Scanned Code Report

AUDITAGENT

Powered by NETHERMIND SECURITY

Code Info

Deep Scan

Scan ID 2	Date November 18, 2025
Organization CMTA	Repository CMTAT
Branch dev	Commit Hash 5148513f...89477867

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/IERC5679.sol
contracts/interfaces/technical/IERC7802.sol
contracts/interfaces/technical/IGetCCIPAdmin.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/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/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/core/VersionModule.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/CCIPModule.sol

contracts/modules/wrapper/options/DebtEngineModule.sol

contracts/modules/wrapper/options/DebtModule.sol

contracts/modules/wrapper/options/ERC20CrossChainModule.sol

contracts/modules/wrapper/options/ERC2771Module.sol

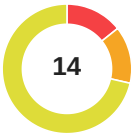
contracts/modules/wrapper/options/ERC7551Module.sol

contracts/modules/wrapper/security/AccessControlModule.sol

Code Statistics

Findings	Contracts Scanned	Lines of Code
14	75	5999

Findings Summary



Total Findings

- High Risk (2)
- Medium Risk (2)
- Low Risk (10)
- Info (0)
- Best Practices (0)

Code Summary

This protocol provides a highly modular and configurable framework for creating standards-compliant security tokens, based on the Swiss Capital Markets and Technology Association (CMTAT) specifications. It extends the ERC-20 standard with a comprehensive suite of features for compliance, on-chain identity, and asset lifecycle management.

The architecture is built around a core token contract that can be enhanced with various optional modules. It also utilizes an 'engine' pattern, allowing it to delegate complex logic to external, swappable smart contracts. This design enables the creation of customized tokens tailored to specific regulatory and business requirements.

Key features of the protocol include:
- **Compliance and Transfer Controls:** The protocol integrates robust mechanisms for enforcing transfer restrictions. This includes the ability to pause all token transfers, freeze individual accounts, or freeze partial balances of an account. Compliance can be managed through a simple built-in allowlist or delegated to a sophisticated external `RuleEngine` for complex, programmable logic, aligning with standards like ERC-1404 and ERC-3643.
- **Rich Asset Metadata:** Tokens created with this framework can store detailed on-chain information, including a unique `tokenId` (e.g., an ISIN), legal `terms` linked via an ERC-1643 compliant document management system, and other descriptive `information`.
- **Role-Based Access Control:** It employs a granular access control system with predefined roles such as `MINTER_ROLE`, `BURNER_ROLE`, `PAUSER_ROLE`, and `ENFORCER_ROLE`. This ensures that critical functions like token issuance, burning, and compliance enforcement are restricted to authorized entities.
- **Asset Lifecycle Management:** The protocol supports the full lifecycle of a tokenized asset, including single and batch minting, burning, and a `deactivateContract` function for handling corporate actions or delisting the asset. It also allows authorized administrators to perform forced transfers.
- **Extensibility and Interoperability:** Optional modules provide advanced functionality, such as support for debt instruments (`DebtModule`), cross-chain transfers via ERC-7802 (`ERC20CrossChainModule`), meta-transactions for gasless user experiences (`ERC2771Module`), and payable token interactions (`ERC1363Module`).
- **State and History Tracking:** The framework can integrate with an external `SnapshotEngine` to record historical token balances and total supply, a critical feature for governance, voting, and dividend distributions.

Main Entry Points and Actors

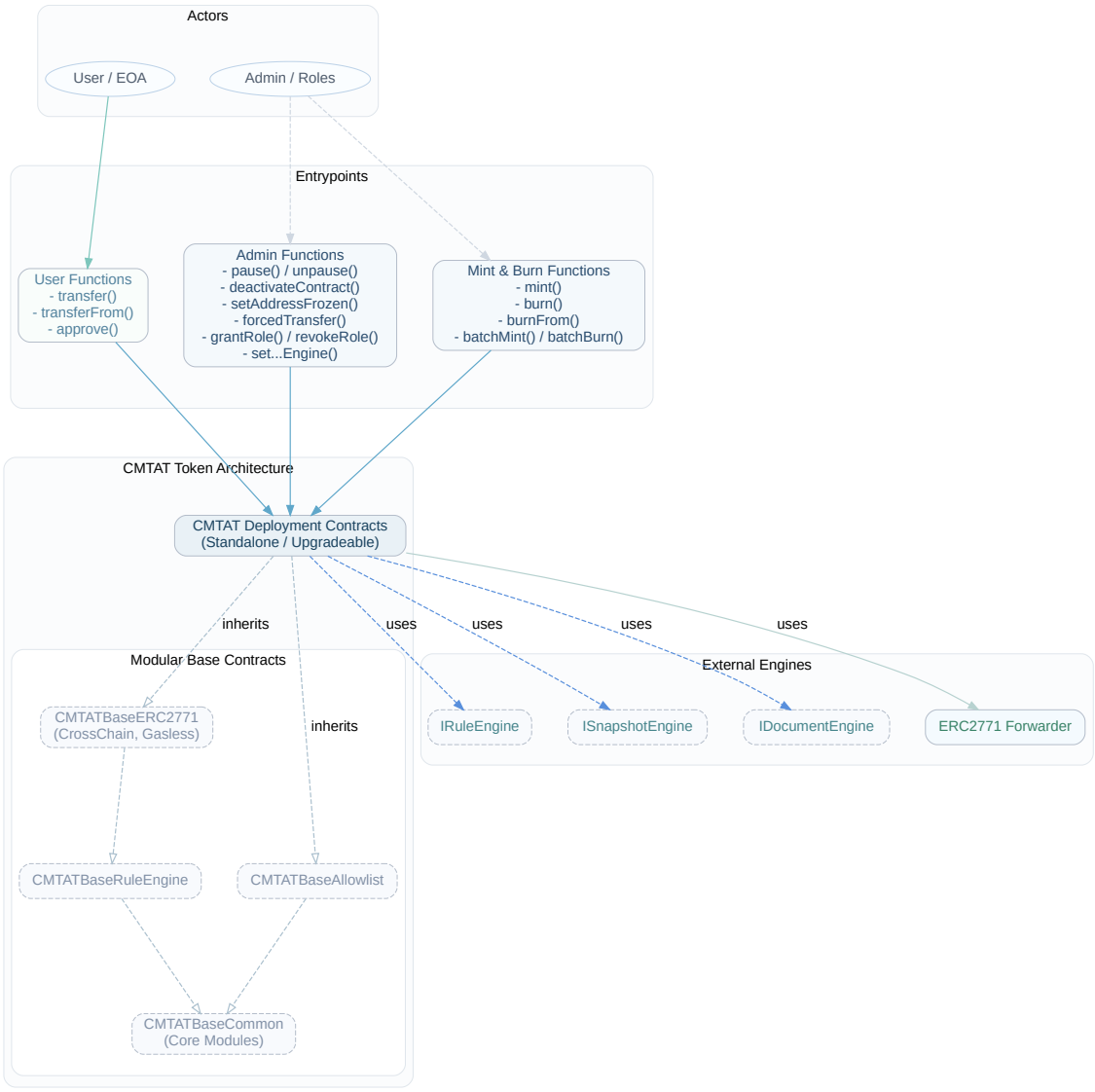
Below is a list of the main functions that allow actors to modify the protocol's state:

- **Token Holder:** An entity that owns the tokens.
- `transfer(address to, uint256 value)`: Transfers tokens to another address.
- `approve(address spender, uint256 value)`: Approves a spender to manage a certain amount of tokens.
- `increaseAllowance(address spender, uint256 addedValue)`: Increases the allowance of a spender.
- `decreaseAllowance(address spender, uint256 subtractedValue)`: Decreases the allowance of a spender.
- `transferAndCall(address to, uint256 value, bytes calldata data)`: (ERC1363) Transfers tokens and calls a function on the recipient contract.
- `approveAndCall(address spender, uint256 value, bytes calldata data)`: (ERC1363) Approves a spender and calls a function on the spender contract.
- **Spender:** An address that has been approved to manage a token holder's tokens.

- `transferFrom(address from, address to, uint256 value)`: Transfers tokens on behalf of a token holder.
- `transferFromAndCall(address from, address to, uint256 value, bytes calldata data)`: (ERC1363) Transfers tokens on behalf of a holder and calls a function on the recipient contract.
- **Minter (`MINTER_ROLE`):** An authorized address that can create new tokens.
- `mint(address account, uint256 value, bytes calldata data)`: Mints new tokens to a specific account.
- `batchMint(address[] calldata accounts, uint256[] calldata values)`: Mints new tokens to multiple accounts in a single transaction.
- `batchTransfer(address[] calldata tos, uint256[] calldata values)`: Transfers tokens from the minter's account to multiple recipients.
- **Burner (`BURNER_ROLE`):** An authorized address that can destroy tokens.
- `burn(address account, uint256 value, bytes calldata data)`: Burns tokens from a specific account.
- `batchBurn(address[] calldata accounts, uint256[] calldata values, bytes memory data)`: Burns tokens from multiple accounts in a single transaction.
- **Pauser (`PAUSER_ROLE`):** An authorized address that can pause and unpause all token transfers.
- `pause()`: Pauses all token transfers.
- `unpause()`: Resumes token transfers.
- **Enforcer (`ENFORCER_ROLE`):** An authorized address that can freeze or unfreeze accounts.
- `setAddressFrozen(address account, bool freeze)`: Sets the frozen status of an entire account.
- `batchSetAddressFrozen(address[] calldata accounts, bool[] calldata freezes)`: Sets the frozen status for multiple accounts.
- **ERC20 Enforcer (`ERC20ENFORCER_ROLE`):** An authorized address that can manage partial token balances.
- `freezePartialTokens(address account, uint256 value, bytes calldata data)`: Freezes a specific amount of tokens in an account.
- `unfreezePartialTokens(address account, uint256 value, bytes calldata data)`: Unfreezes a specific amount of tokens in an account.
- **Allowlist Manager (`ALLOWLIST_ROLE`):** An authorized address that manages the token's allowlist.
- `setAddressAllowlist(address account, bool status, bytes calldata data)`: Adds or removes an account from the allowlist.
- `batchSetAddressAllowlist(address[] calldata accounts, bool[] calldata status)`: Manages multiple accounts on the allowlist.
- `enableAllowlist(bool status)`: Enables or disables the allowlist requirement for transfers.
- **Debt Manager (`DEBT_ROLE`):** An authorized address that can update debt-related information.
- `setDebt(DebtInformation calldata debt_)`: Sets the primary debt information.
- `setCreditEvents(CreditEvents calldata creditEvents_)`: Sets credit event flags and ratings.
- `setDebtInstrument(DebtInstrument calldata debtInstrument_)`: Sets the details of the debt instrument.

- **Cross-Chain Bridge (`CROSS_CHAIN_ROLE`):** A trusted contract that can mint and burn tokens as part of a cross-chain transfer.
- `crosschainMint(address to, uint256 value)`: Mints tokens on the destination chain.
- `crosschainBurn(address from, uint256 value)`: Burns tokens on the source chain.
- **Authorized Burners (`BURNER_FROM_ROLE`, `BURNER_SELF_ROLE`):** Roles for burning tokens with an allowance or from one's own balance.
- `burnFrom(address account, uint256 value)`: Burns tokens from another account using an allowance.
- `burn(uint256 value)`: Burns tokens from the caller's own balance.

Code Diagram



1 of 14 Findings

contracts/modules/1_CMTATBaseAllowlist.sol

contracts/modules/1_CMTATBaseRuleEngine.sol

contracts/modules/2_CMTATBaseERC1404.sol

contracts/modules/internal/ERC20EnforcementModuleInternal.sol

Partial-freeze not enforced on transfer path allows frozen tokens to be spent (breaks frozen ≤ balance invariant)

High Risk

Observed behavior in provided modules:

1) The write-path hook used during transfers, _checkTransferred, does not enforce the active-balance (balance − frozen) constraint.

```
// contracts/modules/1_CMTATBaseAllowlist.sol
function _checkTransferred(address spender, address from, address to, uint256 value)
    internal virtual override(CMTATBaseCommon)
{
    CMTATBaseCommon._checkTransferred(spender, from, to, value);
    if (!ValidationModule._canTransferGenericByModule(spender, from, to)) {
        revert Errors.CMTAT_InvalidTransfer(from, to, value);
    }
    // NOTE: no call to _checkActiveBalance(...) here
}

// contracts/modules/1_CMTATBaseRuleEngine.sol
function _checkTransferred(address spender, address from, address to, uint256 value)
    internal virtual override(CMTATBaseCommon)
{
    CMTATBaseCommon._checkTransferred(spender, from, to, value);
    require(ValidationModuleRuleEngine._transferred(spender, from, to, value),
Errors.CMTAT_InvalidTransfer(from, to, value));
    // ValidationModuleRuleEngine._transferred() only checks _canTransferGenericByModule
    // (freeze/paused), not active balance; no _checkActiveBalance(...) either
}
```

2) The active-balance check exists only in the read-path helpers (canTransfer / canTransferFrom and ERC-1404 detection):

```
// e.g., contracts/modules/1_CMTATBaseAllowlist.sol
if(!ERC20EnforcementModuleInternal._checkActiveBalance(from, value)){ return false; }

// contracts/modules/2_CMTATBaseERC1404.sol
uint256 activeBalance = ERC20Upgradeable.balanceOf(from) - frozenTokensLocal; // view-time
computation only
```

3) The enforcement module exposes the check that should guard the write path:

```
// contracts/modules/internal/ERC20EnforcementModuleInternal.sol
function _checkActiveBalanceAndRevert(address from, uint256 value) internal view {
    require(_checkActiveBalance(from, value),
CMTAT_ERC20EnforcementModule_ValueExceedsActiveBalance());
}
```

Because the actual transfer flow (via _checkTransferred) never calls _checkActiveBalanceAndRevert (nor an equivalent), a holder can transfer more than their active balance when their address is not fully frozen and the contract is not paused. This lets balances decrease while the per-account frozen amount remains unchanged.

Concrete invariant break scenario:
- Pre-state: Alice balance = 100, frozen = 80.
- Alice calls ERC20.transfer(Bob, 50).
- _checkTransferred only checks freeze/paused/allowlist, so the transfer proceeds.
- Post-state: Alice balance = 50, frozen = 80.
- This violates the invariant frozen[alice] ≤ balanceOf[alice]. Furthermore, subsequent view functions that compute active balance with `balanceOf - frozen` can underflow/revert (e.g., detectTransferRestriction and getActiveBalanceOf).

Impact:
- Partial-freeze protection is bypassed on-chain during transfers.
- Invariants (frozen ≤ balance) are violated, enabling inconsistent token states and potential DoS in view/compliance paths that assume the invariant.

Severity Note:
- _checkTransferred is the effective gate in the transfer write path and there is no additional downstream check that enforces active-balance.
- CMTATBaseCommon._checkTransferred does not itself enforce active-balance (based on provided snippets).

2 of 14 Findings

contracts/modules/1_CMTATBaseRuleEngine.sol

Unprotected initialize() allows front-running of proxy initialization

High Risk

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

The public initializer modifier only ensures the function is called once but does not restrict who can call it. An attacker observing the deployment of an uninitialized proxy can front-run the legitimate initialization transaction and invoke `initialize()` with their own address as `admin`, along with malicious or zero addresses for the external engines. This grants the attacker `DEFAULT_ADMIN_ROLE` (and thus all other roles), allows them to mint/burn tokens at will, pause/unpause or deactivate the contract, freeze/unfreeze addresses, and replace engine contracts with attacker-controlled implementations. This results in a complete loss of protocol control and potential theft of all funds.

Severity Note:
- The system deploys this contract behind a proxy and does not always initialize atomically via constructor calldata.
- DEFAULT_ADMIN_ROLE controls critical actions (pause/deactivate/freeze/rule-engine management; potentially mint/burn) in downstream modules.

3 of 14 Findings

contracts/modules/0_CMTATBaseCommon.sol

Missing spender validation in transfer check function

Medium Risk

In CMTATBaseCommon.sol, the transferFrom function passes _msgSender() as the spender parameter to _checkTransferred, but this parameter is marked as unused (/*spender*/) in the function signature. The validation logic does not verify if the spender is frozen or has any restrictions.

```
function transferFrom(
    address from,
    address to,
    uint256 value
)
    public
    virtual
    override(ERC20Upgradeable, ERC20BaseModule)
    returns (bool)
{
    _checkTransferred(_msgSender(), from, to, value);
    return ERC20BaseModule.transferFrom(from, to, value);
}

function _checkTransferred(address /*spender*/, address from, address /* to */, uint256
value) internal virtual {
    ERC20EnforcementModuleInternal._checkActiveBalanceAndRevert(from, value);
}
```

This means that a frozen spender could still execute transfers on behalf of other users if they have allowance, potentially bypassing compliance restrictions. The spender parameter should be validated according to the compliance rules.

Severity Note:
- Compliance policy intends to block frozen/banned spenders from initiating transferFrom.
- At least some users have active allowances (potentially unlimited) to the frozen spender (EOA or contract).

4 of 14 Findings

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

Missing Contract Validation for RuleEngine Address

Medium Risk

The `setRuleEngine` function does not validate that the provided address is actually a contract implementing the required `IRuleEngine` interface:

```
function setRuleEngine(
    IRuleEngine ruleEngine_
) public virtual onlyRuleEngineManager {
     require(ruleEngine_ != ruleEngine(), CMTAT_ValidationModule_SameValue());
    _setRuleEngine(ruleEngine_);
}
```

The function only checks that the new address is different from the current one, but doesn't verify:
1. That the address is a contract (not an EOA)
2. That the contract implements the required `IRuleEngine` interface

If an invalid address is set (EOA or incompatible contract), all subsequent transfers that rely on the RuleEngine will fail, effectively causing a denial of service for token transfers. This could be exploited by a compromised admin to accidentally or intentionally break the token's functionality.

A proper implementation should include:

```
require(address(ruleEngine_).code.length > 0, "Not a contract");
// Optionally: Check interface support via ERC-165
```

Severity Note:
- The host token/module enforces transfer checks via canTransfer/canTransferFrom so that a revert blocks transfers.
- The RuleEngine manager role is held by a privileged admin as implied by onlyRuleEngineManager.

5 of 14 Findings

contracts/modules/0_CMTATBaseCommon.sol

contracts/modules/wrapper/core/PauseModule.sol

Transfers ignore pause/deactivation when using CMTATBaseCommon (transfers remain possible after deactivateContract)

Low Risk

In CMTATBaseCommon, the ERC-20 `transfer` / `transferFrom` paths do not consult the pause/deactivation state. If a final token composition combines PauseModule with CMTATBaseCommon as the provider of ERC-20 transfers, users can still transfer while the contract is paused or even after deactivation, contradicting the invariant's stated goal that deactivation permanently halts transfers.

Problematic code (no pause/deactivation check):

```
function transfer(address to, uint256 value) public virtual override(ERC20Upgradeable)
returns (bool) {
    address from = _msgSender();
    _checkTransferred(address(0), from, to, value);
    ERC20Upgradeable._transfer(from, to, value);
    return true;
}

function transferFrom(address from, address to, uint256 value)
    public
    virtual
    override(ERC20Upgradeable, ERC20BaseModule)
    returns (bool)
{
    _checkTransferred(_msgSender(), from, to, value);
    return ERC20BaseModule.transferFrom(from, to, value);
}
```

These functions neither use `whenNotPaused` nor check `paused()` / `deactivated()` (directly or via a validation module). Thus, if the final contract does not route transfers through a validation layer that enforces pause, transfers succeed even when `PauseModule` reports `paused() == true` and `deactivated() == true`.

Impact relative to the invariant: although `PauseModule` ensures `deactivated() ==> paused()` and blocks `unpause()`, the intended effect "transfers are permanently halted" after deactivation is not guaranteed when using these transfer implementations.

Severity Note:
- The final token inherits PauseModule and CMTATBaseCommon without another module overriding transfer/transferFrom to enforce pause/deactivation.
- No external enforcement layer intercepts transfers to apply pause logic.

6 of 14 Findings

contracts/modules/0_CMTATBaseCommon.sol

Transfers to frozen recipients are possible in CMTATBaseCommon.transfer(), violating the freeze invariant for receivers

Low Risk

The freeze invariant requires that an increase in a receiver's balance implies the receiver was not frozen in the pre-state. In CMTATBaseCommon, `transfer()` and `transferFrom()` call `_checkTransferred(...)`, which only checks the sender's active (unfrozen) balance and ignores the receiver's frozen status; then they call `ERC20Upgradeable._transfer` directly.

Problematic code:

```
function transfer(address to, uint256 value) public virtual override(ERC20Upgradeable)
returns (bool) {
    address from = _msgSender();
    _checkTransferred(address(0), from, to, value); // does not check `to` freeze status
    ERC20Upgradeable._transfer(from, to, value);
    return true;
}

function _checkTransferred(address /*spender*/, address from, address /* to */, uint256
value) internal virtual {
    ERC20EnforcementModuleInternal._checkActiveBalanceAndRevert(from, value);
}
```

Because the `to` address is not validated for freeze status, a transfer to a frozen `to` will succeed, increasing the balance of a frozen account and violating the invariant constraint for receivers.

Severity Note:
- Frozen status is intended to block receiving as well as sending (receiver-freeze invariant).
- ERC20EnforcementModule does not perform a later-stage receiver check in code not provided here.
- Admin forced-transfer functionality indeed exists on the deployed setup (suggested by _authorizeForcedTransfer).

7 of 14 Findings

contracts/modules/internal/ERC20EnforcementModuleInternal.sol

contracts/interfaces/engine/ISnapshotEngine.sol

Reentrancy window between unfreeze and balance update allows freezing based on stale balance, causing frozen > balance after completion

Low Risk

freezePartialTokens() checks the current balance at call time, while _forcedTransfer() first unfreezes and only then updates balances via _transfer/_burn. Per the developer docs and the ISnapshotEngine interface, operateOnTransfer(...) is called from the token's transfer hook before any state change. This creates an external-call window where a privileged callee can reenter and call freezePartialTokens() using the pre-transfer balance, producing a post-transfer state with frozen tokens exceeding the new balance.

Relevant code snippets:

```
// 1) freeze uses current balance at call time
function _freezePartialTokens(address account, uint256 value, bytes memory data) internal
virtual {
    uint256 balance = ERC20Upgradeable.balanceOf(account);
    uint256 frozenBalance = $._frozenTokens[account] + value;
    require(balance >= frozenBalance,
CMTAT_ERC20EnforcementModule_ValueExceedsAvailableBalance());
    $._frozenTokens[account] = frozenBalance;
    emit TokensFrozen(account, value, data);
}

// 2) forced transfer unfreezes first, then calls _transfer/_burn (which triggers external
hooks)
function _forcedTransfer(address from, address to, uint256 value, bytes memory data)
internal virtual {
    _unfreezeTokens(from, value, data); // reduces frozen pre-transfer
    if (to == address(0)) { _burn(from, value); } else { _transfer(from, to, value); }
    emit Enforcement(_msgSender(), from, value, data);
}
```

ISnapshotEngine (developer comment):

```
// should be called inside the {_update} hook so that snapshots are updated prior to any
state changes
function operateOnTransfer(address from, address to, uint256 balanceFrom, uint256 balanceTo,
uint256 totalSupply) external;
```

Attack path (requires the external engine or a reentrant callee to hold freezing privileges):
1) State before: balance_pre = 100, frozen_pre = 80 ⇒ active_pre = 20; admin calls forcedTransfer(from, to, 50, data).
2) _unfreezeTokens lowers frozen to 50.
3) Before the balance is reduced, _transfer/_burn triggers operateOnTransfer(...), allowing reentrancy. The callee calls freezePartialTokens(from, 50). The require passes because balanceOf(from) is still 100, so frozen becomes 100.
4) The original transfer resumes; balance becomes 50. Final state: balance_post = 50, frozen_post = 100.

This violates:
- Invariant 0 and 1: getFrozenTokens(account) <= balanceOf(account) no longer holds after completion (100 > 50).
- Invariant 3: the post-state frozen amount is no longer `frozen_pre - expectedUnfrozenAmount` because an extra freeze occurred intra-call.

Secondary effect: any function computing `active = balance - frozen` (e.g., detectTransferRestriction and getActiveBalanceOf) can now underflow/revert, creating a denial-of-service for reads until an operator intervenes.

Severity Note:
- The token integrates an external engine (e.g., ISnapshotEngine) that is called before balances change and allows reentrancy.
- There exists an external/operator function that ultimately calls _freezePartialTokens with appropriate privileges.
- No reentrancy guard prevents freezing during the pre-state-change callback.
- The privileged module or operator can call back into the token during operateOnTransfer.

8 of 14 Findings

contracts/modules/1_CMTATBaseAllowlist.sol

contracts/modules/internal/ERC20EnforcementModuleInternal.sol

canTransfer/canTransferFrom can return true even when the ERC-20 transfer would revert (insufficient balance when no tokens are frozen)

Low Risk

Invariant 4 requires canTransferFrom to accurately predict the success of a corresponding transferFrom call. The pre-flight checks only consider "active" balance when some tokens are frozen; if no tokens are frozen, the active-balance check always returns true, and the functions do not verify total balance. This can yield true while a real transfer reverts due to insufficient ERC-20 balance.

Relevant code:

```
// contracts/modules/internal/ERC20EnforcementModuleInternal.sol
function _checkActiveBalance(address from, uint256 value) internal view returns(bool){
    uint256 frozenTokensLocal = _getFrozenTokens(from);
    if(frozenTokensLocal > 0 ){
        uint256 activeBalance = ERC20Upgradeable.balanceOf(from) - frozenTokensLocal;
        if(value > activeBalance) {
            return false;                // Only enforced when some tokens are frozen
        }
    }
    return true;                         // No check at all when frozenTokensLocal == 0
}
```

Combined with:

```
// contracts/modules/1_CMTATBaseAllowlist.sol (excerpt)
if(!_canTransferGenericByModule(...)) return false;
else if(!ERC20EnforcementModuleInternal._checkActiveBalance(from, value)) return false;
else return ValidationModuleCore.canTransfer(...); // does not check balances
```

Breaking scenario: from has balance 10, frozenTokens(from) == 0, value = 20; addresses pass module checks (allowlist/freeze/pause). Then:
- _checkActiveBalance(from, 20) returns true (because frozenTokens == 0)
- ValidationModule._canTransferGenericByModule(...) returns true
- canTransfer/canTransferFrom return true
- Actual transfer/transferFrom reverts due to ERC-20 insufficient balance

Thus, the pre-flight result does not match the outcome of the transfer, violating Invariant 4.

9 of 14 Findings

contracts/modules/3_CMTATBaseERC20CrossChain.sol

contracts/modules/5_CMTATBaseERC1363.sol

SnapshotEngine hook is bypassed in _update (direct call to CMTATBaseCommon._update)

Low Risk

The _update override in CMTATBaseERC20CrossChain hard-calls CMTATBaseCommon._update instead of chaining via super. This bypasses any intermediate _update overrides (e.g., in CMTATBaseRuleEngine) that are supposed to invoke the SnapshotEngine before balances/supply are modified.

```
// contracts/modules/3_CMTATBaseERC20CrossChain.sol
function _update(
    address from,
    address to,
    uint256 amount
) internal virtual override(ERC20Upgradeable, CMTATBaseCommon) {
   return CMTATBaseCommon._update(from, to, amount); // hard jump to Common
}
```

The ERC-1363 variant forwards to the same implementation, propagating the bypass:

```
// contracts/modules/5_CMTATBaseERC1363.sol
function _update(
    address from,
    address to,
    uint256 amount
) internal override(ERC20Upgradeable, CMTATBaseERC20CrossChain) {
    CMTATBaseERC20CrossChain._update(from, to, amount);
}
```

Invariant impact: If a snapshotEngine is configured, the expected call `snapshotEngine.operateOnTransfer(from, to, balance_from_pre, balance_to_pre, total_supply_pre)` (performed in the RuleEngine-layer _update before state changes) never executes for mints, burns, or transfers, because execution jumps directly to CMTATBaseCommon._update where balances/supply are modified. Therefore the required pre-state snapshot is not taken, violating the invariant tied to the RuleEngine-layer pre-update hook.

Severity Note:
- SnapshotEngine is not enforcing hard on-chain restrictions and is primarily used for pre-state recording/snapshotting.
- No downstream on-chain logic critically depends on SnapshotEngine firing to maintain core safety invariants (e.g., preventing transfers).

10 of 14 Findings

contracts/modules/3_CMTATBaseERC20CrossChain.sol

RuleEngine spender is hardcoded to address(0) for minter-initiated transfers (and for mint/burn), weakening compliance checks that depend on spender identity

Low Risk

In CMTATBaseERC20CrossChain, the RuleEngine is invoked via `_checkTransferred` with `spender = address(0)` for several balance-changing operations, including the special minter transfer path. Examples:

```
// Mint path
function _mintOverride(address account, uint256 value)
    internal virtual override(CMTATBaseCommon, ERC20MintModuleInternal)
{
    _checkTransferred(address(0), address(0), account, value);
    ERC20MintModuleInternal._mintOverride(account, value);
}

// Burn path
function _burnOverride(address account, uint256 value)
    internal virtual override(CMTATBaseCommon, ERC20BurnModuleInternal)
{
    _checkTransferred(address(0), account, address(0), value);
    ERC20BurnModuleInternal._burnOverride(account, value);
}

// Minter-initiated transfer path
function _minterTransferOverride(address from, address to, uint256 value)
    internal virtual override(CMTATBaseCommon, ERC20MintModuleInternal)
{
    _checkTransferred(address(0), from, to, value);
    ERC20MintModuleInternal._minterTransferOverride(from, to, value);
}
```

Invariant 0 requires that when a RuleEngine is configured, every balance change (transfer, mint, burn) is validated against external compliance rules. The IRuleEngine/IERC7551Compliance interfaces expose `canTransferFrom(spender, from, to, value)` / `transferred(spender, from, to, value)` semantics where the `spender` parameter identifies the actor initiating the movement. By passing `address(0)` instead of the real actor for the minter transfer (and for mint/burn), any compliance logic that relies on the actual `spender` (e.g., KYC/role- or jurisdiction-based checks on the initiating party) cannot be correctly enforced.

Concrete scenario: an address with MINTER permissions uses the minter transfer function to move tokens from `from` to `to`. If the RuleEngine policy would normally restrict that minter from performing such a transfer (based on `spender`), the check is neutralized because it receives `spender = address(0)`. Thus, the movement can be accepted when it should be rejected under spender-aware policies. This violates the intent of Invariant 0 that “all token movements are validated against the external compliance rules,” because a critical input to those rules (the spender’s identity) is dropped during validation.

Severity Note:
- A RuleEngine is configured and actively consulted for these operations.
- The RuleEngine enforces spender-aware rules for mint/burn/minter-transfer (e.g., different allowances by

caller).
- address(0) is not treated as a strict deny in the RuleEngine, or its use materially weakens the intended policy.

11 of 14 Findings

contracts/modules/wrapper/extensions/ERC20EnforcementModule.sol

contracts/modules/internal/ERC20EnforcementModuleInternal.sol

contracts/modules/1_CMTATBaseAllowlist.sol

contracts/modules/1_CMTATBaseRuleEngine.sol

contracts/modules/wrapper/controllers/ValidationModule.sol

contracts/modules/wrapper/controllers/ValidationModuleAllowlist.sol

Forced transfers still enforced by standard validation (freeze/allowlist/pause), defeating enforcement intent

Low Risk

Forced transfer/burn paths do not bypass the generic validation checks. The internal enforcement function performs a normal ERC20 transfer/burn after unfreezing tokens, which triggers the token's _checkTransferred hook that enforces generic restrictions (pause, account freeze, allowlist). This contradicts comments indicating forced operations should be used to move/burn tokens from frozen or non-allowlisted accounts.

Key paths:

1) Forced transfer implementation calls standard _transfer/_burn:

```
// contracts/modules/internal/ERC20EnforcementModuleInternal.sol
function _forcedTransfer(address from, address to, uint256 value, bytes memory data)
internal virtual {
    _unfreezeTokens(from, value, data);
    if(to == address(0)){
        _burn(from, value);
    } else{
        uint256 currentAllowance = allowance(from, to);
        if (currentAllowance > 0 && currentAllowance < type(uint256).max) {
            if (currentAllowance < value) {
                unchecked { _approve(from, to, 0, false); }
            } else{
                unchecked { _approve(from, to, currentAllowance - value, false); }
            }
        }
        _transfer(from, to, value);
    }
    emit Enforcement(_msgSender(), from, value, data);
}
```

2) Post-transfer hook enforces generic validation and reverts if it fails:

```
// contracts/modules/1_CMTATBaseAllowlist.sol
function _checkTransferred(address spender, address from, address to, uint256 value)
internal virtual override(CMTATBaseCommon) {
    CMTATBaseCommon._checkTransferred(spender, from, to, value);
    if (!ValidationModule._canTransferGenericByModule(spender, from, to)) {
        revert Errors.CMTAT_InvalidTransfer(from, to, value);
    }
}

// contracts/modules/1_CMTATBaseRuleEngine.sol
function _checkTransferred(address spender, address from, address to, uint256 value)
internal virtual override(CMTATBaseCommon) {
    CMTATBaseCommon._checkTransferred(spender, from, to, value);
    require(ValidationModuleRuleEngine._transferred(spender, from, to, value),
Errors.CMTAT_InvalidTransfer(from, to, value));
}
```

3) Generic validation blocks transfers if paused or any participant is frozen; and Allowlist extension additionally requires spender/from/to to be allowlisted:

```
// contracts/modules/wrapper/controllers/ValidationModule.sol
function _canTransferGenericByModule(address spender, address from, address to) internal
view virtual returns (bool) {
    if(from == address(0)){
        return _canMintBurnByModule(to);
    } else if(to == address(0)){
        return _canMintBurnByModule(from);
    } else {
        return _canTransferStandardByModule(spender, from, to);
    }
}
function _canMintBurnByModule(address target) internal view virtual returns (bool) {
    if(PauseModule.deactivated() || EnforcementModule.isFrozen(target)){
        return false;
    }
    return true;
}
function _canTransferStandardByModule(address spender, address from, address to) internal
view virtual returns (bool) {
    if (EnforcementModule.isFrozen(spender) || EnforcementModule.isFrozen(from) ||
EnforcementModule.isFrozen(to) || PauseModule.paused())  {
        return false;
    } else {
        return true;
    }
}

// contracts/modules/wrapper/controllers/ValidationModuleAllowlist.sol
function _canMintBurnByModule(address target) internal view virtual
override(ValidationModule) returns (bool) {
    if(_isAllowlistEnabled() && !isAllowlisted(target)){
        return false;
    } else {
        return ValidationModule._canMintBurnByModule(target);
    }
}
function _canTransferStandardByModule(address spender, address from, address to) internal
view virtual override(ValidationModule) returns (bool) {
    if(_isAllowlistEnabled()){
        bool spenderCheck = spender != address(0) && !isAllowlisted(spender);
        if (spenderCheck || !isAllowlisted(from) || !isAllowlisted(to)){
            return false;
        }
    }
    return ValidationModule._canTransferStandardByModule(spender, from, to);
}
```

Effect:
- Forced burn from a frozen or non-allowlisted address will revert because _canMintBurnByModule(...) returns false for frozen/non-allowlisted targets, and the post-transfer hook reverts the operation.
- Forced transfer between addresses where any party is frozen or not allowlisted, or when paused, will also revert due to _canTransferStandardByModule(...).

This defeats the intended purpose of administrative enforcement (e.g., regulatory recovery), risks permanently locking funds on frozen/non-allowlisted accounts, and contradicts inline comments like:

- "cannot burn if target is frozen (used forcedTransfer instead if available)" and
- "Use forcedTransfer (or forcedBurn) to burn tokens from an non-allowlist address".

Severity Note:
- The token’s transfer/burn hooks ultimately invoke _checkTransferred as shown, so _transfer/_burn during forcedTransfer are subject to ValidationModule checks.

12 of 14 Findings

contracts/modules/wrapper/controllers/ValidationModule.sol

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

Inconsistent deactivation handling: canTransfer() can return true while ERC-1404 detectTransferRestriction() reports deactivated

Low Risk

Standard transfer permission in ValidationModule does not consider the deactivated() state, but ERC-1404 restriction reporting does. This can yield contradictory outcomes where compliance checks (canTransfer) return true, yet ERC-1404 returns a non-zero restriction code indicating the contract is deactivated.

Relevant snippets:

```
// contracts/modules/wrapper/controllers/ValidationModule.sol
function _canTransferStandardByModule(address spender, address from, address to)
    internal view returns (bool)
{
    if (EnforcementModule.isFrozen(spender)
        || EnforcementModule.isFrozen(from)
        || EnforcementModule.isFrozen(to)
        || PauseModule.paused())
    {
        return false;
    } else {
        return true; // deactivated() is not checked here
    }
}
```

```
// contracts/modules/wrapper/extensions/ValidationModule/ValidationModuleERC1404.sol
function _detectTransferRestriction(address from, address to, uint256) internal view returns
(uint8 code) {
    if (deactivated()) {
        return uint8(IERC1404Extend.REJECTED_CODE_BASE.TRANSFER_REJECTED_DEACTIVATED);
    } else if (paused()) { /* ... */ }
    // ...
}
```

Impact: when deactivated()==true and the contract is not paused and no participant is frozen, _canTransferStandardByModule() returns true, so canTransfer() may return true, but detectTransferRestriction() returns TRANSFER_REJECTED_DEACTIVATED. This violates the invariant that detectTransferRestriction must return TRANSFER_OK (0) iff canTransfer returns true, and can mislead integrators relying on consistent pre-transfer checks.

Severity Note:
- Actual transfer execution enforces restrictions independently (e.g., via ERC-1404 gating or other checks) and does not solely rely on canTransfer() to block transfers when deactivated().
- No hidden code path makes deactivated() the only intended hard-stop for standard transfers.

13 of 14 Findings

contracts/deployment/CMTATStandalone.sol

Approve function not protected by pause modifier, allowing allowance changes when contract is paused

Low Risk

Invariant 3 explicitly states: "When the contract is paused, all standard ERC-20 transfer and approval functions, such as `transfer`, `transferFrom`, and `approve`, must be disabled. Any attempt to call these functions should not alter token balances or allowances."

The invariant condition specifies:

```
if (old(paused())) {
    // After a call to approve(spender, value) from `from`
    assert(allowance(from, spender) == old(allowance(from, spender)));
}
```

However, the `approve()` function inherits from OpenZeppelin's ERC20 implementation without any pause protection:

```
// OpenZeppelin ERC20.approve() - no pause check
function approve(address spender, uint256 value) public virtual returns (bool) {
    address owner = _msgSender();
    _approve(owner, spender, value);  // No whenNotPaused modifier
    return true;
}
```

While `transfer()` and `transferFrom()` are protected via `_checkTransferred()` which includes a pause check:

```
// In ValidationModuleCore._checkTransferred()
function _checkTransferred(address spender, address from, address to, uint256 value)
internal view {
    if (paused()) {
        revert CMTAT_InvalidTransfer(spender, from, to, value);
    }
    // ...
}
```

The `approve()` function does not call `_checkTransferred()` or any other function that validates pause state.

Impact:
- Users can modify allowances even when the contract is paused
- Violates the stated invariant and security expectations
- While paused transfers cannot execute, attackers could still set up allowances that become active once the contract is unpaused
- Inconsistent pause behavior across ERC-20 functions creates confusion and potential security gaps

14 of 14 Findings

contracts/modules/wrapper/options/ERC2771Module.sol

contracts/deployment/CMTATUpgradeable.sol

contracts/deployment/CMTATUpgradeableUUPS.sol

contracts/deployment/ERC1363/CMTATUpgradeableERC1363.sol

contracts/deployment/ERC7551/CMTATUpgradeableERC7551.sol

contracts/deployment/allowlist/CMTATUpgradeableAllowlist.sol

ERC2771 forwarder is set via constructor in upgradeable deployments, leaving proxy storage uninitialized and breaking _msgSender semantics

Low Risk

In upgradeable variants, the trusted forwarder for ERC-2771 is configured in the implementation constructor, which does not run in the proxy context. As a result, the proxy's storage never receives the trusted forwarder value and `_msgSender()` will behave as if no forwarder is set.

Relevant snippets:

```
// contracts/modules/wrapper/options/ERC2771Module.sol
abstract contract ERC2771Module is ERC2771ContextUpgradeable {
    constructor(address trustedForwarder) ERC2771ContextUpgradeable(trustedForwarder) {
        // Nothing to do
    }
}
```

```
// contracts/deployment/CMTATUpgradeable.sol
constructor(address forwarderIrrevocable) ERC2771Module(forwarderIrrevocable) {
    _disableInitializers();
}
```

```
// contracts/deployment/CMTATUpgradeableUUPS.sol
constructor(address forwarderIrrevocable) ERC2771Module(forwarderIrrevocable) {
    _disableInitializers();
}
```

Because constructors execute on the implementation contract, the assignment occurs in the implementation's storage instead of the proxy's storage. Calls routed through the proxy will observe an unset (zero) trusted forwarder, meaning relayed calls will be treated as regular calls. This causes inconsistent `_msgSender()` behavior across standalone vs. proxy deployments and can unintentionally break role-gated flows that are expected to work with meta-transactions. If operators compensate by granting roles to the relayer itself, that relayer gains broad on-chain authority, increasing operational risk.

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.