Project Report

Cybersecurity Alert System for Suspicious Login Activities

Team Members

Aayan Rashid – 2023002

Muaaz Bin Salman – 2023338

Instructor

Sir Junaid Ahmed

Course

CS-221L

Abstract:

This project presents a robust, user-driven cybersecurity system designed to dynamically detect and flag suspicious login activities. Leveraging core data structures such as Binary Search Trees (BST), stacks, queues, and graphs, this system efficiently monitors login attempts, identifies suspicious patterns, and provides comprehensive reports. The program dynamically processes user input in real-time and employs hierarchical and graph-based analysis to detect clusters of malicious activities. All detected activities are logged into further examination, ensuring traceability and scalability.

System Overview:

The system is built around modular components that handle dynamic data input, real-time analysis, and reporting. Key data structures like queues ensure first-in-first-out (FIFO) processing of login data, while a BST organizes login attempts by timestamp for efficient querying. Graph structures are employed to model relationships between suspicious IPs, enabling cluster analysis using Depth First Search (DFS). The program also integrates file handling to maintain a persistent log of all activities.

Features:

1. Dynamic Data Input

The system collects login attempt data, including:

- **IP Address**: Unique identifier for the source of login.
- **Timestamp**: The exact time of the login attempt.
- **Status**: Indicates success (s/S) or failure (f/F).

Implementation:

- User input is validated through iterative error-handling loops, ensuring correct data entry.
- Each login attempt is stored in a dynamically updated BST, where nodes are sorted by timestamp.

Code Example:

```
LoginAttempt attempt;
cout << "Enter IP address: ";
cin >> attempt.ip;
cin.ignore();
cout << "Enter timestamp: ";
getline(cin, attempt.timestamp);
bool validInput = false;
while (!validInput) {
   cout << "Enter status (s/S for success, f/F for failure): ";
   cin >> attempt.status;
   if (attempt.status == "s" || attempt.status == "S" || attempt.status == "f" || attempt.status == "F")
{
      validInput = true;
   } else {
      cout << "Invalid status. Please enter again.\n";
      cin.clear();
      cin.ignore(numeric_limits<streamsize>::max(), '\n');
   }     }
root = insert(root, attempt);
loginQueue.push(attempt);
```

2. Suspicious Activity Detection:

Suspicious activities are flagged based on patterns, such as multiple failed login attempts (≥3) from the same IP address.

Algorithmic Process

1. Increment failure count for each failed attempt.
2. If the count reaches three, classify the IP as suspicious.
3. Establish connections between suspicious IPs using a graph data structure.
4. Log all suspicious activities in a file for persistent storage.

Code Example:

```
if (attempt.status == "f" || attempt.status == "F") {
   ipFailCount[attempt.ip]++;
   if (ipFailCount[attempt.ip] >= 3) {
      if (suspiciousIPs.find(attempt.ip) == suspiciousIPs.end()) {
         logFile << "Suspicious activity detected from IP: " << attempt.ip << " at " <<
attempt.timestamp << endl;
         recordActivity("activity_log.txt", "Suspicious activity detected from IP: " + attempt.ip + "
at " + attempt.timestamp);
         suspiciousIPs.insert(attempt.ip);
  for (const auto& ip : suspiciousIPs) {
            if (ip != attempt.ip) {
               graph[ip].insert(attempt.ip);
               graph[attempt.ip].insert(ip);
            }
         }
      }
   }
} else {
   ipFailCount[attempt.ip] = 0;          }
```

3. Report Generation:

The system generates a detailed report of all login attempts and flagged suspicious activities. It employs an in-order traversal of the BST to present login data chronologically.

Implementation

- Login data is traversed using an in-order traversal function.
- Reports include IP addresses, timestamps, and login statuses, formatted for readability.
- All data is appended to a text file for persistent logging.

Code Example:

```
void displayReport(const BSTNode* root) {
   cout << "\nAll Login Attempts:\n";
   auto display = [](const LoginAttempt& attempt) {
      cout << attempt.ip << " " << attempt.timestamp << " " << attempt.status << endl;
   };
   inorderTraversal(const_cast<BSTNode*>(root), display);          }
```

4. Clustering Analysis:

Graph traversal algorithms (DFS) are used to identify clusters of suspicious activity. Each cluster represents interconnected suspicious IPs, highlighting coordinated attack patterns.

Implementation

- Suspicious IPs are stored in a graph adjacency list.
- A DFS function traverses the graph to identify connected components.

Code Example:

```
void DFS(const string& node, map<string, set<string>>& graph, map<string, bool>& visited) {
   visited[node] = true;
   cout << node << " ";
   for (const auto& neighbor : graph[node]) {
      if (!visited[neighbor]) {
         DFS(neighbor, graph, visited);          }          }          }
```

5. File Handling:

All activities, including detected suspicious behaviors, are logged in a file (activity_log.txt) for future reference. The system ensures that the file remains open throughout execution for efficient logging.

Code Example:

```
ofstream logFile("activity_log.txt", ios::app);

if (logFile.is_open()) {

   logFile << "Activity logged." << endl;

   logFile.close();

} else {

   cerr << "Unable to open log file.\n";          }
```

Core Data Structures and Algorithms

Binary Search Tree (BST)

Used for efficient storage and retrieval of login attempts, organized by timestamp.

Queue

Ensures FIFO processing of login data, maintaining chronological order.

Graph

Represents relationships between suspicious IPs, facilitating cluster detection.

DFS

Traverses the graph to identify interconnected clusters of suspicious activity.

Conclusion:

This project successfully implements a cybersecurity system that dynamically processes login data, detects suspicious patterns, and generates detailed reports. By integrating multiple data structures and algorithms, the system achieves high efficiency and scalability, making it a powerful tool for real-time cybersecurity monitoring.