Scanned Code Report

AUDITAGENT

Powered by NETHERMIND SECURITY

Code Info

Deep Scan

Scan ID 1	Date July 22, 2025
Organization CMTA	Repository CMTAT
Branch master	Commit Hash 04dad821...f76fad1b

Contracts in scope

contracts/deployment/CMTATStandalone.sol
contracts/deployment/CMTATUpgradeable.sol
contracts/deployment/CMTATUpgradeableUUPS.sol
contracts/deployment/ERC1363/CMTATStandaloneERC1363.sol
contracts/deployment/ERC1363/CMTATUpgradeableERC1363.sol
contracts/deployment/ERC7551/CMTATStandaloneERC7551.sol
contracts/deployment/ERC7551/CMTATUpgradeableERC7551.sol
contracts/deployment/allowlist/CMTATStandaloneAllowlist.sol
contracts/deployment/allowlist/CMTATUpgradeableAllowlist.sol
contracts/deployment/debt/CMTATStandaloneDebt.sol
contracts/deployment/debt/CMTATUpgradeableDebt.sol
contracts/deployment/light/CMTATStandaloneLight.sol
contracts/deployment/light/CMTATUpgradeableLight.sol
contracts/interfaces/engine/IDebtEngine.sol
contracts/interfaces/engine/IDocumentEngine.sol
contracts/interfaces/engine/IRuleEngine.sol
contracts/interfaces/engine/ISnapshotEngine.sol
contracts/interfaces/modules/IAllowlistModule.sol
contracts/interfaces/modules/IDebtModule.sol
contracts/interfaces/modules/IDocumentEngineModule.sol
contracts/interfaces/modules/ISnapshotEngineModule.sol
contracts/interfaces/technical/ICMTATConstructor.sol
contracts/interfaces/technical/IERC20Allowance.sol
contracts/interfaces/technical/IERC7802.sol
contracts/interfaces/technical/IMintBurnToken.sol
contracts/interfaces/tokenization/ICMTAT.sol
contracts/interfaces/tokenization/IERC3643Partial.sol
contracts/interfaces/tokenization/draft-IERC1404.sol
contracts/interfaces/tokenization/draft-IERC1643.sol
contracts/interfaces/tokenization/draft-IERC1643CMTAT.sol
contracts/interfaces/tokenization/draft-IERC7551.sol
contracts/libraries/Errors.sol
contracts/mocks/DebtEngineMock.sol
contracts/mocks/DocumentEngineMock.sol
contracts/mocks/ERC1363ReceiverMock.sol
contracts/mocks/ERC721MockUpgradeable.sol
contracts/mocks/MinimalForwarderMock.sol
contracts/mocks/RuleEngine/CodeList.sol
contracts/mocks/RuleEngine/RuleEngineMock.sol
contracts/mocks/RuleEngine/RuleMock.sol
contracts/mocks/RuleEngine/RuleMockMint.sol
contracts/mocks/RuleEngine/interfaces/IRule.sol
contracts/mocks/RuleEngine/interfaces/IRuleEngineMock.sol
contracts/mocks/SnapshotEngineMock.sol
contracts/mocks/library/snapshot/ICMTATSnapshot.sol

contracts/mocks/library/snapshot/SnapshotErrors.sol

contracts/mocks/library/snapshot/SnapshotModuleBase.sol

contracts/mocks/test/proxy/CMTAT_PROXY_TEST.sol

contracts/mocks/test/proxy/CMTAT_PROXY_TEST_UUPS.sol

contracts/modules/0_CMTATBaseCommon.sol contracts/modules/0_CMTATBaseCore.sol

contracts/modules/0_CMTATBaseGeneric.sol contracts/modules/1_CMTATBaseAllowlist.sol

contracts/modules/1_CMTATBaseRuleEngine.sol contracts/modules/2_CMTATBaseDebt.sol

contracts/modules/2_CMTATBaseERC1404.sol contracts/modules/3_CMTATBaseERC20CrossChain.sol

contracts/modules/4_CMTATBaseERC2771.sol contracts/modules/5_CMTATBaseERC1363.sol

contracts/modules/5_CMTATBaseERC7551.sol contracts/modules/internal/AllowlistModuleInternal.sol

contracts/modules/internal/ERC20BurnModuleInternal.sol

contracts/modules/internal/ERC20EnforcementModuleInternal.sol

contracts/modules/internal/ERC20MintModuleInternal.sol

contracts/modules/internal/EnforcementModuleInternal.sol

contracts/modules/internal/ValidationModuleRuleEngineInternal.sol

contracts/modules/internal/common/EnforcementModuleLibrary.sol

contracts/modules/wrapper/controllers/ValidationModule.sol

contracts/modules/wrapper/controllers/ValidationModuleAllowlist.sol

contracts/modules/wrapper/core/BaseModule.sol contracts/modules/wrapper/core/ERC20BaseModule.sol

contracts/modules/wrapper/core/ERC20BurnModule.sol

contracts/modules/wrapper/core/ERC20MintModule.sol

contracts/modules/wrapper/core/EnforcementModule.sol contracts/modules/wrapper/core/PauseModule.sol

contracts/modules/wrapper/core/ValidationModuleCore.sol

contracts/modules/wrapper/extensions/DocumentEngineModule.sol

contracts/modules/wrapper/extensions/ERC20EnforcementModule.sol

contracts/modules/wrapper/extensions/ExtraInformationModule.sol

contracts/modules/wrapper/extensions/SnapshotEngineModule.sol

contracts/modules/wrapper/extensions/ValidationModule/ValidationModuleERC1404.sol

contracts/modules/wrapper/extensions/ValidationModule/ValidationModuleRuleEngine.sol

contracts/modules/wrapper/options/AllowlistModule.sol

contracts/modules/wrapper/options/DebtEngineModule.sol

contracts/modules/wrapper/options/DebtModule.sol

contracts/modules/wrapper/options/ERC2771Module.sol

contracts/modules/wrapper/options/ERC7551Module.sol

contracts/modules/wrapper/security/AccessControlModule.sol

Code Statistics

Findings	Contracts Scanned	Lines of Code
22	88	6837

Findings Summary

Total Findings

- High Risk (2)
- Medium Risk (3)
- Low Risk (5)
- Info (4)
- Best Practices (8)

Code Summary

The CMTAT (Capital Markets Tokenized Asset Toolkit) is a comprehensive smart contract framework designed for tokenizing assets in compliance with Swiss law and other regulatory requirements. It provides a modular and extensible architecture for creating tokenized securities, debt instruments, and other financial assets on the blockchain.

Core Architecture

CMTAT follows a modular design pattern with a layered architecture:

1. **Base Modules**: Foundational components that provide core ERC-20 functionality with additional features.
2. **Extension Modules**: Add specialized capabilities like enforcement, document management, and snapshots.
3. **Deployment Options**: Various deployment configurations including standalone, upgradeable, and proxy implementations.

The framework is highly customizable through its modular approach, allowing issuers to select only the components needed for their specific use case.

Key Features

- **Compliance Controls**: Built-in mechanisms for regulatory compliance including allowlists, transfer restrictions, and freezing capabilities.
- **Document Management**: Support for attaching legal documents and terms to tokens.
- **Enforcement Mechanisms**: Ability to freeze accounts and tokens, and force transfers when legally required.
- **Upgradeability**: Multiple proxy patterns supported for future upgrades.
- **Gasless Transactions**: Support for meta-transactions via ERC-2771.
- **Cross-Chain Compatibility**: ERC-7802 support for cross-chain token transfers.
- **Debt Instrument Support**: Specialized modules for debt securities with credit events tracking.
- **Snapshot Capabilities**: Point-in-time balance recording for dividends and governance.

Standards Support

CMTAT implements or extends several token standards:
- ERC-20: Base token functionality
- ERC-1404: Transfer restriction standard
- ERC-1643: Document management
- ERC-3643: Security token standard
- ERC-7551: Compliance extensions
- ERC-1363: Payable token
- ERC-7802: Cross-chain transfers

Main Entry Points and Actors

Entry Points

1. **Token Management**:
   - `mint(address account, uint256 value, bytes calldata data)`: Creates new tokens.
   - `burn(address account, uint256 value, bytes calldata data)`: Destroys tokens.
   - `batchMint(address[] calldata accounts, uint256[] calldata values)`: Mints tokens to multiple addresses.
   - `batchBurn(address[] calldata accounts, uint256[] calldata values, bytes memory data)`: Burns tokens from multiple addresses.
   - `burnAndMint(address from, address to, uint256 amountToBurn, uint256 amountToMint, bytes calldata d` Burns and mints in one transaction.

2. **Transfer Functions**:
   - `transfer(address to, uint256 value)`: Standard ERC-20 transfer.
   - `transferFrom(address from, address to, uint256 value)`: Transfer with allowance.
   - `batchTransfer(address[] calldata tos, uint256[] calldata values)`: Transfer to multiple recipients.
   - `forcedTransfer(address from, address to, uint256 value, bytes calldata data)`: Admin-forced transfer.

3. **Compliance Controls**:
   - `setAddressFrozen(address account, bool freeze)`: Freezes/unfreezes an address.
   - `freezePartialTokens(address account, uint256 value, bytes memory data)`: Freezes a portion of tokens.
   - `unfreezePartialTokens(address account, uint256 value, bytes memory data)`: Unfreezes tokens.
   - `setAddressAllowlist(address account, bool status)`: Adds/removes address from allowlist.
   - `enableAllowlist(bool status)`: Enables/disables allowlist functionality.

4. **Contract Controls**:
   - `pause()`: Pauses all transfers.
   - `unpause()`: Resumes transfers.
   - `deactivateContract()`: Permanently deactivates the contract.

5. **Document Management**:
   - `setTerms(IERC1643CMTAT.DocumentInfo calldata terms_)`: Sets token terms.
   - `setTokenId(string calldata tokenId_)`: Sets token identifier.
   - `setInformation(string calldata information_)`: Sets token information.

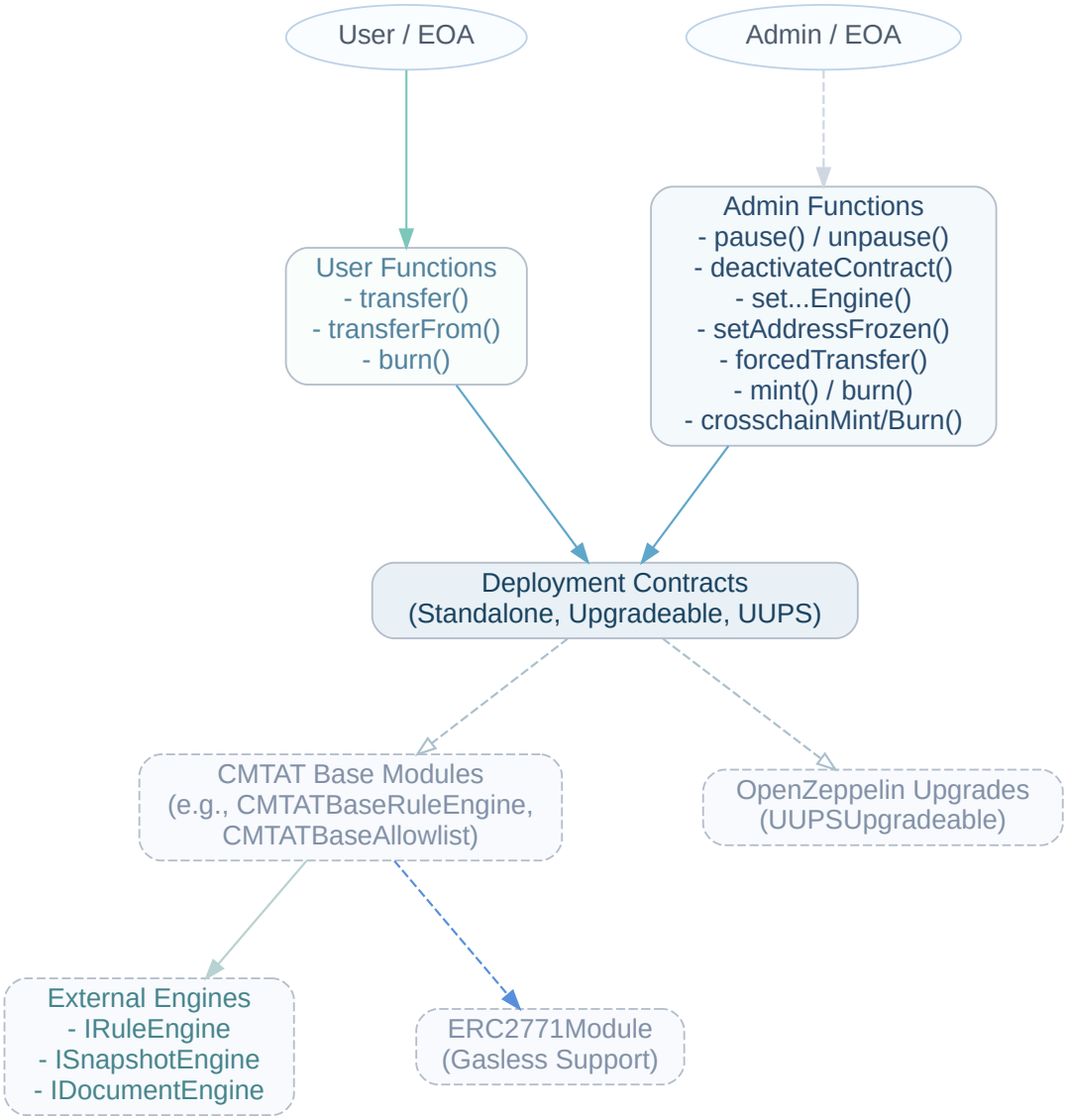
6. **Cross-Chain Operations**:
   - `crosschainMint(address to, uint256 value)`: Mints tokens from cross-chain transfer.
   - `crosschainBurn(address from, uint256 value)`: Burns tokens for cross-chain transfer.

Actors

1. **Admin**: Has the `DEFAULT_ADMIN_ROLE` and can perform all administrative functions.
2. **Minter**: Has the `MINTER_ROLE` and can create new tokens.
3. **Burner**: Has the `BURNER_ROLE` and can destroy tokens.
4. **Pauser**: Has the `PAUSER_ROLE` and can pause/unpause transfers.
5. **Enforcer**: Has the `ENFORCER_ROLE` and can freeze/unfreeze addresses.
6. **ERC20Enforcer**: Has the `ERC20ENFORCER_ROLE` and can freeze/unfreeze specific token amounts.
7. **Document Manager**: Has the `DOCUMENT_ROLE` and can update document references.
8. **Snapshooter**: Has the `SNAPSHOOTER_ROLE` and can manage token snapshots.

9. **Cross-Chain Operator**: Has the `CROSS_CHAIN_ROLE` and can perform cross-chain operations.
10. **Token Holders**: Regular users who can transfer tokens subject to compliance rules.
11. **Allowlist Manager**: Has the `ALLOWLIST_ROLE` and can manage the allowlist.
12. **Debt Manager**: Has the `DEBT_ROLE` and can update debt information.
13. **Proxy Upgrader**: Has the `PROXY_UPGRADE_ROLE` and can upgrade proxy implementations.

Code Diagram



1 of 22 Findings

contracts/deployment/CMTATUpgradeableUUPS.sol

CMTATUpgradeableUUPS contract is not initializable

High Risk

The `initialize` function in `CMTATUpgradeableUUPS` overrides the one from its parent `CMTATBaseRuleEngine` and attempts to call the parent's function directly. Both the child and parent `initialize` functions are decorated with the `initializer` modifier from OpenZeppelin's `Initializable` contract.

When `CMTATUpgradeableUUPS.initialize` is called, its `initializer` modifier will run first, setting the contract's state to 'initialized'. Subsequently, the call to `CMTATBaseRuleEngine.initialize` will execute. This function also has an `initializer` modifier, which will check if the contract is already initialized. Since it is, the modifier will cause the transaction to revert, making it impossible to initialize any contract that inherits from `CMTATUpgradeableUUPS`.

Code snippet from `CMTATUpgradeableUUPS.sol`:

```
    function initialize(  address admin,
        ICMTATConstructor.ERC20Attributes memory ERC20Attributes_,
        ICMTATConstructor.ExtraInformationAttributes memory extraInformationAttributes_,
        ICMTATConstructor.Engine memory engines_ ) public override initializer {
        CMTATBaseRuleEngine.initialize( admin,
            ERC20Attributes_,
            extraInformationAttributes_,
            engines_);
        __UUPSUpgradeable_init_unchained();
    }
```

Code snippet from the parent `CMTATBaseRuleEngine.sol`:

```
    function initialize(
        address admin,
        ICMTATConstructor.ERC20Attributes memory ERC20Attributes_,
        ICMTATConstructor.ExtraInformationAttributes memory extraInformationAttributes_,
        ICMTATConstructor.Engine memory engines_
    ) public virtual initializer {
        __CMTAT_init(
            admin,
            ERC20Attributes_,
            extraInformationAttributes_,
            engines_
        );
    }
```

This nested `initializer` pattern makes the contract undeployable as its initialization will always fail.

2 of 22 Findings

contracts/modules/wrapper/core/ERC20MintModule.sol

batchTransfer bypasses all compliance / enforcement logic

High Risk

The public function `batchTransfer()` implemented in `contracts/modules/wrapper/core/ERC20MintModule.sol` allows a holder that owns the `MINTER_ROLE` to move tokens from its own address to an arbitrary list of recipients without running any of the compliance checks that normally protect a transfer (pause status, address freeze status, allow-list, rule-engine, etc.).

```
function batchTransfer(address[] calldata tos,uint256[] calldata values)
    public override(IERC3643BatchTransfer) onlyRole(MINTER_ROLE) returns (bool success_){
        return _batchTransfer(tos, values);   // <-- no compliance checks
}

// internal helper
function _batchTransfer(...) internal virtual returns (bool){
    ...
    for (uint256 i = 0; i < tos.length; ++i){
        ERC20Upgradeable._transfer(_msgSender(), tos[i], values[i]); // <-- direct call,
skips CMTAT checks
    }
}
```

`_transfer()` is called directly from OpenZeppelin's implementation, therefore **no invocation of `_checkTransferred`, `ValidationModule`, `Allowlist`, or Rule-Engine logic takes place**. A malicious or compromised `MINTER_ROLE` can therefore:

- send tokens to a frozen address;
- send tokens while the contract is `paused();`
- send tokens to an address that is not on the allow-list when the allow-list is enabled;
- violate any bespoke ERC-1404 / RuleEngine restrictions.

Because only role membership is checked, this becomes a single-transaction compliance bypass that defeats the regulatory guarantees the token is supposed to provide.

3 of 22 Findings contracts/deployment/CMTATUpgradeableUUPS.sol

Upgrade authorisation can be performed by Default Admin even without PROXY_UPGRADE_ROLE

Medium Risk

The UUPS implementation restricts upgrades with `onlyRole(PROXY_UPGRADE_ROLE)`, but because of the `hasRole` override described above, any account with `DEFAULT_ADMIN_ROLE` passes the check automatically:

```
function _authorizeUpgrade(address) internal override onlyRole(PROXY_UPGRADE_ROLE) {}
```

This grants the default admin unconditional upgrade power, bypassing the designated `PROXY_UPGRADE_ROLE`. Systems that assume a separation between operational admin and upgrade authority are therefore exposed to single-key upgrade risk.

4 of 22 Findings contracts/deployment/* contracts/modules/wrapper/options/ERC2771Module.sol

Trusted-forwarder address not validated – a malicious forwarder can impersonate any user

Medium Risk

Every deployment contract receives an arbitrary `forwarderIrrevocable` address and passes it to `ERC2771Module`. No check ensures the address really is a well-audited ERC-2771 forwarder. If a wrong or malicious contract is supplied, it can craft calldata that makes `_msgSender()` return **any** address it desires, obtaining roles or performing privileged actions while the on-chain tx appears to come from the forwarder.

Because the forwarder is *irrevocable* (there is no setter), a single misconfiguration at deployment permanently breaks all role-based protections.

5 of 22 Findings

contracts/modules/internal/AllowlistModuleInternal.sol

Missing Initialization for AllowlistModule

Medium Risk

The `AllowlistModuleInternal` enables the allowlist by default during initialization:

```
function __Allowlist_init_unchained() internal onlyInitializing {
     AllowlistModuleInternalStorage storage $ = _getAllowlistModuleInternalStorage();
     $._enableAllowlist = true;
    // no variable to initialize
}
```

This means that if a contract inheriting from `AllowlistModule` is deployed without explicitly disabling the allowlist, transfers will be restricted to allowlisted addresses only. This could lead to unintentional token lockup if no addresses are allowlisted immediately after deployment. Consider changing the default to disabled, or at minimum, clearly documenting this behavior for implementers.

6 of 22 Findings

contracts/mocks/library/snapshot/SnapshotModuleBase.sol

Denial of Service via Unbounded Loop in Snapshotting

Low Risk

The function `_findScheduledMostRecentPastSnapshot` in `SnapshotModuleBase.sol` iterates through the `_scheduledSnapshots` array with a `for` loop to find the most recent snapshot time that is in the past. The complexity of this operation is linear, O(N), where N is the number of scheduled snapshots.

This function is called by `_setCurrentSnapshot`, which is in turn called by `operateOnTransfer` in the `SnapshotEngineMock`. The `operateOnTransfer` function is designed to be called from the main token contract's `_update` hook, which executes on every transfer, mint, and burn operation.

If an account with the `SNAPSHOOTER_ROLE` schedules a large number of snapshots, the gas cost of the `_findScheduledMostRecentPastSnapshot` function can grow significantly. This can cause any subsequent token transfer, mint, or burn to consume an excessive amount of gas, potentially exceeding the block gas limit and causing all transfer-related transactions to fail. This constitutes a Denial of Service (DoS) vulnerability, as a malicious or careless snapshot scheduler can render the token contract's core functionality unusable.

```
// File: contracts/mocks/library/snapshot/SnapshotModuleBase.sol

    function _findScheduledMostRecentPastSnapshot()
        private
        view
        returns (uint256 time, uint256 index)
    {
        SnapshotModuleBaseStorage storage $ = _getSnapshotModuleBaseStorage();
        uint256 currentArraySize = $._scheduledSnapshots.length;
        // ...
        uint256 mostRecent;
        index = currentArraySize;
        // No need of unchecked block since Soliditiy 0.8.22
        for (uint256 i = $._currentSnapshotIndex; i < currentArraySize; ++i ) {
            if ($._scheduledSnapshots[i] <= block.timestamp) {
                mostRecent = $._scheduledSnapshots[i];
                index = i;
            } else {
                // All snapshot are planned in the futur
                break;
            }
        }
        return (mostRecent, index);
    }
```

The call chain that triggers this vulnerability is:

1. `CMTATBaseCommon._update()` (called on every transfer, mint, burn)
2. `snapshotEngine.operateOnTransfer()`
3. `SnapshotEngineMock.operateOnTransfer()` (assuming this mock represents a real engine's logic)
4. `SnapshotModuleBase._setCurrentSnapshot()`
5. `SnapshotModuleBase._findScheduledMostRecentPastSnapshot()`

7 of 22 Findings

contracts/modules/wrapper/options/ERC2771Module.sol

Immutable trusted forwarder creates upgrade challenges

Low Risk

The ERC2771Module sets the trusted forwarder address in the constructor with no mechanism to update it afterward, creating inflexibility in the meta-transaction infrastructure.

```
contract ERC2771Module is ERC2771ContextUpgradeable {
    /// @custom:oz-upgrades-unsafe-allow constructor
    constructor(
        address trustedForwarder
    ) ERC2771ContextUpgradeable(trustedForwarder) {
        // Nothing to do
    }
}
```

This immutability creates several operational risks:

1. If the trusted forwarder is compromised, there's no way to revoke its privileges without redeploying the entire contract
2. If the forwarder needs to be upgraded for security or feature improvements, the change cannot be made
3. In a proxy-based deployment, even upgrading the implementation contract wouldn't change the trusted forwarder

The issue is exacerbated because meta-transactions bypass normal transaction authorization, meaning a compromised forwarder could potentially submit unauthorized transactions that appear to be from legitimate users. For a token representing securities, this rigidity creates unnecessary security and operational risk.

8 of 22 Findings

contracts/modules/wrapper/core/ERC20BaseModule.sol

Missing validation for zero decimals requirement

Low Risk

According to the documentation, CMTAT tokens must have zero decimals to comply with Swiss securities law, but this constraint is not enforced in the contract code.

```
function __ERC20BaseModule_init_unchained(
    uint8 decimals_,
    string memory name_,
    string memory symbol_
) internal virtual onlyInitializing {
    ERC20BaseModuleStorage storage $ = _getERC20BaseModuleStorage();
    $._decimals = decimals_;
    $._symbol = symbol_;
    $._name = name_;
}
```

The code accepts any valid uint8 value for decimals, without validating that it equals zero. This means:

1. A deployment with non-zero decimals could be created, violating the Swiss law requirement
2. Different deployments might have inconsistent decimal settings
3. There's no on-chain enforcement of this critical regulatory requirement

This is particularly concerning because the project documentation explicitly states: "CMTAT enforces 0 decimals, so only integer tokens are issued" and "Integeronly tokens zero decimals align with Swiss ledgered security requirements." Yet this enforcement happens only through correct configuration, not through code constraints.

9 of 22 Findings

contracts/modules/wrapper/options/AllowlistModule.sol

Missing Input Validation in `batchSetAddressAllowlist` Function

Low Risk

The `batchSetAddressAllowlist` function in the `AllowlistModule` contract does not validate that none of the addresses in the `accounts` array is the zero address. While the function does check that the arrays have matching lengths, it doesn't verify the validity of the addresses themselves.

```
function batchSetAddressAllowlist(
    address[] calldata accounts, bool[] calldata status
) public virtual onlyRole(ALLOWLIST_ROLE) {
    _addToAllowlist(accounts, status, "");
}
```

The internal `_addToAllowlist` function calls `EnforcementModuleLibrary._checkInput` which only validates array lengths but not address values:

```
function _checkInput(address[] calldata accounts, bool[] calldata status) internal pure {
    require(accounts.length > 0, CMTAT_Enforcement_EmptyAccounts());
    require(bool(accounts.length == status.length),
CMTAT_Enforcement_AccountsValueslengthMismatch());
}
```

This oversight could allow an administrator to accidentally or maliciously add the zero address to the allowlist, potentially leading to unexpected behavior in token transfers and compromising the security of the token contract.

10 of 22 Findings

contracts/modules/wrapper/options/AllowlistModule.sol

Missing Input Validation in `setAddressAllowlist` Function

Low Risk

The `setAddressAllowlist` function in the `AllowlistModule` contract does not validate that the `account` parameter is not the zero address. This oversight allows administrators to add the zero address to the allowlist, which could lead to unexpected behavior in the token transfer logic.

```
function setAddressAllowlist(address account, bool status) public virtual
onlyRole(ALLOWLIST_ROLE) {
    _addToAllowlist(account, status, "");
}
```

The same issue exists in the overloaded version that accepts data:

```
function setAddressAllowlist(address account, bool status, bytes calldata data) public
virtual onlyRole(ALLOWLIST_ROLE) {
    _addToAllowlist(account, status, data);
}
```

If the zero address is added to the allowlist, it could potentially allow transfers to/from the zero address outside of the intended minting and burning functions, bypassing important validation checks. This could lead to tokens being accidentally sent to the zero address and permanently lost.

11 of 22 Findings

contracts/modules/internal/ERC20EnforcementModuleInternal.sol

Incorrect error parameters in `_unfreezeTokens`

Info

The `_unfreezeTokens` function in `ERC20EnforcementModuleInternal.sol` performs a check to ensure the value of tokens being unfrozen does not exceed the account's balance. If the check fails, it reverts with an `ERC20InsufficientBalance` error. However, the parameters passed to this error are incorrect.

The standard OpenZeppelin `ERC20InsufficientBalance` error is defined as `error ERC20InsufficientBalance(address sender, uint256 balance, uint256 needed)`. The `sender` should be the account with the insufficient balance, and `needed` should be the required amount.

The current implementation reverts with `revert ERC20InsufficientBalance(_msgSender(), balance, value-balance)`. This is incorrect for two reasons:
1. The `sender` is passed as `_msgSender()`, which is the enforcer role, not the account (`from`) whose balance is insufficient.
2. The `needed` amount is passed as `value-balance`, which is the shortfall, not the total `value` required.

Code snippet from `_unfreezeTokens`:

```
    function _unfreezeTokens(address from, uint256 value, bytes memory data) internal
virtual{
        uint256 balance = ERC20Upgradeable.balanceOf(from);
        if(value > balance){
            revert ERC20InsufficientBalance(_msgSender(), balance, value-balance);
        }
        // ...
    }
```

This incorrect error reporting can mislead off-chain tooling and developers trying to debug failed transactions. The correct call should be `revert ERC20InsufficientBalance(from, balance, value);`.

12 of 22 Findings

contracts/mocks/library/snapshot/SnapshotModuleBase.sol

Uninitialized variable used in condition

Info

In `SnapshotModuleBase.sol`, the function `_findScheduledMostRecentPastSnapshot` uses an uninitialized return variable `time` within an `if` condition. Return variables are default-initialized to zero, so `time` will be `0` inside the function body.

Code snippet:

```
    function _findScheduledMostRecentPastSnapshot()
        private
        view
        returns (uint256 time, uint256 index)
    {
        SnapshotModuleBaseStorage storage $ = _getSnapshotModuleBaseStorage();
        uint256 currentArraySize = $._scheduledSnapshots.length;
        // no snapshot or the current snapshot already points on the last snapshot
        if (
            currentArraySize == 0 ||
            (($._currentSnapshotIndex + 1 == currentArraySize) && (time != 0))
        ) {
            return (0, currentArraySize);
        }
        // ...
    }
```

Because `time` is always `0`, the condition `time != 0` is always false. This makes the second part of the `||` condition, which appears to be an optimization to exit early, unreachable. While this does not seem to break the core logic of the function, it renders the optimization ineffective and leads to unnecessary gas consumption by always proceeding to the loop that follows. Since this contract is a mock provided as part of the framework, developers using it may be affected by this inefficiency.

13 of 22 Findings

contracts/modules/wrapper/options/AllowlistModule.sol

contracts/modules/wrapper/core/EnforcementModule.sol

Potential Front-Running in Allow/Blocklisting

Info

Functions like setAddressAllowlist() and setAddressFrozen() can be front-run, allowing entities to perform actions before restrictions take effect. An entity monitoring the mempool could see these transactions and quickly execute transfers before being blocked.

```
function setAddressAllowlist(address account, bool status) public virtual
onlyRole(ALLOWLIST_ROLE) {
    _addToAllowlist(account, status, "");
}
```

For example, if a regulator decides to freeze an address due to suspicious activity, the target could monitor the mempool, see the freeze transaction, and quickly transfer their funds out before the freeze takes effect, defeating the purpose of the enforcement action.

14 of 22 Findings

contracts/modules/wrapper/extensions/DocumentEngineModule.sol

contracts/modules/wrapper/extensions/SnapshotEngineModule.sol

contracts/modules/wrapper/extensions/ValidationModule/ValidationModuleRuleEngine.sol

Potential for External Engine Integration Failures

Info

The contract delegates critical functionality to external engines (Document, Snapshot, and Rule engines) using a pattern where calls are only forwarded if the engine address is non-zero:

```
function getDocument(string memory name) public view virtual override(IERC1643) returns
(Document memory document){
    DocumentEngineModuleStorage storage $ = _getDocumentEngineModuleStorage();
    if(address($._documentEngine) != address(0)){
        return $._documentEngine.getDocument(name);
    } else{
        return Document("", 0x0, 0);
    }
}
```

This creates a risk where misconfigured or outdated external engines could cause unexpected behavior. Additionally, there's no comprehensive validation of engine interfaces at the time of setting them. Consider implementing more robust validation when setting external engines and clear fallback mechanisms for when external engines fail.

15 of 22 Findings

contracts/modules/wrapper/extensions/ERC20EnforcementModule.sol

Lack of Input Validation in `freezePartialTokens` and `unfreezePartialTokens`

Best Practices

The `freezePartialTokens` and `unfreezePartialTokens` functions in the `ERC20EnforcementModule` contract do not validate that the `value` parameter is greater than zero. This allows administrators to execute these functions with a zero value, which would emit events without actually changing any state, potentially misleading observers and monitoring systems.

```
function freezePartialTokens(address account, uint256 value) public virtual
override(IERC3643ERC20Enforcement) onlyRole(ERC20ENFORCER_ROLE){
    _freezePartialTokens(account, value, "");
}

function unfreezePartialTokens(address account, uint256 value) public virtual
override(IERC3643ERC20Enforcement) onlyRole(ERC20ENFORCER_ROLE) {
    _unfreezePartialTokens(account, value, "");
}
```

The same issue exists in the overloaded versions that accept data:

```
function freezePartialTokens(address account, uint256 value, bytes calldata data) public
virtual override(IERC7551ERC20Enforcement) onlyRole(ERC20ENFORCER_ROLE){
    _freezePartialTokens(account, value, data);
}

function unfreezePartialTokens(address account, uint256 value, bytes calldata data) public
virtual override(IERC7551ERC20Enforcement) onlyRole(ERC20ENFORCER_ROLE) {
    _unfreezePartialTokens(account, value, data);
}
```

This could lead to confusion and potential security issues if monitoring systems or other contracts rely on these events to track token freezing and unfreezing activities.

16 of 22 Findings

contracts/modules/wrapper/options/ERC2771Module.sol

Missing Zero Address Validation for Trusted Forwarder

Best Practices

The `ERC2771Module` constructor does not validate that the `trustedForwarder` is not the zero address:

```
constructor(address trustedForwarder) ERC2771ContextUpgradeable(trustedForwarder) {
    // Nothing to do
}
```

If the zero address is passed, it would result in a non-functional meta-transaction system. Consider adding zero address validation for the trusted forwarder.

17 of 22 Findings

contracts/modules/wrapper/options/DebtModule.sol

Insufficient Input Validation for Debt Module Functions

Best Practices

Several functions in the DebtModule lack proper input validation for their parameters. For example, setDebt(), setCreditEvents(), and setDebtInstrument() don't validate their inputs before storing them.

```
function setDebt(ICMTATDebt.DebtInformation calldata debt_) public virtual
override(IDebtModule) onlyRole(DEBT_ROLE) {
    DebtModuleStorage storage $ = _getDebtModuleStorage();
    $._debt = debt_;
    emit DebtLogEvent();
}
```

Without proper validation, it's possible to store invalid, inconsistent, or malformed data that could cause issues with downstream systems that rely on this data. For example, negative values, invalid dates, or empty strings in debt information could cause problems for applications using this data.

18 of 22 Findings

contracts/modules/3_CMTATBaseERC20CrossChain.sol

Inconsistent Pause Protection

Best Practices

The whenNotPaused modifier is applied to some functions like crosschainMint() and crosschainBurn(), but it might not be consistently applied to all sensitive functions throughout the codebase.

```
function crosschainMint(address to, uint256 value) public virtual override(IERC7802)
onlyRole(CROSS_CHAIN_ROLE) whenNotPaused {
    emit CrosschainMint(to, value, _msgSender());
    _mintOverride(to, value);
}
```

Inconsistent application of the whenNotPaused modifier could lead to certain operations still being possible when the contract is paused, potentially undermining the security benefits of the pause mechanism. This could be particularly problematic if critical functions remain active during an emergency pause.

19 of 22 Findings

contracts/deployment/debt/CMTATStandaloneDebt.sol

contracts/deployment/debt/CMTATUpgradeableDebt.sol

contracts/deployment/light/CMTATStandaloneLight.sol

contracts/deployment/light/CMTATUpgradeableLight.sol

Misleading NatSpec Comments in Deployment Contracts

• Best Practices

The NatSpec documentation for the constructors of several deployment contracts (`CMTATStandaloneDebt`, `CMTATUpgradeableDebt`, `CMTATStandaloneLight`, `CMTATUpgradeableLight`) incorrectly states that they accept a `forwarderIrrevocable` parameter for ERC2771 meta-transaction support. However, the actual contract implementations do not inherit the `ERC2771Module`, their constructors do not accept this parameter, and they do not provide gasless transaction functionality.

This discrepancy between the documentation and the implementation is misleading for developers and users of the framework, who might incorrectly assume that these specific token configurations support meta-transactions. While this does not introduce a direct code vulnerability, it represents a significant documentation flaw that can lead to integration errors and incorrect assumptions about the contracts' capabilities.

Example from `CMTATStandaloneDebt.sol`:

```
// File: contracts/deployment/debt/CMTATStandaloneDebt.sol

contract CMTATStandaloneDebt is CMTATBaseDebt {
    /**
     * @notice Contract version for standalone deployment
     * @param forwarderIrrevocable address of the forwarder, required for the gasless
support // This parameter does not exist in the constructor signature.
     * @param admin address of the admin of contract (Access Control)
     * @param ERC20Attributes_ ERC20 name, symbol and decimals
     * @param baseModuleAttributes_ tokenId, terms, information
     * @param engines_ external contract
     */
    /// @custom:oz-upgrades-unsafe-allow constructor
    constructor(
        address admin,
        ICMTATConstructor.ERC20Attributes memory ERC20Attributes_,
        ICMTATConstructor.ExtraInformationAttributes memory extraInformationAttributes_,
        ICMTATConstructor.Engine memory engines_
    ) {
        // ...
    }
}
```

Similar incorrect NatSpec comments are present in `CMTATUpgradeableDebt.sol`, `CMTATStandaloneLight.sol`, and `CMTATUpgradeableLight.sol`.

20 of 22 Findings

contracts/modules/internal/ERC20BurnModuleInternal.sol

contracts/modules/internal/ERC20MintModuleInternal.sol

Missing Granular Events for Important State Changes

Best Practices

Several functions that modify important state variables don't emit their own events, making it difficult to track changes off-chain. While the underlying OpenZeppelin implementations might emit events, additional custom events would improve transparency and auditability.

```
function _burnOverride(address account, uint256 value) internal virtual {
    ERC20Upgradeable._burn(account, value);
}
```

The lack of granular events makes it more difficult to monitor and audit the contract's behavior, especially for specific operations like overrides that might have different semantic meaning than the standard operations.

21 of 22 Findings

contracts/deployment/debt/CMTATUpgradeableDebt.sol

contracts/deployment/light/CMTATUpgradeableLight.sol

Misleading NatSpec comments in constructors of upgradeable contracts

Best Practices

The constructors for several upgradeable contracts contain NatSpec comments that are inconsistent with their actual function signatures. Specifically, they mention a parameter that the constructor does not accept.

For example, in `CMTATUpgradeableDebt.sol`:

```
/**
 * @notice Contract version for the deployment with a proxy
 * @param forwarderIrrevocable address of the forwarder, required for the gasless support
 */
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    // Disable the possibility to initialize the implementation
    _disableInitializers();
}
```

The NatSpec comment includes `@param forwarderIrrevocable`, but the constructor `constructor()` takes no arguments. A similar issue exists in `CMTATUpgradeableLight.sol`.

This discrepancy arises because these contract variants do not inherit the `ERC2771Module`, which is responsible for gasless transaction support and requires the forwarder address upon construction. The comments appear to have been copied from other contracts in the framework that do include this functionality. While this does not pose a direct security risk, it creates misleading documentation that can cause confusion and integration errors for developers using the framework.

22 of 22 Findings

contracts/modules/wrapper/extensions/ExtraInformationModule.sol

Use of block.timestamp for Time-Sensitive Operations

Best Practices

The contract uses block.timestamp for recording the last modification time of documents. While this is less of an issue in recent Ethereum versions, miners can still manipulate block.timestamp to a limited extent.

```
function _setTerms(ExtraInformationModuleStorage storage $, IERC1643CMTAT.DocumentInfo
memory terms_) internal virtual {
    $._terms.name = terms_.name;
    $._terms.doc.documentHash = terms_.documentHash;
    $._terms.doc.uri = terms_.uri;
    $._terms.doc.lastModified = block.timestamp;
    emit Term($._terms);
}
```

For most applications, the potential manipulation of block.timestamp (usually by a few seconds) is not significant. However, if precise timing is crucial for regulatory compliance or other time-sensitive operations, this could be a concern.

Disclaimer

Kindly note, the Audit Agent is currently in beta stage and no guarantee is being given as to the accuracy and/or completeness of any of the outputs the Audit Agent may generate, including without limitation this Report. The results set out in this Report may not be complete nor inclusive of all vulnerabilities. The Audit Agent is provided on an 'as is' basis, without warranties or conditions of any kind, either express or implied, including without limitation as to the outputs of the code scan and the security of any smart contract verified using the Audit Agent.

Blockchain technology remains under development and is subject to unknown risks and flaws. This Report does not indicate the endorsement of any particular project or team, nor guarantee its security. Neither you nor any third party should rely on this Report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset.

To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this Report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.