EUROe Stablecoin Smart Contract Test Coverage Report

100% Statements `37/37` **100%** Branches `10/10` **100%** Functions `15/15` **100%** Lines `38/38`

File ▲		Statements		Branches		Functions		Lines	
EUROe.sol	🟩	100%	37/37	100%	10/10	100%	15/15	100%	38/38

EUROe.sol

100% Statements `37/37` **100%** Branches `10/10` **100%** Functions `15/15` **100%** Lines `38/38`

```
    // SPDX-License-Identifier: MIT
    pragma solidity 0.8.4;

    import "@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol";
    import "@openzeppelin/contracts-upgradeable/token/ERC20/IERC20Upgradeable.sol";
    import "@openzeppelin/contracts-upgradeable/token/ERC20/extensions/ERC20BurnableUpgradeable.sol";
    import "@openzeppelin/contracts-upgradeable/security/PausableUpgradeable.sol";
    import "@openzeppelin/contracts-upgradeable/access/AccessControlUpgradeable.sol";
    import "@openzeppelin/contracts-upgradeable/token/ERC20/extensions/draft-ERC20PermitUpgradeable.sol";
    import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";
    import "@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol";
    import "@openzeppelin/contracts-upgradeable/token/ERC20/utils/SafeERC20Upgradeable.sol";

    /**
    @title A stablecoin ERC20 token contract for EUROe
    @author Membrane Finance
    @notice This contract implements the EUROe stablecoin along with its core functionality, such as minting and burning
    @dev This contract is upgradable. It is implemented as an EIP-1967 transparent upgradable proxy. The PROXYOWNER_ROLE controls upgrades t
     */
    contract EUROe is
        Initializable,
        ERC20Upgradeable,
        ERC20BurnableUpgradeable,
        PausableUpgradeable,
        AccessControlUpgradeable,
        ERC20PermitUpgradeable,
        UUPSUpgradeable
    {
        using SafeERC20Upgradeable for IERC20Upgradeable;

        bytes32 public constant PROXYOWNER_ROLE = keccak256("PROXYOWNER_ROLE");
        bytes32 public constant BLOCKLISTER_ROLE = keccak256("BLOCKLISTER_ROLE");
        bytes32 public constant PAUSER_ROLE = keccak256("PAUSER_ROLE");
        bytes32 public constant UNPAUSER_ROLE = keccak256("UNPAUSER_ROLE");
        bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
        bytes32 public constant BLOCKED_ROLE = keccak256("BLOCKED_ROLE");
        bytes32 public constant RESCUER_ROLE = keccak256("RESCUER_ROLE");
        bytes32 public constant BURNER_ROLE = keccak256("BURNER_ROLE");

        /**
         * @dev Emitted once a minting set has been completed
         * @param id External identifier for the minting set
         */
        event MintingSetCompleted(uint256 indexed id);

        /// @custom:oz-upgrades-unsafe-allow constructor
        constructor() {
6×          _disableInitializers();
        }

        /**
         * @dev Initializes the (upgradeable) contract.
         * @param proxyOwner Address for whom to give the proxyOwner role
         * @param admin Address for whom to give the admin role
         * @param blocklister Address for whom to give the blocklister role
         * @param pauser Address for whom to give the pauser role
         * @param unpauser Address for whom to give the unpauser role
         * @param minter Address for whom to give the minter role
         */
        function initialize(
            address proxyOwner,
            address admin,
            address blocklister,
            address pauser,
            address unpauser,
            address minter,
            address rescuer,
            address burner
        ) external initializer {
3×          __ERC20_init("EUROe Stablecoin", "EUROe");
3×          __ERC20Burnable_init();
3×          __Pausable_init();
3×          __AccessControl_init();
3×          __ERC20Permit_init("EUROe Stablecoin");
3×          __UUPSUpgradeable_init();

3×          _grantRole(PROXYOWNER_ROLE, proxyOwner);
3×          _grantRole(DEFAULT_ADMIN_ROLE, admin);
3×          _grantRole(BLOCKLISTER_ROLE, blocklister);
3×          _grantRole(PAUSER_ROLE, pauser);
3×          _grantRole(UNPAUSER_ROLE, unpauser);
3×          _grantRole(MINTER_ROLE, minter);
3×          _grantRole(RESCUER_ROLE, rescuer);
3×          _grantRole(BURNER_ROLE, burner);

            // Add this contract as blocked so it can't receive its own tokens by accident
```

```
 3×         _grantRole(BLOCKED_ROLE, address(this));

 3×         _setRoleAdmin(BLOCKED_ROLE, BLOCKLISTER_ROLE);
        }

        /// @inheritdoc ERC20Upgradeable
        function decimals() public pure override returns (uint8) {
 1×         return 6;
        }

        /**
         * @dev Pauses the contract
         */
        function pause() external onlyRole(PAUSER_ROLE) {
10×         _pause();
        }

        /**
         * @dev Unpauses the contract
         */
        function unpause() external onlyRole(UNPAUSER_ROLE) {
 1×         _unpause();
        }

        /// @inheritdoc ERC20BurnableUpgradeable
        function burn(uint256 amount) public override onlyRole(BURNER_ROLE) {
 3×         super.burn(amount);
        }

        /// @inheritdoc ERC20BurnableUpgradeable
        function burnFrom(address account, uint256 amount)
            public
            override
            onlyRole(BURNER_ROLE)
        {
 9×         super.burnFrom(account, amount);
        }

        /**
         * @dev Consumes a received permit and burns tokens based on the permit
         * @param owner Source of the permit and allowance
         * @param spender Target of the permit and allowance
         * @param value How many tokens were permitted to be burned
         * @param deadline Until what timestamp the permit is valid
         * @param v The v portion of the permit signature
         * @param r The r portion of the permit signature
         * @param s The s portion of the permit signature
         */
        function burnFromWithPermit(
            address owner,
            address spender,
            uint256 value,
            uint256 deadline,
            uint8 v,
            bytes32 r,
            bytes32 s
        ) public onlyRole(BURNER_ROLE) {
 4×         super.permit(owner, spender, value, deadline, v, r, s);
 4×         super.burnFrom(owner, value);
        }

        /**
         * @dev Mints tokens to the given account
         * @param account The account to mint tokens to
         * @param amount How many tokens to mint
         */
        function mint(address account, uint256 amount)
            external
            onlyRole(MINTER_ROLE)
        {
 4×         _mint(account, amount);
        }

        /**
         * @dev Performs a batch of mints
         * @param targets Array of addresses for which to mint
         * @param amounts Array of amounts to mint for the corresponding addresses
         * @param id An external identifier given for the minting set
         * @param checksum A checksum to make sure none of the input data has changed
         */
        function mintSet(
            address[] calldata targets,
            uint256[] calldata amounts,
            uint256 id,
            bytes32 checksum
        ) external onlyRole(MINTER_ROLE) {
23×         require(targets.length == amounts.length, "Unmatching mint lengths");
22×         require(targets.length > 0, "Nothing to mint");

21×         bytes32 calculated = keccak256(abi.encode(targets, amounts, id));
21×         require(calculated == checksum, "Checksum mismatch");

11×         for (uint256 i = 0; i < targets.length; i++) {
  16×             require(amounts[i] > 0, "Mint amount not greater than 0");
  15×             _mint(targets[i], amounts[i]);
              }
   7×         emit MintingSetCompleted(id);
          }

          /**
           * @dev Modifier that checks that an account is not blocked. Reverts
           * if the account is blocked
           */
          modifier whenNotBlocked(address account) {
 143×         require(!hasRole(BLOCKED_ROLE, account), "Blocked user");
 132×         _;
          }

          /**
           * @dev Checks that the contract is not paused and that neither sender nor receiver are blocked before transferring tokens. See {ERC.
           * @param from source of the transfer
           * @param to target of the transfer
           * @param amount amount of tokens to be transferred
           */
          function _beforeTokenTransfer(
              address from,
              address to,
              uint256 amount
          ) internal override whenNotPaused whenNotBlocked(from) whenNotBlocked(to) {
  63×         super._beforeTokenTransfer(from, to, amount);
          }

          /**
           * @dev Restricts who can upgrade the contract. Executed when anyone tries to upgrade the contract
           * @param newImplementation Address of the new implementation
           */
          function _authorizeUpgrade(address newImplementation)
              internal
              override
              onlyRole(PROXYOWNER_ROLE)
          {}

          /**
           * @dev Returns the address of the implementation behind the proxy
           */
          function getImplementation() external view returns (address) {
   2×         return _getImplementation();
          }

          /**
           * @dev Allows the rescue of an arbitrary token sent accidentally to the contract
           * @param token Which token we want to rescue
           * @param to Where should the rescued tokens be sent to
           * @param amount How many should be rescued
           */
          function rescueERC20(
              IERC20Upgradeable token,
              address to,
              uint256 amount
          ) external onlyRole(RESCUER_ROLE) {
   5×         token.safeTransfer(to, amount);
          }
      }

```