Table of Contents for ERC20 Token Audit Report

Introduction

Overview of the Audit

This audit report provides an in-depth review of the **AaryaV1** ERC20 token contract, developed using OpenZeppelin's upgradeable smart contract framework. The primary objective of this audit is to identify any security vulnerabilities, ensure compliance with the ERC20 standard, and evaluate the contract's functionality, efficiency, and access control mechanisms.

This token has implemented several extensions, including burnability, pausability, permit functionality (EIP-2612), and upgradeability via the UUPS (Universal Upgradeable Proxy Standard) mechanism. Our audit covers all these features to ensure the security and correctness of the contract's behavior.

Scope of the Audit

The audit focused on the following key areas:

1. **Code Correctness and Consistency**: Verifying that the smart contract logic is properly implemented and consistent with the ERC20 standard.
2. **Security**: Checking for vulnerabilities such as reentrancy, integer overflows/underflows, and other commonly known attack vectors in Solidity contracts.
3. **Upgradeability**: Ensuring that the UUPS proxy mechanism is correctly implemented and that no misconfiguration could compromise the upgrade process.
4. **Access Control**: Reviewing the owner's privileges and ensuring that sensitive functions like minting, pausing, and upgrades are adequately restricted.
5. **Gas Efficiency**: Analyzing the code for gas optimizations and potential inefficiencies that could affect the contract's performance.
6. **Compliance with Standards**: Verifying that the contract adheres to the ERC20 standard and its extensions, including ERC20Burnable, ERC20Permit, and Pausable functionalities.

Token Details

- **Token Name**: Aarya
- **Token Symbol**: ARY
- **Decimals**: 18
- **Initial Supply**: 1,411,141 ARY (minted to the contract owner upon initialization)
- **Minting Capability**: Only the contract owner can mint additional tokens.
- **Burnable**: Token holders can burn their tokens, reducing the overall supply.
- **Pausable**: The contract owner has the ability to pause and unpause the token transfers.
- **Permit (EIP-2612)**: The contract supports off-chain approvals via signatures, enhancing user experience.
- **Upgradeability**: The contract utilizes OpenZeppelin's UUPS proxy pattern to allow future upgrades, controlled by the owner.

2. Contract Summary

Contract Name and Overview

The contract under review is named AaryaV1, which is an ERC20 token with additional functionalities such as burnability, pausability, and upgradeability, built using OpenZeppelin's upgradeable smart contract framework. This contract follows the Universal Upgradeable Proxy Standard (UUPS) allowing the contract to be upgraded while retaining its state and address.

The contract is initialized with an initial supply of 1,411,141 ARY tokens, which are minted to the owner's wallet. It provides an owner-controlled minting function to increase the total token supply, and a burn function to reduce the supply. Pausability gives the owner the ability to halt token transfers in case of emergencies. The contract also supports off-chain approvals via EIP-2612, allowing users to approve token transfers using signed messages without needing to interact with the blockchain directly, which enhances user experience and gas efficiency.

Libraries and Inheritance

The contract leverages several OpenZeppelin libraries to ensure robust security and standard compliance. The following OpenZeppelin contracts are used:

1. ERC20Upgradeable: This is the core ERC20 implementation providing the standard token functionality, such as transfers, allowances, and balances.
2. ERC20BurnableUpgradeable: This extension allows any token holder to burn their tokens, thereby reducing the total supply.
3. PausableUpgradeable: Provides the ability to stop all token transfers in the contract. Only the owner can trigger the pause() and unpause() functions.
4. OwnableUpgradeable: Grants ownership control to a single address, typically the contract deployer. The owner has special privileges, such as minting tokens and pausing the contract.
5. ERC20PermitUpgradeable: Implements the ERC2612 standard, allowing token approvals via signatures (permit function), which eliminates the need for users to send a transaction to approve transfers.
6. UUPSUpgradeable: A proxy mechanism used for contract upgradeability. It allows the contract to be upgraded while retaining the same address and state, ensuring future improvements can be made without redeploying a new contract.

The contract inherits from the following OpenZeppelin contracts in the following hierarchy:

- Initializable (from @openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol)
- ERC20Upgradeable
- ERC20BurnableUpgradeable
- PausableUpgradeable
- OwnableUpgradeable
- ERC20PermitUpgradeable
- UUPSUpgradeable

This inheritance structure ensures that all the core and extended functionalities are integrated seamlessly while ensuring upgradeability.

Key Functionalities

1. Initialization:
    - The initialize() function is used to initialize the contract after deployment. This function is called only once and sets the token name ("Aarya"), the symbol ("ARY"), and mints the initial supply of tokens to the owner.
2. Minting:
    - The mint(address to, uint256 amount) function allows the owner to mint new tokens to any address. This function is protected by the onlyOwner modifier, meaning only the contract owner can execute it.
3. Burning:
    - The contract includes the burn(uint256 amount) function (inherited from ERC20BurnableUpgradeable), allowing any token holder to burn their own tokens, reducing the total supply.
4. Pausability:
    - The pause() and unpause() functions allow the owner to temporarily halt all token transfers and minting in case of emergency or critical issues. The transfer restrictions are enforced by the _beforeTokenTransfer() hook, which checks whether the contract is paused.
5. Permit (EIP-2612):
    - The ERC20Permit extension allows users to sign off-chain messages to approve token allowances. This makes the process more gas efficient by eliminating the need for on-chain approval transactions. It is particularly useful in DeFi applications where users interact frequently with multiple smart contracts.
6. Upgradeability:
    - The contract includes the _authorizeUpgrade(address newImplementation) function, which allows the owner to upgrade the contract. This is part of the UUPS proxy pattern, ensuring that upgrades are only performed by the owner. This mechanism enables future contract enhancements without the need for redeployment.
7. Token Transfer Rules:
    - The _beforeTokenTransfer() function is overridden to integrate the pause functionality. It ensures that no token transfers can occur when the contract is paused, safeguarding against unwanted transfers during critical events.

3. Security Assessment

Initialization and Upgradeability

Risks of Misconfiguration

The **AaryaV1** contract employs the `Initializable` pattern, which ensures the contract is initialized only once using the `initialize()` function. This is a critical aspect of upgradeable contracts as improper initialization can lead to serious vulnerabilities, such as the contract being taken over by an attacker.

- **Uninitialized Contracts**: If the contract were not initialized correctly, it could result in an unintended actor becoming the owner, as the `OwnableUpgradeable` library relies on proper initialization. However, the constructor of the contract disables all initializers by calling `_disableInitializers()`, mitigating the risk of misconfiguration during deployment.
- **Initializer Function Exposure**: The `initialize()` function is public and callable only once due to the `initializer` modifier. Ensuring it is invoked during deployment is crucial to prevent any future misconfiguration or exploitation. Any missed initialization could leave the contract vulnerable to re-initialization attacks, where a malicious actor could attempt to initialize the contract and seize control.

Upgrade Process and Security

The **UUPSUpgradeable** mechanism allows this contract to be upgraded over time while retaining the same address and state. While this flexibility is useful, it also introduces risks that must be carefully managed.

- **Controlled Upgrades**: Only the contract owner can execute the `_authorizeUpgrade(address newImplementation)` function, ensuring that unauthorized users cannot push upgrades. This function is secured with the `onlyOwner` modifier, providing strong access control.
- **New Implementation Validation**: Upgrades are inherently risky if the new contract implementation contains bugs or security issues. It's important that any new implementation is thoroughly audited before being authorized. Additionally, since this is a UUPS proxy pattern, it’s essential that each new implementation maintains compatibility with the UUPS structure to avoid bricking the proxy.
- **Upgrade Exploits**: If the upgradeability feature is misused or compromised, the contract could be upgraded to a malicious version, potentially leading to token theft or denial of service. This emphasizes the need for strong governance mechanisms surrounding the upgrade process.

Ownership Control

Owner Privileges

The **AaryaV1** contract uses the `OwnableUpgradeable` pattern, where the owner has extensive control over critical contract functions. The owner privileges include:

- **Minting Tokens**: The `mint()` function allows the owner to mint new tokens, which could be misused to inflate the token supply if access control is not tightly managed.
- **Pausability**: The owner has the ability to pause or unpause the contract, which affects all token transfers. This power can be crucial in emergencies but also poses risks if abused.
- **Upgrading the Contract**: As discussed, the owner can upgrade the contract, which is another powerful privilege. If the ownership account is compromised, attackers could deploy malicious upgrades.

To mitigate these risks, it is crucial that the private key of the owner's address is securely stored and that ownership can be transferred to a multi-signature wallet to improve security.

Role Management

In its current state, the contract uses only the `Ownable` pattern for role management, with the owner having control over all privileged actions. There is no delegation of roles or multiple levels of access, which simplifies control but also centralizes authority in one address. While this reduces complexity, it increases the risk if the owner account is compromised.

To enhance security, future upgrades could implement a more granular role-based access control mechanism, allowing the delegation of specific privileges (e.g., separating the minting role from the upgrade role).

Pausability Features

Pause and Unpause Functions

The contract utilizes the `PausableUpgradeable` functionality to enable or disable token transfers and minting in times of emergency. The owner can call the `pause()` function to halt all token transfers, and the `unpause()` function to resume normal operations.

- **Pausability as a Safety Mechanism**: This feature is particularly useful in case of security vulnerabilities or attacks. For example, if a bug is discovered or the contract is under attack, the owner can quickly pause all activities to prevent further damage.
- **Access Control for Pausing**: Since only the owner can trigger the pause and unpause actions, this reduces the likelihood of unauthorized or accidental pauses. However, it also centralizes power, meaning if the owner account is compromised, an attacker could permanently pause the contract.

Impact on Token Transfers

The `_beforeTokenTransfer()` hook ensures that all token transfers are blocked when the contract is paused. This function overrides the default behavior of the `ERC20` contract and is invoked before any transfer, mint, or burn operation.

- **Transfer Restrictions**: When the contract is paused, no transfers can occur, meaning token holders cannot send or receive tokens during the pause period. This ensures

that the contract remains secure while the issue is resolved, but it could disrupt normal operations if used improperly.

- Mint and Burn Operations: In addition to regular transfers, the minting and burning of tokens are also restricted when the contract is paused. This ensures that the token supply remains stable while the contract is in a paused state.

While the pause mechanism adds an important layer of security, it also requires careful management to avoid unnecessary disruptions. It's essential that pausing and unpausing are used responsibly and that the community is informed whenever these actions occur.

Overall, the security features related to initialization, upgradeability, ownership control, and pausability are well-implemented, but their effectiveness is heavily dependent on the security of the owner account. Recommendations for multi-signature wallets and role-based access control could further improve the robustness of these features.

4. Token Supply Management

Minting and Burning Functions

Minting Function

The AaryaV1 contract includes a minting function, allowing the contract owner to generate new tokens as needed. The minting functionality is implemented using the following function:

```
function mint(address to, uint256 amount) public onlyOwner {
    _mint(to, amount);
}
```

- Owner-Only Control: The minting function is protected by the onlyOwner modifier, meaning that only the contract owner has the ability to mint new tokens. This ensures that no unauthorized minting can take place.
- Uncapped Minting: The contract does not impose a hard cap on the total token supply, meaning that the owner can mint an unlimited number of new tokens at any time. While this offers flexibility in managing the token supply, it also presents risks, as excessive minting could lead to inflation and devaluation of the token. It is important that the owner uses this privilege responsibly to maintain trust and stability within the token ecosystem.
- Supply Expansion: When the mint() function is called, the newly created tokens are added to the specified address (to), increasing the total supply of tokens in circulation. This can be used for a variety of purposes, including rewarding users, incentivizing participation, or adding liquidity to decentralized markets.

Burning Function

The AaryaV1 contract also includes a burning function, allowing token holders to reduce the total supply of tokens by destroying them. The burn functionality is inherited from the ERC20BurnableUpgradeable contract, which provides the following function:

User-Initiated Burning: Any token holder can choose to burn their own tokens, reducing the total supply. This can be useful in scenarios where users want to reduce the token supply to increase scarcity, or for deflationary tokenomics.

Supply Reduction: When a user burns tokens, the specified amount is subtracted from the user's balance and the total token supply. This feature empowers users to have direct control over reducing the circulating supply.

Owner Burn Privileges: In addition to user-initiated burns, the contract owner can burn tokens from any address by using the following function:

```
// Required methods
fn burn(&mut self, value: U256) -> Result<(), Self::Error>;

fn burn_from(
    &mut self,
    account: Address,
    value: U256,
) -> Result<(), Self::Error>;
```

This allows the owner to burn tokens from any address that has approved the owner to spend tokens on their behalf, providing an additional mechanism for supply reduction.

Supply Cap and Management

No Hard Supply Cap

The **AaryaV1** contract does not enforce a hard cap on the total token supply, meaning that the owner can continue to mint new tokens without restriction. While this provides flexibility for future token distribution, it also presents the following risks:

- **Inflationary Risk**: Unrestricted minting can lead to inflation, which could dilute the value of the token if too many tokens are introduced into circulation. This is a common concern for token ecosystems where the supply is not capped, as it could affect investor confidence and token utility.
- **Governance and Trust**: In the absence of a supply cap, the community must trust the owner to manage the minting process responsibly. To mitigate concerns, it is often recommended to implement governance mechanisms or supply caps in future contract upgrades, or to establish clear guidelines on how and when new tokens will be minted.

Supply Management Considerations

- While the contract provides flexible minting and burning capabilities, it is essential to adopt a clear strategy for managing the total token supply. Some potential considerations for supply management include:
- **Establishing a Soft Cap**: The owner could implement a soft cap or guidelines for the maximum number of tokens that can be minted over a specific time period. This would help reduce inflation risks and increase trust within the ecosystem.
- **Periodic Burns**: To counter inflationary pressures from minting, the owner could initiate periodic token burns, reducing the circulating supply to maintain a balance between minting and burning activities.
- **Minting Audits**: Regular audits or governance votes to approve minting activities could provide transparency and allow the community to have a say in how the supply is managed, fostering trust and engagement.
- In conclusion, the **AaryaV1** contract offers robust supply management tools through its minting and burning functions. However, the lack of a hard supply cap means that careful

consideration and responsible governance are required to ensure the long-term stability and value of the token.

5. Access Control and Permissions

onlyOwner Functions

The **AaryaV1** contract implements the OwnableUpgradeable module from OpenZeppelin, which grants the owner special privileges over certain key functions. Functions protected by the onlyOwner modifier include:

- **Minting Tokens**: The mint(address to, uint256 amount) function allows the owner to mint new tokens, giving them control over the token supply.
- **Pause and Unpause**: The pause() and unpause() functions enable the owner to halt or resume all token transfers, offering an emergency shutdown mechanism.
- **Upgrading the Contract**: The _authorizeUpgrade(address newImplementation) function allows the owner to approve upgrades, providing full control over the contract's evolution.

These privileges require careful handling, as they centralize significant power with the owner, making the contract vulnerable if the owner's account is compromised. It's recommended to transfer ownership to a multi-signature wallet for enhanced security.

Privileged Roles and Access Levels

The contract employs a single privileged role, **Owner**, which is granted access to all sensitive functions. There are no additional roles or role-based access controls (RBAC) implemented in the current version, which simplifies permission management but concentrates power in a single entity.

In future versions, adding a more granular RBAC structure could enhance security by delegating specific permissions (such as minting or pausing) to different roles. This would minimize the risks associated with a compromised owner account and provide a more flexible governance model.

6. Compliance with ERC20 Standards

ERC20 Functionality Verification

The **AaryaV1** contract adheres to the ERC20 standard, as defined by the OpenZeppelin ERC20Upgradeable library. The contract successfully implements all core ERC20 functions, including:

- **totalSupply()**: Returns the total supply of the token.
- **balanceOf(address account)**: Returns the balance of a specified address.
- **transfer(address to, uint256 amount)**: Allows token holders to transfer tokens to other addresses.
- **approve(address spender, uint256 amount)**: Approves a spender to transfer tokens on behalf of the token holder.
- **transferFrom(address from, address to, uint256 amount)**: Enables transfers from an address that has been approved as a spender.

These core functions ensure full compliance with the ERC20 standard, making the token compatible with ERC20-based exchanges, wallets, and DeFi platforms.

Extensions (Burnable, Pausable, Permit)

In addition to the base ERC20 functionality, the contract includes several useful extensions:

- Burnable: The ERC20BurnableUpgradeable extension allows token holders to burn their tokens, reducing the total supply. This feature is often used for deflationary token models or to add value through token scarcity.
- Pausable: The PausableUpgradeable extension allows the owner to temporarily disable all token transfers during emergency situations. While paused, neither transfers nor minting can occur, ensuring contract safety in the event of an attack or vulnerability discovery.
- Permit (EIP-2612): The ERC20PermitUpgradeable extension adds support for off-chain signatures for token approvals, reducing the need for gas-expensive on-chain approvals. This enhances user experience by enabling them to grant allowances without sending a transaction.

These extensions expand the contract's capabilities beyond the basic ERC20 standard, providing additional security and flexibility.

7. Gas Efficiency

Analysis of Gas Costs

The AaryaV1 contract is built using gas-efficient libraries from OpenZeppelin, but some functions still incur significant gas costs, particularly:

- Minting: The mint() function can be gas-expensive depending on the amount of tokens minted and the number of accounts involved. Minting large amounts of tokens in a single transaction can lead to high gas consumption.
- Token Transfers: The transfer() and transferFrom() functions have standard gas costs based on the ERC20 implementation. However, the inclusion of the Pausable extension slightly increases gas costs due to the _beforeTokenTransfer() hook.
- Permit Function: While the permit() function introduces a gas-free token approval mechanism (off-chain signing), the initial setup cost for users can include some gas fees for transaction approvals.

Optimizations for Token Transfers and Functions

Several optimizations can be considered to improve gas efficiency:

- Batch Minting: Instead of calling mint() multiple times for different addresses, batch minting could be implemented to reduce the overhead of multiple transactions.
- Use of Events: Minimizing unnecessary event emissions can save gas. The contract is already optimized by emitting only essential events such as Transfer() and Approval().
- Pausable Hook: Although the Pausable extension adds some gas overhead, it is justified by the added security. For use cases with frequent transactions, an optimized transfer function could be created to skip the hook when the contract is unpaused, if pausing is a rare occurrence.

Overall, while the contract is fairly gas-efficient due to the use of OpenZeppelin libraries, there are opportunities to further optimize specific high-frequency functions.

8. Security Vulnerabilities

Potential Vulnerabilities (Reentrancy, Overflow, etc.)

The **AaryaV1** contract avoids many of the common security pitfalls due to the use of OpenZeppelin's well-audited libraries. However, it is important to highlight potential vulnerabilities:

- **Reentrancy**: No direct external calls are made in any of the sensitive functions (such as mint(), burn(), or transfer()), reducing the risk of reentrancy attacks. The contract is inherently safe from such attacks given its current functionality.
- **Integer Overflow/Underflow**: The contract uses Solidity version ^0.8.18, which has built-in overflow and underflow protection. This ensures that arithmetic operations cannot wrap around and lead to unintended behavior.
- **Upgradeable Contract Risks**: As the contract is upgradeable, it carries the inherent risk of upgrade mismanagement. If the upgrade process is not properly controlled, the contract could be upgraded to a malicious version or rendered inoperable.

Audit Findings and Recommendations

1. Secure Ownership

The current AaryaV1 contract centralizes control in a single owner account, which is a significant security concern, especially when using a basic MetaMask wallet for ownership. A single point of failure leaves the entire token system vulnerable to compromise, whether through phishing, social engineering, or a key compromise.

Recommendation:

- Multi-Signature Wallet: It is highly recommended to transfer ownership to a multi-signature wallet (e.g., Gnosis Safe) to reduce the risk associated with a single private key. A multi-signature wallet requires multiple trusted parties to approve critical actions like minting, pausing, or upgrading the contract, drastically lowering the risk of unauthorized control. This approach also enhances accountability by ensuring consensus among trusted parties.
- Hardware Wallets: For individual or testing purposes, using a hardware wallet (e.g., Ledger, Trezor) for the owner account is essential for enhanced security. Hardware wallets provide an additional layer of protection by keeping private keys offline.
- Key Rotation: Periodic rotation of private keys, along with a well-defined process for replacing compromised keys, is a best practice for ownership security.

2. Upgradable Contract Security

The AaryaV1 contract is designed to be upgradeable using the UUPS proxy pattern. While this offers flexibility, it also introduces risks if upgrades are not properly audited or controlled.

Recommendation:

- Thorough Audits: Every new contract implementation or upgrade must undergo a full security audit before deployment. This ensures that any potential vulnerabilities or logic errors in new versions are identified and mitigated before they affect the production environment.
- Upgrade Authorization: Consider introducing governance or multi-signature mechanisms for authorizing contract upgrades, ensuring that any future upgrades are carefully vetted by multiple parties. This mitigates the risk of malicious or accidental upgrades.

3. Minting Control

The contract allows the owner to mint tokens without restrictions, which, while offering flexibility, poses a significant inflationary risk if misused.

Recommendation:

- Supply Cap: To prevent unchecked inflation, consider implementing a maximum supply cap that limits the total number of tokens that can be minted. This would protect the token's value and foster trust among users and investors.
- Governance on Minting: Introduce a community governance mechanism or require multi-signature approval for minting operations to ensure transparency and prevent arbitrary minting by the owner.

- Minting Timelocks: Implement a timelock on minting, which delays mint operations and provides the community or trusted parties time to intervene or object if unexpected minting is attempted.

4. Role-Based Access Control

Currently, all privileged actions (minting, pausing, upgrading) are centralized in the owner role. This concentration of power increases the risk of misuse or compromise.

Recommendation:

- Role-Based Access Control (RBAC): In future iterations, introduce RBAC to distribute responsibilities across different roles. For example:
    - Minting Manager: A dedicated role for minting tokens.
    - Pauser Role: A separate role with permissions to pause or unpause the contract.
    - Upgrader Role: A role limited to contract upgrades.

This division of responsibilities can minimize the risk of compromise or mismanagement by a single individual.

5. Initial Supply Holder Security

During testing, the owner and initial token supply are controlled by a simple MetaMask wallet, which is not secure enough for long-term use or production environments. This setup leaves the initial token holder vulnerable to theft or key loss.

Recommendation:

- Cold Wallet Storage: For long-term storage of the initial supply, consider transferring the tokens to a cold wallet (offline storage). Cold wallets are not connected to the internet, making them immune to online attacks.
- Vesting Contracts: For additional security and control over token distribution, consider using vesting contracts. These smart contracts release the tokens gradually over time, ensuring that large amounts of the initial supply are not accessible all at once. This strategy is particularly useful for preventing the immediate liquidation of a large token supply.
- Delegation of Owner Rights: If using a single account for testing is necessary, consider implementing delegate ownership mechanisms, where ownership can be temporarily delegated to more secure addresses during testing or transition phases.

Additional Security Measures:

- Account Monitoring: Set up monitoring tools like OpenZeppelin Defender or Etherscan alerts to notify you in real-time if critical actions (e.g., minting, transfers, upgrades) are performed on the owner account.
- Private Key Management: Use secret management solutions such as AWS Secrets Manager or HashiCorp Vault to securely store the private keys used for the owner account, especially in production environments.
- Emergency Backup: Maintain a well-documented emergency procedure to recover or transfer ownership if the owner account is compromised or lost.

Conclusion

The AaryaV1 contract demonstrates a solid foundation in terms of functionality and security. However, due to the inherent risks associated with upgradeability and centralized ownership, it is crucial to implement robust access control mechanisms, multi-signature governance, and secure key management practices. By adopting these recommendations, the contract can maintain a higher level of trust, security, and long-term viability in production environments.