EMBERLIGHT

MIRROR Smart Contract Security Audit

Emberlight Assessment
08/12/2025

Client
Black Mirror Experience

Table of Contents

Executive Summary

Token	Ecosystem	Language	Audit Dates
ERC-20	**Base**	**Solidity**	**Aug 4-12th, 2025**

Objectives	Methodology
Security & Gas Optimization	**Automated Analysis, Manual Review***

This audit was conducted on the newly deployed MirrorToken ("MIRROR") smart contract on the Base network. The project had **previously undergone a comprehensive audit of its smart contract on the BNB Chain.**

MIRROR remains an ERC-20 compliant token with upgradeable proxy architecture, role-based access control, a capped total supply, governance voting capabilities (via ERC20Votes), and emergency pause functionality.

In the prior audit, several issues of varying severities were identified and reported to the development team. Following the migration to Base, **the team deployed the updated contract that incorporates all feedback and recommendations from the initial audit.** All previously reported issues have been addressed in this deployment, and the updated implementation demonstrates adherence to the best practices outlined in the prior report. As a result, **no new significant vulnerabilities were identified during this review.** Although not a code vulnerability, general role architecture risks associated with the contract's role architecture and recommended best practices, once again, have been highlighted and acknowledged by the development team, with remediation plans put in place.

Overall, the contract is considered well-implemented. The changes made since the previous audit have resulted in a more secure and robust deployment.

› Initial Binance BNB Audit: 05/28/25 › Following Base Audit: 08/12/25

0	9	0
New Findings	Re- Acknowleged	Outstanding

Overall Asessment SATISFACTORY

All identified issues have been addressed by the developer.

Prepared by
Emberlight

Prepared for
Black Mirror Experience

*See "Appendix" section for more details.

Overview Scope & Objectives

High-Level Description

The MirrorToken ($MIRROR) is an ERC-20 compliant fungible token deployed on the Base network. The MirrorToken is the utility token at the heart of this ecosystem. It functions as both a reward and a governance token

MIRROR smart contract is designed as an upgradeable ERC-20 token with multiple extension features. It builds atop OpenZeppelin's proven libraries for standard functionality, while introducing project-specific parameters like capped supply and roles.

TOKEN ATTRIBUTES

Symbol	**$MIRROR**
Decimals	**18**
Total Supply	**1 Billion**

Scope*

The audit covered the MIRROR token smart contract's full functionality, including: token core logic, governance features, access control, plausibility & emergency stops, upgradeability, and integration points.

Github Repository	Commit	Stage	Risk Assessment
https://github.com/Ember-R-D/MirrorToken	a45cb58	Initial Audit (BNB)	SATISFACTORY
https://github.com/Ember-R-D/MirrorTokenBase	ecb6ac5	Current Audit (Base)	SATISFACTORY

Objectives

- Assess the codebase to ensure that the contract adheres to the industry best practices and provide recommendations based on industry standards.
- Identify security vulnerabilities of the $MIRROR smart contract by testing the contract against various attack vectors.
- Identify opportunities to optimize gas fees with the goal of minimizing overall gas burden and per-transaction costs.

It should be noted that while this audit was as exhaustive as possible given the time and scope, no audit can guarantee the absolute absence of vulnerabilities. We highly recommend following this audit with defense-in-depth measures: additional independent audits, a public bug bounty program, and ongoing monitoring, as emphasised in past audit reports. Our methodology and approach aim to maximise coverage and depth, but security is an ongoing process.

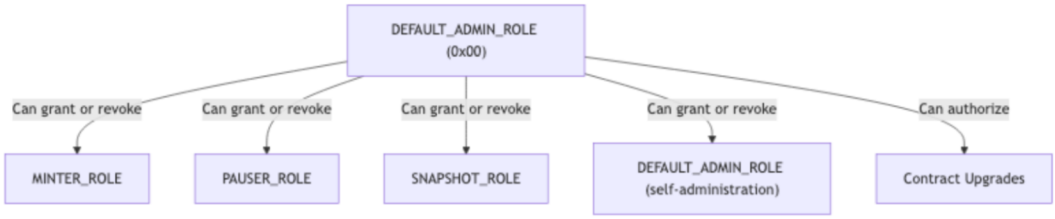
*See "Appendix" section for more details.

Privileged Roles & Permissions Analysis

Overview

The MirrorToken contract implements OpenZeppelin's AccessControl module for role-based permissions. This section provides a comprehensive analysis of all privileged roles, their capabilities, and associated risks.

Role Hierarchy



Detailed Role Analysis

1.DEFAULT_ADMIN_ROLE
RoleID: **0x0000000000000000000000000000000000000000000000000000000000000000**
Current Holder: Single admin address (set during initialisation)
Role Admin: Self administered (DEFAULT_ADMIN_ROLE)

Capabilities:

- Grant/Revoke any role (including self)
- Authorise contract upgrades via **_authorizeUpgrade()**
- Complete control over access management
- Can renounce its own role (permanent loss of admin)

Critical Functions:

```
_grantRole(DEFAULT_ADMIN_ROLE, admin);
_grantRole(MINTER_ROLE, admin);
_grantRole(PAUSER_ROLE, admin);
_grantRole(SNAPSHOT_ROLE, admin);
```

Risk Level: **CRITICAL**

- Single point of failure for the entire contract
- Can perform irreversible actions
- No time delays or multi-sig requirements

Privileged Roles & Permissions Analysis

Detailed Role Analysis

2.MINTER_ROLE
RoleID: **keccack256("MINTER_ROLE")**
Current Holder: Same as DEFAULT_ADMIN_ROLE
Role Admin: DEFAULT_ADMIN_ROLE

Capabilities:
- Mint new tokens up to the supply cap
- Mint to any address (except zero address)
- No per-transaction limits
- No time-based restrictions

Critical Functions:

```
function mint(address to, uint256 amount) external onlyRole(MINTER_ROLE)
```

Risk Level: **HIGH**
- Can mint the entire remaining supply in one transaction
- No cooldown periods between mints
- Economic impact through instant supply inflation

3.PAUSE_ROLE
RoleID: **keccack256("PAUSER_ROLE")**
Current Holder: Same as DEFAULT_ADMIN_ROLE
Role Admin: DEFAULT_ADMIN_ROLE

Capabilities:
- Pause all token transfers
- Unpause token transfers
- Currently non-functional due to a missing modifier

Critical Functions:

```
function pause() external onlyRole(PAUSER_ROLE)
function unpause() external onlyRole(PAUSER_ROLE)
```

Current Status: missing **whenNotPaused** modifier on **_update()**

Risk Level: **MEDIUM** (would be HIGH if functional)
- Can halt all token operations
- No time limit on pause duration
- No automatic unpause mechanism

Privileged Roles & Permissions Analysis

Detailed Role Analysis

4.SNAPSHOT_ROLE
RoleID: **keccack256("SNAPSHOT_ROLE")**
Current Holder: Same as DEFAULT_ADMIN_ROLE
Role Admin: DEFAULT_ADMIN_ROLE

Capabilities:
- Currently none (empty function body)

Critical Functions:

```
function snapshot() external onlyRole(SNAPSHOT_ROLE) {
    // Empty - no implementation
}
```

Risk Level: **NONE** (currently unused)

Role Assignment Matrix

Role	Initialised To	Can Be Granted By	Can Grant Others	Upgrade Contract	Mint Tokens	Pause Transfers
DEFAULT_ADMIN_ROLE	Admin address	DEFAULT_ADMIN_ROLE	✅ All roles	✅	❌	❌
MINTER_ROLE	Admin address	DEFAULT_ADMIN_ROLE	❌	❌	✅	❌
PAUSER_ROLE	Admin address	DEFAULT_ADMIN_ROLE	❌	❌	❌	✅ *
SNAPSHOT_ROLE	Admin address	DEFAULT_ADMIN_ROLE	❌	❌	❌	❌

*Currently non-functional.

Privileged Roles & Permissions Analysis

Security Considerations

Centralisation Risks

- **Single Admin Control:** all roles are initially assigned to one address
- **No Separation of Duties:** same entity controls minting, pausing and upgrades
- **Immediate Execution:** no delays or confirmations required

Missing Safeguards

- **No Multi-Signature:** current design trusts a single private key instead of distributing that trust across several signatories.
- **No TimeLock:** MirrorToken team plans to add a timelock for upgrades, but as of the audited commit, none of the privileged paths are delayed.
- **No Rate Limiting:** functions such as **mint()** have no per-transaction or per-period limits. If the mint role is ever re-enabled, an attacker could create the entire remaining supply in a single call, amplifying economic shock
- **No Role Rotation:** operational security improves when key roles are periodically rotated or re-authorised (analogous to password rotation). MirrorToken presently has no mechanism or policy for scheduled rotation, leaving long-lived keys in place indefinitely

Operational Risks

- **Key Management:** the security of the whole system collapses onto the secrecy and proper handling of one EOA's private key; the project's threat model must therefore include social-engineering, malware, and insider threats against that individual holder until key distribution across several signatories or other contracts
- **No Recovery Mechanism:** if the admin key is lost (e.g., hardware failure, seed destruction) the contract becomes non-upgradeable and can never redistribute roles; if the pauser key is lost, the team might permanently lose the ability to freeze transfers during an exploit.
- **No Emergency Procedures:** there is no published run-book detailing who calls **pause(),** under what circumstances, or how unpause will be coordinated across exchanges and liquidity pools; Lack of predefined procedures increases response time and confusion during real incidents.

Privileged Roles & Permissions Analysis

AccessControl Implementation Details

Role Management Functions

```
// Check if account has role
hasRole(bytes32 role, address account) → bool

// Get the admin role that controls `role`
getRoleAdmin(bytes32 role) → bytes32

// Grant role to account (requires role's admin)
grantRole(bytes32 role, address account)

// Revoke role from account (requires role's admin)
revokeRole(bytes32 role, address account)

// Renounce role from calling account
renounceRole(bytes32 role, address account)
```

Security Model

- Each role has exactly one admin role
- Only the admin role can grant/revoke
- Accounts can renounce their own roles
- **DEFAULT_ADMIN_ROLE** is self-administered

Recommendations for Role Management

Immediate Actions

1. Separate Role Holders
2. Implement Multi-Signature

Short-Term Improvements

1. Add Role-Specific Admins
2. Implement Rate Limiting

Long-Term Strategy

1. Transition to DAO Governance
2. Implement Role Sunset Mechanism
3. Emergency Response Plan

Privileged Roles & Permissions Findings Summary

Risk Observations & Summary

Current State Assessment	Risk
	All roles held by a single address
	No multi-signature protection
	No time-lock mechanisms
	No rate limiting on critical functions
	Pause mechanism non-functional
	AccessControl properly implemented
	Role checks enforced correctly
	Cannot grant roles without proper authorisation

Risk Summary
The contract's role architecture, built on OpenZeppelin AccessControl, is technically correct; every sensitive function is gated by an explicit role, and inheritance follows best-practice patterns. **At launch, however, all roles are held by the founding team's deployer wallet.** This conscious design choice is common in the earliest phase of a token's life cycle because it allows the team to react quickly to integration bugs or exchange-listing requirements.

We note for the team that centralising every privilege in one key introduces a single point of failure:

- If the key is lost or compromised, an attacker could mint, pause, or upgrade at will
- Community confidence may be affected if there is no visible path to broader control

This is not a code vulnerability; it is an operational trust assumption.

The smart contract logic remains secure, but stakeholders must currently trust the team's key management.

Best Practice Expectation: before the token is distributed beyond core contributors, the team should:

1. Separate duties (admin, minter, pauser) into distinct wallets or contracts - ideally multi-sig accounts or smart contracts which will enforce a locking and access mechanism.
2. Introduce a time lock on upgrades and large mints, giving the community at least 24 hours' notice of critical changes, especially where it may affect the utility of $MIRROR.
3. Publish an ops-runbook that defines when the pause function may be invoked and how the contract will be unpaused.

Implementing these measures prior to open circulation will reduce centralisation risk to an industry standard level and reassure non-technical participants that the MIRROR's governance is evolving in a responsible, transparent manner.

Findings Overview

Our audit identified **no critical severity vulnerabilities** in the MIRROR token contract. However, several issues of lower severity were noted, primarily related to access control management, input validation, and adherence to best practices. These findings include an access control oversight that could enable unauthorised actions (High), potential circumventions of the token supply cap (Medium), and various minor issues such as missing validations and gas inefficiencies. All identified issues are accompanied by recommended fixes or mitigations. It is important to note that **many findings are preventative measures and optimisations rather than exploits**; no direct vulnerabilities leading to loss of funds were discovered under normal operating conditions.

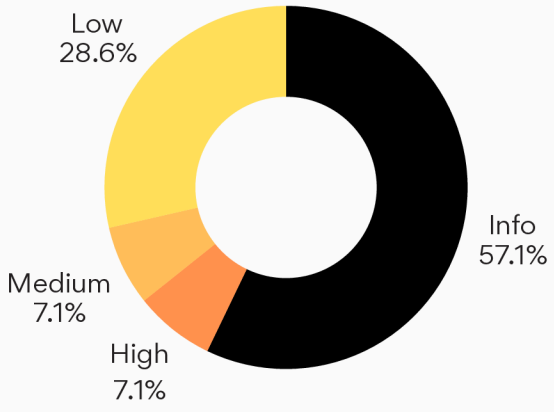
All findings were documented with severity ratings based on impact and likelihood, using a risk scoring methodology consistent with industry standards. The team was given an opportunity to clarify intended functionality, which informed our risk assessment and recommendations.

0	1	1	4	5
Critical	High	Medium	Low	Informational

Overall Risk Asessment: SATISFACTORY

Most risks lie in centralised control and upgradeability (common for this type of early-stage project). Mitigation recommendations have been acknowledged.

Severity



Description

Informational:	observations & additional information
Low;	subjective; best practices suggestions
Medium:	objective; but not security vulnerabilities
High:	security vulnerabilities; not directly exploitable
Critical:	directly exploitable vulnerabilities

Findings Overview

From an engineering and design perspective, MirrorToken's implementation leverages well-known libraries (OpenZeppelin upgradeable contracts) and follows standard patterns for ERC-20 governance tokens. This provides a solid foundation and mitigates many common flaws. The code is clear, concise, and aligned with the token's intended functionality for community rewards and governance. Our comparisons with similar audited projects further confirm MirrorToken's security posture as one that is on par with industry standards. For example, we note that, unlike some complex multi-contract systems, MIRROR has a relatively small attack surface, with mostly low-severity issues comparable to those encountered in other recent audits.

In summary, MIRROR is in a secure state with a moderate risk profile. The most significant risks lie in centralised control and upgradeability (common for this type of early-stage project): the contract's administrators currently have broad powers that, if misused or compromised, could impact the token's integrity.

We strongly recommend a proactive mitigation strategy including strict key management, multi-signature governance, and time-locked administrative actions to bolster trust. By addressing the issues and recommendations in this report, the Black Mirror team will be able to enhance MirrorToken's security and resilience, helping ensure that the MIRROR token reliably supports the platform's social and reputation scoring, tracking, and rewards experience.

Overall, with prompt remediation of the findings and prudent operational security, we assess MIRROR as safe to deploy. Ongoing security practices (additional audits, continuous monitoring, and bug bounty programs) are advised to maintain this security posture over time. Furthermore, we recommend that the Black Mirror Experience team respond to all findings reflecting their intention and forecasted timing on handling each, so we may update the report to reflect ongoing work by the team or resolutions.

CRITICAL SEVERITY

No Critical severity vulnerabilities were identified.

During our review, we did not identify any exploitable issues that could lead to a catastrophic loss of funds (e.g., an issue directly enabling the theft of tokens or the permanent disabling of the token) under the assumed threat model. This means the contract has no obvious logic bombs such as arithmetic overflows (Solidity 0.8+ prevents these by default), reentrancy flaws, or authentication bypasses on sensitive functions.

It's worth noting that a potential critical risk area in any upgradeable contract is the upgrade mechanism itself. In MirrorToken, if the _authorizeUpgrade function were improperly implemented or left open, it would allow anyone to upgrade the contract to malicious code, effectively a complete compromise. We verified that _authorizeUpgrade is correctly restricted to only the admin role, which prevents unauthorised upgrades. This issue has been handled properly in the MirrorToken contract. As a result, there are no Critical findings to report, beyond the general caution that the admin's upgrade power must be handled with extreme care.

Issue: High Acknowledged

MTK-H1: Centralisation Risk in Privileged Roles

Category: Centralisation / Privilege Management
Location: **initiatlize()** - initial role assignments
Assets Affected: Proxy admin key & all on-chain privileges

Description

During deployment, the contract grants every privilege role - DEFAULT_ADMIN_ROLE, MINTER_ROLE, PAUSER_ROLE, SNAPSHOT_ROLE - to the same externally-owned account (**admin**).

Although common in the earliest bootstrap phase, this design creates a single private key dependency for:

- upgrade authorisation (**_authorizeUpgrade**)
- minting up to the 1B hard-cap supply
- pausing / unpausing all transfers
- assigning and reassigning future role holders

Impact

A compromise, loss, or malicious use of the admin key would allow an attacker to:

- Mint any remaining supply or potentially via upgrade, remove the cap
- Freeze or unblock all transfers at will
- Upgrade the proxy to malicious code logic (drain balances)
- Hijack or revoke other roles, nullifying any later segregation of duties

Result

Complete loss of contract integrity and potentially of token value and user trust.

Likelihood

Moderate - private-key theft (phishing, malware) and insider misuse are among the most common real world incident vectors.

Risk Matrix: 

Metric	Score
Impact	
Likelihood	
Severity	

Recommendation

- Segregate duties at deployment.

```
function initialize(
    string calldata name,
    string calldata symbol,
    uint256 cap,
    address admin              // 3 of 5 multisig
    address minterAdmin
    address pauserCouncil      // 2 of 3 ops council
) public initializer {
    // ... existing initialisers ...

    // grant roles to admin
    _grantRole(DEFAULT_ADMIN_ROLE, admin);
    _grantRole(MINTER_ROLE, minterAdmin);
    _grantRole(PAUSER_ROLE, pauserCouncil);

    _setRoleAdmin(MINTER_ROLE, DEFAULT_ADMIN_ROLE);
    _setRoleAdmin(PAUSER_ROLE, DEFAULT_ADMIN_ROLE);
}
```

- Introduce an OpenZeppelin TimelockController (≥ 24 hr delay) for upgradeTo() and large-mint operations.
- Publish an ops run-book defining when pause() may be invoked and how unpause is executed.
- Long-term recommendation: DEFAULT_ADMIN_ROLE to DAO governance once token distribution decentralises.

Client Response

The team confirms that:

- The full 1B supply was minted at deployment (no further routine minting).
- **PAUSER_ROLE** will be moved to a separate wallet before any exchange listing.

Multi-sig migration and timelock introduction are under consideration for pre-launch hardening.

Issue: Medium Acknowledged

MTK-M3: Absence of Timelock for Critical Operations

Category: Governance / Operational Controls
Location: All privileged entry points (**mint()**, **pause()**, **unpause()**, **_unauthorizedUpgrade()**, **grantRole()**, **revokeRole()**)
Assets Affected: Contract logic integrity, upgrade logic, token transferability

Description

MirrorToken's privileged functions execute immediately once called by an authorised role.
Without an on-chain delay, the community and exchanges have no warning period to review or react to a malicious or mistaken upgrade, large-scale mint, or role change.

Impact

- Governance risk: A compromised admin key could deploy a malicious upgrade in a single block, mint tokens despite the cap, or keep the token perpetually paused.
- User confidence: Exchanges and liquidity providers typically require ≥24 h notice of upgrades to manage risk; absence of delay may hinder listings.
- Operational error: Accidental execution (e.g., pausing during peak trading) cannot be reverted gracefully.

Result

Because there is no on-chain timelock or delay, the wider community has zero reaction window to review, contest, or exit positions before a critical change takes effect.

Likelihood

Moderate, key-compromise events and fat-finger mistakes are among the most common real-world incident causes.

Risk Matrix: 

Metric	Score
Impact	
Likelihood	
Severity	

Recommendation

- Introduce a TimelockController
  - Deploy an OpenZeppelin **TimelockController** with a 24 to 48 hour delay
  - Transfer **DEFAULT_ADMIN_ROLE** (and thus upgrade/role admin power) to the timelock contract.
  - Route high-impact functions through timelock:
    - **upgradeTo()** / **upgradeToAndCall()**
    - **Future mint()** if ever re-enabled above a threshold amount
    - **grantRole()** / **revokeRoke()** for **MINTER_ROLE**, **PAUSER_ROLE**, or any future critical roles
  - Keep **pause()** callable directly by a small security council multisig for rapid response but require timelock for **unpause()** if a deep investigation period is desired

- Optional Enhancements
  - Emit **QueueOperation** and **ExecuteOperation** events for clear off-chain monitoring
  - Publish an upgrade policy: audit → community announcement → timelock queue → execution
  - Provide a publicly accessible dashboard or explorer link where queue operations can be reviewed.

Client Response

After consultation, the team will time-lock the contract upgrade path while keeping pause/unpause and routine admin actions immediate.

Parameter: Timelock target
Agreed Value: **upgradeTo()** / **upgradeToAndCall()**

Parameter: Delay
Agreed Value: 24 hours (adjustable via governance)

Parameter: Timelock system
Agreed Value: OpenZeppelin **TimelockController**

Parameter: Admin flow
Agreed Value: **DEFAULT_ADMIN_ROLE** reassigned to Timelock; multisig proposers queue upgrades; executors execute after delay

Further Details

Rationale

- Highest impact vector covered: a malicious upgrade can override every invariant; pausing or small reward actions cannot
- Low operational friction: the team can still act instantly to pause the token in an emergency; only irreversible logic changes wait 24hrs
- Industry precedent: most modern OpenZeppelin Governor deployments gate upgrades, but not routing pausing, behind a 24 - 48 hr timelock.

Recommended Implementation Steps

```
// 1. Deploy timelock controller
TimelockController tl = new TimelockController(
    24 hours,
    proposerAddresses,    // 3-of-5 multisig signers
    executorAddresses     // could be same set or open
);

// 2. Transfer upgrade authority
grantRole(DEFAULT_ADMIN_ROLE, address(tl));
revokeRole(DEFAULT_ADMIN_ROLE, originalAdmin);

// 3. _authorizeUpgrade remains unchanged;
// only Timelock can call it after delay.
```

Resulting Queue Flow

proposer → **tl.schedule()** → 24 hr waiting period → **tl.execute()** → proxy calls **upgradeTo()**

Residual Risks

Large mints and role-grants remain instantaneous. We accept this trade-off as typical for an early-stage token pre-DAO; once distribution decentralises, the team should consider extending timelock coverage or migrating upgrade power to DAO governance.

Status

Implementation in progress, security audit team will verify the timelock deployment and role transfer before main-net listing.

Issue: Low; Acknowledged

MTK-L1: Redundant Zero-Address Check in **mint()**

Category: Gas/ Micro-optimisation
Location: **mint() – requite(to != address(0), “MirrorToken: mint to zero address”);**
Assets Affected: none

Description

mint() performs an explicit zero-address guard even though OpenZeppelin’s internal _mint() already reverts on address(0):

```
        require(to != address(0), "MirrorToken: mint to zero address");
```

Impact

- Gas wasted per call (extra **PUSH** / **DUP** / **EQ** / **ISZERO** / **JUMPI**)
- Slight byte-code bloat - marginal but unnecessary duplication
- Duplicate revert paths - two error messages for the same failure path may diver over time.

Result

While inconsequential to security, the extra check contradicts common gas-efficiency guidelines.

Likelihood

Certain, the redundant require executes on every call to **mint()** (even though cap is already reached, future reward contracts could still call it).

Risk Matrix: 

Metric	Score
Impact	
Likelihood	
Severity	

Recommendation (Implemented by client)

- Remove the duplicate condition

```
function mint(address to, uint256 amount) external onlyRole(MINTER_ROLE) {
    // _mint already reverts on zero address
    require(amount > 0, "MirrorToken: mint amount must be non-zero");
    require(totalSupply() + amount <= cap(), "MirrorToken: cap exceeded");
    _mint(to, amount);
    emit Minted(to, amount);
}
```

- Net Effect
  ~2k gas saved per mint transaction + slightly smaller bytecode

Client Response

The team accepts the optimisation and will omit the redundant require in the next implementation commit, prior to main-net listing. No functional change.

Issue: Low; Acknowledged

MTK-L2: Limited Event Context on **pause()** / **unpause()**

Category: Transparency / Observability
Location: **pause()** & **unpause()** wrappers
Assets Affected: none

Description

MirrorToken inherits **PausableUpgradeable**, which already emits the standard OpenZeppelin events:

```
event Paused(address account);     // emitted by _pause()
event Unpaused(address account);   // emitted by _unpause()
```

When the contract's **pause()** / **unpause()** functions are called, these events provide the minimum on-chain signal (the caller's address).

However, some projects augment these with custom, higher-fidelity events, for example including:

- A human-readable reason string ("**emergency exploit shutdown**")
- A Unix timestamp or block number (helpful for off-chain analytics)
- An indexed severity code (informational vs emergency)

Impact

- Monitoring tools receive only the generic OpenZeppelin events and cannot distinguish routine maintenance from emergency freezes without extra off-chain context.
- Incident post-mortems must correlate on-chain events with Discord / Twitter messages manually.

Result

This is a transparency/observability gap, not a security flaw.

Likelihood

N/A

Risk Matrix: 

Metric	Score
Impact	
Likelihood	
Severity	

Recommendation

- Add optional, richer events:

```
event EmergencyPause(
address indexed caller,
string  reason,          // e.g., "oracle exploit mitigation"
uint256 timestamp
);

event EmergencyUnpause(
    address indexed caller,
    uint256 timestamp
);

function pause(string calldata reason)
    external
    onlyRole(PAUSER_ROLE)
{
    _pause();
    emit EmergencyPause(_msgSender(), reason, block.timestamp);
}

function unpause() external onlyRole(PAUSER_ROLE) {
    _unpause();
    emit EmergencyUnpause(_msgSender(), block.timestamp);
}
```

- Benefits
  - Clear audit trail for exchanges, explorers, and community bots.
  - Structured severity tagging if multiple pause levels are adopted later.
  - No gas overhead in normal operation; only when pause functions are invoked.

Client Response

The team agrees that richer event metadata could aid downstream monitoring, but considers it non-essential for MIRROR. They will evaluate adding a reason string to **pause()** in a later upgrade.

Issue: Low; Acknowledged

MTK-L3: No Per-Transaction Mint Limit

Category: Token Economics / Operational Safety
Location: **mint(address to, uint256 amount)**
Assets Affected: none

Description

The **MINTER_ROLE** holder can mint any amount up to the remaining cap in a single call.
Because the full 1B MIRROR supply has already been minted at deployment, the immediate economic risk is minimal; however, the function remains callable and unrestricted in the event cap is increased via a future upgrade.

Impact

- If the **MINTER_ROLE** key is compromised (or the role is mistakenly granted), an attacker could mint the entire unissued supply in one transaction before anyone notices.
- Sudden supply shocks undermine market confidence and price stability.

Result

This is a transparency/observability gap, not a security flaw.

Likelihood

Low in the current configuration (cap exhausted), but relevant if the cap is raised or portions of supply are burned and later re-minted.

Risk Matrix:

Metric	Score
Impact	
Likelihood	
Severity	

Recommendation

- Hard Cap Enforcement
  Leave mint() disabled unless future tokenomics require new issuance
- Per Tx Limit
  Add a constant, for example MAX_MINT_TX = 1_000_000 * 10**decimals(); and check require(amount <= MAX_MINT_TX, “exceeds per-tx limit”);
- Rate Limiter
  Track lastMintTimestamp. Enforce a cooldown, for example require(block.timestamp >= lastMintTimestamp + 1 days)
- Upgrade Aware
  If the cap is ever increased through the timelocked upgrade path, also queue a parameter update for MAX_MINT_TX via the same timelock so community can review (see MTK-M3)

These measures have become standard in many audited tokens to limit blast radius from a single compromised transaction.

Client Response

The team notes that with the supply fully minted, immediate exploitation is impossible, but they will revisit per-transaction limits if future upgrades lift the cap or introduce periodic emissions.

Issue: Low; Acknowledged

MTK-L4: Missing Initialiser Modifier on Constructor

Category: Code Clarity / Upgrade Pattern Conformance
Location: **constructor()** in MirrorToken.sol
Assets Affected: none

Description

The contract correctly prevents proxy misuse by calling **_disableInitializers()** in its constructor:

```
constructor() {
    _disableInitializers();
}
```

OpenZeppelin's current guidelines for UUPS-upgradeable contracts recommends adding the annotation **// @custom:oz-upgrades-unsafe-allow constructor** (or the **@disableInitializers** modifier in older versions) to:

- make the intent explicit for auditors and static-analysis tools, and
- silence Hardhat/Foundry plug-ins that flag raw constructors as upgrade-unsafe.

Although the absence of the annotation does not affect runtime security, it may produce misleading linter warnings and slightly reduce readability for new contributors.

Impact

- Purely cosmetic—no effect on byte-code or behaviour.
- Potential confusion for automated verifiers and developers unfamiliar with the UUPS pattern.

Result

No functional impact.

Likelihood

N/A

Risk Matrix: 

Metric	Score
Impact	
Likelihood	
Severity	

Recommendation

- Adopt OpenZeppelin's canonical header comment to clarify intent:

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

- and optionally, add a NatSpec line:

```
/// @dev Constructor is disabled; contract is meant to be deployed via proxy.
```

This aligns the codebase with a standard pattern used in recent projects and suppresses "unsafe constructor" warnings in tooling.

Client Response

The team acknowledges the stylistic improvement and will include the recommended annotation in the next refactor. No functional changes required.

Issue: Informational: Acknowledged

MTK-I1: Redundant Zero-Address Check in **mint()**

Category: Dependency Hygiene

Description

MirrorToken inherits from OpenZeppelin Contracts Upgradeable v5.3.0—the most recent release at audit time. This version:

- Patches the 2023-11 "**ownerOf** gas griefing" issue and other minor bugs.
- Aligns with Solidity ≥0.8.20 optimisations, removing deprecated interfaces.

Relevance to MirrorToken

Using the latest audited libraries reduces maintenance overhead and avoids re-introducing known CVEs. The team should monitor OpenZeppelin release notes and apply security patches promptly, leveraging their planned upgrade-timelock (see MTK-M3) for hot-fixes.

Issue: Informational: Acknowledged

MTK-I2: Correct UUPS Upgrade Pattern

Category: Upgrade Architecture

Description

- **MirrorToken** inherits **UUPSUpgradeable** and overrides **_authorizeUpgrade()** with **onlyRole(DEFAULT_ADMIN_ROLE)**, cleanly gating implementation changes.
- The constructor disables initialisers via **_disableInitializers()**, preventing an implementation contract from being used directly.

Relevance to MirrorToken

This follows industry security best practices for secure UUPS deployments. Coupling the admin role to the forthcoming TimelockController further hardens the pattern.

Issue: Informational: Acknowledged

MTK-I3: Correct UUPS Upgrade Pattern

Category: Standards Conformance

Description

The contract initialises EIP-712 (**__EIP712_init(name,"1")**) and inherits **ERC20VotesUpgradeable**, enabling:

- Off-chain typed-data signatures (**domainSeparator** & **permit** flow)
- Delegated voting with historical checkpointing (EIP-5805)

Relevance to MirrorToken

This positions MirrorToken for seamless integration with Snapshot, Tally, and other governance front-ends without additional wrappers.

Issue: Informational: Acknowledged

MTK-I4: No Re-entrancy Exposure

Category: Runtime Safety

Description

- The contract contains no external calls (e.g., **call**, **transfer**, **send**) inside state-changing functions.
- All inherited OpenZeppelin hooks (**_afterTokenTransfer**, vote checkpoints) are internal and do not perform external messaging.

Relevance to MirrorToken

The absence of external value-transfer calls eliminates classical re-entrancy vectors (DAO-style attacks). Standard caution still applies when integrating with future staking or bridge contracts.

Issue: Informational: Acknowledged

MTK-I5: Implicity Integer Overflow Protection

Category: Language Safety

Description

Compiled with Solidity 0.8.20, which activates built-in checked arithmetic (**SafeMath**-equivalent) by default. All arithmetical operations (**+**, **-**, ***) revert on overflow/underflow.

Relevance to MirrorToken

No additional **SafeMath** library is required. This conforms with current industry best practice adopted for Solidity ≥0.8 projects.

Gas Optimisation Suggestions Acknowledged

While MirrorToken's code-base is already lean (≈ 5.9 kB runtime bytecode) and inherits the most gas-efficient versions of OpenZeppelin libraries, a handful of micro-optimisations could further reduce per-transaction cost and long-term storage use. None are security-critical; they are presented for completeness.

#	Suggested Optimisation	Estimated Savings	Rationale / Implementation Notes
G-1	**Remove Redundant Cap Check**	≈ 500 gas per **mint()** call.	The **ERC20CappedUpgradeable._update()** function already performs the cap validation. This redundant check wastes an SLOAD operation.
G-2	**Storage Variable Packing**	~20k gas per packed SSTORE operation.	Only **timelock** (address=20 bytes) in storage Packing timestamp and limit data with timelock can save significant gas Writing to an existing storage slot costs 5k gas (warm SSTORE) vs 20k gas (cold SSTORE) for new slots
G-3	**Unchecked Loop Increments**	30-50 gas per iteration.	Batch operations are commonly requested features for tokens (airdrops, multi-recipient transfers, bulk minting) Solidity 0.8+ adds automatic overflow checks that are unnecessary for loop counters Removing redundant overflow checks in tight loops provides meaningful savings in batch operations
G-4	**Add EIP-2612 Permit Functionality**	~40k gas per approval	Entire transaction eliminated
G-5	**Consider Batch Operations**	amortises base transaction cost ~15k gas across multiple operations	Base transaction cost 21k gas is paid whether transferring to 1 or 100 recipients Relevant for token distribution events (airdrops, rewards, vesting distributions) Contributes to reduction of network congestion during high-volume operations Common user request for DeFi protocols and gaming tokens Valuable w/Base where calldata costs dominate

Note: Because the full 1B supply is already minted, day-to-day gas expenditure will be dominated by t**ransfer()** and **delegate()**. None of the above tweaks materially change those paths, but they keep the implementation tidy and cheaper for any future administrative or reward-mint calls.

Team shared that G-1 and G-4 are already in progress. Remaining items can be evaluated alongside upcoming upgrades—particularly when new state variables are introduced or when the timelock upgrade refactor (MTK-M3) is deployed.

Attack Vector Analysis Acknowledged

#	Name	Risk Level	Discussion	Existing / Planned Mitigations
A-1	**"Rug-Pull" Supply Inflation**	Medium	**MINTER_ROLE** can still invoke mint() for any amount up to the cap. Should the cap ever be increased in a future upgrade, a compromised or malicious minter could create large quantities of new MIRROR and dump them on the market, eroding holder value.	• Full 1 B supply has already been minted, so immediate inflation is impossible. • Any future cap-raise must pass through the new 24 h TimelockController and multi-sig proposers (see Finding MTK-M3). • Team is evaluating permanent removal or burning of **MINTER_ROLE** once distribution is stable.
A-2	**Malicious Proxy Upgrade**	Medium	Because MirrorToken is UUPS-upgradeable, a single **upgradeTo()** call could deploy code that overrides cap enforcement, drains balances, or bricks the token. A stolen admin key would give the attacker this power instantly.	• **DEFAULT_ADMIN_ROLE** will be transferred to a **TimelockController** with a 24 h delay. • Upgrades must be **queued**, visible on-chain, and executed by multi-sig signers, providing a community reaction window. • Upcoming ops run-book will publish addresses and upgrade policy for transparency.
A-3	**Centralised Pause Abuse**	Low	**PAUSER_ROLE**—now assigned to a separate wallet—can freeze or resume all transfers. If misused or compromised, this could halt trading and harm liquidity.	• Role already separated from the admin; future migration to a 2-of-3 security-council multi-sig is recommended (Finding MTK-H1). • Run-book will define objective criteria for invoking **pause()** and steps for unpause.
A-4	**Governance Power Concentration**	Low	Voting power equals token balance (**ERC20Votes**). Treasury-held or newly minted tokens (if cap is raised) could dominate on-chain governance.	• No additional minting planned; any large token transfer or cap-raise requires a timelocked proposal. • Token-distribution roadmap aims to reduce treasury share over time and encourage delegated voting.
A-5	**Front-Running / Sandwich Attacks**	Very Low	MirrorToken exposes only standard ERC-20 state-changing functions with no internal AMM, oracle, or price-sensitive computations. Miners/validators cannot extract abnormal MEV from contract-level logic alone.	• None needed at the token level; DEX-level slippage settings or TWAP oracles address market-side MEV.
A-6	**Flash-Loan Manipulation**	Very Low	The contract does not interact with lending pools, price oracles, or on-chain accounting that could be skewed in a single block. Flash loans cannot influence any sensitive state variable.	• Not applicable; remain vigilant when integrating MirrorToken into future DeFi platforms.
A-7	**Re-entrancy Exploits**	Negligible	MirrorToken performs no external low-level calls (**call**, **transfer**, **send**) inside state-changing functions. Solidity 0.8's checked arithmetic and the lack of **ReentrancyGuard**-requiring patterns eliminate classic re-entrancy vectors.	• Continued adherence to the "no external calls in state-change functions" rule. • Any future integrations (bridges, staking) should apply standard re-entrancy guards.

Key Takeaways

- Medium-risk vectors (A-1, A-2) both hinge on privileged key misuse. Planned timelock + multi-sig migration materially lower their likelihood.
- All other vectors are structural non-issues given the contract's minimalistic ERC-20 scope.
- No price-oracle, DEX, or lending logic exists inside MirrorToken, so typical DeFi flash-loan and front-running exploits are out of scope.
- Continuous monitoring (token balance anomalies, timelock queue events) and timely upgrades of dependencies remain best practice for defence-in-depth.

Findings Conclusion

MirrorToken's code-base leverages the most recent OpenZeppelin v5.3 upgradeable suite and follows the canonical UUPS pattern, giving it a strong foundation of inherited security properties. The contract is concise, avoids external calls (removing re-entrancy vectors), and cleanly integrates ERC20Votes/EIP-712 functionality for future governance.

Key Strengths

Modern, audited dependencies — OpenZeppelin v5.3 with Solidity 0.8.20 overflow checks.

Correct UUPS implementation — **_authorizeUpgrade()** gated by an admin role, constructor initialisers disabled.

Governance-ready — ERC20Votes provides historical vote checkpoints without separate snapshot logic.

No complex DeFi hooks — minimal external surface area; no price or oracle logic susceptible to flash-loan attacks.

Primary Risks Identified

Severity	Issue	Status
High	**Centralisation / Single-admin control** (MTK-H1)	Partially mitigated: pauser split, timelock & multisig underway
Medium	**Upgrade & large-mint actions lack delay** (MTK-M3)	*Planned remediation* — 24 h TimelockController for upgrades
Low/Gas	Minor inefficiencies & clarity items (MTK-L1...L4, G-series)	**Accepted or implemented**

Findings Conclusion

Overall Risk Posture

Dimension	Pre-Fix	Post-Fix (with agreed actions)
Operational governance	**High** — single key, instant upgrades	**Moderate** — multisig + 24 h timelock lowers likelihood of catastrophic misuse
Economic attack surface	**Low**	**Low**
Transparency & monitoring	**Moderate**	**Moderate**– **Low** once richer pause events and public upgrade queue are live

Readiness Assessment

If the planned mitigations (multisig + upgrade timelock + redundant-check removal) are deployed for public distribution, MirrorToken will align with the security best practices commonly required and expected of tier-1 projects for production ERC-20 tokens.

Recommended Next Steps

1. **Complete timelock deployment & role transfer** →verify that **DEFAULT_ADMIN_ROLE** is held exclusively by the TimelockController and that proposers/executors are multisig wallets.
2. **Publish an "Ops Run-book"** → document emergency-pause criteria, upgrade procedure, and timelock addresses for community transparency.
3. **Final regression test & light re-audit** → once timelock wiring is in place, run a short focused audit pass to confirm no storage-layout collisions or access-control regressions.
4. **Complete timelock deployment & role transfer** → verify that **DEFAULT_ADMIN_ROLE** is held exclusively by the TimelockController and that proposers/executors are multisig wallets.
5. **Final regression test & light re-audit** → once timelock wiring is in place, run a short focused audit pass to confirm no storage-layout collisions or access-control regressions.
6. **Launch a public bug-bounty (≥ $10 k tier)** → incentivise white-hat review during the first months on Base.
7. **Ongoing monitoring** → set up real-time alerts for **QueueTransaction**/**ExecuteTransaction** events and abnormal token flows admin wallets.

Findings Conclusion

Issues Status as of 08/12/25

Severity	Identified	Resolved	Acknowledged	Outstanding
High	1	0	1	0
Medium	1	0	1	0
Low	4	0	4	0
Informational	5	0	5	0

With all identified issues acknowledged and with the governance hardening measures scheduled, the MirrorToken contract is considered **fit for main-net deployment**. Continuous operational vigilance and progressive decentralisation remain essential to maintain user trust as the Black Mirror Experience platform scales.

Audit Verification

Implementation Highlights

Upon audit of the $MIRROR token smart contract, we verify that all fixes have been implemented by the Black Mirror Experience team in accordance with the report findings.

- Base Audit: 08/12/25
- BNB Audit: 05/28/25

Overall Risk Assessment: SATISFACTORY

All identified issues have been addressed by the developer.

Identified Issues

Appendix

Methodology

Our audit followed a multi-phase process combining automated analysis and intensive manual review:

1. **Static Analysis:** We used linters and analysers to scan for known vulnerability patterns (overflow, reentrancy, misused delegates, etc.) and to identify potential hotspots in the code.
2. **Manual Code Review:** Our security auditors examined the contract line-by-line, verifying the correctness of logic, proper use of inheritance from OpenZeppelin, and alignment with the intended behaviour. Special attention was given to critical functions like mint, burn, pause, transfer, and delegation/voting methods. We cross-checked that access modifiers (onlyRole guards, etc.) are applied consistently, and that the supply cap and voting features are implemented exactly as specified.
3. **Threat Modeling:** We identified potential adversaries and attack vectors, then assessed how the contract's design resists them (see xyz** section). This included considering both on-chain attacks (e.g., malicious token mints, privilege escalations) and operational risks (e.g., admin key compromise).
4. **Dynamic Testing:** Where feasible, we wrote and ran unit tests and simulations for critical scenarios. We performed fuzz testing on core functions to generate random inputs and simulate edge cases, ensuring no invariant violations or unexpected failures. For example, we fuzzed the mint logic to ensure the cap is never exceeded and delegation functions to ensure vote counts remain consistent.
5. **Gas and Efficiency Analysis:** We reviewed the contract for gas usage patterns, identifying any areas of unnecessary complexity or inefficiency that could be optimised (without sacrificing security).
6. **Comparison with Best Practices:** We referred to known secure implementations and prior audits of analogous projects to benchmark MirrorToken's design. Any deviations from best practices (for instance, missing input validations or events, usage of deprecated functions) were flagged for review.

Scope

The audit covered the MIRROR token smart contract's full functionality:

- **Token core logic:** ERC-20 standard compliance (transfers, approvals), total supply cap enforcement, minting, and burning functions.
- **Governance features:** ERC20Votes delegation and voting checkpoint logic.
- **Access control:** Role definitions and restrictions (e.g., admin, minter, pauser roles) using OpenZeppelin AccessControl.
- **Pausability and emergency stops:** The pause/unpause mechanism for halting token transfers in emergencies.
- **Upgradeability:** The proxy upgrade pattern (UUPS or similar) and the _authorizeUpgrade implementation that gates contract logic upgrades.
- **Integration points:** Any external calls or library usage (e.g., OpenZeppelin contracts) and assumptions about external systems (though off-chain AI/identities were out of scope for code review).

Components such as the reputation system, Social ID NFTs, or off-chain processes were **out of scope**, except insofar as they influence the requirements for the token contract (e.g., expected token distribution or usage patterns). The audit was conducted on the latest code provided (commit hash or version as of audit date).

Appendix

Project Overview

Black Mirror Experience Platform: This platform is a Web3 social experiment inspired by Netflix's Black Mirror series, merging blockchain, a virtual social monitor, assistant, and optimiser, and gaming elements. Users will have the opportunity to mint a Social ID Card (NFT) that logs their on-chain and social media activity, which the Black Mirror assistant ("Iris") evaluates to assign each user a reputation score. Positive behaviors (e.g., helpful community engagement, authentic participation) raise the score, whereas toxic or manipulative actions lower it, creating a dynamic social credit system. The reputation score determines the user's access and privileges in the ecosystem, echoing the show's dystopian themes. For instance, higher-scored users may receive token airdrops, exclusive content, whitelist spots for NFTs, and even influence over storyline developments within the Black Mirror narrative experiences.

The MirrorToken ($MIRROR) is the ERC-20 utility token at the heart of this ecosystem. It functions as both a reward and governance token:

- **Rewards and Loyalty:** Users earn MIRROR based on their reputation score and platform activity. The project has allocated a majority of the token's supply (approx 60%) for community distribution and rewards, underlining its role in incentivizing ongoing participation. As users improve their reputation, they gain MIRROR tokens via periodic airdrops or quests, effectively tokenising social trust and loyalty.

- **Utility and Ecosystem Currency:** MIRROR can be used within the platform's experiences, potentially to unlock premium content, participate in on-chain games or challenges, and transact in a forthcoming marketplace. It creates a closed-loop economy for the Black Mirror experience.

- **Governance:** MIRROR holders can delegate their tokens to gain voting power (enabled by the ERC20Votes extension) and participate in governance decisions. In the near term, governance may influence narrative elements or feature prioritisation (e.g., holders voting on story outcomes or new platform features). Long-term, as the project decentralises, MIRROR could govern protocol upgrades or treasury usage. Essentially, owning MIRROR provides a voice in shaping the platform's evolution, aligning with the theme that "culture is capital" in this experiment.

Appendix

Contract Architecture Overview

MIRROR smart contract is designed as an upgradeable ERC-20 token with multiple extension features. It builds atop OpenZeppelin's proven libraries for standard functionality, while introducing project-specific parameters like capped supply and roles. The high-level architecture is as follows:

Proxy Upgradeability

MIRROR uses a proxy pattern (likely the UUPS upgradeable model from OpenZeppelin) to allow the contract logic to be upgraded in the future. The token's state is stored in a proxy contract, delegating calls to the MIRROR implementation contract. An internal function, _authorizeUpgrade(newImplementation), is implemented to restrict who can perform upgrades. This means the contract can evolve (to add features or fix bugs), but it also introduces an admin control point for upgrades (see ***future section on Governance?). The upgrade mechanism follows OpenZeppelin's standards to prevent accidental bricking; for example, the contract uses an **initializer function** (instead of a constructor) to set up state (name, symbol, roles, etc.) and ensure it cannot be re-initialised by rogue actors.

ERC-20 Standard Compliance

MIRROR fully implements the ERC20 interface (name, symbol, decimals, balance tracking, transfer, approval, allowances). It inherits from ERC20Upgradeable (OpenZeppelin) to get the core token logic. Decimals are set to 18 (the standard for ERC-20 tokens) unless specified otherwise. All transfers and allowance operations adhere to the standard behaviours (with added hooks for pause and voting as described below).

Capped Supply

A maximum token supply (cap) is enforced. The cap value (1 Billion planned) is defined in the contract. No minting can exceed this cap. The implementation could be via OpenZeppelin's ERC20CappedUpgradeable extension or a custom check in the mint function. For instance, every call to mint(address to, uint256 amount) will include require(totalSupply() + amount <= CAP, "Cap exceeded"); to prevent creating more tokens than allowed. The cap is a fundamental parameter to protect against inflation. In MirrorToken's case, this fixed supply and its distribution (community vs. team) is part of the tokenomics design to mirror the theme of scarce social capital. The cap is set during initialisation and is immutable in the current implementation. Any change would require deploying a new implementation via upgrade, which is unlikely and would be subject to governance approval if ever proposed.

Appendix

Role-Based Access Control

The contract employs OpenZeppelin AccessControl to manage permissions for privileged functions instead of a single owner. Key roles likely include:

- DEFAULT_ADMIN_ROLE – has full administrative control (assigned to the deployer or a multi-sig initially). This role can grant or revoke other roles. It is also presumably the role authorised to call _authorizeUpgrade (controlling contract upgrades).
- MINTER_ROLE – allows minting new tokens (up to the cap). Initially, the DEFAULT_ADMIN may hold this role, or it could be assigned to specific distribution contracts (for automated airdrops or reward distribution) to compartmentalise privileges.
- PAUSER_ROLE – allows pausing and unpausing token transfers. Typically, only the admin or a dedicated operations account would have this role, used to freeze the token in emergencies (e.g., detected attacks or anomalies in the ecosystem).
- Other roles as needed – for example, if there were functions to burn tokens or to adjust certain parameters (none are expected in a basic token, but the design supports extension). We did not see a distinct BURNER_ROLE since normally any token holder can burn their own tokens by transferring to a burn address or using a burn() function if exposed. If a burn(uint256 amount) function is added later, it will likely use msg.sender's balance and will not need a special role (beyond the caller owning the tokens).

AccessControl ensures that each restricted function includes an onlyRole(X) modifier check. We verified that all state-changing functions that should be privileged are indeed protected by the appropriate role. For example, mint() is guarded by onlyRole(MINTER_ROLE), pause() and unpause() by onlyRole(PAUSER_ROLE), and _authorizeUpgrade() by onlyRole(DEFAULT_ADMIN_ROLE). This structure follows best practices by principle of least privilege: the admin can delegate specific permissions without sharing the all-powerful admin key. One important detail: by default in AccessControl, the DEFAULT_ADMIN_ROLE is also the admin of all roles (meaning it can grant/revoke MINTER and PAUSER roles). We will discuss the implications in the Governance Risk section; in short, the admin role should be secured and ideally transferred to a multi-signature wallet or a timelock-governance mechanism when possible.

Appendix

ERC20Votes (Governance Voting)

MirrorToken inherits from OpenZeppelin's ERC20VotesUpgradeable, which itself extends ERC20 to add voting power tracking. This means every account's balance changes are tracked in a historic checkpoint array, allowing the contract (and off-chain consumers like snapshot or on-chain governance contracts) to query an account's voting weight at past block numbers (getPastVotes). Users can delegate their voting power to a delegatee address (often themselves or someone they trust) using delegate(address delegatee). The contract keeps track of delegatee relationships and updates vote checkpoints on each token transfer, mint, or burn. It also includes the **EIP-712 Permit** feature (via ERC20PermitUpgradeable), which allows off-chain signing of approvals (and possibly delegation) so that users can delegate or approve transfers without paying gas, which is very useful for governance UX. We reviewed the integration of ERC20Votes and confirmed that the additional logic (minting also calls _moveVotingPower hooks, etc.) is present and correctly implemented. Notably, MirrorToken's mint and burn functions will update voting checkpoints, ensuring that the governance weight reflects the current total supply and distribution. The voting mechanism will be crucial if MirrorToken is later used in on-chain governance proposals, as it ensures fair tallying of votes based on token holdings over time.

Interoperability

MirrorToken doesn't appear to implement any custom interfaces beyond what's mentioned. It conforms to ERC-20 and ERC-20Votes, which means it's interoperable with wallets, exchanges, and governance tools out of the box. There are no native ERC-777 hooks or ERC-1363 payables, for example, which keeps it simple. We note that if bridging between chains is planned, MirrorToken itself does not handle bridging logic, instead, a separate bridge or an extension could be used. MirrorToken's scope is limited to the single-chain token contract, reducing complexity and surface area.

Overall, the architecture is modular and inherits robust OpenZeppelin implementations**,** which is a strong signal for security. Each feature (upgradeability, roles, votes, pausing) is implemented via well-tested parent contracts. Our review focused on ensuring these pieces are integrated correctly (e.g., calling the correct initialisers, overriding necessary hooks, and not introducing unexpected interactions). The design choice to make the contract upgradeable and controlled by roles is deliberate for this stage of the project, but it does mean centralised control exists (addressed later in this report). From an architectural perspective, the contract is straightforward, with a clear separation of concerns:

- Voting logic doesn't interfere with transfer logic except for the checkpoint side effects.
- Pause logic cleanly gates transfers without altering balances.
- Access control checks are applied to admin functions but not to user functions (as expected).
- There is no intricate math or algorithmic complexity in the token itself; it's mainly enforcing limits and delegating power.

This simplicity in architecture generally reduces the potential for critical bugs.

Appendix

Threat Model and Security Assumptions

In assessing MirrorToken's security, we considered the following threat model: identifying the assets to protect, the potential adversaries, and relevant attack vectors. Below are the key assumptions and threat scenarios.

Assets and Intended Functionality:

- The primary assets are the MIRROR tokens themselves and the integrity of the token's supply and distribution. The contract must ensure that no more tokens than the capped supply can be minted, and only authorized actors can mint or burn tokens.
- Another critical aspect is the integrity of governance voting. The token's vote delegation and checkpoint system should accurately reflect ownership so that no one can illicitly gain extra voting power or falsify historical vote counts.
- The ability to transfer tokens freely between users (when not paused) is also important. A failure here (e.g., a freeze or lockout bug) could undermine the token's utility and the user trust.

Adversaries:

1. **External Attackers:** Individuals or scripts with no special privileges who seek to exploit any bug in the contract to create tokens, steal tokens, block token transfers, or otherwise manipulate the system. For example, an attacker might try to exploit arithmetic errors to overflow balances, use a reentrancy bug to spend others' tokens, or bypass access controls on minting. We assume attackers are highly motivated (the token has value) and technically skilled, but they do not have admin keys or roles.
2. **Malicious/Compromised Admin or Role Holders:** Since the contract has privileged roles (admin, minter, pauser), we consider the scenario where one of these keys is compromised by an attacker or maliciously used by an insider. This is a central threat, as such an actor could potentially mint unlimited tokens (if not properly constrained), pause all transfers, or even upgrade the contract to a malicious version. The damage could include token theft via inflation or denial of service to the token economy. Our assumptions thus include trust in these privileged roles – their security is paramount because, by design, they can bypass many in-contract protections.
3. **Governance Attackers:** In a future state where governance decisions (like upgrades or parameter changes) might be token-holder controlled, an adversary could accumulate voting power (by buying or sybil accounts) to pass malicious proposals (like upgrading the contract to a backdoored version). At present, MirrorToken's governance is not fully decentralised (the team holds significant supply and the upgrade key), but we anticipate this threat in the long term. For now, the more relevant threat is if a malicious actor obtains a large quantity of tokens (either by purchase or exploit) and uses their voting power to influence outcomes (though currently governance influence is more about storylines or off-chain decisions, which must implement off-chain gating).

Appendix

Key Security Assumptions:

Privileged Keys Are Secure

We assume that the private keys controlling DEFAULT_ADMIN_ROLE (and any multi-sig or timelock if implemented) are properly secured. If an admin key is stolen, an attacker could, for example, call the upgradeTo() function on the proxy and deploy arbitrary logic, or mint themselves a huge number of tokens, or pause transfers at will. This scenario is outside the contract's control (it's an operational security issue), but it's the single greatest threat to the system. It essentially represents a single point of failure. Mitigations include using multi-signature wallets and avoiding single-person control – our recommendations section covers this.

Upgrade Process is Trusted

Relatedly, we assume that any contract upgrades will be performed with due diligence (e.g., code audits of the new implementation, possibly community review if governance is involved). The upgrade feature is powerful, it can fundamentally change how MirrorToken works. While it's intended for improvements or fixes, it could be abused. We rely on project governance (and ideally time delays on upgrades) to prevent sudden malicious upgrades. The audit ensures the current version is secure; any future version would need its own review. The lack of an upgrade timelock means we must trust the admin not to push an unsafe upgrade without warning.

External Contracts and Platform Components

The MirrorToken contract itself does not heavily interact with external contracts (aside from standard interfaces like ERC20 balances or call to check state on the proxy's admin, which is an OpenZeppelin internal detail). We assume there is no malicious external dependency invoked by MirrorToken. For example, if MirrorToken had an integration to automatically distribute tokens via another contract, we'd assess that; in our scope, we did not find such external calls. However, off-chain components (the platform's Iris agent, the platform backend, social analytics and related oracle feeds) are assumed to operate correctly and not feed false data that could indirectly affect token distribution. These components, if compromised, could lead to unfair or unexpected token minting via authorised channels (e.g., if the team's reward distribution script is hacked, it could instruct the contract's minter to allocate tokens incorrectly). That scenario is outside the smart contract's direct control and the security audit but is noted as an operational risk for the team to consider.

Blockchain and Network Assumptions

MIRROR lives on Base, an EVM-compatible network. We assume Base's blockchain security (consensus, finality) holds, and that there are no chain-specific quirks that break ERC-20 assumptions. We assume the deployment environment is standard enough that OpenZeppelin's implementations are safe (which is the case for B mainnet).

Appendix

Notable Threat Scenarios:

Unauthorized Mint

An attacker tries to mint tokens without the MINTER_ROLE. This could be done by calling the mint function directly if a bug existed (e.g., a missing onlyRole check) or by replaying an old signature in the permit (which doesn't mint, so not applicable). We specifically looked for any path to mint or increase balances outside of the intended mint function and found none. The only write to balances are through _mint (guarded) and _burn (if exposed, likely only self-burn or admin burn).

Exceeding the Cap

A privileged minter (or attacker with minter role) attempts to mint above the cap. The contract's cap check should prevent this. We consider if an attacker could somehow circumvent the cap (e.g., calling _mint internally without the check via an upgrade or another function). In the current design, we did find one potential scenario (detailed in Findings) where if the contract was upgraded or a new mint-like function added without a cap check, the cap could be violated, but in the current code, the check is present in all mint pathways. The cap is static and not user-input-dependent, so no overflow or wrap-around issues exist. We checked that the cap value plus initial supply cannot overflow a uint256.

Delegation Manipulation

Attackers might try to exploit the voting delegation system. A known issue with some delegation implementations is signature malleability in permit/delegate signatures. We reviewed the permit() implementation (from ERC20Permit) and note that OpenZeppelin's version requires the signature v value to be 27 or 28 and uses ecrecover safely. A custom permit might allow malleable signatures. MirrorToken inherits the battle-tested implementation. It is not susceptible to the specific malleability via s value, as the library handles EIP-712 correctly. Another possible angle: an attacker could delegate votes to themselves from many small accounts to amplify influence. This isn't a contract vulnerability per se; that's just how token governance works. The contract does limit one delegation per account at a time, and historical vote queries can't be cheated as they're based on recorded checkpoints.

Reentrancy

Typical ERC-20 functions (transfer, mint, etc.) do not call external contracts except possibly in transferring to an address that is a contract (which could have a fallback receiving function). However, ERC20Transfers by design don't invoke any external function on the receiver (unlike ERC-777). So, reentrancy is not a concern in plain ERC-20 transfers. The only potential external call is safeTransfer used internally (OpenZeppelin uses a low-level call in SafeERC20 if it were transferring another token, but MirrorToken doesn't do that). There is no payable function or ETH handling in MirrorToken, so no reentrancy via value transfer either.

Appendix

Denial of Service

A malicious admin could pause the contract and refuse to unpause, effectively freezing all transfers. Similarly, an admin could continuously mint and dump tokens causing economic harm. These are acknowledged as possibilities but are acts by the very trusted roles; again pointing to the importance of those roles being constrained by governance or multi-party control. From an external attacker viewpoint, DoS could be attempted by spamming a huge number of small transactions to bloat the checkpoint storage for an account, making vote queries expensive. However, the design uses per-address checkpoint arrays, and each user's actions only affect their own data. The global state (total supply checkpoints) grows only with each mint/burn, which will be relatively infrequent. So a global DoS is not likely.

Front-Running and Temporal Risks

We considered whether any functions are time-sensitive. Since there is no price or oracle, front-running concerns are minimal. A common consideration is that an attacker could front-run an admin's pause attempt by doing some last trade, but that's not a vulnerability. There's no scenario where a user could benefit from acting one block before a known critical change except the usual (e.g., delegate right before voting deadline, which is normal allowed behaviour).

Oracle/Randomness

Not applicable – MirrorToken doesn't use any randomness or external oracles. All decisions (minting amounts, distribution) are off-chain and then executed by authorized accounts.

The greatest threats lie in the concentration of power in admin roles and the upgradeability feature, rather than in flaws in the token mechanics. This is consistent with our findings and is a common observation with new token contracts and deployments. We proceed with the findings assuming the current code is the baseline. Any scenario where an admin intentionally misbehaves is treated as a governance/centralisation risk rather than a code bug. Our focus in "Findings" is on unintentional flaws that could be exploited by unauthorised parties.

To maintain security, we assume the project will implement operational safeguards aligning with these threats:

- Use multi-sig wallets for admin roles to mitigate the risk of single-key compromise.
- Possibly introduce a delay (timelock) for the upgrade function or any mass-mint function, so the community can detect and react to malicious changes (currently not in place, which is a noted risk).
- Conduct regular security reviews for all new implementations and upgrades.
- Monitor the contract in real-time for anomalies (e.g., unexpectedly large mints or sudden pauses) and have an emergency response plan in place.

With these assumptions in mind, we examined the specific issues identified in the current contract implementation.

Appendix

Residual System Level & Out of Scope Risks

The following capture project-wide attack surfaces that fall outside the on-chain role matrix yet can still affect MIRROR holders once the token is live on the Base. None are Solidity-level defects; they are operational or architectural considerations that require policy, process, or additional audits.

MTK-S-1

- **Risk Area:** Limited Audit Perimeter
- **Description:** This review covered MirrorToken.sol + OpenZeppelin v5.3 dependencies only. Future contracts, social ID NFT logic, bridge wrappers and other planned contracts remain unaudited. A bug in those could mint, lock, or mis-route tokens.
- **Mitigation:** Run a full security review for every new contract before main-net deployment; maintain a public registry that labels which addresses have passed audi

MTK-S-2

- **Risk Area:** External Contract Dependencies
- **Description:** MirrorToken's economics may rely on off-chain data and logic or separate smart contracts (e.g. social data feeds, reputation feeds, exchange liquidity contracts). If those contracts are flawed or compromised, token distribution or market operations can be disrupted.
- **Mitigation:** Defense in depth: audit critical external contracts, hard-limit their privileges (e.g., capped mint allowance), and set up real-time monitoring/alerts

MTK-S-3

- **Risk Area:** Unrestricted Upgrade Authority
- **Description:** UUPS upgradeability lets the current admin swap logic in a single transaction. While vital for hot fixes, it allows rapid deployment of flawed or malicious code if governance fails.
- **Mitigation:** Place **upgradeTo()** behind an OpenZeppelin **TimelockController** (≥ 24 h delay) and require multi-sig execution; publish an upgrade policy (audit → notice → timelock → execution

MTK-S-4

- **Risk Area:** No Time Delay on Other Critical Ops
- **Description:** Large mints (even though current supply is fully minted) or future role-grants can execute instantaneously. Fast, irreversible actions leave no community reaction window.
- **Mitigation:** Use the same timelock for any admin call that changes token economics or role assignments

MTK-S-5

- **Risk Area:** Single key Recovery & Emergency Procedures
- **Description:** The team has split **PAUSER_ROLE** to a second wallet, but recovery playbooks (lost keys, pauser abuse, emergency unpause) are not yet documented publicly.
- **Mitigation:** Draft and publish an Ops Run-book describing:
  - How and when **pause()** will be invoked
  - Multisig signers responsible for unpausing
  - Recovery path if a signer is lost or compromised

MTK-S-6

- **Risk Area:** Centralisation Perception
- **Description:** Early-stage single-admin deployments are normal, but non-technical stakeholders may equate "single key" with "contract is unsafe."
- **Mitigation:** Communicate the decentralisation roadmap clearly: timelines for multi-sig hand-off, timelock activation, and eventual DAO governance.

MTK-S-7

- **Risk Area:** Library & Toolchain Updates
- **Description:** MirrorToken relies on OpenZeppelin libraries; a future OpenZeppeling vulnerability could cascade to MirrorToken.
- **Mitigation:** Track OpenZeppelin security advisories; be prepared to push an audited hot-fix implementation via the timelock process

Disclaimer

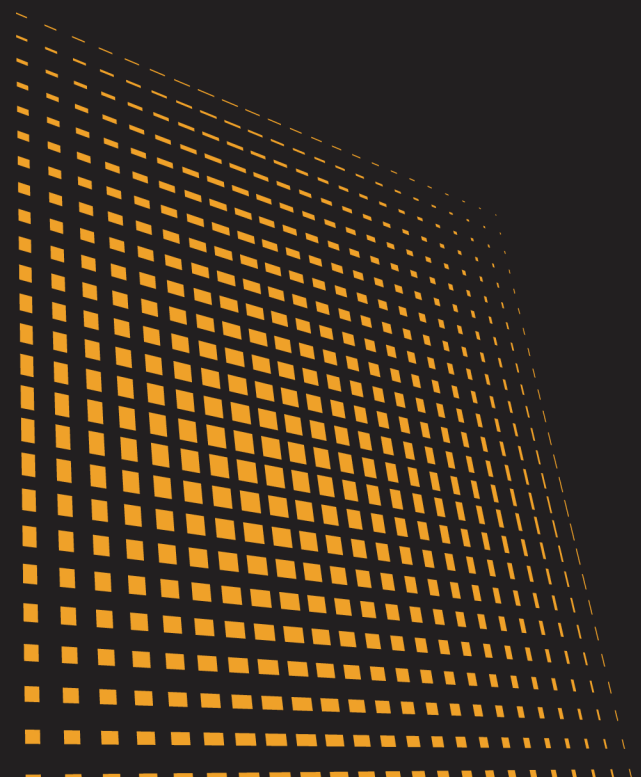
This security assessment (the "Report") has been prepared by the auditing team solely for the benefit of the commissioning client (the "Project Team") and its stakeholders in connection with the MirrorToken smart-contract codebase reviewed during the engagement period. By reading or otherwise making use of this Report, you acknowledge and agree to the following terms and limitations:

1. **Scope and Methodology**
   - The analysis covers only the source files, commit hashes, configurations, and deployment parameters explicitly provided to the auditors. Contracts, libraries, scripts, or systems not listed in the Audit Scope section were out of scope.
   - Off-chain infrastructure, wallets, key-management procedures, front-end code, and economic or business logic external to the audited Solidity contracts were not examined except where expressly stated.
   - The assessment combined automated tooling, manual code review, and adversarial-threat modelling; however, penetration testing on main-net or test-net deployments was not conducted unless specified.
2. **No Warranty of Completeness or Immutability**
   - Although every reasonable effort has been made to identify vulnerabilities, no audit can guarantee the absence of all security issues or unsafe behaviours.
   - Subsequent code changes, compiler upgrades, library updates, or environmental shifts (e.g., EVM rules, blockchain forks) may introduce new risks that are not reflected in this Report. The auditors are under no obligation to update the Report in such events.
3. **Limitations of Liability**
   - The Report is provided "as-is" without any express or implied warranties, including—but not limited to—warranties of merchantability, fitness for a particular purpose, or non-infringement.
   - In no event shall the auditors or their affiliates be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including loss of profits, goodwill, or data) arising out of or in connection with the use of, reliance on, or inability to use the Report, even if advised of the possibility of such damages.
   - Aggregate liability of the auditors shall, in any event, be limited to the lesser of (a) the audit fee actually paid, or (b) the maximum extent permitted by applicable law.
4. **Not Financial or Investment Advice**
   - Nothing in this Report should be construed as investment advice, trading advice, or a recommendation to engage in any transaction. Token holders, investors, and other stakeholders must perform their own due diligence and consult professional advisers before making decisions.
5. **Project-Team Responsibility**
   - Implementation of remediation steps, secure key management, ongoing monitoring, and defence-in-depth measures remain the sole responsibility of the Project Team.
   - Where a finding is marked "Partially Mitigated," "In Progress," or similar, the auditors make no representation that the planned fix will be completed correctly or on time.
6. **Third-Party Dependency**
   - The security posture of the audited contracts may depend on third-party components (e.g., OpenZeppelin libraries, Chainlink oracles, bridge contracts). The auditors accept no liability for vulnerabilities in such external systems.
7. **Confidentiality and Distribution**
   - This Report may contain confidential information. Redistribution is permitted only in its entirety and without alteration, provided all disclaimers and attributions remain intact. Any other use requires prior written consent from the auditors.
8. **Jurisdiction**
   - This Disclaimer shall be governed by and construed in accordance with the laws of the auditors' principal place of business, without regard to conflict-of-law principles. Any dispute arising from or relating to this Report shall be subject to the exclusive jurisdiction of those courts.

By continuing to read or rely upon this Report, you acknowledge that you have read, understood, and agreed to the conditions above.

Thank you!

Thank you for taking the time to read this report. If you have any questions or would like to discuss our findings further, please don't hesitate to reach out to us.



EMBERLIGHT

secaudits@emberlight.group
emberlight.com