Secureum Quiz

## Structure:

The Certora RACE-X will be composed of 24 questions split into 3 sections:

Part 1: Propositional Logic - 10 Questions.

Part 2: Properties of DeFi Systems - 12 questions.

Part 3: Questions On The Released Videos - 2 questions.

**Total time**: 64 + 32 mins = 96 mins

## Common Language:

The RACE-X uses well known logic symbols.. The above table presents them to make sure they are clear to everyone.

|  |  |  |
| --- | --- | --- |
| Symbol | Free Language | Logic Meaning |
| ⋀ | And | And |
| ￢ | Not | Not |
| ⋁ | Or | Or |
| ⇒ | Imply/Implies | Implication |
| ⇔ | Iff/Equivalent | Equivalency |

## Instructions:

i

1. Please read all these instructions very carefully before starting the quiz  
   **Structure**: The Certora RACE-X is composed of **24 questions** split into 3 sections —  
   Part 1: *Propositional Logic* (Q1-Q10)  
   Part 2: *Properties of DeFi Systems* (Q11-Q22)  
   Part 3: Questions on (earlier shared) Certora Videos (Q23-Q24)

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Free Language** | **Logic Meaning** |
| ⋀ | And | And |
| ￢ | Not | Not |
| ⋁ | Or | Or |
| ⇒ | Imply/Implies | Implication |
| ⇔ | Iff/Equivalent | Equivalency |

**Common Language**: The RACE-X uses well known logic symbols. The above table presents them here to make sure they are clear to everyone.

1. This is a **single-attempt** multiple-choice RACE-X.
2. The **time limit is 96 minutes**. It **does *not* allow you to stop and resume the quiz**. So please take the RACE-X when you are completely ready.
3. All questions have check-boxes and so **please choose all and only the correct answers**. There may only be 1 correct answer or more than 1. Partial points will be granted.
4. You must answer a question to proceed to the next one. You are allowed to scroll back to previous questions to change their answers.
5. **Warning: Strictly no exceptions can be made.**

## Part 1: Propositional Logic

1. [Q1] When is the expression p ⇒ q **false**? D   
     
   (A): ￢p ⋀ ￢q   
   (B): p ⋀ q   
   (C): ￢p ⋀ q  
   (D): None of the above

## 

## [Q2] Is the following expression **true**? C p ⋁ q ⇒ p (A): True in all cases (B): False in all cases (C): True for some cases and false for others (D): None of the above

1. [Q3] Is the following expression **true**? B  
    (p ⋀ (q ⋁ ￢p)) ⋀ ￢q  
     
   (A): True in all cases  
   (B): False in all cases  
   (C): True for some cases and false for others  
   (D): None of the above
2. [Q4] Is the following expression **true**? A  
    ￢(￢p ⋁ ￢q ⋁ ￢r) ⇔ p ⋀ q ⋀ r  
     
   (A): True in all cases  
   (B): False in all cases  
   (C): True for some cases and false for others  
   (D): None of the above
3. [Q5] Is the following expression **true**? C  
    ￢((￢p ⋁ q) ⋀ (p ⋁ ￢q))  
     
   (A): True in all cases  
   (B): False in all cases  
   (C): True for some cases and false for others  
   (D): None of the above
4. [Q6] Is the following expression **true**? C  
    ￢((p ⋀ q) ⋁ (￢p ⋀ ￢q))  
     
   (A): True in all cases  
   (B): False in all cases  
   (C): True for some cases and false for others  
   (D): None of the above
5. [Q7] Given the expression: **M**: p ⋁ ￢p  
    Which of the following expressions implies the above given expression **M**? ABCD  
     
   (A): p ⋁ ￢p  
   (B): p ⋀ ￢p  
   (C): (p ⋀ ￢p) ⋁ ￢p  
   (D): (￢p ⋁ p) ⋀ ￢p
6. [Q8] Given the expression: **M**: p ⋀ ￢p   
   Which of the following expressions are implied by the above given expression **M**? ABCD  
     
   (A): p ⋁ ￢p  
   (B): p ⋀ ￢p  
   (C): (p ⋀ ￢q) ⋁ ￢r  
   (D): (￢p ⋁ p) ⋀ ￢p
7. [Q9] Is the following expression **true**?   
    (p ⇒ (q ⇒ r)) ⇒ ((p ⇒ q) ⇒ r) C  
     
   (A): True in all cases  
   (B): False in all cases  
   (C): True for some cases and false for others  
   (D): None of the above

1. [Q10] Is the following expression **true**?  
    ((p ⇒ q) ⇒ r) ⇒ (p ⇒ (q ⇒ r)) A  
     
   (A): True in all cases  
   (B): False in all cases  
   (C): True for some cases and false for others  
   (D): None of the above

## Part 2: Properties Of DeFi Systems

Given the below four properties for [OpenZeppelin's ERC20 implementation](https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol), answer **Q11-Q13**:   
  
**P1**: Forall user: *balanceOf(user)* can only change on *mint()*, *burn()*, *transfer()*, *transferFrom()*  
**P2**: *totalSupply()* is the sum of *balanceOf()* over all users   
**P3**: Forall user: *balanceOf(user) <= totalSupply()***P4**: Forall user: *balanceOf(user)* can only change on operation performed when *msg.sender==user* or when *allowance(user, msg.sender)* is not zero

1. [Q11] Which of the below are correct properties of ERC20? ABC  
   (A): Property P1(B): Property P2  
   (C): Property P3  
   (D): Property P4

given the following buggy version of *transferFrom()*:

function transferFrom(address \_from, address \_to, uint256 \_value) public returns (bool) {

require(\_to != address(0));

require(\_value <= balances[\_from]);

require(\_value <= allowed[\_from][msg.sender]);

balances[\_from] = balances[\_from].add(\_value);

balances[\_to] = balances[\_to].sub(\_value);

allowed[\_from][msg.sender] = allowed[\_from][msg.sender].sub(\_value);

emit Transfer(\_from, \_to, \_value);

return true;

}

1. [Q12] Which of the below properties are violated? D  
     
   (A): Property P1  
   (B): Property P2  
   (C): Property P3  
   (D): None of the above
2. [Q13] Is there an implication between **P3** and **P2**? A  
     
   (A): Yes, P2 ⇒ P3  
   (B): Yes, P3 ⇒ P2  
   (C): Yes, P2 ⇔ P3  
   (D): No

Assuming a correct implementation of transferFrom, as in [OpenZeppelin's ERC20 implementation](https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol), and given the following pseudo unit-test code:

uint256 bFrom = balances[bFrom];  
uint256 bTo = balances[to];  
transferFrom(from, to, x);

assert($exp$);

1. [Q14] Which of the following expressions are valid choices (should always hold) for *$exp$*? BC  
     
   (A): balances[from] == bFrom - x  
   (B): from != to || balances[to] == bTo  
   (C): balances[to] + balances[from] == bFrom + bTo  
   (D): None of the above

The below contract (same contract for Qs 15-17) has a bug:

contract test {

// Assume other required functionality is correctly implemented

uint256 private constant MAX\_FUND\_RAISE = 100 ether;

mapping (address => uint256) contributions;

function contribute() external payable {

require(address(this).balance != MAX\_FUND\_RAISE);

contributions[msg.sender] += msg.value;

}

}

1. [Q15] Which of the following invariants should hold on a correct implementation of the contribute function? BCD  
     
   P1: Forall user: address(this).balance ≤ contributions[user]  
   P2: Forall user: contributions[user] ≤ address(this).balance  
   P3: Forall user: contributions[user] ≤ MAX\_FUND\_RAISE  
   P4: address(this).balance ≤ MAX\_FUND\_RAISE  
     
   (A): P1  
   (B): P2  
   (C): P3  
   (D): P4

1. [Q16] Which of the following invariant(s) that is/are supposed to hold is/are violated due to the buggy implementation? CD  
     
   (A): P1  
   (B): P2  
   (C): P3  
   (D): P4
2. [Q17] The revert characteristic (conditions in which the function should revert) of a correct implementation of contribute is: B  
     
   (A): *msg.value == 0*  
   (B): *address(this).balance + msg.value > MAX\_FUND\_RAISE*  
   (C): *msg.value == MAX\_FUND\_RAISE*  
   (D): None of the above

In the below contract (same contract for Qs 18-19):

### pragma solidity 0.7.0;

### contract InSecureumToken {

### mapping(address => uint) private balances; uint public decimals = 10\*\*18; // decimals of the token uint public totalSupply; // total supply uint MAX\_SUPPLY = 100 ether; // Maximum total supply event Mint(address indexed destination, uint amount);

### function balanceOf(address u) public returns (uint256) { return balances[u]; }

### function ethBalance(address u) public returns (uint256) { return u.balance; }

### function transfer(address to, uint amount) public { // save the balance in local variables // so that we can re-use them multiple times // without paying for SLOAD on every access

### uint balance\_from = balances[msg.sender]; uint balance\_to = balances[to]; require(balance\_from >= amount); balances[msg.sender] = balance\_from - amount; balances[to] = safeAdd(balance\_to, amount); }

### /// @notice Allow users to buy a token. 1 ether = 10 tokens /// @dev Users can send more ether than token to be bought, to donate a fee to the protocol team.

### function buy(uint desired\_tokens) public payable { // Check if enough ether has been sent uint required\_wei\_sent = (desired\_tokens / 10) \* decimals; require(msg.value >= required\_wei\_sent); // Mint the tokens totalSupply = safeAdd(totalSupply, desired\_tokens); balances[msg.sender] = safeAdd(balances[msg.sender], desired\_tokens);

### emit Mint(msg.sender, desired\_tokens); }

### /// @notice Add two values. Revert if overflow

### function safeAdd(uint a, uint b) pure internal returns(uint) { if (a + b < a) { revert(); } return a + b; }

### }

Given the following two properties:  
  
**P1**: *totalSupply()* is the sum of *balanceOf()* over all users.  
**P2**: Monotonicity of totalSupply vs the contract's ether balance:   
 (a) totalSupply is increased iff (⇔) this.balance is increased and  
 (b) totalSupply is decreased iff (⇔) this.balance is decreased

1. [Q18] Which of the existing issues in the code violates which property? BC  
     
   (A): An issue in buy() violates P1  
   (B): An issue in buy() violates P2  
   (C): An issue in transfer() violates P1  
   (D): An issue in transfer() violates P2

1. [Q19] Assuming a correct implementation of *buy()* and *transfer()*, which properties should hold? B  
     
   (A): The order of operation *buy()* and *transfer()* is not important, i.e. first calling buy and then transfer has the same outcome as first calling transfer and then buy.  
   (B): transfer is additive i.e. performing transfer in two steps:

transfer(to,x);

transfer(to,y);

Is equivalent to performing it in one step:

transfer(to,x+y);

(C): buy is additive i.e. performing buy in two steps:

buy(x1){value:x};

buy(y1){value:y};

Is equivalent to performing it in one step:

buy(x1+y1){value:x+y};

(D): None of the above

1. [Q20] In [OpenZeppelin’s implementation of ERC721](https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol), which of the following properties are correct specification assuming *user != 0*? C  
     
   (A): ownerOf(tokenId) == user ⇔ balanceOf(user) == tokenId  
   (B): ownerOf(tokenId) == user ⇒ balanceOf(user) == 1  
   (C): ownerOf(tokenId) == user ⇒ balanceOf(user) ≥ 1  
   (D): None of the above
2. [Q21] In [OpenZeppelin’s implementation of ERC721](https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol), which of the following is necessarily correct after a successful (non-reverting) call to *transferFrom(from, to, tokenId)*? AC  
     
   (A): *ownerOf(tokenId) == to*(B): *ownerOf(tokenId) == from*  
   (C): balanceOf(to) ≥ 1  
   (D): balanceOf(from) ≥ balanceOf(to)

1. [Q22] In [OpenZeppelin’s implementation of ERC721 Enumerable](https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/extensions/ERC721Enumerable.sol), which of the following expressions is true? AC  
     
   (A): tokenByIndex(i) == j ⇒ i < totalSupply()  
   (B): i < totalSupply() ⇒ tokenByIndex(i) ≠ 0  
   (C): (u != o && tokenOfOwnerByIndex(o, i) == j) ⇒ ownerOf(j) ≠ u  
   (D): None of the above

## Part 3: Auditing and Formal Verification

1. [Q23] Based on the lecture on ["Auditing and Formal Verification: Better together"](https://www.youtube.com/watch?v=VGSsPIsbb6U) by Certora's CEO Mooly Sagiv, which of the following is generally accepted? CD  
     
   (A): Formal verification eliminates the need for auditing  
   (B): Auditing eliminates the need for formal verification  
   (C): Auditing may find bugs after a project been formally verified  
   (D): Formal verification may find bugs after a project has been audited
2. [Q24] Based on the lecture on ["Auditing and Formal Verification: Better together"](https://www.youtube.com/watch?v=VGSsPIsbb6U) by Certora's CEO Mooly Sagiv, the takeaways are: ABC  
     
   (A): Spec is the law  
   (B): Writing correct spec is challenging  
   (C): Spec should be audited  
   (D): None of the above