

Smart Contract Security Audit Report

The SlowMist Security Team received the UUSD team's application for smart contract security audit of the UUSD on 2026.03.09. The following are the details and results of this smart contract security audit:

Token Name :

UUSD

The contract address :

Implementation Address

https://bscscan.com/address/0xa4f44c290cc693fc0c985d679281c61e99d9be9a

Proxy Address

https://bscscan.com/token/0x61a10e8556bed032ea176330e7f17d6a12a10000

The audit items and results :

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

NO.	Audit Items	Result
1	Replay Vulnerability	Passed
2	Denial of Service Vulnerability	Passed
3	Race Conditions Vulnerability	Passed
4	Authority Control Vulnerability Audit	Passed
5	Integer Overflow and Underflow Vulnerability	Passed
6	Gas Optimization Audit	Passed
7	Design Logic Audit	Passed
8	Uninitialized Storage Pointers Vulnerability	Passed
9	Arithmetic Accuracy Deviation Vulnerability	Passed
10	"False top-up" Vulnerability	Passed
11	Malicious Event Log Audit	Passed
12	Scoping and Declarations Audit	Passed

NO.	Audit Items	Result
13	Safety Design Audit	Passed
14	Non-privacy/Non-dark Coin Audit	Passed

Audit Result : Passed

Audit Number : 0X002603100001

Audit Date : 2026.03.09 - 2026.03.10

Audit Team : SlowMist Security Team

Summary conclusion : This is a token contract that contains the Upgradeable sections. The total amount of contract tokens can be changed, owner and minter role can burn their owner tokens through the burn and minterBurn functions. SafeMath security module is used, which is a recommended approach. The contract does not have the Overflow and the Race Conditions issue.

During the audit, we found the following information:

1. The owner role can mint tokens arbitrarily through the mintTo function, and there is no upper limit for the mint amount.
2. The owner role can add/ remove the minter role, the minter role can mint tokens arbitrarily through the mintTo function, and there is no upper limit for the mint amount.
3. The owner role can freeze/ unfreeze any account. The frozen account can not transfer or recieve token. After verifying the deployed contract, the owner role was set to a 3 out of 5 multisig contract 0x009c5467667dcD2Ab8C310E84B105cC0b570a8F4.

The source code:

```
// SPDX-License-Identifier: MIT
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.8.28;

import "@openzeppelin/contracts-
upgradeable/token/ERC20/extensions/ERC20PermitUpgradeable.sol";
import "@openzeppelin/contracts-upgradeable/access/Ownable2StepUpgradeable.sol";
import "@openzeppelin/contracts-upgradeable/utils/PausableUpgradeable.sol";
import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";
```

```
/**
 * @title UUSD - Unity USD
 * @notice AI Economy Base Settlement Layer Stablecoin
 * @dev Upgradeable ERC20 with EIP-2612, EIP-3009, freeze, pause, and multi-role
support
 *
 * Key behaviors:
 * - Pause: Halts transfers, mints, burns, and approvals
 *   (cancelAuthorization remains available for user protection)
 * - Freeze: Blocks specific accounts from:
 *   - Sending or receiving tokens (as from/to)
 *   - Approving or being approved (as owner/spender)
 *   - Spending existing allowances (as spender in transferFrom)
 *   - Minting or burning (as minter)
 * - EIP-3009 valid window: (validAfter, validBefore) - open interval per spec
 *
 * @custom:website https://uusd.ai
 */
contract UUSD is ERC20PermitUpgradeable, Ownable2StepUpgradeable, PausableUpgradeable
{
    using ECDSA for bytes32;

    // ============ Constants ============

    uint256 public constant MAX_BATCH_SIZE = 200;

    // EIP-3009 type hashes
    bytes32 public constant TRANSFER_WITH_AUTHORIZATION_TYPEHASH = keccak256(
        "TransferWithAuthorization(address from,address to,uint256 value,uint256
validAfter,uint256 validBefore,bytes32 nonce)"
    );
    bytes32 public constant RECEIVE_WITH_AUTHORIZATION_TYPEHASH = keccak256(
        "ReceiveWithAuthorization(address from,address to,uint256 value,uint256
validAfter,uint256 validBefore,bytes32 nonce)"
    );
    bytes32 public constant CANCEL_AUTHORIZATION_TYPEHASH = keccak256(
        "CancelAuthorization(address authorizer,bytes32 nonce)"
    );

    // ============ State Variables ============

    /// @notice Admin address for freeze/pause operations
    address public admin;

    /// @notice Mapping of frozen accounts
    mapping(address => bool) public frozen;

    /// @notice Mapping of minter addresses
    mapping(address => bool) public isMinter;
```

```
/// @notice Mapping of minter allowances
mapping(address => uint256) public minterAllowance;

/// @notice EIP-3009: Mapping of used authorization nonces
mapping(address => mapping(bytes32 => bool)) public authorizationState;

/// @dev Reserved storage space for future upgrades
uint256[50] private __gap;

// ============ Events ============

event Mint(address indexed minter, address indexed to, uint256 amount);
event Burn(address indexed burner, uint256 amount);
event Freeze(address indexed caller, address indexed account);
event Unfreeze(address indexed caller, address indexed account);
event AdminSet(address indexed previousAdmin, address indexed newAdmin);
event AdminRemoved(address indexed admin);
event MinterAdded(address indexed minter);
event MinterRemoved(address indexed minter);
event MinterAllowanceSet(address indexed minter, uint256 allowance);
event AuthorizationUsed(address indexed authorizer, bytes32 indexed nonce);
event AuthorizationCanceled(address indexed authorizer, bytes32 indexed nonce);

// ============ Errors ============

error AccountFrozen(address account);
error NotAdminOrOwner();
error NotMinter();
error MinterAllowanceExceeded(uint256 requested, uint256 available);
error BatchSizeExceeded(uint256 size, uint256 max);
error BatchLengthMismatch();
error ZeroAddress();
error AuthorizationExpired();
error AuthorizationNotYetValid();
error AuthorizationAlreadyUsed();
error CallerMustBePayee();
error InvalidSignature();
error RenounceOwnershipDisabled();

// ============ Modifiers ============

modifier notFrozen(address account) {
    if (frozen[account]) revert AccountFrozen(account);
    _;
}

modifier onlyAdminOrOwner() {
    if (msg.sender != admin && msg.sender != owner()) revert NotAdminOrOwner();
```

```
        _;
    }

    modifier onlyMinter() {
        if (!isMinter[msg.sender]) revert NotMinter();
        _;
    }

    // ============ Initializer ============

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

    /**
     * @notice Initializes the UUSD token
     * @param _name Token name
     * @param _symbol Token symbol
     * @param _initialOwner Initial owner address
     */
    function initialize(string memory _name, string memory _symbol, address
_initialOwner) public initializer {
        if (_initialOwner == address(0)) revert ZeroAddress();
        __ERC20_init(_name, _symbol);
        __ERC20Permit_init(_name);
        __Ownable_init(_initialOwner);
        __Ownable2Step_init();
        __Pausable_init();
    }

    // ============ Owner Functions ============

    /**
     * @notice Mint tokens to owner's address (Owner only, no limit)
     * @param amount Amount to mint
     */
    //SlowMist// The owner role can mint tokens arbitrarily through the mint function,
and there is no upper limit for the mint amount
    function mint(uint256 amount) external onlyOwner returns (bool) {
        _mint(msg.sender, amount);
        emit Mint(msg.sender, msg.sender, amount);
        return true;
    }

    /**
     * @notice Mint tokens to a specific address (Owner only, no limit)
     * @param to Recipient address
     * @param amount Amount to mint
```

```
     */
    //SlowMist// The owner role can mint tokens arbitrarily through the mintTo function, and there is no upper limit for the mint amount
    function mintTo(address to, uint256 amount) external onlyOwner notFrozen(to) returns (bool) {
        if (to == address(0)) revert ZeroAddress();
        _mint(to, amount);
        emit Mint(msg.sender, to, amount);
        return true;
    }

    /**
     * @notice Burn tokens from owner's balance
     * @param amount Amount to burn
     */
    function burn(uint256 amount) external onlyOwner returns (bool) {
        _burn(msg.sender, amount);
        emit Burn(msg.sender, amount);
        return true;
    }

    /**
     * @notice Set admin address
     * @param newAdmin New admin address (can be zero to remove)
     */
    function setAdmin(address newAdmin) external onlyOwner {
        address previousAdmin = admin;
        admin = newAdmin;
        if (newAdmin == address(0)) {
            emit AdminRemoved(previousAdmin);
        } else {
            emit AdminSet(previousAdmin, newAdmin);
        }
    }

    /**
     * @notice Add a minter
     * @param minter Address to add as minter
     */
    function addMinter(address minter) external onlyOwner {
        if (minter == address(0)) revert ZeroAddress();
        isMinter[minter] = true;
        emit MinterAdded(minter);
    }

    /**
     * @notice Remove a minter
     * @param minter Address to remove
     */
```

```
    function removeMinter(address minter) external onlyOwner {
        if (minter == address(0)) revert ZeroAddress();
        isMinter[minter] = false;
        minterAllowance[minter] = 0;
        emit MinterRemoved(minter);
    }

    /**
     * @notice Set minter allowance
     * @param minter Minter address
     * @param amount Allowance amount
     */
    function setMinterAllowance(address minter, uint256 amount) external onlyOwner {
        if (minter == address(0)) revert ZeroAddress();
        if (!isMinter[minter]) revert NotMinter();
        minterAllowance[minter] = amount;
        emit MinterAllowanceSet(minter, amount);
    }

    /**
     * @notice Disable renounceOwnership to prevent accidental lockout
     */
    function renounceOwnership() public view override onlyOwner {
        revert RenounceOwnershipDisabled();
    }

    // ============ Admin/Owner Functions ============

    /**
     * @notice Freeze an account
     * @param account Address to freeze
     */
    function freeze(address account) external onlyAdminOrOwner {
        frozen[account] = true;
        emit Freeze(msg.sender, account);
    }

    /**
     * @notice Unfreeze an account
     * @param account Address to unfreeze
     */
    function unfreeze(address account) external onlyAdminOrOwner {
        delete frozen[account];
        emit Unfreeze(msg.sender, account);
    }

    /**
     * @notice Pause all token operations (transfers, mints, burns, approvals)
     * @dev This is an emergency stop mechanism that halts all token movement
```

```
     */
    function pause() external onlyAdminOrOwner {
        _pause();
    }

    /**
     * @notice Unpause all token operations
     */
    function unpause() external onlyAdminOrOwner {
        _unpause();
    }

    // ============ Minter Functions ============

    /**
     * @notice Mint tokens (Minter with allowance limit)
     * @param to Recipient address
     * @param amount Amount to mint
     */
    //SlowMist// The minter role can mint tokens through the minterMint function, but the amount minted is limited by the minterAllowance set by the owner
    function minterMint(address to, uint256 amount) external onlyMinter notFrozen(msg.sender) notFrozen(to) returns (bool) {
        if (to == address(0)) revert ZeroAddress();
        uint256 available = minterAllowance[msg.sender];
        if (available < amount) {
            revert MinterAllowanceExceeded(amount, available);
        }
        minterAllowance[msg.sender] = available - amount;
        _mint(to, amount);
        emit Mint(msg.sender, to, amount);
        return true;
    }

    /**
     * @notice Burn tokens from minter's own balance
     * @param amount Amount to burn
     */
    function minterBurn(uint256 amount) external onlyMinter notFrozen(msg.sender) returns (bool) {
        _burn(msg.sender, amount);
        emit Burn(msg.sender, amount);
        return true;
    }

    // ============ Batch Transfer ============

    /**
     * @notice Batch transfer to multiple recipients
```

```
     * @param recipients Array of recipient addresses
     * @param amounts Array of amounts
     */
    function batchTransfer(
        address[] calldata recipients,
        uint256[] calldata amounts
    ) external whenNotPaused notFrozen(msg.sender) returns (bool) {
        uint256 length = recipients.length;
        if (length != amounts.length) revert BatchLengthMismatch();
        if (length > MAX_BATCH_SIZE) revert BatchSizeExceeded(length, MAX_BATCH_SIZE);

        for (uint256 i = 0; i < length; ) {
            // frozen check for recipient is done in _update
            _transfer(msg.sender, recipients[i], amounts[i]);
            unchecked { ++i; }
        }
        return true;
    }

    // ============ EIP-3009 Functions ============

    /**
     * @notice Execute a transfer with a signed authorization
     * @param from Payer's address
     * @param to Payee's address
     * @param value Amount to transfer
     * @param validAfter Timestamp after which the authorization is valid
     * @param validBefore Timestamp before which the authorization is valid
     * @param nonce Unique nonce
     * @param v Signature v
     * @param r Signature r
     * @param s Signature s
     */
    function transferWithAuthorization(
        address from,
        address to,
        uint256 value,
        uint256 validAfter,
        uint256 validBefore,
        bytes32 nonce,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external whenNotPaused notFrozen(from) notFrozen(to) {
        _requireValidAuthorization(from, validAfter, validBefore, nonce);

        bytes32 structHash = keccak256(
            abi.encode(
                TRANSFER_WITH_AUTHORIZATION_TYPEHASH,
```

```
                from,
                to,
                value,
                validAfter,
                validBefore,
                nonce
            )
        );

        _validateSignature(from, structHash, v, r, s);
        _markAuthorizationUsed(from, nonce);
        _transfer(from, to, value);
    }

    /**
     * @notice Receive a transfer with a signed authorization from payer
     * @dev This allows the payee to trigger the transfer
     * @param from Payer's address
     * @param to Payee's address (must be msg.sender)
     * @param value Amount to transfer
     * @param validAfter Timestamp after which the authorization is valid
     * @param validBefore Timestamp before which the authorization is valid
     * @param nonce Unique nonce
     * @param v Signature v
     * @param r Signature r
     * @param s Signature s
     */
    function receiveWithAuthorization(
        address from,
        address to,
        uint256 value,
        uint256 validAfter,
        uint256 validBefore,
        bytes32 nonce,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external whenNotPaused notFrozen(from) notFrozen(to) {
        if (to != msg.sender) revert CallerMustBePayee();
        _requireValidAuthorization(from, validAfter, validBefore, nonce);

        bytes32 structHash = keccak256(
            abi.encode(
                RECEIVE_WITH_AUTHORIZATION_TYPEHASH,
                from,
                to,
                value,
                validAfter,
                validBefore,
```

```
                nonce
            )
        );

        _validateSignature(from, structHash, v, r, s);
        _markAuthorizationUsed(from, nonce);
        _transfer(from, to, value);
    }

    /**
     * @notice Cancel an authorization
     * @param authorizer Authorizer's address
     * @param nonce Nonce to cancel
     * @param v Signature v
     * @param r Signature r
     * @param s Signature s
     */
    function cancelAuthorization(
        address authorizer,
        bytes32 nonce,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external {
        if (authorizationState[authorizer][nonce]) revert AuthorizationAlreadyUsed();

        bytes32 structHash = keccak256(
            abi.encode(CANCEL_AUTHORIZATION_TYPEHASH, authorizer, nonce)
        );

        _validateSignature(authorizer, structHash, v, r, s);
        authorizationState[authorizer][nonce] = true;
        emit AuthorizationCanceled(authorizer, nonce);
    }

    // ============ Internal Functions ============

    function _requireValidAuthorization(
        address authorizer,
        uint256 validAfter,
        uint256 validBefore,
        bytes32 nonce
    ) internal view {
        // EIP-3009: valid window is (validAfter, validBefore) - open interval
        if (block.timestamp <= validAfter) revert AuthorizationNotYetValid();
        if (block.timestamp >= validBefore) revert AuthorizationExpired();
        if (authorizationState[authorizer][nonce]) revert AuthorizationAlreadyUsed();
    }
```

```
    function _validateSignature(
        address signer,
        bytes32 structHash,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) internal view {
        bytes32 digest = _hashTypedDataV4(structHash);
        address recovered = ECDSA.recover(digest, v, r, s);
        if (recovered != signer) revert InvalidSignature();
    }

    function _markAuthorizationUsed(address authorizer, bytes32 nonce) internal {
        authorizationState[authorizer][nonce] = true;
        emit AuthorizationUsed(authorizer, nonce);
    }

    /**
     * @dev Override transferFrom to add freeze check for spender
     * This prevents frozen accounts from spending their existing allowances
     */
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public override notFrozen(msg.sender) returns (bool) {
        return super.transferFrom(from, to, amount);
    }

    /**
     * @dev Override _update to add pause and freeze checks
     */
    function _update(
        address from,
        address to,
        uint256 amount
    ) internal override whenNotPaused {
        // Skip freeze check for mint (from == 0) and burn (to == 0)
        if (from != address(0) && frozen[from]) revert AccountFrozen(from);
        //SlowMist// This kind of check is very good, avoiding user mistakes leading
to the loss of tokens during the transfer
        if (to != address(0) && frozen[to]) revert AccountFrozen(to);
        super._update(from, to, amount);
    }

    /**
     * @dev Override _approve to add pause and freeze checks
     */
    function _approve(
```

```
        address owner_,
        address spender,
        uint256 amount,
        bool emitEvent
    ) internal override whenNotPaused notFrozen(owner_) notFrozen(spender) {
        super._approve(owner_, spender, amount, emitEvent);
    }
}
```

Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.



Official Website

www.slowmist.com



E-mail

team@slowmist.com



Twitter

@SlowMist_Team



Github

https://github.com/slowmist