# Compromised User Detector

```
import struct
import socket
import datetime
from collections import defaultdict, Counter

def parse_binary_logs(file_path):
    """Parse binary log file according to the specified format."""
    login_attempts = []

    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 IPv4 address (4 bytes)
        ip_bytes = data[offset:offset+4]
        ip_address = socket.inet_ntoa(ip_bytes)
        offset += 4

        # Read timestamp (4-byte Unix timestamp)
        timestamp = struct.unpack('>I', data[offset:offset+4])[0]
        datetime_obj = datetime.datetime.fromtimestamp(timestamp, tz=datetime.timezone.utc)
        offset += 4

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

```
        # Store the parsed login attempt
        login_attempts.append({
            'username': username,
            'ip_address': ip_address,
            'timestamp': timestamp,
            'datetime': datetime_obj,
            'success': success
        })

    return login_attempts

def analyze_logs(login_attempts):
    """Basic analysis of the log data to extract key metrics."""
    # Get the earliest timestamp (start date of the log)
    earliest_timestamp = min(login_attempts, key=lambda x: x['timestamp'])['timestamp']
    start_date_utc = datetime.datetime.fromtimestamp(earliest_timestamp,
tz=datetime.timezone.utc)

    # Count unique usernames
    unique_usernames = set(attempt['username'] for attempt in login_attempts)

    # Count unique IP addresses
    unique_ips = set(attempt['ip_address'] for attempt in login_attempts)

    # Count total login attempts
    total_attempts = len(login_attempts)

    return {
        'start_date_utc': start_date_utc,
        'total_attempts': total_attempts,
        'unique_usernames': len(unique_usernames),
        'unique_ips': len(unique_ips),
        'usernames': unique_usernames,
        'ip_addresses': unique_ips
    }

def analyze_login_patterns(login_attempts):
    """Analyze login patterns to identify potentially compromised users."""
    # Track login data per user
    user_data = defaultdict(lambda: {
        'ips': set(),
```

```
        'successful_logins': 0,
        'failed_logins': 0,
        'login_times': [],
        'success_after_failure': 0,
        'unique_ips_count': 0,
        'unusual_hour_logins': 0,
        'successful_after_multiple_failures': 0,
        'geographic_anomalies': 0
    })

    # Group login attempts by user
    user_attempts = defaultdict(list)
    for attempt in login_attempts:
        user = attempt['username']
        user_attempts[user].append(attempt)

        # Track IPs used
        user_data[user]['ips'].add(attempt['ip_address'])

        # Track login success/failure
        if attempt['success']:
            user_data[user]['successful_logins'] += 1
        else:
            user_data[user]['failed_logins'] += 1

        # Track login times
        user_data[user]['login_times'].append(attempt['timestamp'])

        # Track unusual hour logins (between 1am and 5am)
        hour = attempt['datetime'].hour
        if 1 <= hour <= 5:
            user_data[user]['unusual_hour_logins'] += 1

    # Calculate additional metrics
    for user, attempts in user_attempts.items():
        # Sort attempts by timestamp
        sorted_attempts = sorted(attempts, key=lambda x: x['timestamp'])

        # Check for successful login after failures
        consecutive_failures = 0
        for i in range(1, len(sorted_attempts)):
```

```
            current = sorted_attempts[i]
            previous = sorted_attempts[i-1]

            # Check for IP hopping (successful login from different IP)
            if (not previous['success'] and
                current['success'] and
                current['ip_address'] != previous['ip_address']):
                user_data[user]['success_after_failure'] += 1

            # Count consecutive failures
            if not previous['success']:
                consecutive_failures += 1
            else:
                consecutive_failures = 0

            # Successful login after multiple consecutive failures
            if current['success'] and consecutive_failures >= 3:
                user_data[user]['successful_after_multiple_failures'] += 1
                consecutive_failures = 0

        # Count unique IPs
        user_data[user]['unique_ips_count'] = len(user_data[user]['ips'])

        # Calculate rough geographic anomalies based on IP
        # This is simplified - in reality, you'd use IP geolocation
        if user_data[user]['unique_ips_count'] >= 3:
            ips_first_octet = [int(ip.split('.')[0]) for ip in user_data[user]['ips']]
            unique_first_octets = len(set(ips_first_octet))
            if unique_first_octets >= 2:  # Different network classes suggest geographic
spread
                user_data[user]['geographic_anomalies'] += unique_first_octets

    # Calculate comprehensive suspicion scores
    suspicious_users = []
    for user, data in user_data.items():
        # Initialize base suspicion score
        suspicion_score = 0

        # Factor 1: Multiple IPs used
        ip_anomaly_score = (data['unique_ips_count'] - 1) * 5  # Expect one normal IP
        suspicion_score += max(0, ip_anomaly_score)
```

```
        # Factor 2: Successful logins after failures from different IPs
        suspicion_score += data['success_after_failure'] * 15

        # Factor 3: Failed login ratio
        if data['successful_logins'] > 0:
            failure_ratio = data['failed_logins'] / data['successful_logins']
            suspicion_score += min(failure_ratio * 3, 20)  # Cap at 20 points
        elif data['failed_logins'] > 5:  # Only failures, no successes
            suspicion_score += 20

        # Factor 4: Unusual hour logins
        suspicion_score += data['unusual_hour_logins'] * 2

        # Factor 5: Successful after multiple failures
        suspicion_score += data['successful_after_multiple_failures'] * 25

        # Factor 6: Geographic anomalies
        suspicion_score += data['geographic_anomalies'] * 8

        # Add context for this score
        reasons = []
        if data['unique_ips_count'] > 1:
            reasons.append(f"Used {data['unique_ips_count']} different IPs")
        if data['success_after_failure'] > 0:
            reasons.append(f"Successful login after failure from different IP:
{data['success_after_failure']} times")
        if data['failed_logins'] > 3:
            reasons.append(f"High number of failed logins: {data['failed_logins']}")
        if data['unusual_hour_logins'] > 0:
            reasons.append(f"Logins during unusual hours: {data['unusual_hour_logins']}")
        if data['successful_after_multiple_failures'] > 0:
            reasons.append(f"Successful login after multiple failures:
{data['successful_after_multiple_failures']}")
        if data['geographic_anomalies'] > 0:
            reasons.append(f"Potential geographic anomalies detected")

        suspicious_users.append((user, suspicion_score, data, reasons))

    # Sort by suspicion score
    suspicious_users.sort(key=lambda x: x[1], reverse=True)
```

```
    # Identify the most likely compromised user
    most_likely_compromised = suspicious_users[0][0] if suspicious_users else None

    return suspicious_users, user_data, most_likely_compromised

def main():
    file_path = input("Enter the path to the binary log file: ")

    try:
        login_attempts = parse_binary_logs(file_path)
        print(f"Successfully parsed {len(login_attempts)} login attempts.")

        # Basic log analysis for the requested metrics
        log_metrics = analyze_logs(login_attempts)

        print("\n===== LOG METRICS =====")
        print(f"Start date of the log (UTC): {log_metrics['start_date_utc'].strftime('%Y-%m-%d
%H:%M:%S UTC')}")
        print(f"Total login attempts recorded: {log_metrics['total_attempts']}")
        print(f"Number of unique usernames: {log_metrics['unique_usernames']}")
        print(f"Number of unique IP addresses: {log_metrics['unique_ips']}")

        # Advanced pattern analysis for suspicious activity
        suspicious_users, user_data, most_likely_compromised =
analyze_login_patterns(login_attempts)

        print("\n===== COMPROMISED USER IDENTIFICATION =====")
        if most_likely_compromised:
            print(f"\n□□MOST LIKELY COMPROMISED USER: {most_likely_compromised} □□")

            # Find this user in the suspicious_users list
            for user, score, data, reasons in suspicious_users:
                if user == most_likely_compromised:
                    print(f"Suspicion Score: {score:.2f}")
                    print("\nReasons for suspicion:")
                    for i, reason in enumerate(reasons, 1):
                        print(f"  {i}. {reason}")
                    print("\nDetailed metrics:")
                    print(f"  Unique IPs: {data['unique_ips_count']}")
                    print(f"  Successful logins: {data['successful_logins']}")
```

```
                    print(f"  Failed logins: {data['failed_logins']}")
                    print(f"  Logins during unusual hours: {data['unusual_hour_logins']}")
                    print(f"  IP addresses used: {', '.join(data['ips'])}")
                    break
        else:
            print("No compromised user identified.")

        print("\n===== OTHER SUSPICIOUS USERS =====")
        if len(suspicious_users) > 1:
            for user, score, data, reasons in suspicious_users[1:6]:  # Show top 5 after the
most suspicious
                if score > 10:  # Only show users with meaningful suspicion scores
                    print(f"\nUsername: {user} (Suspicion Score: {score:.2f})")
                    print("Reasons for suspicion:")
                    for i, reason in enumerate(reasons, 1):
                        print(f"  {i}. {reason}")
        else:
            print("No other suspicious users identified.")

    except Exception as e:
        print(f"Error processing the file: {e}")
        import traceback
        traceback.print_exc()

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

---

Revision #1

Created 2025-11-25 18:03:24 UTC by David Rizzo

Updated 2025-11-25 18:03:45 UTC by David Rizzo