Lecture Notes: Lesson 4

ERC-20 Token Creation

Cryptocurrency Course

Contents

1 Introduction to Token Standards

1.1 What Are Token Standards?

Token standards define common interfaces for smart contracts, enabling interoperability across the Ethereum ecosystem.

Benefits of Standards

Standardization enables:

- **Wallet Integration**: Any wallet supporting the standard works with any token
- **Exchange Listing**: Exchanges implement standard once for all tokens
- **DApp Compatibility**: DApps interact with tokens uniformly
- **Composability**: Protocols can build on each other
- **Reduced Development**: Leverage existing tools and libraries

1.2 Common Token Standards

Standard	Purpose
ERC-20	Fungible tokens (currencies, utility tokens)
ERC-721	Non-fungible tokens (NFTs, unique items)
ERC-1155	Multi-token (fungible + non-fungible in one contract)
ERC-777	Advanced fungible tokens with hooks
ERC-4626	Tokenized vaults (yield-bearing tokens)

2 ERC-20 Token Standard

2.1 Historical Context

ERC-20 was proposed by Fabian Vogelsteller in November 2015 as Ethereum Request for Comments #20. It became the de facto standard for fungible tokens.

Adoption: Thousands of tokens use ERC-20, including:

- USDT (Tether) - Stablecoin
- LINK (Chainlink) - Oracle network token
- UNI (Uniswap) - DEX governance token
- DAI (MakerDAO) - Decentralized stablecoin

2.2 Interface Specification

The ERC-20 standard defines 6 mandatory functions and 2 events.

2.2.1 Required Functions

ERC-20 Interface

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

interface IERC20 {
    // Returns total token supply
    function totalSupply() external view returns (uint256);

    // Returns balance of an account
    function balanceOf(address account) external view returns (
       uint256);

    // Transfers tokens from caller to recipient
    function transfer(address recipient, uint256 amount)
        external returns (bool);

    // Returns remaining tokens that spender can transfer
    // on behalf of owner
    function allowance(address owner, address spender)
        external view returns (uint256);

    // Sets amount as the allowance of spender over caller's tokens
    function approve(address spender, uint256 amount)
        external returns (bool);

    // Transfers tokens from sender to recipient using allowance
    function transferFrom(address sender, address recipient,
       uint256 amount)
        external returns (bool);

    // Emitted when tokens are transferred
    event Transfer(address indexed from, address indexed to,
       uint256 value);

    // Emitted when allowance is set
    event Approval(address indexed owner, address indexed spender,
       uint256 value);
}
```

2.2.2 Optional Metadata Functions

Not required but widely implemented:

```solidity
function name() public view returns (string memory);
function symbol() public view returns (string memory);
function decimals() public view returns (uint8);
```

Decimals Explanation

decimals specifies display precision, not blockchain storage.
Example: Token with 18 decimals

- Blockchain stores: `1000000000000000000` (1 with 18 zeros)
- User sees: `1.0` tokens
- Formula: Display = Storage/10^{decimals}

Common values: 18 (matches ETH), 6 (USDC), 8 (Bitcoin-style)

2.3 Core Mechanics

2.3.1 Token Transfer Flow

Direct Transfer (`transfer`):

1. User calls `transfer(recipient, amount)`
2. Contract checks: `balanceOf[msg.sender] >= amount`
3. If true: `balanceOf[msg.sender] -= amount`
4. And: `balanceOf[recipient] += amount`
5. Emit `Transfer(msg.sender, recipient, amount)`

Delegated Transfer (`approve` + `transferFrom`):

1. Owner calls `approve(spender, amount)`
2. Sets: `allowance[owner][spender] = amount`
3. Emit `Approval(owner, spender, amount)`
4. Later, spender calls `transferFrom(owner, recipient, amount)`
5. Contract checks: `allowance[owner][spender] >= amount`
6. If true: Reduce allowance, transfer tokens

Approve-TransferFrom Use Case

Scenario: Using Uniswap DEX to swap tokens

1. User approves Uniswap contract: `token.approve(uniswap, 1000)`
2. User initiates swap on Uniswap UI
3. Uniswap contract calls: `token.transferFrom(user, uniswap, 1000)`
4. Swap executes without requiring user's private key

This pattern enables smart contracts to interact with user tokens safely.

2.3.2 State Variables

Minimal ERC-20 implementation requires:

```
mapping(address => uint256) private _balances;
mapping(address => mapping(address => uint256)) private _allowances;
uint256 private _totalSupply;
```

Storage Structure:

- `_balances`: Maps each address to its token balance
- `_allowances`: Nested mapping for approved spenders
- `_totalSupply`: Sum of all tokens in existence

3 Implementation with OpenZeppelin

3.1 Why Use OpenZeppelin?

OpenZeppelin Contracts is the gold standard library for secure smart contract development.

Advantages:

- Audited by multiple security firms
- Battle-tested (used by thousands of projects)
- Modular and extensible
- Well-documented
- Regularly updated

3.2 Basic ERC-20 Token

Simple Token Implementation

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MyToken is ERC20 {
    constructor(uint256 initialSupply) ERC20("MyToken", "MTK") {
        _mint(msg.sender, initialSupply * 10**decimals());
    }
}
```

Deployment:

```
// scripts/deploy.js
const initialSupply = 1000000; // 1 million tokens
const MyToken = await ethers.getContractFactory("MyToken");
const token = await MyToken.deploy(initialSupply);
await token.deployed();
console.log("Token deployed to:", token.address);
```

This 7-line contract provides:

- Full ERC-20 compliance
- Minting of initial supply
- All transfer and allowance functions
- Safe arithmetic (no overflow/underflow)

3.3 Advanced Features

3.3.1 Mintable Token

Allow authorized addresses to create new tokens:

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

contract MintableToken is ERC20, Ownable {
    constructor() ERC20("MintableToken", "MINT") {}

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

3.3.2 Burnable Token

Allow token holders to destroy their tokens:

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

```

```
contract BurnableToken is ERC20, ERC20Burnable {
    constructor(uint256 initialSupply) ERC20("BurnableToken", "BURN") {
        _mint(msg.sender, initialSupply);
    }
}
```

Usage:

```
// Burn your own tokens
token.burn(1000);

// Burn tokens you have allowance for
token.burnFrom(otherAddress, 500);
```

3.3.3 Capped Supply

Enforce maximum total supply:

```
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Capped.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract CappedToken is ERC20, ERC20Capped, Ownable {
    constructor(uint256 cap) ERC20("CappedToken", "CAP")
        ERC20Capped(cap * 10**decimals()) {}

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

    function _mint(address account, uint256 amount)
        internal virtual override(ERC20, ERC20Capped) {
        super._mint(account, amount);
    }
}
```

3.4 Comprehensive Token Example

Full-Featured Token

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/
    ERC20Burnable.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Capped.
    sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/security/Pausable.sol";

contract ComprehensiveToken is ERC20, ERC20Burnable,
    ERC20Capped, Ownable, Pausable {

    // Events
    event TokensMinted(address indexed to, uint256 amount);
    event TokensBurned(address indexed from, uint256 amount);

    constructor(
        string memory name,
        string memory symbol,
        uint256 cap,
        uint256 initialSupply
    ) ERC20(name, symbol) ERC20Capped(cap * 10**decimals()) {
        require(initialSupply <= cap, "Initial␣supply␣exceeds␣cap")
            ;
        _mint(msg.sender, initialSupply * 10**decimals());
    }

    // Minting function (only owner)
    function mint(address to, uint256 amount) public onlyOwner {
        _mint(to, amount);
        emit TokensMinted(to, amount);
    }

    // Enhanced burn with event
    function burn(uint256 amount) public override {
        super.burn(amount);
        emit TokensBurned(msg.sender, amount);
    }

    // Pause/unpause transfers
    function pause() public onlyOwner {
        _pause();
    }

    function unpause() public onlyOwner {
        _unpause();
    }

    // Override required by Solidity
    function _mint(address account, uint256 amount)
        internal virtual override(ERC20, ERC20Capped) {
        super._mint(account, amount);
    }

    // Ensure transfers respect pause state
    function _beforeTokenTransfer(
        address from,
        address to,
        uint256 amount
```

4 Token Economics and Supply Models

4.1 Supply Mechanisms

4.1.1 Fixed Supply

Total supply created at deployment, never changes.

Example: Bitcoin (21M cap), Wrapped Bitcoin (WBTC)

Implementation:

```
constructor() ERC20("FixedToken", "FIX") {
    _mint(msg.sender, 21_000_000 * 10**decimals());
}
```

Characteristics:

- Deflationary pressure if tokens are lost
- Scarcity-driven value
- Predictable supply schedule

4.1.2 Inflationary Supply

New tokens continuously minted.

Example: Ethereum (before EIP-1559 burn), staking rewards

Implementation:

```
uint256 public constant ANNUAL_INFLATION_RATE = 2; // 2% per year

function mintInflation() public onlyOwner {
    uint256 timeSinceLastMint = block.timestamp - lastMintTimestamp;
    uint256 annualSeconds = 365 days;
    uint256 inflationAmount = (totalSupply() * ANNUAL_INFLATION_RATE
        * timeSinceLastMint) / (100 * annualSeconds);

    _mint(treasury, inflationAmount);
    lastMintTimestamp = block.timestamp;
}
```

Characteristics:

- Rewards participation (staking, mining)
- Can decrease token value over time
- Funds development or ecosystem growth

4.1.3 Elastic Supply

Supply adjusts to maintain price target (rebase tokens).

Example: Ampleforth (AMPL)

Mechanism: If price ¿ \$1.05, increase all balances proportionally. If price ¡ \$0.95, decrease balances.

Rebase Tokens

Elastic supply tokens are NOT standard ERC-20 compatible:

- Balance changes without transfers
- Breaks DeFi integrations expecting constant balances
- Requires special handling by exchanges/wallets

4.2 Distribution Strategies

4.2.1 Initial Coin Offering (ICO)

- Sell tokens to public before launch
- Raised $20B+ in 2017-2018 peak
- Often unregulated, high fraud risk
- Regulatory crackdowns in most jurisdictions

4.2.2 Airdrop

- Free distribution to specific addresses
- Reward early users or protocol participants
- Marketing and community building

Airdrop Implementation

```
contract Airdrop {
    IERC20 public token;
    mapping(address => bool) public claimed;

    function claimAirdrop() public {
        require(!claimed[msg.sender], "Already␣claimed");
        require(isEligible(msg.sender), "Not␣eligible");

        claimed[msg.sender] = true;
        token.transfer(msg.sender, 100 * 10**18); // 100 tokens
    }

    function isEligible(address user) internal view returns (bool)
       {
        // Example: Must have interacted before block X
        // Implementation depends on criteria
        return true;
    }
}
```

4.2.3 Liquidity Mining

- Reward users who provide liquidity to DEXs
- Tokens distributed over time based on participation
- Popularized by Compound, Uniswap, etc.

4.2.4 Vesting

Lock tokens and release gradually over time.

Simple Vesting

```
contract TokenVesting {
    IERC20 public token;
    address public beneficiary;
    uint256 public cliff;         // When vesting starts
    uint256 public duration;      // Vesting period
    uint256 public released;

    constructor(
        IERC20 _token,
        address _beneficiary,
        uint256 _cliff,
        uint256 _duration
    ) {
        token = _token;
        beneficiary = _beneficiary;
        cliff = block.timestamp + _cliff;
        duration = _duration;
    }

    function release() public {
        require(block.timestamp >= cliff, "Cliff not reached");

        uint256 vested = vestedAmount();
        uint256 releasable = vested - released;
        require(releasable > 0, "No tokens to release");

        released += releasable;
        token.transfer(beneficiary, releasable);
    }

    function vestedAmount() public view returns (uint256) {
        uint256 totalBalance = token.balanceOf(address(this)) +
           released;

        if (block.timestamp < cliff) {
            return 0;
        } else if (block.timestamp >= cliff + duration) {
            return totalBalance;
        } else {
            return (totalBalance * (block.timestamp - cliff)) /
               duration;
        }
    }
}
```

4.3 Tokenomics Design Considerations

1. **Utility**: What does the token do?
    - Governance (voting rights)
    - Access (pay for services)

- Staking (earn rewards, secure network)
- Dividends (share revenue)

2. **Supply Schedule**: When are tokens released?
   - Avoid dumping large amounts at once
   - Align incentives with long-term growth
   - Consider market absorption capacity
3. **Allocation**: Who gets tokens?
   - Team (10-20%, vested)
   - Investors (10-30%, vested)
   - Community/Ecosystem (40-60%)
   - Treasury/Reserve (10-20%)
4. **Value Accrual**: How does token capture value?
   - Token burns (reduce supply)
   - Buybacks (market support)
   - Staking yield (incentivize holding)
   - Revenue sharing (cashflow to holders)

5 Security Considerations

5.1 Common Vulnerabilities

5.1.1 Integer Overflow/Underflow

Pre-Solidity 0.8.0 Issue:

```
// Vulnerable (Solidity < 0.8.0)
uint256 balance = 0;
balance -= 1; // Underflows to 2^256 - 1
```

Solution: Solidity 0.8+ has built-in overflow checking. For older versions, use SafeMath library.

5.1.2 Approval Race Condition

Problem: Changing approval from N to M can be exploited.

Attack Scenario:

1. Alice approves Bob for 100 tokens
2. Alice decides to change to 50, sends `approve(Bob, 50)`
3. Bob sees pending transaction, front-runs with `transferFrom(Alice, Bob, 100)`
4. Bob's transaction executes first (uses old 100 allowance)
5. Alice's transaction executes (sets allowance to 50)
6. Bob calls `transferFrom(Alice, Bob, 50)` again
7. Total stolen: 150 tokens

Mitigation:

```
// Set to 0 first, then set to new value
token.approve(spender, 0);
token.approve(spender, newAmount);

// Or use increaseAllowance/decreaseAllowance
token.increaseAllowance(spender, 50);
token.decreaseAllowance(spender, 30);
```

5.1.3 Unlimited Approvals

Common Pattern: `approve(spender, type(uint256).max)`

Risk: If spender contract is compromised, all tokens can be stolen.

Approval Best Practices

- Approve only what's needed for immediate transaction
- Revoke approvals after use
- Use tools like `https://revoke.cash` to audit/revoke approvals
- Consider using permit (EIP-2612) for gasless approvals

5.2 Audit and Testing

5.2.1 Testing Framework

Hardhat Test Example

```
const { expect } = require("chai");

describe("MyToken", function() {
  let token, owner, addr1, addr2;

  beforeEach(async function() {
    [owner, addr1, addr2] = await ethers.getSigners();
    const Token = await ethers.getContractFactory("MyToken");
    token = await Token.deploy(1000000);
  });

  it("Should assign total supply to owner", async function() {
    const ownerBalance = await token.balanceOf(owner.address);
    expect(await token.totalSupply()).to.equal(ownerBalance);
  });

  it("Should transfer tokens correctly", async function() {
    await token.transfer(addr1.address, 50);
    expect(await token.balanceOf(addr1.address)).to.equal(50);
  });

  it("Should fail if sender doesn't have enough tokens", async
     function() {
    const initialOwnerBalance = await token.balanceOf(owner.address
       );
    await expect(
      token.connect(addr1).transfer(owner.address, 1)
    ).to.be.revertedWith("ERC20: transfer amount exceeds balance");
  });

  it("Should handle allowance correctly", async function() {
    await token.approve(addr1.address, 100);
    expect(await token.allowance(owner.address, addr1.address))
      .to.equal(100);

    await token.connect(addr1).transferFrom(
      owner.address, addr2.address, 50
    );
    expect(await token.balanceOf(addr2.address)).to.equal(50);
    expect(await token.allowance(owner.address, addr1.address))
      .to.equal(50);
  });
});
```

5.2.2 Security Audit Checklist

1. **Access Control**
   - Only authorized addresses can mint
   - Owner cannot steal user tokens

- Ownership transfer is secure

2. **Arithmetic**
   - No overflow/underflow possible
   - Rounding errors minimized
   - Division by zero prevented
3. **External Calls**
   - Reentrancy protection where needed
   - Check return values
   - Gas limits considered
4. **Token Mechanics**
   - Transfer logic correct
   - Allowance handling secure
   - Events emitted properly
   - Decimal handling consistent
5. **Deployment**
   - Constructor parameters validated
   - Initial supply distributed correctly
   - Contract ownership set properly

6 Deployment and Verification

6.1 Deployment Process

Complete Deployment Script

```
const hre = require("hardhat");

async function main() {
  // Deployment parameters
  const NAME = "My Project Token";
  const SYMBOL = "MPT";
  const CAP = 100_000_000;         // 100 million cap
  const INITIAL_SUPPLY = 10_000_000; // 10 million initial

  console.log("Deploying token...");

  const Token = await hre.ethers.getContractFactory(
    "ComprehensiveToken"
  );
  const token = await Token.deploy(
    NAME,
    SYMBOL,
    CAP,
    INITIAL_SUPPLY
  );

  await token.deployed();

  console.log("Token deployed to:", token.address);
  console.log("Name:", await token.name());
  console.log("Symbol:", await token.symbol());
  console.log("Decimals:", await token.decimals());
  console.log("Total Supply:", ethers.utils.formatEther(
    await token.totalSupply()
  ));
  console.log("Cap:", ethers.utils.formatEther(await token.cap()));

  // Wait for 5 confirmations before verifying
  console.log("Waiting for block confirmations...");
  await token.deployTransaction.wait(5);

  // Verify on Etherscan
  console.log("Verifying contract...");
  await hre.run("verify:verify", {
    address: token.address,
    constructorArguments: [NAME, SYMBOL, CAP, INITIAL_SUPPLY],
  });

  console.log("Deployment complete!");
}

main().catch((error) => {
  console.error(error);
  process.exitCode = 1;
});
```

Execution:

```
npx hardhat run scripts/deploy.js --network sepolia
```

6.2 Contract Verification

Contract verification publishes source code on block explorers, enabling:

- Users to read contract code
- Direct interaction via explorer UI
- Trust and transparency
- Automated security checks

hardhat.config.js:

```
require("@nomiclabs/hardhat-etherscan");

module.exports = {
  solidity: "0.8.20",
  networks: {
    sepolia: {
      url: process.env.SEPOLIA_RPC_URL,
      accounts: [process.env.PRIVATE_KEY]
    }
  },
  etherscan: {
    apiKey: process.env.ETHERSCAN_API_KEY
  }
};
```

7 Study Questions

7.1 Conceptual Questions

1. Explain the difference between `transfer` and `transferFrom`.
2. Why is the approve-transferFrom pattern necessary for DEX interactions?
3. What are the risks of unlimited token approvals?
4. How does `decimals` affect token display vs. storage?
5. Compare fixed supply vs. inflationary token economics.

7.2 Implementation Exercises

1. Create an ERC-20 token with 2% transfer tax that goes to a treasury.
2. Implement a token with blacklist functionality to freeze malicious accounts.
3. Build a token that pays dividends to holders from contract balance.
4. Design a governance token where voting power increases with holding duration.

7.3 Security Analysis

1. Identify the approval race condition vulnerability and propose mitigation.
2. Write tests to verify all critical token functions.
3. Analyze a given token contract for potential security issues.
4. Design a safe token vesting contract with multiple beneficiaries.

7.4 Advanced Topics

1. Implement EIP-2612 (permit) for gasless approvals.
2. Create a token with snapshot functionality for airdrops.
3. Build a wrapper token (like WETH) that converts ETH to ERC-20.
4. Design a rebasing token that adjusts supply based on oracle price.

8 Further Reading

- EIP-20: Token Standard Specification
- OpenZeppelin ERC-20 Documentation and Code
- EIP-2612: Permit Extension (Gasless Approvals)
- EIP-4626: Tokenized Vault Standard
- ConsenSys: Token Implementation Best Practices
- Trail of Bits: Token Security Checklist
- DeFi Developer Roadmap: Token Standards
- Ethereum.org: Token Standards Overview