BLOCKCHAIN ARCHITECTURE AND DESIGN-II

CSC 403

CA2

Submitted By:

Gauri Sharma

Registration No.:

12108982

Roll No.:

21

B.Tech Hons. (Cse) In Cyber Security and Blockchain

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

LOVELY PROFESSIONAL UNIVERSITY

PHAGWARA, PUNJAB

Question No. Assigned: 3
Original Code :

```
function addUsers(address[] calldata admins, address[] calldata regularUsers, bytes calldata signature) external {
    if (!isAdmin[msg.sender]) {
        bytes32 hash = keccak256(abi.encodePacked(admins, regularUsers));
        address signer = hash.toEthSignedMessageHash().recover(signature);
        require(isAdmin[signer], "Only admins can add users.");
    }
    for (uint256 i = 0; i < admins.length; i++) {
        isAdmin[admins[i]] = true;
    }
    for (uint256 i = 0; i < regularUsers.length; i++) {
        isRegularUser[regularUsers[i]] = true;
    }
}
```

Identify the issues in the smart contract and fix the issue that you identify. Explain the core reason why the issue happens.

Answer:-

Original Code and Identified Problems

1. **Lack of Input Validation**

```
for (uint256 i = 0; i < admins.length; i++) {
    isAdmin[admins[i]] = true;
}
for (uint256 i = 0; i < regularUsers.length; i++) {
        isRegularUser[regularUsers[i]] = true;
}
```

Issue:

- The function does not validate that addresses in admins and regularUsers are non-zero. It also doesn't limit the number of users, which could lead to gas exhaustion.
- Also the function should ensure a check on non zero address to prevent DoS attacks.

2. **Replay Attack Vulnerability**

   function addUsers(address[] calldata admins, address[] calldata regularUsers, bytes calldata signature) external {

   bytes32 hash = keccak256(abi.encodePacked(admins, regularUsers));
   address signer = hash.toEthSignedMessageHash().recover(signature);

   Issues:

   - No nonce is included in the function, making it possible for a valid signature to be reused multiple times, leading to replay attacks.
   - Also validate the nonce as a unique transaction identifier and increment it after each successful call to prevent replay attacks.

3. **Hash Collision Vulnerability**

   bytes32 hash = keccak256(abi.encodePacked(admins, regularUsers));

   Issues:

   - Using abi.encodePacked() for hashing can create hash collisions if different inputs produce the same hash.

4. **Atomicity and Race Condition Prevention(Does not occur generally as solidity handles one transaction at a time)**

   ```
   for (uint256 i = 0; i < admins.length; i++) {
      isAdmin[admins[i]] = true;
   }
   for (uint256 i = 0; i < regularUsers.length; i++) {
      isRegularUser[regularUsers[i]] = true;
   }
   ```

   Issues:

   - Separate loops for updating isAdmin and isRegularUser mappings may result in incomplete state updates if the function fails midway.

CORRECTED CODE WITH EXPLAINATION

```
function addUsers(
    address[] calldata admins,
    address[] calldata regularUsers,
```

uint256 nonce,

bytes calldata signature

) external **nonReentrant**{

nonReentrant: Just a check / Confirmation to prevent any renterancy attack due to external call

1. Initial Validations

require(isAdmin[msg.sender], "Caller is not an admin"); // Ensure caller is an admin

require(nonce == adminTxNonce[msg.sender], "Invalid nonce"); // Verify unique nonce for replay protection

require(admins.length + regularUsers.length <= 100, "Too many users to add"); // Limit total users to avoid gas limit issues

Solution:

- **CheckAdmin Status**: The require(isAdmin[msg.sender], "Caller is not an admin"); line checks if the msg.sender (the caller of the function) is an admin. If not, it reverts the transaction with an error message, "Caller is not an admin." This ensures only admins can call this function.
- **Nonce Validation for Replay Protection**: The require(nonce == adminTxNonce[msg.sender], "Invalid nonce"); line checks if the provided nonce matches the expected adminTxNonce for the msg.sender. The nonce is a unique identifier used to prevent replay attacks by ensuring each transaction can only be executed once per admin.
- **User Limit on User Additions**: The require(admins.length + regularUsers.length <= 100, "Too many users to add"); line limits the number of users that can be added in a single transaction to prevent excessive gas costs. If the combined count of admins and regularUsers exceeds 100, it reverts with the message "Too many users to add."

2. Hash Calculation (using abi.encode and including contract address to avoid hash collisions and cross-contract replay)

bytes32 hash = keccak256(abi.encode(address(this), admins, regularUsers, nonce));

address signer = hash.toEthSignedMessageHash().recover(signature); // Recover signer address from the signature

Solution:

- Unique and Secure Hashing: Combines address(this), admins, regularUsers, and nonce in the abi.encode function to prevent hash collisions and cross-

contract replay.

- **address(this)**: Includes the contract's address to prevent hash collisions with identical data in other contracts. It ensures that the signature cannot be reused in a different contract with similar code.

- hash.toEthSignedMessageHash() converts the hash into an Ethereum Signed Message. This adds a prefix to the hash, indicating it's an Ethereum-signed message.

3. Signature Verification

require(isAdmin[signer], "Only admins can add users"); // Ensure signer is an authorized admin

Solution:

- Checks that the signer (derived from the signature) is indeed an admin, which confirms authorization, if a regular user is adding another user it only allows the ones that are validated by the admin.

4. Update Mappings in a Single Loop

```
for (uint256 i = 0; i < admins.length; i++) {
    require(admins[i] != address(0), "Invalid admin address"); // Check for zero address in admins
    isAdmin[admins[i]] = true;
}
for (uint256 i = 0; i < regularUsers.length; i++) {
    require(regularUsers[i] != address(0), "Invalid regular user address"); // Check for zero address in regular users
    isRegularUser[regularUsers[i]] = true;
}
```

Solution:

- **Validation of Each Address:** Checks that addresses in both admins and regularUsers are non-zero, preventing unintended roles for invalid addresses.

- **Add to isAdmin Mapping**: isAdmin[admins[i]] = true; marks each address in admins as an admin by setting its corresponding value in isAdmin to true.

5. Increment Nonce after Successful Execution

adminTxNonce[msg.sender]++;

}

Solution:

- After all validations and updates, adminTxNonce[msg.sender]++ increments the nonce for msg.sender. This ensures that the same nonce cannot be reused in future transactions, providing replay protection. This action only occurs if the function completes successfully, so the nonce is only updated once per unique valid transaction.