }

    /**
     * @dev Creates a `value` amount of tokens and assigns them to `account`, by transferring it from address(0).
     * Relies on the `_update` mechanism
     *
     * Emits a {Transfer} event with `from` set to the zero address.
     *
     * NOTE: This function is not virtual, {_update} should be overridden instead.
     */
    function _mint(address account, uint256 value) internal {
        if (account == address(0)) {
            revert ERC20InvalidReceiver(address(0));
        }
        _update(address(0), account, value);
    }

    /**
     * @dev Destroys a `value` amount of tokens from `account`, lowering the total supply.
     * Relies on the `_update` mechanism.
     *
     * Emits a {Transfer} event with `to` set to the zero address.
     *
     * NOTE: This function is not virtual, {_update} should be overridden instead
     */
    function _burn(address account, uint256 value) internal {
        if (account == address(0)) {
```

```
            revert ERC20InvalidSender(address(0));
        }
        _update(account, address(0), value);
    }

    /**
     * @dev Sets `value` as the allowance of `spender` over the `owner` s tokens.
     *
     * This internal function is equivalent to `approve`, and can be used to
     * e.g. set automatic allowances for certain subsystems, etc.
     *
     * Emits an {Approval} event.
     *
     * Requirements:
     *
     * - `owner` cannot be the zero address.
     * - `spender` cannot be the zero address.
     *
     * Overrides to this logic should be done to the variant with an additional `bool emitEvent` argument.
     */
    function _approve(address owner, address spender, uint256 value) internal {
        _approve(owner, spender, value, true);
    }

    /**
     * @dev Variant of {_approve} with an optional flag to enable or disable the {Approval} event.
     *
     * By default (when calling {_approve}) the flag is set to true. On the other hand, approval changes made by
     * `_spendAllowance` during the `transferFrom` operation set the flag to false. This saves gas by not emitting any
     * `Approval` event during `transferFrom` operations.
     *
     * Anyone who wishes to continue emitting `Approval` events on the`transferFrom` operation can force the flag to
     * true using the following override:
     * ```
     * function _approve(address owner, address spender, uint256 value, bool) internal virtual override {
     *     super._approve(owner, spender, value, true);
     * }
     * ```
     *
     * Requirements are the same as {_approve}.
     */
    function _approve(address owner, address spender, uint256 value, bool emitEvent) internal virtual {
        if (owner == address(0)) {
            revert ERC20InvalidApprover(address(0));
        }
        if (spender == address(0)) {
            revert ERC20InvalidSpender(address(0));
        }
        _allowances[owner][spender] = value;
        if (emitEvent) {
            emit Approval(owner, spender, value);
        }
    }
```

```
    /**
     * @dev Updates `owner` s allowance for `spender` based on spent `value`.
     *
     * Does not update the allowance value in case of infinite allowance.
     * Revert if not enough allowance is available.
     *
     * Does not emit an {Approval} event.
     */
    function _spendAllowance(address owner, address spender, uint256 value) internal virtual {
        uint256 currentAllowance = allowance(owner, spender);
        if (currentAllowance != type(uint256).max) {
            if (currentAllowance < value) {
                revert ERC20InsufficientAllowance(spender, currentAllowance, value);
            }
            unchecked {
                _approve(owner, spender, currentAllowance - value, false);
            }
        }
    }
}


// OpenZeppelin Contracts (last updated v4.9.0) (security/ReentrancyGuard.sol)

pragma solidity ^0.8.0;

/**
 * @dev Contract module that helps prevent reentrant calls to a function.
 *
 * Inheriting from `ReentrancyGuard` will make the {nonReentrant} modifier
 * available, which can be applied to functions to make sure there are no nested
 * (reentrant) calls to them.
 *
 * Note that because there is a single `nonReentrant` guard, functions marked as
 * `nonReentrant` may not call one another. This can be worked around by making
 * those functions `private`, and then adding `external` `nonReentrant` entry
 * points to them.
 *
 * TIP: If you would like to learn more about reentrancy and alternative ways
 * to protect against it, check out our blog post
 * https://blog.openzeppelin.com/reentrancy-after-istanbul/[Reentrancy After Istanbul].
 */
abstract contract ReentrancyGuard {
    // Booleans are more expensive than uint256 or any type that takes up a full
    // word because each write operation emits an extra SLOAD to first read the
    // slot's contents, replace the bits taken up by the boolean, and then write
    // back. This is the compiler's defense against contract upgrades and
    // pointer aliasing, and it cannot be disabled.

    // The values being non-zero value makes deployment a bit more expensive,
    // but in exchange the refund on every call to nonReentrant will be lower in
    // amount. Since refunds are capped to a percentage of the total
    // transaction's gas, it is best to keep them low in cases like this one, to
    // increase the likelihood of the full refund coming into effect.
```

```
    uint256 private constant _NOT_ENTERED = 1;
    uint256 private constant _ENTERED = 2;

    uint256 private _status;

    constructor() {
        _status = _NOT_ENTERED;
    }

    /**
     * @dev Prevents a contract from calling itself, directly or indirectly.
     * Calling a `nonReentrant` function from another `nonReentrant`
     * function is not supported. It is possible to prevent this from happening
     * by making the `nonReentrant` function external, and making it call a
     * `private` function that does the actual work.
     */
    modifier nonReentrant() {
        _nonReentrantBefore();
        _;
        _nonReentrantAfter();
    }

    function _nonReentrantBefore() private {
        // On the first call to nonReentrant, _status will be _NOT_ENTERED
        require(_status != _ENTERED, "ReentrancyGuard: reentrant call");

        // Any calls to nonReentrant after this point will fail
        _status = _ENTERED;
    }

    function _nonReentrantAfter() private {
        // By storing the original value once again, a refund is triggered (see
        // https://eips.ethereum.org/EIPS/eip-2200)
        _status = _NOT_ENTERED;
    }

    /**
     * @dev Returns true if the reentrancy guard is currently set to "entered", which indicates there is a
     * `nonReentrant` function in the call stack.
     */
    function _reentrancyGuardEntered() internal view returns (bool) {
        return _status == _ENTERED;
    }
}


contract REDSF is Context, IERC20, Ownable2Step, ReentrancyGuard {
    string public name = "REDSF";
    string public symbol = "REDSF";
    uint256 public buyTax = 0;
    uint256 public sellTax = 0;
    uint256 public transferTax = 0; // Default transfer tax is 0
    uint8 private _decimals = 18;
    address public taxReceiver;
```

```
event TaxCollected(address indexed taxReceiver, uint256 amount);
event TaxUpdated(string taxType, uint256 newTaxRate);
event TaxReceiverUpdated(address indexed oldReceiver, address indexed newReceiver);

uint256 private constant MAX_TAX_PERCENTAGE = 100;
uint256 private constant DECIMALS = 18;
uint256 private constant INITIAL_SUPPLY = 1_000_000_000 * 10 ** DECIMALS;

uint256 public totalSupply = INITIAL_SUPPLY;
mapping(address => uint256) private balances;
mapping(address => mapping(address => uint256)) private allowances;
mapping(address => bool) private _isExcludedFromFee;
mapping(address => bool) private _isDex; // Track known DEX addresses

// Custom errors
error InvalidAddress(address addr);
error InsufficientBalance(address account, uint256 balance, uint256 required);
error TransferFailed(address from, address to, uint256 amount);

constructor(address _taxReceiver) Ownable(msg.sender) {
    require(msg.sender != address(0), "Invalid owner address");
    require(_taxReceiver != address(0), "Invalid tax receiver address");
    taxReceiver = _taxReceiver;
    balances[msg.sender] = INITIAL_SUPPLY;
    _isExcludedFromFee[msg.sender] = true;
}

function balanceOf(address account) external view override returns (uint256) {
    return balances[account];
}

function decimals() public view returns (uint8) {
    return _decimals;
}


function setBuyTax(uint256 newTax) external onlyOwner {
    require(newTax >= 0 && newTax < MAX_TAX_PERCENTAGE, "Tax percentage must be between 0 and 100");
    buyTax = newTax;
    emit TaxUpdated("Buy", newTax);
}

function setSellTax(uint256 newTax) external onlyOwner {
    require(newTax >= 0 && newTax < MAX_TAX_PERCENTAGE, "Tax percentage must be between 0 and 100");
    sellTax = newTax;
    emit TaxUpdated("Sell", newTax);
}

function setTransferTax(uint256 newTax) external onlyOwner {
    require(newTax >= 0 && newTax < MAX_TAX_PERCENTAGE, "Tax percentage must be between 0 and 100");
    transferTax = newTax;
    emit TaxUpdated("Transfer", newTax);
}
```

```
function setTaxReceiver(address newTaxReceiver) external onlyOwner {
    require(newTaxReceiver != address(0), "Invalid tax receiver address");
    address oldTaxReceiver = taxReceiver;
    taxReceiver = newTaxReceiver;
    emit TaxReceiverUpdated(oldTaxReceiver, newTaxReceiver);
}

function transfer(address to, uint256 amount) external override nonReentrant returns (bool) {
    _transfer(msg.sender, to, amount);
    return true;
}

function transferFrom(address from, address to, uint256 amount) external override nonReentrant returns (bool) {
    uint256 currentAllowance = allowances[from][msg.sender];
    require(currentAllowance >= amount, "ERC20: transfer amount exceeds allowance");
    unchecked {
        allowances[from][msg.sender] = currentAllowance - amount;
    }
    _transfer(from, to, amount);
    return true;
}

function _transfer(address from, address to, uint256 amount) internal {
    if (to == address(0)) {
        revert InvalidAddress(to);
    }
    uint256 senderBalance = balances[from];
    if (senderBalance < amount) {
        revert InsufficientBalance(from, senderBalance, amount);
    }

    uint256 amountAfterFee = amount;
    uint256 fee = 0;
    if (!_isExcludedFromFee[from] && !_isExcludedFromFee[to]) {
        uint256 taxRate;
        if (_isDex[from]) {
            // If the sender is a DEX, it's a buy
            taxRate = buyTax;
        } else if (_isDex[to]) {
            // If the receiver is a DEX, it's a sell
            taxRate = sellTax;
        } else {
            // If neither is a DEX, it's a transfer
            taxRate = transferTax;
        }

        fee = (amount * taxRate) / MAX_TAX_PERCENTAGE;
        amountAfterFee = amount - fee;
    }

    // Effects - Update all balances in one block
    balances[from] -= amount;
    balances[to] += amountAfterFee;
    if (fee > 0) {
```

```
        balances[taxReceiver] += fee;
    }

    // Events
    emit Transfer(from, to, amountAfterFee);
    if (fee > 0) {
        emit Transfer(address(this), taxReceiver, fee);
        emit TaxCollected(taxReceiver, fee);
    }
}

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) {
    allowances[msg.sender][spender] = amount;
    emit Approval(msg.sender, spender, amount);
    return true;
}

function increaseAllowance(address spender, uint256 addedValue) external returns (bool) {
    uint256 currentAllowance = allowances[msg.sender][spender];
    uint256 newAllowance = currentAllowance + addedValue;
    require(newAllowance >= currentAllowance, "ERC20: allowance overflow");
    _approve(msg.sender, spender, newAllowance);
    return true;
}

function decreaseAllowance(address spender, uint256 subtractedValue) external returns (bool) {
    uint256 currentAllowance = allowances[msg.sender][spender];
    require(currentAllowance >= subtractedValue, "ERC20: decreased allowance below zero");
    uint256 newAllowance = currentAllowance - subtractedValue;
    _approve(msg.sender, spender, newAllowance);
    return true;
}

function _approve(address owner, address spender, uint256 amount) internal {
    allowances[owner][spender] = amount;
    emit Approval(owner, spender, amount);
}

function excludeFromFee(address account) external onlyOwner {
    _isExcludedFromFee[account] = true;
}

function includeInFee(address account) external onlyOwner {
    _isExcludedFromFee[account] = false;
```

```
    }

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

    // Function to add a DEX address
    function addDexAddress(address dexAddress) external onlyOwner {
        _isDex[dexAddress] = true;
    }

    // Function to remove a DEX address
    function removeDexAddress(address dexAddress) external onlyOwner {
        _isDex[dexAddress] = false;
    }
}
```

Terrence Nibbles, CCE, CCA Auditor #17865