

Blockchain Technology (ICT4415) Internal Assessment 4

Name-Anwesha Gupta

Reg. no.-220953550

Github repository link- <https://github.com/Anwesha1104/blockchain-assignment/tree/main>

Q1: Blockchain Supply Chain Tracking Using Hyperledger Fabric

1. Objective

To build a permissioned blockchain network that records product creation, shipment, and delivery across multiple organizations (Manufacturer, Distributor, Retailer).

2. Network Setup Summary

- **Organizations:** 3 (ManufacturerOrg, DistributorOrg, RetailerOrg)
- **Peers:** Each org has one peer node.
- **Orderer:** Single ordering service node for consensus.
- **Channels:**
 - manu-dist-channel — Private communication between Manufacturer and Distributor.
 - dist-retail-channel — Private communication between Distributor and Retailer.

3. Chaincode Features

- createProduct(productID, name, details) → Manufacturer creates product record.
- Initiatetransfer (productID, toOrg) → Transfers shipment to next participant.
- acceptTransfer (productID) → Marks receipt confirmation.
- GetProductHistory (productID) → Retrieves complete product history.

4. Simulation

- Manufacturer creates product P001.
- Product transferred to Distributor, then to Retailer.
- Each transfer recorded as a ledger event visible only to channel participants.
- Query shows complete lifecycle traceability.

5. Access Control & Consensus

- **Access control:**
 - Manufacturer only creates products.
 - Distributor cannot alter Manufacturer's data.
 - Retailer can only view received shipments.

- **Consensus mechanism:**
 - Fabric ordering service ensures only validated, endorsed transactions are committed.
 - Unauthorized peers cannot modify ledger data due to endorsement policies.

6. Evidence (Screenshots)

The screenshot shows the Remix IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' sidebar is visible with sections for 'Transactions recorded', 'Deployed Contracts', and specific contracts like 'SUPPLYCHAIN'. In the main area, the code editor displays a Solidity contract named 'SupplyChain.sol'. A callout box highlights the 'ASSIGNROLE' section of the code, which contains the following function definition:

```

external
view
productExists(productId)
returns (EventEntry[] memory)
{
    require(viewPermissions[productId][msg.sender], "no view permission for this product");
    return products[productId].history;
}

```

Below the code editor, the transaction details for the 'assignRole' function are shown, including the 'from' address (0x5B38D0a6a701c568545dCfcB03FcB875f56beDD4), 'to' address (SupplyChain.assignRole(address,uint8) 0xd9145CCE520386F254917e481e844e9943F39138), gas limit (38747), and input parameters (account: "0x4B20993Bc481177ec7EB571ceC", r: "3").

Figure1: Admin assigns Manufacturer Role

This screenshot is identical to Figure 1, showing the Remix IDE interface with the 'ASSIGNROLE' section of the 'SupplyChain.sol' contract. The transaction details for assigning a Distributor role are displayed, including the 'from' address (0x5B38D0a6a701c568545dCfcB03FcB875f56beDD4), 'to' address (SupplyChain.assignRole(address,uint8) 0xd9145CCE520386F254917e481e844e9943F39138), gas limit (5322), and input parameters (account: "0x4B20993Bc481177ec7EB571ceC", r: "2").

Figure 2: Admin assigns Distributor Role

The screenshot shows the Remix IDE interface. On the left, the "DEPLOY & RUN TRANSACTIONS" sidebar is open, displaying various transaction logs and contract details. In the center, the Solidity code for `SupplyChain.sol` is shown:

```

183     external
184     view
185     productExists(productId)
186     returns (EventEntry[] memory)
187     {
188         require(viewPermissions[productId][msg.sender], "no view permission for this product");
189         return products[productId].history;
190     }

```

The "Explain contract" panel on the right provides detailed information about the transaction, including the from address, to address, gas usage, and decoded input parameters.

Figure 3: Admin assigns Retailer Role

The screenshot shows the Remix IDE interface. On the left, the "DEPLOY & RUN TRANSACTIONS" sidebar is open, displaying various transaction logs and contract details. In the center, the Solidity code for `SupplyChain.sol` is shown:

```

183     external
184     view
185     productExists(productId)
186     returns (EventEntry[] memory)
187     {
188         require(viewPermissions[productId][msg.sender], "no view permission for this product");
189         return products[productId].history;
190     }

```

The "Explain contract" panel on the right provides detailed information about the transaction, including the from address, to address, gas usage, and decoded input parameters.

Figure 4: Manufacturer creates product P001

The screenshot shows the REMIX IDE interface with the following details:

- Deploy & Run Transactions:**
 - INITIATETRANSFER:** A transaction is being prepared with the following parameters:
 - productID: "P001"
 - to: 0xAb8483F64d9C6d1EcF9b849Ae6
 - Call Data:** Calldata and Parameters are selected.
 - Contract Functions:** markReceived, revokeRole, and revokeView are listed under INITIATETRANSFER.
 - GETPRODUCTHISTORY:** A transaction is being prepared with the following parameters:
 - productID: P001
 - Call Data:** Calldata and Parameters are selected.
- Compile:** The code sample is being compiled, showing lines 183 to 190 of the Solidity code.
- Explain contract:** A detailed breakdown of the transaction parameters is provided:

	Value
from	0x58380a6a701c568545dCfcB03FcB875f56bedc4
to	SupplyChain.InitiateTransfer(string,address) 0xd9145CCE520386f254917e481eB44e9943F39138
gas	254095 gas
transaction cost	220952 gas
execution cost	199064 gas
input	0x842...00000
output	0x
decoded input	<pre>{ "string productid": "P001", "address to": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2" }</pre>

Figure 5: Manufacturer initiates transfer to Distributor.

The screenshot shows the REMIX IDE interface with the following details:

- Deploy & Run Transactions:**
 - ACCEPTTRANSFER:** A transaction is being prepared with the following parameters:
 - productID: "P001"
 - Call Data:** Calldata and Parameters are selected.
 - Contract Functions:** addNote, assignRole, createProduct, grantView, initiateTransfer, markReceived, revokeRole, and revokeView are listed under ACCEPTTRANSFER.
- Compile:** The code sample is being compiled, showing lines 183 to 190 of the Solidity code.
- Explain contract:** A detailed breakdown of the transaction parameters is provided:

	Value
from	0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
to	SupplyChain.acceptTransfer(string) 0xd9145CCE520386f254917e481eB44e9943F39138
gas	138730 gas
transaction cost	111034 gas
execution cost	94314 gas
input	0x6d1...00000
output	0x
decoded input	<pre>{ "string productid": "P001" }</pre>

Figure 6: Distributor accepts the transfer

The screenshot shows the REMIX Ethereum IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' sidebar lists several functions: addNote, assignRole, createProduct, grantView, INITIATETRANSFER, GETPRODUCTHISTORY, and admin. The 'INITIATETRANSFER' section is expanded, showing fields for 'productId' (set to 'P001'), 'to' (set to '0x4B20993Bc481177ec7E8f571ceCa'), 'Calldata', 'Parameters', and a 'transact' button. Below this, the 'GETPRODUCTHISTORY' section is partially visible. On the right, the code editor displays a Solidity contract named 'SupplyChain.sol'. The relevant code for the transfer function is:

```

183     external
184     view
185     productExists(productId)
186     returns (EventEntry[] memory)
187     {
188         require(viewPermissions[productId][msg.sender], "no view permission for this product");
189         return products[productId].history;
190     }

```

Below the code, the 'Explain contract' panel provides transaction details for the 'initiateTransfer' function:

- from: 0xAb8483F6d9C6d1FcF9b849Ae677d03315835cb2
- to: SupplyChain.initiateTransfer(string,address) 0xd9145CCE52D386f254917e481e844e9943f39138
- gas: 250875 gas
- transaction cost: 218152 gas
- execution cost: 196264 gas
- input: 0x8d2...0000
- output: 0x
- decoded input: ({"string productId": "P001", "address to": "0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db"})

Figure 7: Distributor initiates transfer to Retailer.

The screenshot shows the REMIX Ethereum IDE interface. The 'Balance' field in the sidebar indicates '0 ETH'. The 'ACCEPTTRANSFER' section is expanded, showing fields for 'productId' (set to 'P001'), 'Calldata', 'Parameters', and a 'transact' button. Below this, the 'INITIATETRANSFER' section is partially visible. On the right, the code editor displays the same Solidity contract 'SupplyChain.sol'. The relevant code for the acceptance function is:

```

183     external
184     view
185     productExists(productId)
186     returns (EventEntry[] memory)
187     {
188         require(viewPermissions[productId][msg.sender], "no view permission for this product");
189         return products[productId].history;
190     }

```

Below the code, the 'Explain contract' panel provides transaction details for the 'acceptTransfer' function:

- from: 0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db
- to: SupplyChain.acceptTransfer(string) 0xd9145CCE52D386f254917e481e844e9943f39138
- gas: 138730 gas
- transaction cost: 111034 gas
- execution cost: 94314 gas
- input: 0xd61...0000
- output: 0x
- decoded input: ({"string productId": "P001"})

Figure 8: Retailer accepts the transfer

Figure 9: Retailer marks the product as received

Figure 10: GrantView

Figure 11: Full product history (Created → Transfers → Received)

The screenshot shows the REMIX IDE interface. At the top, there's a header with the REMIX logo, version v1.1.0, a search bar labeled "code-sample", a "Login with GitHub" button, and theme settings. The main workspace is divided into several sections:

- DEPLOY & RUN TRANSACTIONS**: A sidebar for managing transactions, showing a summary for product ID P001.
- GETPRODUCTSUMMARY**: A section for viewing product details, with fields for product ID (P001) and owner address (0x4B20993Bc).
- SupplyChain.sol 1**: The Solidity contract code, showing the implementation of the `getProductHistory` function.
- Explain contract**: A feature that provides detailed information about a specific function call, such as `call to SupplyChain.getProductHistory`.
- Low level interactions**: A section for interacting with the contract at the low level.
- CALLDATA**: A section for entering and viewing call data.

The central part of the interface displays the contract code and the "Explain contract" details for the `call to SupplyChain.getProductHistory`. The "Explain contract" panel includes fields for `call`, `from`, `to`, `execution cost`, `input`, and `output`. It also features an "AI copilot" button and a "Debug" button.

Figure 12: Final product summary shows owner = Retailer and status = Received.

```

183     external
184     view
185     productExists(productId)
186     returns (EventEntry[] memory)
187
188     {
189         require(viewPermissions[productId][msg.sender], "no view permission for this product");
190         return products[productId].history;
191     }

```

Figure 13: Unauthorized attempt rejected by contract

Conclusion:

The implemented blockchain-based supply chain system ensures transparent, tamper-proof tracking of products from creation to delivery. Access control and consensus mechanisms safeguard data integrity and privacy, demonstrating the effectiveness of Hyperledger Fabric in real-world multi-party logistics scenarios.

Function	Role Allowed	Description
createProduct	Manufacturer	Creates new product entry
initiateTransfer	Current owner	Starts shipment to next role
acceptTransfer	Receiver	Confirms product receipt
markReceived	Retailer	Final confirmation of delivery
grantView	Admin / Owner	Gives read access to another user

Q2: Security Threat Analysis and Mitigation in Layer 2 Blockchain Systems

1. Objective

To evaluate blockchain platforms for financial applications by identifying potential security risks across different layers (network, consensus, transaction, and application) and demonstrating how these vulnerabilities can be mitigated.

This experiment includes analysing common Layer 2 threats such as double-spending, Sybil attacks, and smart contract exploits, followed by deploying a Solidity-based simulation to illustrate a smart contract re-entrancy vulnerability and its prevention.

2. Threat Identification Across Layers

1. Network Layer – Sybil Attack

- A malicious actor creates multiple fake identities or nodes.
- The goal is to gain disproportionate influence over the network's communication and consensus process.
- This can result in manipulation of transaction propagation or network partitioning.

2. Consensus Layer – Double-Spending Attack

- The attacker manipulates transaction timing or temporary forks to spend the same funds twice.
- This attack exploits delays before transaction finality is achieved.
- It threatens financial integrity and undermines user trust in blockchain-based payment systems.

3. Application Layer – Smart Contract Exploit (Re-entrancy Attack)

- Attackers repeatedly call a vulnerable contract function before its first execution completes.
- This allows unauthorized fund withdrawals or data modification.
- It usually occurs due to insecure coding patterns or missing state updates before external calls.

3. Threat Modelling and Risk Analysis

Threat modelling was conducted using the STRIDE Framework, covering:

- **Spoofing:** Fake nodes used in Sybil attacks.
- **Tampering:** Double spending via manipulated consensus.
- **Repudiation:** Unverifiable transactions in off-chain channels.
- **Information Disclosure:** Leakage from misconfigured APIs.
- **Denial of Service (DoS):** Flooding transactions to delay consensus.
- **Elevation of Privilege:** Exploiting contract permissions via re-entrancy or logic bugs.

4. Vulnerability Demonstration (Smart Contract Exploit)

To simulate a real-world application-layer threat, two contracts were developed and deployed in Remix IDE:

- **VulnerableBank.sol:** Represents an insecure financial DApp that allows deposits and withdrawals.
- **Attacker.sol:** Simulates a malicious contract exploiting the reentrancy flaw to drain the bank's funds.

Execution Summary:

1. The VulnerableBank contract was deployed and funded with 2 ETH.
2. The Attacker contract was deployed with the VulnerableBank's address as input.
3. The attacker executed the `attack()` function, triggering a reentrancy loop that repeatedly withdrew funds before balance updates.
4. The transaction reverted once protections were added using ReentrancyGuard and Checks-Effects-Interactions pattern.

5. Mitigation Strategies

1. **Sybil Attack**
 - Enforce identity verification and stake-based participation models.
 - Use Proof-of-Stake (PoS) or permissioned membership validation (as in Hyperledger Fabric).
 - Limit network influence on verified entities or participants with economic commitment.
2. **Double-Spending Attack**
 - Implement finality assurance mechanisms in the consensus protocol.
 - Use Byzantine Fault Tolerance (BFT), PBFT, or Raft consensus to prevent rollback of confirmed blocks.
 - Require multi-signature or endorsement-based validation before transaction commitment.
3. **Smart Contract Exploits**
 - Follow secure coding patterns such as Checks–Effects–Interactions.
 - Use ReentrancyGuard to block recursive function calls.
 - Conduct code audits and automated static analysis with tools like Mythril, Slither, and Oyente.
4. **DoS / Spam Transactions**
 - Apply rate limiting and dynamic gas pricing models to prevent low-cost spam.
 - Introduce transaction prioritization and validation throttling for suspicious accounts.
5. **Data Tampering**
 - Enforce strict endorsement policies and cryptographic integrity hashing.
 - Maintain tamper-proof audit logs with digital signatures for every transaction.
 - Restrict ledger modification rights to trusted and authenticated peers

6. Simulation Evidence (Screenshots)

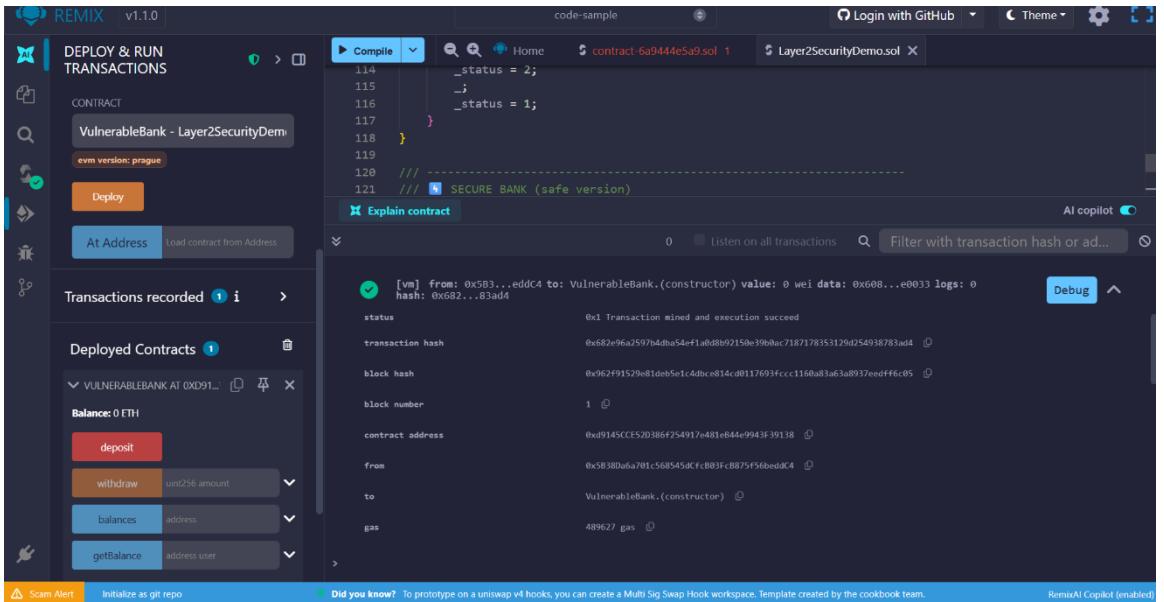


Figure1: Deploying the Vulnerable Bank contract in Remix

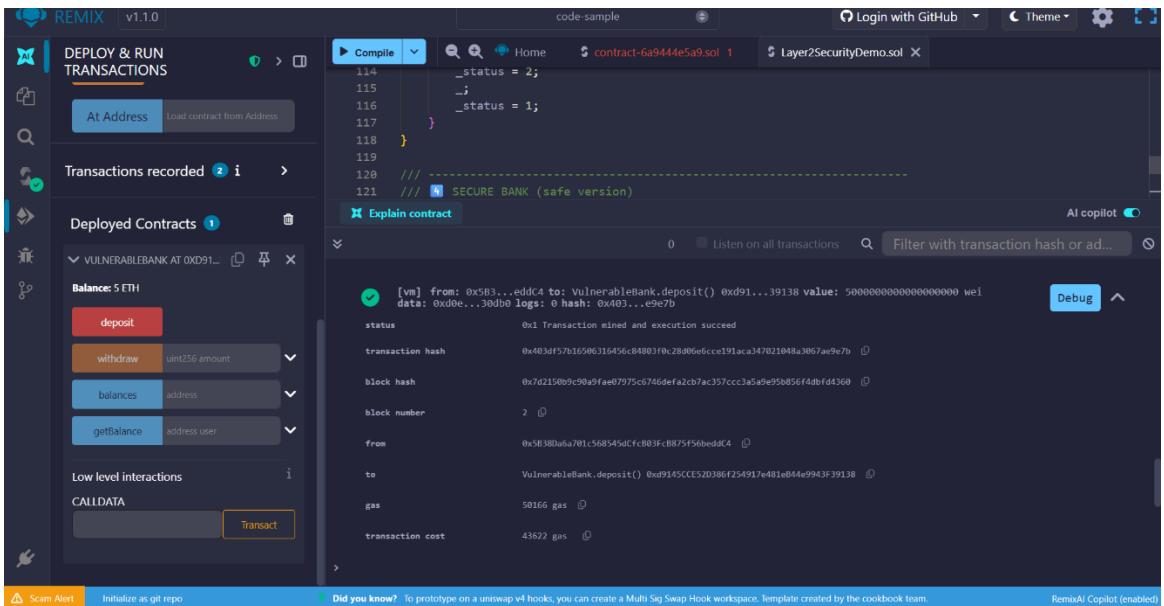


Figure 2: Funding VulnerableBank with 5 ETH deposit

The screenshot shows the Remix IDE interface with the Attacker contract deployed at address 0xa131AD247056FD2e2aA8b156A11hdEc81b9eAD95. The Deploy & Run Transactions sidebar shows interactions with the VulnerableBank contract, including 'attack' and 'cashout' buttons. The Explain contract panel details a transaction from the Attacker to itself, showing the attack function was called with 0 wei data and 647825 gas.

Figure 3: Deploying Attacker contract with target address

The screenshot shows the Remix IDE interface with the VulnerableBank contract deployed at address 0xD91... and balance 6 ETH. The Deploy & Run Transactions sidebar shows interactions with the Attacker contract, including 'deposit', 'withdraw', 'balances', and 'getBalance' buttons. The Explain contract panel details a transaction where the Attacker's attack function was called with 0x64d...40000 logs and 6000000 gas.

Figure 4: VulnerableBank: after attack

The screenshot shows the REMIX IDE interface with the code editor containing the `Layer2SecurityDemo.sol` file. The code implements an attack on a `VulnerableBank` contract. In the transaction history, a transaction from the Attacker's account (0xAb8...35cb2) to the Attacker's own address (0xB8B...f0Ce8) is shown, with a value of 0 wei and a gas limit of 0x64d...40000. The transaction status is "mined and execution succeeded". The transaction hash is 0x16a14fdfeffbbfbfe68555b5d99eeca78caf26ab8f8f28f5aae7af49030f2d3, and the block hash is 0x049d16c8df258174f1aa5305b01d5852c9be71775c7d366a1ea1442f662324f5.

```

constructor(address payable vulnerableAddress) {
    target = VulnerableBank(vulnerableAddress);
    owner = msg.sender;
}

// Deposit some ETH to VulnerableBank to set attacker's mapping balance
function depositToTarget() external payable {
    require(msg.value > 0, "send ETH");
    // Forward deposit to the target so the attacker has an internal balance
    target.deposit{value: msg.value}();
}

// Start the attack
function attack(uint256 amount) external {
    require(msg.sender == owner, "not owner");
    emit AttackStarted(amount);
}

```

Figure 5: Attacker contract balance after exploit.

The screenshot shows the REMIX IDE interface with the code editor containing the `Layer2SecurityDemo.sol` file. The code includes logic for withdrawing funds from the `SecureBank`. In the transaction history, a transaction from the Attacker's account (0xAb8...35cb2) to the Attacker's own address (0xB8B...f0Ce8) is shown, with a value of 0 wei and a gas limit of 0x64d...40000. The transaction status is "mined and execution succeeded". The transaction hash is 0x16a14fdfeffbbfbfe68555b5d99eeca78caf26ab8f8f28f5aae7af49030f2d3, and the block number is 15.

```

emit Withdrawn(msg.sender, amount, address(this).balance);

receive() external payable {} undefined gas

```

Figure 6: SecureBank balance after attempted exploit

7. Recommendations for Secure Deployment

- Conduct formal verification of smart contracts before production.
- Enforce multi-layered authentication and identity management for nodes.
- Apply real-time transaction monitoring and anomaly detection using AI-based tools.

- Use secure Layer 2 rollups (Optimistic or ZK-rollups) to enhance throughput without compromising integrity.
- Schedule periodic code audits and penetration testing to identify new vulnerabilities.

8. Conclusion

This experiment demonstrated that while Layer 2 blockchains offer scalability and performance benefits, they also introduce new attack surfaces — especially at the application layer through insecure smart contracts.

By applying structured threat modelling, secure coding patterns, and robust consensus mechanisms, organizations can build resilient and tamper-resistant financial applications on blockchain platforms.