Solidity OpenZeppelin Quiz

Quiz 1: Understanding OpenZeppelin

Instructions: Neri is researching secure contract development to protect against Hackana. Which statement correctly describes OpenZeppelin?

```
// Option A: OpenZeppelin is a testing framework for Solidity contracts
// Option B: OpenZeppelin is a library of secure, reusable smart contract
components
// Option C: OpenZeppelin is a Solidity compiler with additional security features
// Option D: OpenZeppelin is a blockchain that specializes in security-focused
applications
```

Which statement correctly describes OpenZeppelin?

- A) Option A
- B) Option B
- C) Option C
- D) Option D

Answer: B) Option B

Explanation: OpenZeppelin is indeed a library of secure, reusable smart contract components. It provides:

- Well-tested implementations of standard contracts like ERC20 and ERC721 tokens
- Access control mechanisms like `Ownable` and role-based access
- Security utilities to prevent common vulnerabilities
- Upgrade patterns for deploying upgradable contracts

It's widely used in the Ethereum ecosystem to save development time and improve security by building upon code that has been audited and tested extensively. Developers can focus on their specific application logic rather than reimplementing standard functionalities.

Quiz 2: Using OpenZeppelin Contracts

Instructions: Neri is creating a token for the barangay to reward citizens for reporting Hackana activity. Which code correctly implements an ERC20 token using OpenZeppelin?

```
pragma solidity ^0.8.0;

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

contract BarangayToken is ERC20 {
    constructor() ERC20("Barangay Token", "BTK") {
        _mint(msg.sender, 1000000 * 10 ** decimals());
    }
}

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

contract BarangayToken {
    ERC20 token;

    constructor() {
        token = new ERC20("Barangay Token", "BTK");
        token._mint(msg.sender, 1000000 * 10 ** 18);
    }
}

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

contract BarangayToken {
    constructor() {
        ERC20("Barangay Token", "BTK");
        _mint(msg.sender, 1000000 * 10 ** 18);
    }
}

// Option D
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract BarangayToken is IERC20 {
    string public name = "Barangay Token";
    string public symbol = "BTK";
    uint8 public decimals = 18;
    uint256 public totalSupply = 1000000 * 10 ** 18;

    mapping(address => uint256) balances;

    constructor() {
        balances[msg.sender] = totalSupply;
    }

    function balanceOf(address account) public view override returns (uint256) {
        return balances[account];
    }

    // Other IERC20 function implementations...
}
```

Which option correctly implements an ERC20 token using OpenZeppelin?

- A) Option A
- B) Option B
- C) Option C

- D) Option D

Answer: A) Option A

Explanation: Option A correctly implements an ERC20 token using OpenZeppelin with these key elements:

1. It properly imports the ERC20 base contract from OpenZeppelin
2. It inherits from the ERC20 contract using the `is` keyword
3. It passes the token name and symbol to the ERC20 constructor
4. It uses the `_mint` function to create the initial token supply

The other options have issues:

- Option B doesn't inherit from ERC20, but tries to create a new instance and access a protected function
- Option C doesn't inherit from ERC20 and tries to call the constructor incorrectly
- Option D implements IERC20 from scratch, which defeats the purpose of using OpenZeppelin's battle-tested implementation

OpenZeppelin is most powerful when used through inheritance, allowing your contract to extend its functionality while maintaining all the security features of the base implementation.

Quiz 3: OpenZeppelin Access Control

Instructions: Neri is securing a funds management contract against Hackana attacks. Which OpenZeppelin approach best implements role-based access control?

```
pragma solidity ^0.8.0;

// Option A
import "@openzeppelin/contracts/access/Ownable.sol";

contract FundManager1 is Ownable {
    function withdrawFunds(uint256 amount) public onlyOwner {
        payable(owner()).transfer(amount);
    }
}

// Option B
import "@openzeppelin/contracts/access/AccessControl.sol";

contract FundManager2 is AccessControl {
    bytes32 public constant WITHDRAWER_ROLE = keccak256("WITHDRAWER_ROLE");

    constructor() {
        _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _setupRole(WITHDRAWER_ROLE, msg.sender);
    }

    function withdrawFunds(uint256 amount) public onlyRole(WITHDRAWER_ROLE) {
        payable(msg.sender).transfer(amount);
    }
    function addWithdrawer(address account) public onlyRole(DEFAULT_ADMIN_ROLE) {
        grantRole(WITHDRAWER_ROLE, account);
    }
}

// Option C
contract FundManager3 {
    address public owner;
    mapping(address => bool) public withdrawers;

    constructor() {
        owner = msg.sender;
        withdrawers[msg.sender] = true;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Not owner");
        _;
    }

    modifier onlyWithdrawer() {
        require(withdrawers[msg.sender], "Not authorized");
        _;
    }

    function withdrawFunds(uint256 amount) public onlyWithdrawer {
        payable(msg.sender).transfer(amount);
    }

    function addWithdrawer(address account) public onlyOwner {
        withdrawers[account] = true;
    }
}

// Option D
import "@openzeppelin/contracts/access/Ownable.sol";

contract FundManager4 is Ownable {
    mapping(address => bool) public withdrawers;

    function withdrawFunds(uint256 amount) public {
        require(withdrawers[msg.sender] || msg.sender == owner(), "Not
authorized");
        payable(msg.sender).transfer(amount);
    }

    function addWithdrawer(address account) public onlyOwner {
        withdrawers[account] = true;
    }
}
```

Which option provides the best role-based access control implementation using OpenZeppelin?

- A) Option A
- B) Option B
- C) Option C
- D) Option D

Answer: B) Option B

Explanation: Option B provides the best role-based access control implementation using OpenZeppelin's `AccessControl` contract, which is specifically designed for managing multiple roles and permissions.

Key advantages of Option B:

1. Uses OpenZeppelin's tested and audited `AccessControl` contract
2. Defines explicit roles with constants (`WITHDRAWER_ROLE`)
3. Uses the standardized `onlyRole` modifier for permission checks
4. Provides a function to add new withdrawers that's restricted to admins
5. Uses the built-in role hierarchy with `DEFAULT_ADMIN_ROLE`

The other options have limitations:

- Option A uses `Ownable` which only supports a single owner, not multiple roles
- Option C implements access control from scratch, which is error-prone and lacks the security audits of OpenZeppelin
- Option D mixes `Ownable` with a custom withdrawers mapping, which works but is less standardized and structured than `AccessControl`

For applications requiring multiple roles with different permissions, OpenZeppelin's `AccessControl` contract (Option B) provides the most robust and flexible solution.

Quiz 4: OpenZeppelin Contract Security

Instructions: Neri is reviewing contracts for security vulnerabilities that Hackana might exploit. Which OpenZeppelin feature should she use to prevent reentrancy attacks?

```
pragma solidity ^0.8.0;

// Option A
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract SecureVault1 is ReentrancyGuard {
    mapping(address => uint256) private _balances;

    function withdraw() public nonReentrant {
        uint256 amount = _balances[msg.sender];
        require(amount > 0, "No balance to withdraw");

        _balances[msg.sender] = 0;
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");
    }
    function deposit() public payable {
        _balances[msg.sender] += msg.value;
    }
}

// Option B
import "@openzeppelin/contracts/security/Pausable.sol";

contract SecureVault2 is Pausable {
    mapping(address => uint256) private _balances;

    function withdraw() public whenNotPaused {
        uint256 amount = _balances[msg.sender];
        require(amount > 0, "No balance to withdraw");

        _balances[msg.sender] = 0;
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");
    }

    function deposit() public payable {
        _balances[msg.sender] += msg.value;
    }
}

// Option C
import "@openzeppelin/contracts/utils/Address.sol";

contract SecureVault3 {
    using Address for address;

    mapping(address => uint256) private _balances;

    function withdraw() public {
        uint256 amount = _balances[msg.sender];
        require(amount > 0, "No balance to withdraw");

        _balances[msg.sender] = 0;
        address(msg.sender).sendValue(amount);
    }

    function deposit() public payable {
        _balances[msg.sender] += msg.value;
    }
}

// Option D
import "@openzeppelin/contracts/utils/math/SafeMath.sol";

contract SecureVault4 {
    using SafeMath for uint256;

    mapping(address => uint256) private _balances;

    function withdraw() public {
        uint256 amount = _balances[msg.sender];
        require(amount > 0, "No balance to withdraw");

        _balances[msg.sender] = 0;
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Transfer failed");
    }

    function deposit() public payable {
        _balances[msg.sender] = _balances[msg.sender].add(msg.value);
    }
}
```

Which option best prevents reentrancy attacks using OpenZeppelin?

- A) Option A
- B) Option B
- C) Option C
- D) Option D

Answer: A) Option A

Explanation: Option A correctly uses OpenZeppelin's `ReentrancyGuard` contract with the `nonReentrant` modifier to prevent reentrancy attacks.

Reentrancy attacks occur when an external contract call allows the called contract to make recursive calls back to the original function before the first execution completes. OpenZeppelin's `ReentrancyGuard` prevents this by using a mutex pattern.

Key features of Option A:

1. Inherits from `ReentrancyGuard`
2. Uses the `nonReentrant` modifier on the vulnerable `withdraw()` function
3. Updates state variables before making external calls (also a good practice)

The other options have issues:

- Option B uses `Pausable`, which helps with emergency stops but doesn't prevent reentrancy
- Option C uses `Address.sendValue()`, which is safer than raw `.call` but doesn't prevent reentrancy on its own
- Option D uses `SafeMath`, which prevents overflow/underflow but doesn't help with reentrancy

For functions that make external calls while managing contract state, using `ReentrancyGuard` is a critical security practice that OpenZeppelin makes easy to implement.