FxToken ERC20 Smart Contract Audit

Derive, 5 August 2025

hello@iosiro.com

Contents

1. Introduction 1
2. Disclaimer 2
3. Methodology 3
4. Audit findings 4
   - IO-DRV-FXT-001 Block manager should not be able to block the zero address . . . 5
   - IO-DRV-FXT-002 Incorrect initilization . . . 6
   - IO-DRV-FXT-003 Burning tokens can be trust minimized . . . 7
   - IO-DRV-FXT-004 Redundant block check for mint recipient . . . 8
   - IO-DRV-FXT-005 Minting from a blocked address is permitted . . . 9
   - IO-DRV-FXT-006 Blocked addresses can still spend an allowance . . . 10
5. Specification 11
6. Test coverage report 13
7. Initial audit code 14
8. Final audit code 18

1. Introduction

iosiro was commissioned by Derive to perform a smart contract audit of the FxToken ERC20 Smart Contract. One auditor conducted the audit and review between 2025-07-29 and 2025-08-05, using half an audit day.

Overview

The audit identified two low-risk and four informational issues, all of which were related to design or best practice considerations. The informational issues raised in this report are considerations that should be discussed to ensure the deliberate implementation and intended functionality are in line with the system specification. It should be noted that there is a centralization risk when interacting with this contract, as users with the `MINTER` role can mint and burn tokens at any point. No critical or high-risk vulnerabilities were found. The contract's core logic and access controls were generally well-implemented; however, certain considerations and concerns were raised to improve adherence to best practices.

All identified vulnerabilities and issues have been successfully resolved as of the date of this report, demonstrating a strong commitment to security and proactive risk management.

	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Resolved	0	0	0	2	3
Closed	0	0	0	0	1

Scope

The assessment focused on the source file listed below, with all other files considered out of scope. Any out-of-scope code interacting with the assessed code was presumed to operate correctly without introducing functional or security vulnerabilities.

- **Project name:** FxToken ERC20 Smart Contract Audit
- **Files:** FxToken.sol
- **MD5 hash at audit start:** 2b014686156cd95f300c490e9a2bc32b
- **MD5 hash at audit completion:** 050670b80ed6b7ec40d9d997ca3576a9

A specification is available in the Specification section of this report. Full code listings of both the initial version of the audited contract and the final revision are provided in Section 7 and Section 8 of this report. The final audit code can also be found at the following GitHub link.

2. Disclaimer

This report aims to provide an overview of the assessed smart contracts' risk exposure and a guide to improving their security posture by addressing identified issues. The audit, limited to specific source code at the time of review, sought to:

- Identify potential security flaws.
- Verify that the smart contracts' functionality aligns with their documentation.

Off-chain components, such as backend web application code, keeper functionality, and deployment scripts were explicitly not in-scope of this audit.

Given the unregulated nature and ease of cryptocurrency transfers, operations involving these assets face a high risk from cyber attacks. Maintaining the highest security level is crucial, necessitating a proactive and adaptive approach that accounts for the experimental and rapidly evolving nature of blockchain technology. To encourage secure code development, developers should:

- Integrate security throughout the development lifecycle.
- Employ defensive programming to mitigate the risks posed by unexpected events.
- Adhere to current best practices wherever possible.

3. Methodology

The audit was conducted using the techniques described below.

Code review	The source code was manually inspected to identify potential security flaws. Code review is a useful approach for detecting security flaws, discrepancies between the specification and implementation, design improvements, and high-risk areas of the system.
Dynamic analysis	The contracts were compiled, deployed, and tested in a test environment, both manually and through the test suite provided. Dynamic analysis was used to identify additional edge cases, confirm that the code was functional, and to validate the reported issues.
Automated analysis	Automated tooling was used to detect the presence of various types of security vulnerabilities. Static analysis results were reviewed manually and any false positives were removed. Any true positive results are included in this report.

4. Audit findings

The table below provides an overview of the audit's findings. Detailed write-ups are provided below.

ID	Issue	Risk	Status
IO-DRV-FXT-001	Block manager should not be able to block the zero address	Low	Resolved
IO-DRV-FXT-002	Incorrect initilization	Low	Resolved
IO-DRV-FXT-003	Burning tokens can be trust minimized	Informational	Closed
IO-DRV-FXT-004	Redundant block check for mint recipient	Informational	Resolved
IO-DRV-FXT-005	Minting from a blocked address is permitted	Informational	Resolved
IO-DRV-FXT-006	Blocked addresses can still spend an allowance	Informational	Resolved

Each issue identified during the audit has been assigned a risk rating. The rating is determined based on the criteria outlined below.

Critical risk The issue could result in the theft of funds from the contract or its users.

High risk The issue could result in the loss of funds for the contract owner or its users.

Medium risk The issue resulted in the code being dysfunctional or the specification being implemented incorrectly.

Low risk A design or best practice issue that could affect the ordinary functioning of the contract.

Informational An improvement related to best practice or a suboptimal design pattern.

In addition to a risk rating, each issue is assigned a status:

Open The issue remained present in the code as of the final commit reviewed and may still pose a risk.

Resolved The issue was identified during the audit and has since been satisfactorily addressed, removing the risk it posed.

Closed The issue was identified during the audit and acknowledged by the developers as an acceptable risk without actioning any change.

IO-DRV-FXT-001

Block manager should not be able to block the zero address

Low	Resolved	FxToken.sol#50

It is possible for the block manager to block `address(0)`. Blocking this address would break burning and minting functionality.

Recommendation

The `setBlocked` function should implement a check that ensures the address being blocked is not the zero address.

Client response

Fixed in the final code.

IO-DRV-FXT-002

Incorrect initilization

Low	Resolved	FxToken.sol#36

Arguments are passed to the function `__ERC20_init_unchained` inside of the FxToken's `initialize` function in the incorrect order, resulting in the token's name and symbol having the incorrect values.

Recommendation

The correct order to initialize the token contract is `(_name, _symbol)`, this should be changed in the source code.

Client response

Fixed in the final code.

IO-DRV-FXT-003

Burning tokens can be trust minimized

Informational	Closed	FxToken.sol#72

The token contract allows for any user's tokens to be burned without requiring an allowance from the minter; this does not follow best practices and increases the trust given to the minter to an unnecessary degree.

Recommendation

As this level of trust does not follow best practices, it is advised that the necessity of this functionality is carefully considered. If burning is needed in some capacity, then it is recommended that minters are only given allowance to burn blocked users. However, this would require the ability to unblock them beforehand, as blocked users cannot have their tokens burned, alternatively, the update functionlity could allow for the burn of blocked users' funds.

Client response

It should be noted that this issue was marked as closed, as after discussions with the Derive team, it was determined that this is intentional behavior. This feature is meant to allow the redemption of the underlying collateral owned by the custodian.

IO-DRV-FXT-004

Redundant block check for mint recipient

Informational | Resolved | FxToken.sol#67

When minting new tokens, the `mint` function checks whether the token receiver is currently blocked. This check is redundant as a check for the same condition is included in the subsequently called `_update` function.

Recommendation

The check in the `mint` function is unnecessary and should be removed.

Client response

Fixed in the final code.

IO-DRV-FXT-005

Minting from a blocked address is permitted

Informational | Resolved | FxToken.sol#65

A block manager can block a minter, but this will not prevent the minter from minting tokens to other addresses.

Recommendation

Consider explicitly preventing blocked minters from minting, unless this is intended functionality.

Client response

Fixed in the final code.

IO-DRV-FXT-006

Blocked addresses can still spend an allowance

Informational | Resolved | FxToken.sol#80

A blocked address cannot send or receive tokens; however, it can still spend any allowance it might have to transfer the tokens of other addresses. Currently, the block functionality only prevents the movement of blocked funds.

Recommendation

If the intention was to censor blocked addresses completely, the allowance mechanism should be modified to prevent blocked addresses from spending allowances.

Client response

Fixed in the final code.

5. Specification

The following section outlines the system's intended functionality at a high level, based on its implementation in the codebase. Any perceived points of conflict should be highlighted with the auditing team to determine the source of the discrepancy.

- The FxToken contract will be deployed behind a proxy such as a Transparent Proxy.
- This context was kept in mind and assumed for this audit; however, as the proxy implementation logic was not provided, it will not be in scope for tests or auditing.
- The implementation contract inherits the `Initializable` contract, which is utilized to be deployed behind a proxy.
- The implementation contract cannot be initialized directly.
- The proxy contract will be initialized by a proxy delegate call to the implementation contract.

Access control

- The token contract inherits the `AccessControlUpgradeable` contract. This means the contract has the ability to create and manage user roles.
- The deployer who initializes the contract is the default admin for all roles.
- The contract has two custom-defined roles: `BLOCK_MANAGER` and `MINTER`.
- The minter role is the only role that can mint tokens. The minter role can burn any user's token.
- The contract has implemented a custom blocking mechanism on top of this.
- The blocking mechanism stops blocked users from being able to transfer or receive tokens.
- The blocking mechanism is able to stop users with the minter role from minting to blocked users; however, they can still burn blocked users.
- A block manager can block themselves, and they can still block others while blocked
- A user can be unblocked.
- A block manager can block and unblock themselves.
- Only the block manager role can manage the block list.

Minting, burning, and transfers

- The tokens can only be minted or burned by someone with the `MINTER` role.
- Anyone can transfer tokens as long as they are not blocked and have an allowance.

- There is no built-in supply cap. Tokens can be minted and burned anytime by the `MINTER` role.
- The token contract allows for any user's tokens to be burned without requiring an allowance from the minter.
- Blocked users cannot transfer tokens nor have tokens transferred to them.
- Blocked users cannot use an allowance that was granted to them.
- Minters can mint tokens to themselves.
- Minters can be blocked by the block manager, and they cannot mint tokens to other users or burn other users' tokens while blocked.

6. Test coverage report

The coverage report of the provided tests as of the final day of the audit is given below.

File	% Lines	% Statements	% Branches	% Funcs
src/FxToken.sol	94.74% (36/38)	94.12% (32/34)	84.62% (11/13)	100.00% (10/10)
Total	94.74% (36/38)	94.12% (32/34)	84.62% (11/13)	100.00% (10/10)

7. Initial audit code

FxToken.sol

```
pragma solidity ^0.8.27;

import {AccessControlUpgradeable} from "openzeppelin-upgradeable/access/AccessControlUpgradeable.sol";
import {ERC20Upgradeable, Initializable} from "openzeppelin-upgradeable/token/ERC20/ERC20Upgradeable.sol";



contract FxToken is Initializable, ERC20Upgradeable, AccessControlUpgradeable {
    bytes32 public constant MINTER_ROLE = keccak256("MINTER");
    bytes32 public constant BLOCK_MANAGER_ROLE = keccak256("BLOCK_MANAGER");
    // keccak256(abi.encode(uint256(keccak256("FxToken")) - 1)) & ~bytes32(uint256(0xff))
    bytes32 private constant FxTokenStorageLocation = 0xfb8997de7bd810675586dece12917931ae29ba246c9d4d120b17fca6e2b68f00;



    /// @custom:storage-location erc7201:FxToken
    struct FxTokenStorage {
        uint8 decimals;
        mapping(address user => bool blocked) isBlocked;
    }

    function _getStorage() internal pure returns (FxTokenStorage storage s) {
        bytes32 position = FxTokenStorageLocation;
        assembly {
            s.slot := position
        }
    }
```

```
    ////////////
    // Setup //
    ////////////

    constructor() {
        _disableInitializers();
    }

    function initialize(string memory _symbol, string memory _name,
uint _decimals) external initializer {
        __ERC20_init_unchained(_symbol, _name);
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);

        FxTokenStorage storage s = _getStorage();
        s.decimals = uint8(_decimals);
    }

    ////////////////
    // Block List //
    ////////////////

    function setBlocked(address user, bool blocked) public
onlyRole(BLOCK_MANAGER_ROLE) {
        FxTokenStorage storage s = _getStorage();
        s.isBlocked[user] = blocked;
        emit Blocked(user, blocked);
    }

    function isBlocked(address user) public view returns (bool) {
        return _getStorage().isBlocked[user];
    }

    ///////////////
    // Mint/Burn //
    ///////////////
```

```
    function mint(address to, uint256 amount) public
onlyRole(MINTER_ROLE) {
        FxTokenStorage storage s = _getStorage();
        require(!s.isBlocked[to], "FxToken: recipient is blocked");
        _mint(to, amount);
    }

    function burn(address from, uint256 amount) public
onlyRole(MINTER_ROLE) {
        _burn(from, amount);
    }

    ////////////////////
    // ERC20 Overrides //
    ////////////////////

    function _update(address from, address to, uint256 value) internal
override {
        FxTokenStorage storage s = _getStorage();
        if (s.isBlocked[from]) {
            revert ERC20InvalidSender(from);
        }
        if (s.isBlocked[to]) {
            revert ERC20InvalidReceiver(to);
        }
        super._update(from, to, value);
    }

    function decimals() public view virtual override returns (uint8) {
        return _getStorage().decimals;
    }

    ////////////
    // Events //
    ////////////
```

```
    event Blocked(address indexed user, bool blocked);
}
```

8. Final audit code

FxToken.sol (with resolved changes)

```
pragma solidity ^0.8.27;

import {AccessControlUpgradeable} from "@openzeppelin-upgradeable/access/AccessControlUpgradeable.sol";
import {ERC20Upgradeable, Initializable} from "@openzeppelin-upgradeable/token/ERC20/ERC20Upgradeable.sol";

contract FxToken is Initializable, ERC20Upgradeable, AccessControlUpgradeable {
    bytes32 public constant MINTER_ROLE = keccak256("MINTER");
    bytes32 public constant BLOCK_MANAGER_ROLE = keccak256("BLOCK_MANAGER");
    // keccak256(abi.encode(uint256(keccak256("FxToken")) - 1)) & ~bytes32(uint256(0xff))
    bytes32 private constant FxTokenStorageLocation = 0xfb8997de7bd810675586dece12917931ae29ba246c9d4d120b17fca6e2b68f00;


    /// @custom:storage-location erc7201:FxToken
    struct FxTokenStorage {
        uint8 decimals;
        mapping(address user => bool blocked) isBlocked;
    }

    function _getStorage() internal pure returns (FxTokenStorage storage s) {
        bytes32 position = FxTokenStorageLocation;
        assembly {
            s.slot := position
        }
    }

    //////////
```

```
    // Setup //
    ///////////

    constructor() {
        _disableInitializers();
    }

    function initialize(string memory _name, string memory _symbol,
uint _decimals) external initializer {
        __ERC20_init_unchained(_name, _symbol);
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);

        FxTokenStorage storage s = _getStorage();
        s.decimals = uint8(_decimals);
    }

    ///////////////
    // Block List //
    ///////////////

    function setBlocked(address user, bool blocked) public
onlyRole(BLOCK_MANAGER_ROLE) {
        require(user != address(0), "FxToken: cannot block zero
address");
        FxTokenStorage storage s = _getStorage();
        s.isBlocked[user] = blocked;
        emit Blocked(user, blocked);
    }

    function isBlocked(address user) public view returns (bool) {
        return _getStorage().isBlocked[user];
    }

    //////////////
    // Mint/Burn //
    //////////////
```

```
    function mint(address to, uint256 amount) public
onlyRole(MINTER_ROLE) {
        FxTokenStorage storage s = _getStorage();
        require(!s.isBlocked[msg.sender], "FxToken: minter is
blocked");
        _mint(to, amount);
    }

    function burn(address from, uint256 amount) public
onlyRole(MINTER_ROLE) {
        FxTokenStorage storage s = _getStorage();
        require(!s.isBlocked[msg.sender], "FxToken: minter is
blocked");

        if (from == address(0)) {
            revert ERC20InvalidSender(address(0));
        }
        // Skip the _update call to avoid checking blocked status
        super._update(from, address(0), amount);
    }

    ////////////////////
    // ERC20 Overrides //
    ////////////////////

    function _update(address from, address to, uint256 value) internal
override {
        FxTokenStorage storage s = _getStorage();
        require(!s.isBlocked[from], "FxToken: sender is blocked");
        require(!s.isBlocked[to], "FxToken: recipient is blocked");
        super._update(from, to, value);
    }

    function _spendAllowance(address owner, address spender, uint256
value) internal override {
```

```
        FxTokenStorage storage s = _getStorage();
        require(!s.isBlocked[spender], "FxToken: spender is blocked");
        super._spendAllowance(owner, spender, value);
    }

    function decimals() public view virtual override returns (uint8) {
        return _getStorage().decimals;
    }

    ////////////
    // Events //
    ////////////

    event Blocked(address indexed user, bool blocked);
}
```