L19: Token Lifecycle Management

Module B: Ethereum & Smart Contracts

Blockchain & Cryptocurrency Course

December 2025

Learning Objectives

By the end of this lesson, you will be able to:

- Implement various minting strategies (owner, public, allowlist, merkle tree)
- Design burning mechanisms for deflationary tokenomics
- Use Pausable pattern for emergency circuit breakers
- Apply access control patterns (Ownable, AccessControl, multi-sig)
- Understand upgradeability patterns (Transparent Proxy, UUPS)
- Implement time-locked operations and governance mechanisms

Token Lifecycle Stages

Complete lifecycle of a token contract:

1. **Deployment:**
   - Contract creation and initialization
   - Owner/admin setup
   - Initial token distribution (or reserve for minting)
2. **Minting:**
   - Creating new tokens (increasing supply)
   - Controlled by governance, owner, or public
3. **Operation:**
   - Normal transfers, approvals, staking
   - May include pause/unpause capability
4. **Burning:**
   - Destroying tokens (decreasing supply)
   - Voluntary or forced mechanisms
5. **Upgrade/Migration:**
   - Proxy upgrades or token swaps
   - Governance-controlled or multi-sig

Minting Strategy 1: Owner-Controlled

Simple admin-based minting:

```
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract OwnerMintToken is ERC20, Ownable {
    constructor() ERC20("Owner Mint Token", "OMT") {}

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

Advantages:

- Simple implementation
- Flexible timing and recipients
- Useful for airdrops, rewards, team allocation

Disadvantages:

- Centralized control (trust in owner)
- Risk of unlimited inflation
- Regulatory concerns (security classification)

Minting Strategy 2: Public Mint with Cap

Anyone can mint up to a maximum supply:

```
contract PublicMintToken is ERC20 {
    uint256 public constant MAX_SUPPLY = 1_000_000 * 10**18;
    uint256 public constant MINT_PRICE = 0.01 ether;
    uint256 public constant MAX_PER_TX = 10 * 10**18;

    constructor() ERC20("Public Mint Token", "PMT") {}

    function mint(uint256 amount) public payable {
        require(totalSupply() + amount <= MAX_SUPPLY, "Max supply exceeded");
        require(amount <= MAX_PER_TX, "Exceeds max per transaction");
        require(msg.value >= amount * MINT_PRICE / 10**18, "Insufficient payment");

        _mint(msg.sender, amount);
    }

    function withdraw() public onlyOwner {
        payable(owner()).transfer(address(this).balance);
    }
}
```

Use Cases: NFT mints, fair launch tokens, crowdfunding

Minting Strategy 3: Allowlist (Whitelist)

Restrict minting to approved addresses:

```
contract AllowlistMintToken is ERC20, Ownable {
    mapping(address => bool) public allowlist;
    mapping(address => bool) public hasMinted;

    uint256 public constant MINT_AMOUNT = 100 * 10**18;

    constructor() ERC20("Allowlist Token", "ALT") {}

    function addToAllowlist(address[] calldata addresses) public onlyOwner {
        for (uint i = 0; i < addresses.length; i++) {
            allowlist[addresses[i]] = true;
        }
    }

    function mint() public {
        require(allowlist[msg.sender], "Not on allowlist");
        require(!hasMinted[msg.sender], "Already minted");

        hasMinted[msg.sender] = true;
        _mint(msg.sender, MINT_AMOUNT);
    }
}
```

Challenge: Gas-expensive to add large allowlists (20,000 gas per address)

Minting Strategy 4: Merkle Tree Allowlist

Gas-efficient allowlist using Merkle proofs:

```
import "@openzeppelin/contracts/utils/cryptography/MerkleProof.sol";

contract MerkleMintToken is ERC20 {
    bytes32 public merkleRoot;
    mapping(address => bool) public hasClaimed;

    constructor(bytes32 _merkleRoot) ERC20("Merkle Token", "MTK") {
        merkleRoot = _merkleRoot;
    }

    function claim(uint256 amount, bytes32[] calldata merkleProof) public {
        require(!hasClaimed[msg.sender], "Already claimed");

        bytes32 leaf = keccak256(abi.encodePacked(msg.sender, amount));
        require(MerkleProof.verify(merkleProof, merkleRoot, leaf), "Invalid proof");

        hasClaimed[msg.sender] = true;
        _mint(msg.sender, amount);
    }
}
```

Advantages: Store single root hash (32 bytes) instead of entire allowlist
Gas Cost: Approximately 20,000 gas for root, regardless of allowlist size

Merkle Tree Allowlist Example

Off-chain computation:

Step 1: Create allowlist with amounts

- Alice: 100 tokens
- Bob: 200 tokens
- Charlie: 150 tokens
- ... 10,000 more addresses

Step 2: Build Merkle tree

- Leaf 1: `keccak256(abi.encodePacked(Alice, 100))`
- Leaf 2: `keccak256(abi.encodePacked(Bob, 200))`
- ... hash pairs recursively to build tree
- **Root:** `0xabc123...` (single 32-byte hash)

Step 3: Deploy contract with root

- Store only `0xabc123...` on-chain (20,000 gas)

Step 4: Users claim with proof

- Alice provides: `amount=100`, `proof=[hash2, hash34, hash5678, ...]`
- Contract verifies proof against root

Destroying tokens to reduce supply:

```
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol";

contract BurnableToken is ERC20Burnable {
    constructor() ERC20("Burnable Token", "BURN") {
        _mint(msg.sender, 1_000_000 * 10**18);
    }

    // Inherited from ERC20Burnable:
    // function burn(uint256 amount) public {
    //     _burn(msg.sender, amount);
    // }

    // function burnFrom(address account, uint256 amount) public {
    //     uint256 currentAllowance = allowance(account, msg.sender);
    //     require(currentAllowance >= amount, "Insufficient allowance");
    //     _approve(account, msg.sender, currentAllowance - amount);
    //     _burn(account, amount);
    // }
}
```

Burn Strategies:

- **Voluntary:** Users burn their own tokens (e.g., for utility)
- **Fee Burn:** Transaction fees burned automatically (EIP-1559 model)
- **Buyback & Burn:** Protocol buys tokens from market and burns them

Deflationary Tokenomics

Design patterns for reducing supply over time:

1. **Transaction Fee Burn:**

```
function _transfer(address from, address to, uint256 amount) internal override {
    uint256 burnAmount = amount * 2 / 100;  // 2% burn on every transfer
    uint256 sendAmount = amount - burnAmount;

    super._transfer(from, address(0), burnAmount);  // Burn
    super._transfer(from, to, sendAmount);          // Transfer remainder
}
```

2. **Staking Burn Requirement:**
    - Users must burn X tokens to stake Y tokens
    - Creates scarcity for staking participation
3. **Governance Proposal Burn:**
    - Burn tokens to submit governance proposal
    - Prevents spam proposals

Access Control: Ownable Pattern

Single owner with full control:

```
import "@openzeppelin/contracts/access/Ownable.sol";

contract OwnedToken is ERC20, Ownable {
    constructor() ERC20("Owned Token", "OWN") {}

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

    function transferOwnership(address newOwner) public override onlyOwner {
        require(newOwner != address(0), "Invalid new owner");
        super.transferOwnership(newOwner);
    }

    function renounceOwnership() public override onlyOwner {
        // Irreversibly give up ownership (contract becomes unmanaged)
        super.renounceOwnership();
    }
}
```

Risk: Single point of failure (owner key compromise = full control loss)
Mitigation: Use multi-sig wallet as owner (Gnosis Safe)

Access Control: Multi-Signature Wallets

Require multiple approvals for critical operations:

Gnosis Safe Example:

- 3-of-5 multi-sig: Requires 3 out of 5 owners to approve transaction
- Prevents single key compromise
- Common setup: Deploy token with Gnosis Safe as owner

Workflow:

1. Owner 1 proposes transaction (e.g., `mint(alice, 1000)`)
2. Owners 2 and 3 approve transaction
3. Transaction executes automatically when threshold reached
4. Transaction hash logged on-chain for transparency

Real-World Usage:

- Uniswap: 4-of-7 multi-sig controls protocol fees
- Compound: Timelock + multi-sig for governance execution
- Curve: 3-of-5 emergency multi-sig for pausing

Upgradeability: Why and Risks

Motivation for upgradeable contracts:

- Fix critical bugs without redeploying
- Add new features (e.g., staking, governance)
- Migrate to more efficient implementation
- Comply with changing regulations

Fundamental Challenge:

- Smart contracts are immutable after deployment
- Bytecode cannot be changed

Solution: Proxy Pattern

- **Proxy Contract:** Fixed address, users interact with this
- **Implementation Contract:** Contains logic, can be swapped
- Proxy uses `delegatecall` to execute implementation logic

Risks:

- Admin can rug pull by upgrading to malicious implementation
- Storage layout collisions between versions
- Requires trust in upgrade governance

Upgradeability: Transparent Proxy Pattern

Separate admin and user interfaces:

```
// Proxy Contract (deployed once, never changes)
contract TransparentProxy {
    address public implementation;
    address public admin;

    constructor(address _implementation) {
        implementation = _implementation;
        admin = msg.sender;
    }

    function upgradeTo(address newImplementation) external {
        require(msg.sender == admin, "Not admin");
        implementation = newImplementation;
    }

    fallback() external payable {
        address impl = implementation;
        assembly {
            calldatacopy(0, 0, calldatasize())
            let result := delegatecall(gas(), impl, 0, calldatasize(), 0, 0)
            returndatacopy(0, 0, returndatasize())
            switch result
            case 0 { revert(0, returndatasize()) }
            default { return(0, returndatasize()) }
        }
    }
}
```

Key Takeaways

1. **Minting Strategies:** Owner-controlled (centralized), public mint (open), allowlist (exclusive), Merkle tree (gas-efficient)
2. **Burning:** Voluntary burn, fee burn, buyback and burn for deflationary tokenomics
3. **Pausable:** Emergency circuit breaker halts transfers during critical bugs
4. **Access Control:** Ownable (single admin), AccessControl (role-based), multi-sig (distributed trust)
5. **Upgradeability:** Transparent Proxy (separate admin/user), UUPS (upgrade logic in implementation), with storage layout risks
6. **Timelock:** Delay critical operations to allow community reaction and exit
7. **Governance:** Token-weighted voting enables decentralized control of protocol parameters

Discussion Questions

1. What are the security tradeoffs between Merkle tree allowlists and simple mapping-based allowlists?
2. Should all tokens be pausable, or does this introduce too much centralization risk?
3. How can upgradeable contracts maintain credible neutrality when admins can change the code?
4. What is the optimal timelock delay for different types of governance actions (parameter changes vs emergency actions)?
5. How do deflationary tokenomics (burning) affect long-term protocol sustainability compared to inflationary models?

Next Lesson Preview: L20 - Lab: Token Analysis

Coming up next (hands-on lab):

- Analyzing USDC and DAI contracts on Etherscan
- Examining token holder distribution and centralization
- Tracking transaction patterns and whale movements
- Identifying upgrade events and governance actions
- Deploying your own ERC-20 token with custom features
- Verifying and interacting with deployed token

Preparation:

- Review Etherscan contract reading interface
- Familiarize with USDC and DAI token pages
- Prepare Sepolia testnet ETH for deployment