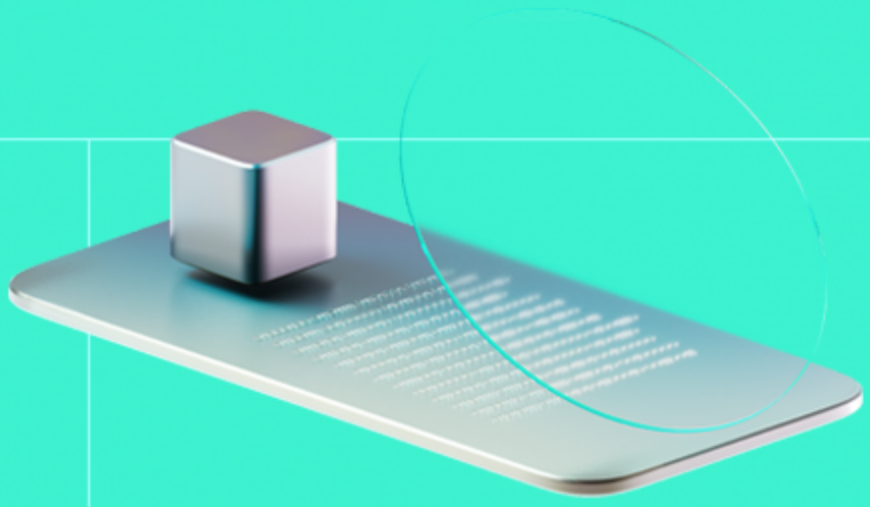
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: Newrails

Date: 12/03/2026

We express our gratitude to the Newrails team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

`Stablecoin` is an upgradeable, role-controlled ERC-20 stablecoin with configurable decimals, per-minter mint quotas, pausing/blacklisting enforcement, EIP-2612 permit, and EIP-712 authorization-based transfer/receive/burn flows plus token/ETH rescue functions.

Document

Name	Smart Contract Code Review and Security Analysis Report for Newrails
Audited By	Kornel Światłowski
Approved By	Seher Saylik
Website	www.newrails.xyz
Changelog	23/02/2026 - Preliminary Report
	12/03/2026 - Final Report
Platform	Monad
Language	Solidity
Tags	ERC20, Signatures, Upgradable, Permit
Methodology	https://docs.hacken.io/methodologies/smart-contracts

Review Scope

Repository	https://github.com/Newrails-xyz/NewrailsEMT-EVM
Commit	8fbd0eb24548926cd98afb0f985d46be3eee9579
Final Commit	8fbd0eb24548926cd98afb0f985d46be3eee9579

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

9	0	9	0
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	0
Medium	0
Low	4

Vulnerability	Severity	Status
F-2026-15201 - Improper Allowance Update Allows Mint Of Unauthorized Excessive Token	Low	Accepted
F-2026-15205 - Blacklist Does Not Restrict Privileged Contract Interactions	Low	Accepted
F-2026-15211 - Inconsistent Blacklist Enforcement Across Burn Functions	Low	Accepted
F-2026-15215 - Redundant receive Function	Low	Accepted
F-2026-15207 - Floating Pragma	Info	Accepted
F-2026-15210 - Missing Validation of Decimal Value During Initialization	Info	Accepted
F-2026-15219 - Public Visibility Used for Functions Intended for External Calls	Info	Accepted
F-2026-15220 - Redundant Inheritance of Initializable and ERC20Upgradeable	Info	Accepted
F-2026-15221 - Blacklisted Addresses Can Act as Relayers in Signature-Based Transfers	Info	Accepted

Documentation quality

- Functional requirements are not provided.
  - No roles description.
  - No documentation.
  - No Readme.md file or any other file with instructions or description.
  - No whitepaper.
  - No Tokenomics.
  - NatSpec is sufficient.
- Technical description is limited.
  - NatSpec is sufficient.
  - No run instructions.
  - No technical specification.

Code quality

- The development environment is configured.
- NatSpec covers the code and is detailed.

Test coverage

Code coverage of the project is **57.14%** (branch coverage).

- Deployment and basic user interactions are thoroughly tested.
- Negative case coverage is missed.

Table of Contents

System Overview

Stablecoin - Upgradeable ERC-20 token intended to operate as a stablecoin. It uses OpenZeppelin upgradeable modules (ERC20, Permit, Pausable, AccessControl, UUPS) and adds custom features on top: configurable decimals, per-minter mint quotas controlled by a separate approver role, and multiple administrative controls (pause/unpause, blacklist management, upgrade authorization, and rescue operations). Blacklisting and pausing are enforced at the token primitive level (transfer/mint/burn updates and approvals), so restrictions apply consistently across regular ERC20 operations, transferFrom, and signature-based authorization flows. The contract also implements EIP-712 authorization-based actions (transferWithAuthorization, receiveWithAuthorization, burnFromWithAuthorization) to enable gasless-style transfers/burns with one-time nonces and validity windows.

Blacklistable - Blacklist module with dedicated storage slot, isBlacklisted() query, and notBlacklisted(address) modifier used to block blacklisted participants in token flows.

Roles - Library defining bytes32 constants for the roles used by Stablecoin.

Privileged roles

Stablecoin uses an **AccessControlUpgradeable** contract from OpenZeppelin to implement role-based access control. It defines the following roles with these capabilities:

- **ADMIN_ROLE** can:
  - `grantRole` / `revokeRole`: manage role assignments (as role admin)
- **MINTER_APPROVER_ROLE** can:
  - `approveMinter`: set/update a minter's remaining mint allowance (quota)
- **MINTER_ROLE** can:
  - `mint`: mint tokens to a recipient and decrease minter allowance
- **BURNER_ROLE** can:
  - `burn`: burn tokens from caller balance
  - `burnFromWithAuthorization`: burn tokens from from via EIP-712 authorization
- **PAUSER_ROLE** can:
  - `pause`: pause transfers and approvals
  - `unpause`: unpause transfers and approvals
- **BLACKLISTER_ROLE** can:
  - `blacklist`: add an account to the blacklist
  - `unblacklist`: remove an account from the blacklist
- **RESCUER_ROLE** can:
  - `rescueERC20`: recover ERC-20 tokens held by the contract
  - `rescueETH`: recover native ETH held by the contract
- **UPGRADER_ROLE** can:
  - `upgradeToAndCall`: perform UUPS upgrade (via `_authorizeUpgrade`)

Potential Risks

- Centralized Control of Minting Process: The token contract’s design allows for centralized control over the minting process, posing a risk of unauthorized token issuance, potentially diluting the token value and undermining trust in the project's economic governance.
- Insufficient Multi-signature Controls for Critical Functions: The lack of multi-signature requirements for key operations centralizes decision-making power, increasing vulnerability to single points of failure or malicious insider actions, potentially leading to unauthorized transactions or configuration changes.
- Flexibility and Risk in Contract Upgrades: The project's contracts are upgradable, allowing the administrator to update the contract logic at any time. While this provides flexibility in addressing issues and evolving the project, it also introduces risks if upgrade processes are not properly managed or secured, potentially allowing for unauthorized changes that could compromise the project's integrity and security.
- Absence of Upgrade Window Constraints: The contract suite allows for immediate upgrades without a mandatory review or waiting period, increasing the risk of rapid deployment of malicious or flawed code, potentially compromising the system's integrity and user assets.

Findings

Vulnerability Details

F-2026-15201 - Improper Allowance Update Allows Mint Of Unauthorized Excessive Token - Low

Description:

The minting of new ERC20 tokens requires the caller to hold the `MINTER_ROLE`, the destination address not to be blacklisted, and the minter to have sufficient remaining allowance. The minter allowance is granted by addresses holding the `MINTER_APPROVER_ROLE` through the `approveMinter()` function, which sets a new allowance value for a specified minter address.

```
function approveMinter(address minter, uint256 newAllowance) external onlyRol
e(Roles.MINTER_APPROVER_ROLE) {
    if (minter == address(0)) {
        revert InvalidAddress();
    }
    _minterAllowance[minter] = newAllowance;

    emit MinterApproved(minter, newAllowance);
}

function mint(address to, uint256 amount) external onlyRole(Roles.MINTER_ROLE
) notBlacklisted(to) {
    address caller = _msgSender();
    uint256 available = _minterAllowance[caller];
    if (amount > available) {
        revert MinterAllowanceExceeded(amount, available);
    }

    _minterAllowance[caller] = available - amount;
    _mint(to, amount);
    emit Mint(caller, to, amount);
}
```

The `approveMinter()` function overwrites the existing allowance with the provided `newAllowance` value. Because both `mint()` and `approveMinter()` are publicly callable transactions, a front-running scenario is introduced. A minter with a non-zero existing allowance can observe a pending `approveMinter()` transaction in the public mempool and submit a higher-priority `mint()` transaction to consume the remaining allowance before the approval is executed.

In such a scenario, the allowance is reduced to zero by the frontrun `mint()` call. When the `approveMinter()` transaction is later executed, the allowance is reset to the new value, effectively allowing the minter to mint more assets than intended.

For example, if a minter has an allowance of 100 and the allowance is intended to be increased to 200, the minter can front-run the approval and mint 100 assets first. Afterward, the approval transaction sets the allowance to 200, enabling the minting of an additional 200 assets. As a result, a total of 300 assets can be minted instead of the intended 200.

Assets:

- src/Stablecoin.sol [https://github.com/Newrails-xyz/NewrailsEMT-EVM]

Status: Accepted

Classification

Impact Rate: 4/5

Likelihood Rate: 2/5

Exploitability: Semi-Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation: **It is recommended to** prevent allowance overwrite front-running by either:

- introducing additive allowance adjustment functions (such as explicit increase and decrease mechanisms) via `increaseAllowance` and `decreaseAllowance`, ensuring allowance changes are applied relative to the current value rather than overwriting it; or
- enforcing allowance updates through a dedicated mint pause mechanism that temporarily disables minting while allowance changes are applied, without affecting unrelated contract functionality.

Resolution: The issue was accepted by the NewRails team.

F-2026-15205 - Blacklist Does Not Restrict Privileged Contract Interactions - Low

Description:

The `Stablecoin` contract supports blacklisting of addresses to prevent interaction with ERC20 assets. A blacklisted address is prevented from transferring, granting allowance, spending allowance, or receiving ERC20 assets. The contract also defines multiple administrative roles that grant privileged access to sensitive functionality.

The current role assignment logic allows administrative roles to be granted to addresses that are already blacklisted, and it also allows privileged roles to be blacklisted without revoking their permissions. This creates an inconsistent state in which an address marked as restricted from token interactions can still retain administrative privileges and interact with the `Stablecoin` contract through privileged functions.

A blacklisted address holding an administrative role can still perform the following actions:

- Receive minting allowance
- Grant minting allowance
- Mint new tokens
- Burn tokens via `burn()` and `burnFromWithAuthorization()`
- Transfer ERC20 assets via `rescueERC20()`
- Transfer native assets from the contract
- Transfer tokens via `transferWithAuthorization()`

This behavior undermines the intended guarantees of the blacklist mechanism. Addresses expected to be fully restricted may continue to exercise high-impact administrative capabilities, leading to unexpected or undesirable system behavior.

Assets:

- src/Blacklistable.sol [https://github.com/Newrails-xyz/NewrailsEMT-EVM]
- src/Stablecoin.sol [https://github.com/Newrails-xyz/NewrailsEMT-EVM]

Status: Accepted

Classification

Impact Rate: 3/5

Likelihood Rate: 2/5

Exploitability: Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation: **It is recommended to:**

- Enforce blacklist checks during role assignment by overriding the `_grantRole()` function and validating that the target address is not blacklisted before granting any privileged role.
- Add a validation check in the blacklist flow to prevent blacklisting of addresses that hold privileged roles. Privileged roles should be revoked by an address holding `ADMIN_ROLE` before an address can be blacklisted.

Resolution: The issue was accepted by the NewRails team.

Evidences

PoC

Reproduce:

```
function test_grant_minter_role_to_blacklisted_address() public {
    vm.prank(blacklister);
    stable.blacklist(user3);

    console.log("hasRole(Roles.MINTER_ROLE, user3): ", stable.hasRole(Roles.M
INTER_ROLE, user3));
    console.log("isBlacklisted(user3):              ", stable.isBlacklisted(u
ser3));
    assertEq(stable.hasRole(Roles.MINTER_ROLE, user3), false);
    assertTrue(stable.isBlacklisted(user3));

    vm.prank(admin);
    stable.grantRole(Roles.MINTER_ROLE, user3);

    assertEq(stable.hasRole(Roles.MINTER_ROLE, user3), true);
    assertTrue(stable.isBlacklisted(user3));
    console.log("--------------------------------");
    console.log("After granting minter role to blacklisted address");
    console.log("hasRole(Roles.MINTER_ROLE, user3): ", stable.hasRole(Roles.M
INTER_ROLE, user3));
    console.log("isBlacklisted(user3):              ", stable.isBlacklisted(u
ser3));
}
```

Results:

```
[PASS] test_grant_minter_role_to_blacklisted_address() (gas: 98071)
Logs:
  hasRole(Roles.MINTER_ROLE, user3):  false
  isBlacklisted(user3):               true
  --------------------------------
  After granting minter role to blacklisted address
  hasRole(Roles.MINTER_ROLE, user3):  true
  isBlacklisted(user3):               true
```

F-2026-15211 - Inconsistent Blacklist Enforcement Across Burn Functions - Low

Description:

The `BURNER_ROLE` is authorized to invoke two burn functions: `burn()` and `burnFromWithAuthorization()`. The `burn()` function burns assets from the caller address, while `burnFromWithAuthorization()` burns assets from a specified address, provided a valid authorization signature is supplied.

The `burnFromWithAuthorization()` function enforces a blacklist check, requiring the source address not to be blacklisted. As a result, asset burning through this flow is prevented for blacklisted addresses. However, the same blacklist validation is not enforced in the `burn()` function.

```
function burn(uint256 amount) external onlyRole(Roles.BURNER_ROLE) {
    address caller = _msgSender();
    _burn(caller, amount);
    emit Burn(caller, caller, amount);
}

function burnFromWithAuthorization(...) external onlyRole(Roles.BURNER_ROLE)
notBlacklisted(from) {
    _useAuthorization(from, nonce, validAfter, validBefore,
        keccak256(abi.encode(
            BURN_FROM_WITH_AUTHORIZATION_TYPEHASH, from, value, validAfter, v
alidBefore, nonce)),
        signature);
    _burn(from, value);
    emit Burn(_msgSender(), from, value);
}
```

An address holding the `BURNER_ROLE` may itself be marked as blacklisted. In such a state, asset burning via `burn()` remains possible despite blacklist restrictions being applied in other burn flows. This results in inconsistent enforcement of blacklist rules across burn operations and creates ambiguity in the intended restrictions applied to blacklisted addresses.

Assets:

- src/Stablecoin.sol [https://github.com/Newrails-xyz/NewrailsEMT-EVM]

Status: Accepted

Classification

Impact Rate: 3/5

Likelihood Rate: 2/5

Exploitability: Semi-Dependent

Complexity: Simple

The issue was accepted by the NewRails team.

Severity: Low

Recommendations

Remediation: **It is recommended to** enforce blacklist validation consistently across all burn-related functions by applying the same blacklist checks in `burn()` as those enforced in `burnFromWithAuthorization()`, ensuring uniform behavior for blacklisted addresses.

F-2026-15215 - Redundant receive Function - Low

Description:

The `Stablecoin` contract includes an empty `receive()` function, allowing the contract to accept native assets. The contract logic does not use native assets for any operational purpose, as all functionality is based exclusively on ERC20 assets.

```
receive() external payable {}
```

The presence of the `receive()` function enables native assets to be transferred to the contract unintentionally. Such transfers are not tracked, processed, or used by the contract logic and cannot be recovered through standard user interactions. Recovery of these assets requires execution of a privileged administrative function.

If the `receive()` function is removed, the contract will no longer accept native assets through standard transfer mechanisms. In this case, the only remaining way for native assets to be transferred to the contract would be via a `selfdestruct` operation from another contract. Under this design, the `rescueETH()` function becomes redundant, as native asset transfers would no longer be supported as part of the intended contract behavior.

This design introduces unnecessary risk of asset lockup, increases reliance on administrative intervention, and may cause confusion regarding supported asset types and expected contract behavior.

Assets:

- src/Stablecoin.sol [https://github.com/Newrails-xyz/NewrailsEMT-EVM]

Status: Accepted

Classification

Impact Rate: 3/5

Likelihood Rate: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: **It is recommended to** remove the `receive()` function to prevent accidental native asset deposits and clearly restrict the contract to ERC20-only asset handling. As a consequence, the `rescueETH()` function should also be removed, eliminating unnecessary administrative recovery logic and reducing the risk of unintended asset lockup.

Resolution: The issue was accepted by the NewRails team.

F-2026-15207 - Floating Pragma - Info

Description:

In Solidity development, the pragma directive specifies the compiler version to be used, ensuring consistent compilation and reducing the risk of issues caused by version changes. However, using a floating pragma (e.g., `^0.8.xx`) introduces uncertainty, as it allows contracts to be compiled with any version within a specified range. This can result in discrepancies between the compiler used in testing and the one used in deployment, increasing the likelihood of vulnerabilities or unexpected behavior due to changes in compiler versions.

The project currently uses floating pragma declarations (`^0.8.24`) in its Solidity contracts. This increases the risk of deploying with a compiler version different from the one tested, potentially reintroducing known bugs from older versions or causing unexpected behavior with newer versions. These inconsistencies could result in security vulnerabilities, system instability, or financial loss. Locking the pragma version to a specific, tested version is essential to prevent these risks and ensure consistent contract behavior.

Assets:

- src/Blacklistable.sol [https://github.com/Newrails-xyz/NewrailsEMT-EVM]
- src/Stablecoin.sol [https://github.com/Newrails-xyz/NewrailsEMT-EVM]
- src/libraries/Roles.sol [https://github.com/Newrails-xyz/NewrailsEMT-EVM]

Status: Accepted

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: It is recommended to **lock the pragma version** to the specific version that was used during development and testing. This ensures that the contract will always be compiled with a known, stable compiler version, preventing

unexpected changes in behavior due to compiler updates. For example, instead of using `^0.8.24`, explicitly define the version with `pragma solidity 0.8.24;`.

Before selecting a version, review known bugs and vulnerabilities associated with each Solidity compiler release. This can be done by referencing the official Solidity compiler release notes: Solidity GitHub releases or Solidity Bugs by Version. Choose a compiler version with a good track record for stability and security.

Resolution: The issue was accepted by the NewRails team.

F-2026-15210 - Missing Validation of Decimal Value During Initialization - Info

Description:

The Stablecoin contract accepts multiple parameters during initialization, including the token name, symbol, and decimal precision. The `decimals_` argument is not sufficiently validated and may be set to `0`.

```
function initialize(
    string memory name_, string memory symbol_, uint8 decimals_, address admi
n) public initializer {
    if (admin == address(0)) {
        revert InvalidAddress();
    }
    // ...

    _tokenDecimals = decimals_;

    // ...
}
```

A zero-decimal configuration may lead to unexpected behavior in downstream integrations. External contracts and off-chain systems commonly assume a non-zero decimal value when performing arithmetic operations, balance normalization, or display logic. Setting the decimal precision to zero may therefore result in incorrect calculations, rounding issues, or integration failures.

Assets:

- src/Stablecoin.sol [https://github.com/Newrails-xyz/NewrailsEMT-EVM]

Status: Accepted

Classification

Impact Rate: 3/5

Likelihood Rate: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: **It is recommended to** enforce validation of the `decimals_` parameter during initialization by requiring a non-zero and reasonable decimal value to prevent misconfiguration and integration inconsistencies.

Resolution: The issue was accepted by the NewRails team.

F-2026-15219 - Public Visibility Used for Functions Intended for External Calls - Info

Description:

The `initialize()` function in the `Stablecoin` contract is declared with `public` visibility but is not invoked internally.

In Solidity, functions intended solely for external invocation should be declared as `external` rather than `public`. While both visibility specifiers allow functions to be called from outside the contract, `public` functions can also be invoked internally, whereas `external` functions cannot.

When called externally, `external` functions are more gas-efficient than `public` functions because they avoid unnecessary internal memory copying. Declaring such functions as `public` introduces avoidable gas overhead and represents a suboptimal visibility choice.

Assets:

- src/Stablecoin.sol [https://github.com/Newrails-xyz/NewrailsEMT-EVM]

Status: Accepted

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: **It is recommended to** change the visibility of functions that are not invoked internally from `public` to `external`, ensuring more appropriate access control semantics and improving gas efficiency for external calls.

Resolution: The issue was accepted by the NewRails team.

F-2026-15220 - Redundant Inheritance of Initializable and ERC20Upgradeable - Info

Description:

The `Stablecoin` contract redundantly inherits from `Initializable` and `ERC20Upgradeable`, despite these base contracts already being included indirectly through other inherited OpenZeppelin upgradeable modules.

```
contract Stablecoin is
    Initializable,
    ERC20Upgradeable,
    ERC20PausableUpgradeable,
    ERC20PermitUpgradeable,
    AccessControlUpgradeable,
    UUPSUpgradeable,
    Blacklistable
{
    ...
}
```

`Initializable` is already inherited via multiple upgradeable parent contracts, such as `AccessControlUpgradeable` and `UUPSUpgradeable`. Similarly, `ERC20Upgradeable` is inherited transitively because both `ERC20PausableUpgradeable` and `ERC20PermitUpgradeable` extend `ERC20Upgradeable`.

This redundant inheritance does not introduce an immediate security vulnerability. However, it increases the complexity of the inheritance graph, raises cognitive overhead during code review, and may increase the risk of errors during future refactors or upgrades. In particular, redundant base classes can complicate method resolution order, override behavior, and expectations around contract linearization.

Assets:

- src/Stablecoin.sol [https://github.com/Newrails-xyz/NewrailsEMT-EVM]

Status: Accepted

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: **It is recommended to** remove redundant base contract inheritance and rely solely on the transitive inheritance provided by the required OpenZeppelin upgradeable modules, simplifying the inheritance hierarchy and reducing the risk of future upgrade or override inconsistencies.

Resolution: The issue was accepted by the NewRails team.

F-2026-15221 - Blacklisted Addresses Can Act as Relayers in Signature-Based Transfers - Info

Description:

The blacklist mechanism is enforced on token movement and allowance usage by applying `notBlacklisted` checks to the `from`, `to`, and `spender` addresses through overrides of `_update`, `_approve`, and `_spendAllowance`. This design ensures that blacklisted addresses cannot directly send, receive, or spend ERC20 assets.

However, the blacklist does not restrict the transaction caller (`msg.sender`) when invoking signature-based flows such as `transferWithAuthorization(...)`, `burnFromWithAuthorization(…)` and `receiveWithAuthorization(...)`, provided that `msg.sender` is not the `from`, `to`, or `spender` address being validated. As a result, a blacklisted address may still submit transactions as a relayer, executing authorized transfers on behalf of non-blacklisted signers.

This behavior allows blacklisted addresses to interact with the contract by facilitating authorized token movements, even though direct token interaction is restricted. If the intended blacklist policy is to prevent blacklisted addresses from interacting with the contract in any capacity, this represents an enforcement gap. If the intended policy is limited to preventing blacklisted addresses from sending, receiving, or spending assets, then the current behavior is consistent with that design.

Assets:

- src/Stablecoin.sol [https://github.com/Newrails-xyz/NewrailsEMT-EVM]

Status: Accepted

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation: **It is recommended to** explicitly define and enforce the intended scope of the blacklist policy by either extending blacklist checks to `msg.sender` in

signature-based execution paths or clearly documenting that relayer participation by blacklisted addresses is permitted when they are not token participants in the transfer.

Resolution: The issue was accepted by the NewRails team.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

As part of Hacken's ongoing quality assurance process, we may conduct re-audits of select projects. These re-audits are performed independently from the original audit and are intended solely for internal quality control and improvement. Updated reports resulting from such re-audits will be shared privately with the respective clients and may be published on the Hacken website only with their explicit consent.
The sole authoritative source for finalized and most up-to-date versions of all reports remains the Audits section at https://hacken.io/audits/.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Definitions

Severities

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution.

Potential Risks

The "Potential Risks" section identifies issues that are not direct security vulnerabilities but could still affect the project's performance, reliability, or user trust. These risks arise from design choices, architectural decisions, or operational practices that, while not immediately exploitable, may lead to problems under certain conditions. Additionally, potential risks can impact the quality of the audit itself, as they may involve external factors or components beyond the scope of the audit, leading to incomplete assessments or oversight of key areas. This section aims to provide a broader perspective on factors that could affect the project's long-term security, functionality, and the comprehensiveness of the audit findings.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details	
Repository	https://github.com/Newrails-xyz/NewrailsEMT-EVM
Commit	8fbd0eb24548926cd98afb0f985d46be3eee9579
Final Commit	8fbd0eb24548926cd98afb0f985d46be3eee9579
Whitepaper	-
Requirements	-
Technical Requirements	-

Asset	Type
src/Blacklistable.sol [https://github.com/Newrails-xyz/NewrailsEMT-EVM]	Smart Contract
src/libraries/Roles.sol [https://github.com/Newrails-xyz/NewrailsEMT-EVM]	Smart Contract
src/Stablecoin.sol [https://github.com/Newrails-xyz/NewrailsEMT-EVM]	Smart Contract

Appendix 3. Additional Valuables

Additional Recommendations

The smart contracts in the scope of this audit could benefit from the introduction of automatic emergency actions for critical activities, such as unauthorized operations like ownership changes or proxy upgrades, as well as unexpected fund manipulations, including large withdrawals or minting events. Adding such mechanisms would enable the protocol to react automatically to unusual activity, ensuring that the contract remains secure and functions as intended.

To improve functionality, these emergency actions could be designed to trigger under specific conditions, such as:

- Detecting changes to ownership or critical permissions.
- Monitoring large or unexpected transactions and minting events.
- Pausing operations when irregularities are identified.

These enhancements would provide an added layer of security, making the contract more robust and better equipped to handle unexpected situations while maintaining smooth operations.

Frameworks and Methodologies

This security assessment was conducted in alignment with recognised penetration testing standards, methodologies and guidelines, including the NIST SP 800-115 – Technical Guide to Information Security Testing and Assessment, and the Penetration Testing Execution Standard (PTES), These assets provide a structured foundation for planning, executing, and documenting technical evaluations such as vulnerability assessments, exploitation activities, and security code reviews. Hacken's internal penetration testing methodology extends these principles to Web2 and Web3 environments to ensure consistency, repeatability, and verifiable outcomes.