Binary Log Parser and Anomaly Detector

```
#!/usr/bin/env python3
"""
Binary Log Parser and Anomaly Detector

This script parses a custom binary format for login attempt logs and identifies
potentially compromised accounts based on anomalous behavior.

Format:
- username_length: 4-byte integer (big-endian)
- username: variable-length string
- ip: 4-byte IPv4 address
- timestamp: 4-byte Unix timestamp (big-endian)
- success: 1-byte boolean

Usage:
    python log_analyzer.py --input <log_file> [--output <output_file>] [--sql <sql_file>]
"""

import argparse
import struct
import socket
import sqlite3
import json
import os
import sys
from datetime import datetime
from collections import defaultdict

def parse_binary_log(file_path):
    """
    Parse the binary log file according to the specified format.

    Args:
```

```
    file_path: Path to the binary log file

Returns:
    List of login attempt records
"""
logs = []

try:
    with open(file_path, 'rb') as f:
        data = f.read()

    offset = 0
    while offset < len(data):
        # Read username length (4-byte integer, big-endian)
        username_length = struct.unpack('>I', data[offset:offset+4])[0]
        offset += 4

        # Read username (variable length string)
        username = data[offset:offset+username_length].decode('utf-8')
        offset += username_length

        # Read IP address (4-byte IPv4 address)
        ip_bytes = data[offset:offset+4]
        ip_address = socket.inet_ntoa(ip_bytes)
        offset += 4

        # Read timestamp (4-byte Unix timestamp, big-endian)
        timestamp = struct.unpack('>I', data[offset:offset+4])[0]
        login_time = datetime.fromtimestamp(timestamp)
        offset += 4

        # Read success flag (1-byte boolean)
        success = data[offset] == 1
        offset += 1

        # Add the parsed entry to our array
        logs.append({
            'username': username,
            'ip_address': ip_address,
            'timestamp': timestamp,
            'login_time': login_time,
```

```
                'success': success
            })

        print(f"Successfully parsed {len(logs)} login attempts")
        return logs

    except Exception as e:
        print(f"Error parsing log file: {str(e)}")
        sys.exit(1)

def detect_anomalies(logs):
    """
    Analyze logs to identify potentially compromised accounts.

    Args:
        logs: List of parsed login attempt records

    Returns:
        List of users with anomaly scores and suspicious behavior details
    """
    # Group logs by username
    user_logs = defaultdict(list)
    for log in logs:
        user_logs[log['username']].append(log)

    anomalies = []

    # Business hours (assuming 9 AM to 5 PM)
    business_start_hour = 9
    business_end_hour = 17

    # Time threshold for rapid location changes (in seconds)
    location_change_threshold = 3600  # 1 hour

    # Analyze each user's login patterns
    for username, user_log in user_logs.items():
        # Sort logs by timestamp
        user_log.sort(key=lambda x: x['timestamp'])

        # Calculate anomaly indicators
        unique_ips = set(log['ip_address'] for log in user_log)
```

```
        failed_attempts = sum(1 for log in user_log if not log['success'])
        successful_attempts = sum(1 for log in user_log if log['success'])

        # Check for rapid location changes
        rapid_location_changes = 0
        for i in range(1, len(user_log)):
            current_log = user_log[i]
            previous_log = user_log[i-1]

            if current_log['ip_address'] != previous_log['ip_address']:
                time_diff = current_log['timestamp'] - previous_log['timestamp']
                if time_diff < location_change_threshold:
                    rapid_location_changes += 1

        # Calculate after-hours logins
        after_hours_logins = sum(
            1 for log in user_log
            if log['login_time'].hour < business_start_hour or log['login_time'].hour >=
business_end_hour
        )

        # Calculate anomaly score based on these factors
        # Weights can be adjusted based on the relative importance of each factor
        anomaly_score = (
            (len(unique_ips) * 10) +
            (failed_attempts * 5) +
            (rapid_location_changes * 20) +
            (after_hours_logins * 3)
        )

        anomalies.append({
            'username': username,
            'anomaly_score': anomaly_score,
            'unique_ips': len(unique_ips),
            'ip_addresses': list(unique_ips),
            'failed_attempts': failed_attempts,
            'successful_attempts': successful_attempts,
            'rapid_location_changes': rapid_location_changes,
            'after_hours_logins': after_hours_logins,
            'total_attempts': len(user_log)
        })
```

```
    # Sort by anomaly score (descending)
    anomalies.sort(key=lambda x: x['anomaly_score'], reverse=True)

    return anomalies

def create_database(logs, db_path=':memory:'):
    """
    Create a SQLite database with the login data

    Args:
        logs: List of parsed login attempt records
        db_path: Path to save the SQLite database (default: in-memory)

    Returns:
        SQLite connection
    """
    conn = sqlite3.connect(db_path)
    cursor = conn.cursor()

    # Create table
    cursor.execute('''
    CREATE TABLE login_attempts (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        username TEXT NOT NULL,
        ip_address TEXT NOT NULL,
        timestamp INTEGER NOT NULL,
        login_time TEXT NOT NULL,
        success INTEGER NOT NULL
    )
    ''')

    # Create indexes
    cursor.execute('CREATE INDEX idx_username ON login_attempts(username)')
    cursor.execute('CREATE INDEX idx_ip_address ON login_attempts(ip_address)')
    cursor.execute('CREATE INDEX idx_timestamp ON login_attempts(timestamp)')
    cursor.execute('CREATE INDEX idx_success ON login_attempts(success)')

    # Insert data
    for log in logs:
        cursor.execute(
```

```
            'INSERT INTO login_attempts (username, ip_address, timestamp, login_time, success)
VALUES (?, ?, ?, ?, ?)',
            (
                log['username'],
                log['ip_address'],
                log['timestamp'],
                log['login_time'].isoformat(),
                1 if log['success'] else 0
            )
        )

    conn.commit()
    return conn

def generate_sql_script():
    """
    Generate a SQL script for creating the table and analyzing login data

    Returns:
        SQL script as a string
    """
    return '''-- Create a table to store login attempts
CREATE TABLE login_attempts (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT NOT NULL,
    ip_address TEXT NOT NULL,
    timestamp INTEGER NOT NULL,
    login_time TEXT NOT NULL,
    success INTEGER NOT NULL
);

-- Create indexes for efficient searching
CREATE INDEX idx_username ON login_attempts(username);
CREATE INDEX idx_ip_address ON login_attempts(ip_address);
CREATE INDEX idx_timestamp ON login_attempts(timestamp);
CREATE INDEX idx_success ON login_attempts(success);

-- Query to find users with multiple IP addresses
SELECT
    username,
    COUNT(DISTINCT ip_address) AS unique_ip_count
```

```
FROM
    login_attempts
GROUP BY
    username
HAVING
    unique_ip_count > 1
ORDER BY
    unique_ip_count DESC;

-- Query to find failed login attempts followed by successful ones
SELECT
    a.username,
    COUNT(*) AS suspicious_patterns
FROM
    login_attempts a
JOIN
    login_attempts b
ON
    a.username = b.username
    AND a.timestamp < b.timestamp
    AND a.success = 0
    AND b.success = 1
    AND (b.timestamp - a.timestamp) < 300 -- Within 5 minutes
GROUP BY
    a.username
ORDER BY
    suspicious_patterns DESC;

-- Query to find rapid login attempts from different locations
SELECT
    a.username,
    a.ip_address AS ip1,
    b.ip_address AS ip2,
    datetime(a.login_time) AS time1,
    datetime(b.login_time) AS time2,
    (julianday(b.login_time) - julianday(a.login_time)) * 24 * 60 AS minutes_between
FROM
    login_attempts a
JOIN
    login_attempts b
ON
```

```
    a.username = b.username
    AND a.ip_address != b.ip_address
    AND a.id < b.id
    AND (julianday(b.login_time) - julianday(a.login_time)) * 24 * 60 < 60 -- Less than 60
minutes apart
ORDER BY
    minutes_between ASC;

-- Query to find users with after-hours login activity
SELECT
    username,
    COUNT(*) AS after_hours_logins
FROM
    login_attempts
WHERE
    (strftime('%H', login_time) < '09' OR strftime('%H', login_time) >= '17')
GROUP BY
    username
ORDER BY
    after_hours_logins DESC;

-- Comprehensive anomaly detection query
WITH
    unique_ips AS (
        SELECT
            username,
            COUNT(DISTINCT ip_address) AS ip_count
        FROM
            login_attempts
        GROUP BY
            username
    ),
    failed_logins AS (
        SELECT
            username,
            SUM(CASE WHEN success = 0 THEN 1 ELSE 0 END) AS failed_count
        FROM
            login_attempts
        GROUP BY
            username
    ),
```

```
    after_hours AS (
        SELECT
            username,
            COUNT(*) AS after_hours_count
        FROM
            login_attempts
        WHERE
            (strftime('%H', login_time) < '09' OR strftime('%H', login_time) >= '17')
        GROUP BY
            username
    ),
    rapid_location_changes AS (
        SELECT
            a.username,
            COUNT(*) AS rapid_changes
        FROM
            login_attempts a
        JOIN
            login_attempts b
        ON
            a.username = b.username
            AND a.ip_address != b.ip_address
            AND a.id < b.id
            AND (b.timestamp - a.timestamp) < 3600 -- Less than 1 hour apart
        GROUP BY
            a.username
    )
SELECT
    u.username,
    COALESCE(u.ip_count, 0) AS unique_ip_count,
    COALESCE(f.failed_count, 0) AS failed_logins,
    COALESCE(a.after_hours_count, 0) AS after_hours_logins,
    COALESCE(r.rapid_changes, 0) AS rapid_location_changes,
    (COALESCE(u.ip_count, 0) * 10) +
    (COALESCE(f.failed_count, 0) * 5) +
    (COALESCE(r.rapid_changes, 0) * 20) +
    (COALESCE(a.after_hours_count, 0) * 3) AS anomaly_score
FROM
    unique_ips u
LEFT JOIN
    failed_logins f ON u.username = f.username
```

```
LEFT JOIN
    after_hours a ON u.username = a.username
LEFT JOIN
    rapid_location_changes r ON u.username = r.username
ORDER BY
    anomaly_score DESC
LIMIT 10;
'''

def analyze_compromised_user(conn, username):
    """
    Perform detailed analysis on a potentially compromised user

    Args:
        conn: SQLite connection
        username: Username to analyze

    Returns:
        Dictionary with detailed analysis
    """
    cursor = conn.cursor()

    # Get all login attempts for this user
    cursor.execute(
        '''
        SELECT
            timestamp,
            login_time,
            ip_address,
            success
        FROM
            login_attempts
        WHERE
            username = ?
        ORDER BY
            timestamp ASC
        ''',
        (username,)
    )

    logins = cursor.fetchall()
```

```
# Analyze suspicious patterns
suspicious_events = []
previous_ip = None
previous_time = None

for timestamp, login_time, ip_address, success in logins:
    if previous_ip and previous_ip != ip_address:
        time_diff = timestamp - previous_time
        if time_diff < 3600:  # Less than 1 hour
            suspicious_events.append({
                'event_type': 'rapid_location_change',
                'previous_ip': previous_ip,
                'new_ip': ip_address,
                'minutes_between': time_diff / 60
            })

    previous_ip = ip_address
    previous_time = timestamp

# Get login success rate
cursor.execute(
    '''
    SELECT
        COUNT(*) AS total,
        SUM(CASE WHEN success = 1 THEN 1 ELSE 0 END) AS successful
    FROM
        login_attempts
    WHERE
        username = ?
    ''',
    (username,)
)

total, successful = cursor.fetchone()
success_rate = (successful / total) * 100 if total > 0 else 0

return {
    'username': username,
    'login_count': total,
    'success_rate': success_rate,
```

```
        'suspicious_events': suspicious_events,
        'login_history': [
            {
                'timestamp': timestamp,
                'login_time': login_time,
                'ip_address': ip_address,
                'success': bool(success)
            }
            for timestamp, login_time, ip_address, success in logins
        ]
    }

def main():
    """
    Main function to process arguments and run the analysis
    """
    parser = argparse.ArgumentParser(description='Analyze binary login logs for compromised
accounts')
    parser.add_argument('--input', '-i', required=True, help='Path to binary log file')
    parser.add_argument('--output', '-o', help='Path to save analysis results (JSON)')
    parser.add_argument('--sql', '-s', help='Path to save SQL script')
    parser.add_argument('--db', '-d', help='Path to save SQLite database')
    parser.add_argument('--verbose', '-v', action='store_true', help='Enable verbose output')

    args = parser.parse_args()

    # Parse the binary log file
    print(f"Parsing binary log file: {args.input}")
    logs = parse_binary_log(args.input)

    # Analyze for anomalies
    print("Analyzing for suspicious behavior...")
    anomalies = detect_anomalies(logs)

    # Print top suspicious users
    print("\nTop potentially compromised accounts:")
    for i, anomaly in enumerate(anomalies[:5]):
        print(f"{i+1}. Username: {anomaly['username']}")
        print(f"   Anomaly Score: {anomaly['anomaly_score']}")
        print(f"   Unique IPs: {anomaly['unique_ips']}")
        print(f"   Failed/Successful Logins:
```

```
{anomaly['failed_attempts']}/{anomaly['successful_attempts']}")
        print(f"   Rapid Location Changes: {anomaly['rapid_location_changes']}")
        print(f"   After-Hours Logins: {anomaly['after_hours_logins']}")
        print()

    # Identify the most likely compromised user
    if anomalies:
        compromised_user = anomalies[0]['username']
        print(f"RESULT: The most likely compromised account is: {compromised_user}")

        # Create database for SQL analysis
        db_path = args.db if args.db else ':memory:'
        conn = create_database(logs, db_path)

        # Get detailed analysis for the compromised user
        detailed_analysis = analyze_compromised_user(conn, compromised_user)

        if args.verbose:
            print("\nDetailed analysis for the compromised account:")
            print(f"Login history for {compromised_user}:")
            for entry in detailed_analysis['login_history']:
                status = "SUCCESS" if entry['success'] else "FAILED"
                print(f"{entry['login_time']} | {entry['ip_address']} | {status}")

            if detailed_analysis['suspicious_events']:
                print("\nSuspicious events:")
                for event in detailed_analysis['suspicious_events']:
                    print(f"IP changed from {event['previous_ip']} to {event['new_ip']} "
                          f"in {event['minutes_between']:.1f} minutes")
    else:
        print("No anomalies detected in the log data")

    # Save results to output file
    if args.output:
        with open(args.output, 'w') as f:
            json.dump({
                'summary': {
                    'total_logs': len(logs),
                    'total_users': len({log['username'] for log in logs}),
                    'compromised_user': compromised_user if anomalies else None
                },
```

```
                'anomalies': anomalies,
                'detailed_analysis': detailed_analysis if anomalies else None
            }, f, indent=4, default=str)
        print(f"Analysis results saved to {args.output}")

    # Save SQL script
    if args.sql:
        with open(args.sql, 'w') as f:
            f.write(generate_sql_script())
        print(f"SQL script saved to {args.sql}")

    # Report if database was saved
    if args.db:
        print(f"SQLite database saved to {args.db}")

if __name__ == "__main__":
    main()
```

Revision #1

Created 2025-11-25 18:08:05 UTC by David Rizzo

Updated 2025-11-25 18:08:17 UTC by David Rizzo