National Cyber League

This volume documents my practical performance in the National Cyber League (NCL) Competition, a premier, scenario-based cybersecurity event that mirrors challenges faced in the professional workforce. It serves as a verifiable extension of my NCL Scouting Report, detailing the methodologies, tools, and successful solutions for complex, time-sensitive challenges.

- Cryptography
- Individual Game Technical Writeup
- Scripts
  - Binary Parser
  - Compromised User Detector
  - GPG Verify
  - Hash Identifier
  - HMAC
  - HMAC Integrity Checker
  - Liber8tion Cracker
  - PDF to Hashcat
  - PDF to John
  - Steg
  - Binary Log Parser and Anomaly Detector

Cryptography

Tools Used

- RapidTables
  - https://www.rapidtables.com/web/tools/index.html
    - Base64
    - Binary
    - Hex String
- Cryptii
  - https://cryptii.com/
  - Decode
- Decode
  - https://www.dcode.fr/
  - RSA Cipher
- Kali Linux
  - Strings

Challange 1

Using rapidtables I was able to decode the encoded strings. The first string was a hexadecimal string. I ommited the 0x and pasted the remaining string into the deocoder. `0x73636f7270696f6e` `scorpion` The decoded string is **scorpion**.

I then proceeded to the next string. The string is a base64 string. I then pased that into the decoder. `c2NyaWJibGU` `scribble` The decoded string is **scribble**.

The next string was a binary string. `01110011 01100101 01100011 01110101 01110010 01100101 01101100 01111001` `securely` The decoded string is **securely**.

The last string was a double encoded string. Original word was first encoded into a base64 and then that base64 was encoded into a binary string. I first decoded the binary to reveal the base64

and then decoded the base64 to reveal the original string. `01100010 01000111 00111001 01110011 01100010 01000111 01101100 01110111 01100010 00110011 01000001 00111101` `bG9sbGlwb3A=` `lollipop` The decoded string is **lollipop**.

Challange 2

Using crpytii I took the encrpyred string and decoded it using a ceaser cipher with a shift of 13. I used the Large Language Model Gemma to aid in my decription. The model was able to tell me the most logical cipher based on the encoding. `iveghny ynxr` `virtual lake` The decoded string is **virtual lake**.

Challange 3

Using cryptii and gima I was able to determine that the cipher is a alphabetical substituion using the alphabet in reverse. `hzuvob lyerlfh xzev` `safely obvious cave` The decoded string is **safely obvious cave**.

Challange 4

Based on the ... and the -- I was able to determine this was morse code. I used crpytii to convert the string. `.... . / ... . -.-. .-. . - / --- ..-. / --. . - - .. -. --. / .- .... . .- -.. / .. ... / --. . - - .. -. --. / ... - .- .-. - . -.. / ... -.- -.-- / -.. -.- ...- -... / ----. ---.. .---- -....` `the secret of getting ahead is getting started sky dkvb 9816` The decoded string is **the secret of getting ahead is getter strated sky dkvb 9816**.

Challange 5

The notes for this challange is that there were handwritten notes with 3 & 5. Using this information I looked for ciphers on cryptii where the numbers could be insterted and keys. I was able to figure out this was a Rail Cipher. 3 correlated to the first message and 5 the second. `Cair eruSA-0org sgaeudrpesr K-II98.ue cn seYQ3` `Courage is grace under pressure SKY-AIQI-9380.` The deconded first string is **Courage is grace under pressure SKY-AIQI-9380.** `F daS-eefn  n KZ3eheadty.YI8lta`

`oiwy-Q0. r aI2` `Feel the fear and do it anyway. SKY-IQIZ-3802.` The decoded second string is **Feel the fear and do it anyway. SKY-IQIZ-3802.**

Challange 6

The notes for this challange was that the keyword is **qizkwcgqbs**. The let me to a vigenere cipher because it uses a key. Using cryptii and the keyword I was able to decipher the phrase. `Y ln xkv lubj swlzqvkht, A vmzb pjk bbua we ddgs ILQ-GQYU-8026` `I do not fear computers, I fear the lack of them SKY-QIZK-8026` The decrypted string is **I do not fear computers, I fear the lack of them SKY-QIZK-8026**.

Challange 7

For this challange I used Kali Linux. The provided file had a string encoded into it. I first tried looking through the metadata and through steghide. The strign was hidden in the strings metadata of the file. Using the following command `strings Step1.jpg | grep SKY` I was able to pull tjust the string I needed out of the file. `SKY-TVJI-2063` The decoded string is **SKY-TVJI-2063**.

Challage 8

This challange is decoding a string using RSA. I used dcode.fr and their RSA decrypter to decipher the text. The provided values were N, C, and E. I was then able to use that to pull P adn Q from dcode. Then With that I was able to decode the string. C is the encoded string. `n = 1079` `e = 43` `c = 996 894 379 631 894 82 379 852 631 677 677 194 893` `p = 13` `q = 83` `SKY-KRYG-5530` The decoded string is **SKY-KRYG-5530**.

Individual Game Technical Writeup

Table of Contents

- Introduction
- Open Source Intelligence (OSINT)
- Cryptography
- Password Cracking
- Log Analysis
- Network Traffic Analysis
- Forensics
- Scanning & Reconnaissance
- HMAC Integrity Verification
- Conclusion

Introduction

This document provides a technical analysis of my participation in the NCL Individual Game challenges, breaking down the methodology, solutions, and techniques used across multiple categories.

Open Source Intelligence (OSINT)

Challenge 1: Code of Conduct

Verified competition terms and conditions:

- No peer collaboration allowed

- Responsible AI usage required

Challenge 2: Honor (Easy)

Analyzed an image from a data breach using ExifTool:

```
exiftool dog.jpg
```

Key Findings:

- Discovered hex-encoded flag in copyright metadata: `534B592D4C494D492D31333337`
- Decoded to ASCII: `SKY-LIMI-1337`

Challenge 3: Controversial Challenge (Medium)

Identified messaging platform and associated individual:

- Visual analysis revealed Signal platform
- Researched "signalgate" context
- Identified "S M" as Stephen Miller

Challenge 4: Nostalgia (Hard)

Analyzed historical photo location:

- Identified Dutch text "Electr Klompenmakeru" (Electric Clog Maker)
- Located business as Ratterman Wooden Shoes
- Address: Noorddammerlaan 22, 1185, ZA, Amstelveen

Challenge 6: Github in Action (Hard)

Investigated GitHub repositories:

- Found user spmedia's website: http://edmond.ma/jor
- Identified anti-phishing repo: PhishingSecLists
- Located pull request by Ashley Tolbert

Cryptography

Challenge 1: The Bases (Easy)

Decoding various encoded strings:

1. Binary to ASCII:
   - Original Message: `01001011 01100001 01101110 01100111 01100001 01010010 00110000 00110000 00110101 00111000`
   - Decoded Result: `KangaR0058`
2. Octal to ASCII:
   - Original Message: `113 145 160 162 151 127 141 164 151 070 063`
   - Decoded Result: `KepriWati83`
3. Octal to ASCII:
   - Original Message: `052 052 051 053 115 117 110 100 105 097 108`
   - Decoded Result: `4435sundial`

Challenge 2: Super Shift (Easy)

Various shift ciphers to decode:

1. Caesar Cipher (shift 13):
   - Original Message: `84cbagvnpfhaoveq`
   - Decoded Result: `84pontiacsunbird`
2. Caesar Cipher (shift 5):
   - Original Message: `Mfnwiwjxxnsl3`
   - Decoded Result: `Hairdressing3`
3. Morse Code + Caesar Cipher (shift 17):
   - Original Message: `-.. --.. - -.-- ..-. . . ...- --.. .--- ... .-. - -...`
   - Intermediate Result: `Dztyfeevzjsrtb`
   - Final Decoded Result: `Michonneisback`

Challenge 3: Pizza Time (Easy)

Rail fence cipher decoding:

1. Rail Fence (key 3):
   - Original Message: `Usezfiysnestpzasrmtltsilhiioaot`
   - Decoded Result: `Unlesshtepizzaisfromitalytossit`
2. Rail Fence (key 6, offset 1):
   - Original Message: `sdIslrstneoathdnowhgakkoeirtl`
   - Decoded Result: `Indarknesslooktowardsthelight`

Challenge 4: Signed (Medium)

Verified GPG signatures with a script that:

- Loops through signature files
- Verifies each against corresponding document
- Reports verification failures

```
# Key functionality of gpg_verify.sh
for sig_file in *.sig; do
  original_file="${sig_file%.sig}"
  # Verify signature and report results
  gpg --verify "$sig_file" "$original_file"
  # Report success or failure
done
```

Findings:

1. Tampered file: Email_48.txt containing HEX `4b767470753861707675a`
2. Decoded to: `Domin8tion`

Password Cracking

Challenge 1: Hash me outside! (Easy)

Generated hashes for provided passwords:

1. MD5 hash: `2b164ea92dbd46f72318e55ec634a83a` `echo -n "white1561lotus" | md5sum`

2. SHA1 hash: `516f652fa03b7f711ada6a1acdd9786cde89dc8a` `echo -n "6891JasmineDragon" | sha1sum`
3. SHA256 hash: `0b09f41de4769201222f1e4a42acdc3a63750700f5d3fd2b37eb643282ba303c` `echo -n "317paisho698" | sha256sum`

Challenge 2: We Will RockYou (Easy)

Cracked MD5 hashes with the RockYou wordlist:

- `Hashcat -m 0 wewillrockyou.txt /usr/share/wordlists/rockyou.txt`
- `3da1dd44e86ce30ff07d32065e9b68c3`: queen24
- `5243dc768dfca6d80993b4803bed95e4`: freddie11
- `a6c8f8fe09042f4ab28d0048575cd9d4`: mercury5134

Challenge 3: Oph the Grid (Medium)

Used Ophcrack with rainbow tables to crack Windows LM/NTLM hashes:

Process:

1. Downloaded rainbow tables
2. Created hash file
3. Imported hashes and tables into Ophcrack
4. Obtained results:
   - `BA50E19589950A959C5014AE4718A7EE:74A942B14C50D4ED03D9A4CC8866199C`: 25332535
   - `2A2A4346DDF2630ABFE061A5F4DBCCB2:8295ABD305A649454D0709E350A8C601`: shanegrace
   - `876CA1B27D8B258D6966CA05A9CDAE2D:F8F9BF0623560DA52451C1893821087A`: beautyelaine

Challenge 4: Totally Safe PDF (Medium)

1. Used a pdf2john script that:

- Extracts encryption parameters from PDFs
- Formats hash for John the Ripper
- Tracks encryption algorithm, permissions, and revision

```
# Key functionality of pdf2john.py
# Extract PDF security parameters
# Format hash string for John the Ripper
# Example: "$pdf$4*4*128*-3904*1*16*hash-data-here*32*more-hash-data*32*final-hash-part"
```

2. Ran John the Ripper against the hash:
    - Used rockyou.txt wordlist for password cracking

Results:

- Password: pdfscott86
- Flag: SKY-PDFS-2472

Challenge 5: put0nth3ma5k (medium)

Used John the Ripper with mask attack:

- Pattern: "SKY-MASK-?a?a?a?a"
- `john -mask="SKY-MASK-?a?a?a?a" put0nth3ma5k.txt` **Results:**
- `$1$MASK$.IxEV.UcEJNjNX.UCE7/A/`: SKY-MASK-2552
- `$1$MASK$u2nKEYGuYo1DYtu2K/yAn/`: SKY-MASK-4778
- `$1$MASK$56Jw4nrfMvazyiOu68Lge.`: SKY-MASK-9310

Log Analysis

Challenge 1: Ancient History (Easy)

Analyzed HTTP logs:

Queries Used:

1. Extracted domain of the third standard HTTP request:
    - Result: http://httpforever.com/js/init.min.js
2. Identified timestamp for server response:
    - Result: 1743959680.158
3. Found IP of www.delta.com in first CONNECT request:

- Result: 96.16.70.40

4. Counted NONE_NONE/000 errors:
   - Result: 40
5. Counted successful connections to push.services.mozilla.com:
   - Result: 134
6. Counted total POST requests:
   - Result: 8
7. Found third most accessed domain:
   - Result: firefox.settings.services.mozilla.com

Challenge 2: Leaked (Medium)

Analyzed leaked SQL data from a social media database:

SQL Queries:

1. Count of compromised users:
   - Simple count query revealing 982 affected users
2. First account join date:
   - Identified earliest timestamp
   - Converted to: March 22, 2025 12:01:38 AM UTC
3. Email with most followers:
   - Sorted by follower count in descending order
   - Result: kvolantge@cityinthe.cloud (1000 followers)
4. Count of verified users:
   - Filtered for verified status
   - Result: 464 verified users
5. Most common phone area code state:
   - Extracted area codes from phone numbers
   - Counted occurrences and identified most common
   - Result: 364 (Kentucky)

Challenge 3: Logins (Hard)

Created login analysis scripts that:

- Parse binary log format with proper byte offsets
- Calculate suspicion scores based on multiple factors
- Identify potentially compromised accounts

Methodology:

- Parsed binary login logs with custom script
- Tracked suspicious login patterns including:
  - Multiple IP addresses per user
  - Failed login attempts followed by successful logins
  - Unusual login hours (1am-5am)
  - Successful logins after multiple failures
  - Geographic anomalies based on IP patterns
- Implemented weighted scoring system for suspicious behaviors
- Calculated comprehensive suspicion scores for each user
- Identified most likely compromised account based on highest score

```
# Key functionality of binary_parser.py
def parse_binary_logs(file_path):
    # Read file and parse using correct binary structure:
    # - Username length (4 bytes)
    # - Username (variable length)
    # - IPv4 address (4 bytes)
    # - Timestamp (4 bytes)
    # - Success flag (1 byte)

# Enhanced in compromised_user_detector.py with:
# - Login time analysis (unusual hours)
# - Repeated failure pattern detection
# - Geographic anomaly detection
# - Comprehensive scoring algorithm
```

Results:

1. Log start date (UTC): March 18, 2024
2. Login attempt events recorded: 3579
3. Unique usernames: 174

4. Unique IP addresses: 208
5. Compromised user: [Username identified by algorithm]

Network Traffic Analysis

Challenge 1: Lost in Resolution (Easy)

Analyzed DNS traffic in a PCAP file:

Wireshark Filters:

1. DNS transaction ID in frame 36:

   ```
   frame.number == 36
   ```

   Result: 0x75b8

2. Email provider:

   ```
   udp.port == 53 or tcp.port == 53
   ```

   Result: Proton (from packet 2505)

3. Second A record for chatgpt.com in frame 10061:

   ```
   frame.number == 10061
   ```

   Result: 172.64.155.209

4. Transaction ID for first pwn.college query:

   ```
   dns.qry.name == "pwn.college"
   ```

   Result: 0xaeee

5. Flag in DNS records for flag.com.localdomain:

   ```
   dns.qry.name contains "flag.com.localdomain"
   ```

   Result: SKY-DENS-5353

Challenge 2: Wifi (Medium)

Analyzed WiFi capture files:

- Identified router MAC: c0:4a:00:80:76:e4
- ESSID: Wii Fii
- Victim MAC: 02:38:aa:ae:9f:e6
- Channel: 4
- Cracked password with aircrack-ng: soccer17
- `aircrack-ng -w /usr/share/wordlists/rockyou.txt wifi.cap`

Forensics

Challenge 1: Overuse (Easy)

Analyzed image with hidden data:

```
strings ForYou.jpg  # Revealed embedded filenames
binwalk -e ForYou.jpg  # Extracted hidden files
```

Used a steganography script that:

- Extracts LSB data from images
- Analyzes color planes for anomalies
- Detects hidden files by signatures

```
# Key functionality of steg.py
# Extract LSB data
# Check image metadata
# Analyze bit distribution
# Look for file signatures
```

Process:

1. Extracted strings:

   ```
   strings ForYou.jpg
   ```

   Discovered file names:

   - 1Scroll.jpg
   - 2NeverGoingToGIve.txt

- 3Sky.jpg
- 4Congrats.txt
- 5Wise.jpg
- 6Bussin.txt
- 7Buzz.jpg
- 8More.txt

2. Extracted hidden files:

```
binwalk -e ForYou.jpg
```

3. Found flags:
   - From 6Bussin.txt: SKY-BUSN-4419
   - Base64 encoded flag: SKY-UATE-1057

Results:

- Found flags in embedded files:
  - SKY-BUSN-4419
  - SKY-UATE-1057

Challenge 2: Oops (Medium)

Recovered deleted file from a disk image:

Approach:

1. Opened image in Autopsy forensic tool
2. Navigated to deleted files section
3. Recovered file with flag: SKY-UNDL-3373

Challenge 3

Analyzed a file to determine its original format:

Findings:

1. Identified as 3D printing instructions (STL file)

Scanning & Reconnaissance

Challenge 1: Portscan (Easy)

Network reconnaissance:

```
ifconfig
nmap 10.8.93.0/24
nmap -sV 10.8.93.100
curl 10.8.93.100/flag.txt  # Found SKY-HTTP-4553
```

Findings:

1. Lowest port: 17
2. Highest port: 4000
3. Service on port 80: version 0.6
4. Flag found with curl command:

   ```
   curl 10.8.93.100/flag.txt
   ```

   Result: SKY-HTTP-4553

Challenge 2: Dig (Medium)

DNS record investigation:

- Used dig to query various record types
- Found IPv4, IPv6, TXT records
- Discovered flag: SKY-XJPO-5751

Results:

1. IPv4 addresses: 23.151.187.212, 43.71.247.55
2. IPv6 address: 2ecd:b3d:2f0c:e72b:da9:f4ee:81e:d62d
3. Flag TXT record: SKY-XJPO-5751
4. Redirect domain: r3d1r3ct3d.liber8.cityinthe.cloud
5. TTL for redirect: 600 seconds

6. Primary mail exchanger: mx1.liber8.cityinthe.cloud

HMAC Integrity Verification

Created HMAC verification script that:

- Calculates SHA-256 HMAC signatures
- Verifies file integrity
- Detects tampering patterns in DNS records

```
# Key functionality of hmac_integrity_checker.py
def calculate_hmac(message):
    # Calculate HMAC using SHA-256

def verify_hmac(message, signature):
    # Verify using constant-time comparison

def detect_tampering(log_entry):
    # Check for suspicious domains/patterns
    # Score risk level
    # Identify possible attack vectors
```

Script Functionality:

- Calculates HMAC signatures for messages using the SHA-256 algorithm
- Verifies signatures using constant-time comparison to prevent timing attacks
- Processes batches of message and signature files
- Identifies mismatched or tampered message/signature pairs
- Analyzes tampering patterns for potential security threats

The tool enabled me to:

1. Count messages with mismatched HMACs
2. Identify stored HMAC values and validate message integrity

Conclusion

The NCL Individual Game required diverse cybersecurity skills including:

- Scripting (Python, Bash)
- Forensic analysis
- Cryptography
- Network traffic analysis
- Database queries
- Steganography
- Password cracking

Custom tools were essential for automating complex tasks and providing deeper insights into the challenge data. The experience demonstrated the importance of both technical depth and breadth in cybersecurity analysis.

Scripts

Scripts

Binary Parser

```
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
})

# 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'])

# 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
    for i in range(1, len(sorted_attempts)):
        if (not sorted_attempts[i-1]['success'] and
            sorted_attempts[i]['success'] and
            sorted_attempts[i]['ip_address'] != sorted_attempts[i-1]['ip_address']):
            user_data[user]['success_after_failure'] += 1
```

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

    # Identify suspicious users based on multiple criteria
    suspicious_users = []
    for user, data in user_data.items():
        suspicion_score = 0

        # Multiple IPs used (especially if significantly more than other users)
        if data['unique_ips_count'] > 3:
            suspicion_score += data['unique_ips_count']

        # High number of failed logins followed by successful ones from different IPs
        if data['success_after_failure'] > 0:
            suspicion_score += data['success_after_failure'] * 10

        # High ratio of failed to successful logins
        if data['successful_logins'] > 0:
            failure_ratio = data['failed_logins'] / data['successful_logins']
            if failure_ratio > 3:
                suspicion_score += failure_ratio

        if suspicion_score > 10:
            suspicious_users.append((user, suspicion_score, data))

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

    return suspicious_users, user_data

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 = analyze_login_patterns(login_attempts)

        print("\n===== SUSPICIOUS ACTIVITY ANALYSIS =====")
        print(f"Total users analyzed: {len(user_data)}")

        if suspicious_users:
            print("\nPotentially compromised users (sorted by suspicion score):")
            for user, score, data in suspicious_users:
                print(f"\nUsername: {user} (Suspicion Score: {score:.2f})")
                print(f"  Unique IPs: {data['unique_ips_count']}")
                print(f"  Successful logins: {data['successful_logins']}")
                print(f"  Failed logins: {data['failed_logins']}")
                print(f"  Successful logins after failures from different IPs:
{data['success_after_failure']}")
                print(f"  IP addresses used: {', '.join(data['ips'])}")
        else:
            print("\nNo suspicious users identified.")

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

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

Scripts

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()
```

Scripts

GPG Verify

```
#!/bin/bash

echo "Verifying signature files in the current directory..."

for sig_file in *.sig; do
  if [[ -f "$sig_file" ]]; then
    original_file="${sig_file%.sig}"
    echo ""
    echo "Verifying signature for \"$original_file\" using \"$sig_file\"..."
    gpg --verify "$sig_file" "$original_file"
    if [ $? -ne 0 ]; then
      echo "[ERROR] Signature verification failed for \"$original_file\". The file may have
been tampered with or the signature is invalid."
    else
      echo "[OK] Signature verification successful for \"$original_file\"."
    fi
  fi
done

echo ""
echo "Verification process complete."
```

Scripts

Hash Identifier

```
#!/usr/bin/env python3

import re
import sys
import hashlib
from collections import defaultdict

def identify_hash(hash_string):
    """Identify the type of hash based on pattern, length, and character set."""

    # Clean the hash string
    hash_string = hash_string.strip()

    # Check for empty string
    if not hash_string:
        return "Empty string"

    # Check for common hash formats with special syntax
    if hash_string.startswith('$1$'):
        return "MD5 (Unix)"
    if hash_string.startswith('$2a$') or hash_string.startswith('$2b$') or
hash_string.startswith('$2y$'):
        return "Bcrypt"
    if hash_string.startswith('$5$'):
        return "SHA-256 (Unix)"
    if hash_string.startswith('$6$'):
        return "SHA-512 (Unix)"
    if hash_string.startswith('$pbkdf2-sha256$'):
        return "PBKDF2-SHA256"
    if hash_string.startswith('$sha1$'):
        return "SHA-1 (Unix)"
    if hash_string.startswith('$pdf$'):
        return "PDF (Hashcat format)"
    if hash_string.startswith('$P$') or hash_string.startswith('$H$'):
```

```
    return "PHPass (WordPress/phpBB)"
if hash_string.startswith('$apr1$'):
    return "APR1-MD5"
if re.match(r'^[a-fA-F0-9]{32}:[a-fA-F0-9]{32}$', hash_string):
    return "MD5(Half:Salt)"

# Check for common hash lengths
hash_length = len(hash_string)
possible_types = []

# Check if the hash is hexadecimal
if re.match(r'^[a-fA-F0-9]+$', hash_string):
    if hash_length == 32:
        possible_types.append("MD5")
        possible_types.append("MD4")
        possible_types.append("NTLM")
        possible_types.append("RIPEMD-128")
    elif hash_length == 40:
        possible_types.append("SHA-1")
        possible_types.append("RIPEMD-160")
    elif hash_length == 64:
        possible_types.append("SHA-256")
        possible_types.append("RIPEMD-256")
    elif hash_length == 96:
        possible_types.append("SHA-384")
    elif hash_length == 128:
        possible_types.append("SHA-512")
        possible_types.append("Whirlpool")
    elif hash_length == 16:
        possible_types.append("MySQL323")
        possible_types.append("DES(Oracle)")
    elif hash_length == 41 and hash_string.startswith('*'):
        possible_types.append("MySQL5")
    elif hash_length == 56:
        possible_types.append("SHA-224")
    elif hash_length == 8:
        possible_types.append("CRC32")
        possible_types.append("ADLER32")

# Check for Base64 character set (with potential padding)
```

```
    if re.match(r'^[A-Za-z0-9+/]+={0,2}$', hash_string):
        if hash_length == 24:
            possible_types.append("MD5 (Base64)")
        elif hash_length == 28:
            possible_types.append("SHA-1 (Base64)")
        elif hash_length == 44:
            possible_types.append("SHA-256 (Base64)")
        elif hash_length == 88:
            possible_types.append("SHA-512 (Base64)")
        else:
            possible_types.append("Base64 encoded")

    # No specific hash type identified, give general suggestion
    if not possible_types:
        if re.match(r'^[a-fA-F0-9]+$', hash_string):
            return f"Unknown hash (Hexadecimal, {hash_length} chars)"
        else:
            return f"Unknown format (possibly not a standard hash, or custom format)"

    return " or ".join(possible_types)

def main():
    if len(sys.argv) != 2:
        print("Usage: python hash_identifier.py <hash_file>")
        sys.exit(1)

    hash_file = sys.argv[1]

    try:
        with open(hash_file, 'r') as f:
            lines = f.readlines()

        print(f"Analyzing {len(lines)} hashes from {hash_file}...\n")

        hash_types = defaultdict(int)

        for i, line in enumerate(lines, 1):
            hash_string = line.strip()
            if not hash_string or hash_string.startswith('#'):
                continue
```

```
            hash_type = identify_hash(hash_string)
            hash_types[hash_type] += 1

            # Print the first few and last few hash identifications
            if i <= 3 or i > len(lines) - 3:
                print(f"Line {i}: {hash_string[:40]}{'...' if len(hash_string) > 40 else ''} -
> {hash_type}")
            elif i == 4 and len(lines) > 6:
                print(f"... ({len(lines) - 6} more hashes) ...")

        print("\nSummary of hash types:")
        for hash_type, count in sorted(hash_types.items(), key=lambda x: x[1], reverse=True):
            print(f"  {hash_type}: {count}")

    except FileNotFoundError:
        print(f"Error: File '{hash_file}' not found.")
        sys.exit(1)
    except Exception as e:
        print(f"Error: {e}")
        sys.exit(1)

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

Scripts

HMAC

```
#!/usr/bin/env python3
"""
Simple HMAC Verification Script

This script verifies HMAC signatures for message files by:
1. Finding all message_#.txt and message_#.hmac file pairs
2. Checking each line to verify the HMAC integrity
3. Reporting only basic verification results without additional analysis

Usage:
    python simple_hmac_verify.py --directory <logs_directory> --key <hmac_key>
"""

import hmac
import hashlib
import os
import sys
import re
import glob
import argparse
from datetime import datetime

# The valid signing key
VALID_KEY = 'ciCloud-API-20240315-4f7b9c'

def calculate_hmac(message, key):
    """Calculate HMAC signature for a message."""
    key_bytes = key.encode('utf-8')
    message_bytes = message.encode('utf-8')
    signature = hmac.new(key_bytes, message_bytes, hashlib.sha256)
    return signature.hexdigest()

def verify_hmac(message, signature, key):
    """Verify if a message's HMAC signature is valid."""
```

```
    calculated_signature = calculate_hmac(message, key)
    return hmac.compare_digest(calculated_signature, signature)

def read_file(file_path):
    """Read a file and return its lines."""
    with open(file_path, 'r') as f:
        return [line.rstrip() for line in f.readlines()]

def find_file_pairs(directory):
    """Find matching message/HMAC file pairs in the directory."""
    file_pairs = []

    # Find all message_*.txt files
    message_files = glob.glob(os.path.join(directory, "message_*.txt"))

    for message_file in message_files:
        # Extract the number part
        match = re.search(r'message_(\d+)\.txt$', message_file)
        if match:
            number = match.group(1)
            hmac_file = os.path.join(directory, f"message_{number}.hmac")

            # Check if the corresponding HMAC file exists
            if os.path.exists(hmac_file):
                file_pairs.append((message_file, hmac_file))

    return file_pairs

def process_file_pair(message_file, hmac_file, key):
    """Process a single message/HMAC file pair."""
    # Extract file number for identification
    match = re.search(r'message_(\d+)\.txt$', message_file)
    file_id = match.group(1) if match else os.path.basename(message_file)

    try:
        # Read files
        message_lines = read_file(message_file)
        hmac_lines = read_file(hmac_file)

        total_lines = min(len(message_lines), len(hmac_lines))
```

```
        valid_lines = 0
        invalid_lines = 0
        mismatched_entries = []

        print(f"Processing file {file_id}: {os.path.basename(message_file)}")
        print(f"  - Total lines: {total_lines}")

        # Process each line
        for i in range(total_lines):
            message = message_lines[i]
            signature = hmac_lines[i]

            # Skip empty lines
            if not message or not signature:
                continue

            # Debug: Print first few characters of message and signature
            if i < 3:  # Just print a few examples for debugging
                print(f"  - Line {i+1} check:")
                print(f"    Message: {message[:30]}{'...' if len(message) > 30 else ''}")
                print(f"    Signature: {signature[:30]}{'...' if len(signature) > 30 else
''}")
                print(f"    Calculated: {calculate_hmac(message, key)[:30]}...")

            # Verify HMAC
            is_valid = verify_hmac(message, signature, key)

            if is_valid:
                valid_lines += 1
            else:
                invalid_lines += 1
                mismatched_entries.append({
                    'line': i + 1,
                    'message': message,
                    'provided_signature': signature,
                    'calculated_signature': calculate_hmac(message, key)
                })

        result = {
            'file_id': file_id,
```

```
            'message_file': message_file,
            'hmac_file': hmac_file,
            'total_lines': total_lines,
            'valid_lines': valid_lines,
            'invalid_lines': invalid_lines,
            'mismatched_entries': mismatched_entries[:10]  # Only include first 10 for brevity
        }

        print(f"  - Valid lines: {valid_lines}")
        print(f"  - Invalid lines: {invalid_lines}")
        print(f"  - Integrity: {'INTACT' if invalid_lines == 0 else 'COMPROMISED'}")
        print()

        return result

    except Exception as e:
        print(f"Error processing file pair ({message_file}, {hmac_file}): {e}")
        return {
            'file_id': file_id,
            'message_file': message_file,
            'hmac_file': hmac_file,
            'error': str(e)
        }

def main():
    """Main entry point for the script."""
    parser = argparse.ArgumentParser(description='Simple HMAC Verification')
    parser.add_argument('--directory', '-d', required=True, help='Directory containing log
files')
    parser.add_argument('--key', '-k', default=VALID_KEY, help=f'HMAC signing key (default:
{VALID_KEY})')
    parser.add_argument('--verbose', '-v', action='store_true', help='Enable verbose output')

    args = parser.parse_args()

    try:
        start_time = datetime.now()
        print(f"Starting HMAC verification in {args.directory}")
        print(f"Using key: {args.key}")
        print(f"Started at: {start_time.isoformat()}")
```

```
print("-" * 60)

# Find all file pairs
file_pairs = find_file_pairs(args.directory)

if not file_pairs:
    print(f"No matching message/HMAC file pairs found in {args.directory}")
    sys.exit(1)

print(f"Found {len(file_pairs)} file pairs")
print("-" * 60)

# Process each file pair
results = []
total_files = len(file_pairs)
files_with_errors = 0
files_with_mismatches = 0
total_lines_processed = 0
total_mismatched_lines = 0

for message_file, hmac_file in file_pairs:
    result = process_file_pair(message_file, hmac_file, args.key)
    results.append(result)

    if 'error' in result:
        files_with_errors += 1
    else:
        total_lines_processed += result['total_lines']
        total_mismatched_lines += result['invalid_lines']

        if result['invalid_lines'] > 0:
            files_with_mismatches += 1

# Summary
print("-" * 60)
print("VERIFICATION SUMMARY")
print("-" * 60)
print(f"Total file pairs processed: {total_files}")
print(f"Files with errors: {files_with_errors}")
print(f"Files with mismatched HMACs: {files_with_mismatches}")
```

```
        print(f"Total lines processed: {total_lines_processed}")
        print(f"Total mismatched lines: {total_mismatched_lines}")

        end_time = datetime.now()
        duration = end_time - start_time
        print(f"Duration: {duration.total_seconds():.2f} seconds")

        # List files with mismatches
        if files_with_mismatches > 0:
            print("\nFiles with mismatched HMACs:")
            for result in results:
                if 'invalid_lines' in result and result['invalid_lines'] > 0:
                    print(f"- {os.path.basename(result['message_file'])}:
{result['invalid_lines']} mismatched lines")

                    # Show example of first mismatched entry
                    if args.verbose and result['mismatched_entries']:
                        first_mismatch = result['mismatched_entries'][0]
                        print(f"  Example (line {first_mismatch['line']}):")
                        print(f"  Message: {first_mismatch['message'][:50]}...")
                        print(f"  Provided HMAC: {first_mismatch['provided_signature']}")
                        print(f"  Calculated HMAC: {first_mismatch['calculated_signature']}")
                        print()

    except Exception as e:
        print(f"Error: {e}")
        import traceback
        traceback.print_exc()
        sys.exit(1)

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

Scripts

HMAC Integrity Checker

```
#!/usr/bin/env python3
"""
DNS Subdomain Batch Integrity Checker

This script processes multiple message/HMAC file pairs in a directory, following the pattern:
message_#.txt and message_#.hmac

It automatically detects and verifies all matching pairs in the specified directory,
generating a comprehensive report of integrity issues across all files.

Usage:
    python dns_batch_integrity.py --directory <logs_directory> --output <output_dir>
"""

import hmac
import hashlib
import sys
import os
import re
import argparse
import json
import glob
from datetime import datetime
from typing import Dict, List, Any, Tuple, Set

# The valid signing key
VALID_KEY = 'ciCloud-API-20240315-4f7b9c'

class DNSSubdomainBatchChecker:
    def __init__(self, key: str = VALID_KEY):
        """
        Initialize the DNS Subdomain Integrity Checker.

        Args:
```

```
            key: The HMAC signing key
        """
        self.key = key

        # Initialize common DNS patterns to check for tampering
        self.common_subdomains = {
            'www', 'mail', 'api', 'admin', 'portal', 'test', 'dev', 'staging',
            'secure', 'vpn', 'internal', 'mx', 'smtp', 'pop', 'imap', 'webmail',
            'remote', 'cdn', 'dns', 'ns1', 'ns2', 'ldap', 'db', 'mysql', 'ftp'
        }

        # Suspicious TLDs often used in attacks
        self.suspicious_tlds = {
            'xyz', 'top', 'club', 'cyou', 'icu', 'rest', 'space', 'casa',
            'monster', 'bar', 'gq', 'tk', 'ml', 'cf', 'ga'
        }

        # Common character substitutions used in spoofing
        self.char_substitutions = {
            '0': 'o', 'o': '0',
            '1': 'l', 'l': '1', 'i': '1',
            '5': 's', 's': '5',
            '3': 'e', 'e': '3',
            '4': 'a', 'a': '4',
            '6': 'g', 'g': '6',
            '7': 't', 't': '7',
            '8': 'b', 'b': '8'
        }

    def calculate_hmac(self, message: str) -> str:
        """
        Calculate HMAC signature for a message.

        Args:
            message: The message to sign

        Returns:
            The HMAC signature (hex encoded)
        """
        key_bytes = self.key.encode('utf-8')
```

```
    message_bytes = message.encode('utf-8')
    signature = hmac.new(key_bytes, message_bytes, hashlib.sha256)
    return signature.hexdigest()

def verify_hmac(self, message: str, signature: str) -> bool:
    """
    Verify if a message's HMAC signature is valid.

    Args:
        message: The message to verify
        signature: The provided HMAC signature

    Returns:
        True if signature is valid, False otherwise
    """
    calculated_signature = self.calculate_hmac(message)
    # Use constant-time comparison to prevent timing attacks
    return hmac.compare_digest(calculated_signature, signature)

def read_file(self, file_path: str) -> List[str]:
    """
    Read a file and return its lines.

    Args:
        file_path: Path to the file

    Returns:
        List of lines from the file
    """
    with open(file_path, 'r') as f:
        return [line.rstrip() for line in f.readlines()]

def find_file_pairs(self, directory: str) -> List[Tuple[str, str]]:
    """
    Find matching message/HMAC file pairs in the directory.

    Args:
        directory: Directory to search for files

    Returns:
```

```
        List of tuples (message_file_path, hmac_file_path)
    """
    file_pairs = []

    # Find all message_*.txt files
    message_files = glob.glob(os.path.join(directory, "message_*.txt"))

    for message_file in message_files:
        # Extract the number part
        match = re.search(r'message_(\d+)\.txt$', message_file)
        if match:
            number = match.group(1)
            hmac_file = os.path.join(directory, f"message_{number}.hmac")

            # Check if the corresponding HMAC file exists
            if os.path.exists(hmac_file):
                file_pairs.append((message_file, hmac_file))

    return file_pairs

def extract_domain_info(self, log_entry: str) -> Dict[str, Any]:
    """
    Extract domain and subdomain information from a log entry.

    Args:
        log_entry: A log entry string

    Returns:
        Dictionary with extracted domain information
    """
    domain_info = {
        'has_domain': False,
        'domain': '',
        'subdomain': '',
        'tld': ''
    }

    # Try to find domain patterns in the log entry
    # This regex looks for domain.tld or subdomain.domain.tld patterns
    domain_matches = re.findall(r'([a-zA-Z0-9][-a-zA-Z0-9]*(\.[a-zA-Z0-9][-a-zA-Z0-
```

```
9]*)+)', log_entry)

        if domain_matches:
            domain_info['has_domain'] = True
            full_domain = domain_matches[0][0]
            domain_info['domain'] = full_domain

            # Split by dots to extract subdomain and TLD
            parts = full_domain.split('.')

            if len(parts) >= 2:
                domain_info['tld'] = parts[-1].lower()

                if len(parts) > 2:
                    domain_info['subdomain'] = '.'.join(parts[:-2])

        return domain_info

    def detect_tampering(self, log_entry: str) -> Dict[str, Any]:
        """
        Detect possible tampering in a DNS log entry.

        Args:
            log_entry: A log entry string

        Returns:
            Dictionary with tampering analysis
        """
        analysis = {
            'is_suspicious': False,
            'tampering_patterns': set(),
            'possible_original': '',
            'risk_level': 'low',
            'reasons': []
        }

        # Extract any domain information from the log entry
        domain_info = self.extract_domain_info(log_entry)

        if domain_info['has_domain']:
```

```
            # Check for suspicious TLDs
            if domain_info['tld'] in self.suspicious_tlds:
                analysis['is_suspicious'] = True
                analysis['tampering_patterns'].add('suspicious_tld')
                analysis['risk_level'] = 'medium'
                analysis['reasons'].append(f"Suspicious TLD found: .{domain_info['tld']}")

            # Check for subdomain issues
            if domain_info['subdomain']:
                subdomain = domain_info['subdomain']

                # Check for character substitutions
                for char in subdomain:
                    if char in self.char_substitutions:
                        analysis['is_suspicious'] = True
                        analysis['tampering_patterns'].add('character_substitution')
                        analysis['risk_level'] = 'high'
                        analysis['reasons'].append(f"Possible character substitution: '{char}'
might be '{self.char_substitutions[char]}'")

                        # Generate a possible original by replacing the character
                        possible_original = log_entry.replace(subdomain,
                                                              subdomain.replace(char,
self.char_substitutions[char]))
                        analysis['possible_original'] = possible_original

                # Check for similar but different subdomains
                for common_sub in self.common_subdomains:
                    if subdomain != common_sub and self.levenshtein_distance(subdomain,
common_sub) <= 2:
                        analysis['is_suspicious'] = True
                        analysis['tampering_patterns'].add('similar_subdomain')
                        analysis['risk_level'] = 'high'
                        analysis['reasons'].append(f"Subdomain '{subdomain}' is suspiciously
similar to common subdomain '{common_sub}'")

                        # Generate a possible original version
                        possible_original = log_entry.replace(subdomain, common_sub)
                        analysis['possible_original'] = possible_original
```

```
                # Check for unusually long subdomains (potential data exfiltration)
                if len(subdomain) > 30:
                    analysis['is_suspicious'] = True
                    analysis['tampering_patterns'].add('exfiltration_subdomain')
                    analysis['risk_level'] = 'high'
                    analysis['reasons'].append(f"Unusually long subdomain (length:
{len(subdomain)}) may indicate data exfiltration")

        # Check for IP address patterns
        ip_matches = re.findall(r'\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b', log_entry)
        if ip_matches:
            # Check for suspicious IP ranges
            for ip in ip_matches:
                octets = [int(octet) for octet in ip.split('.')]

                # Check for loopback or private IP misuse
                if octets[0] == 127 or (octets[0] == 10) or \
                   (octets[0] == 172 and 16 <= octets[1] <= 31) or \
                   (octets[0] == 192 and octets[1] == 168):
                    analysis['is_suspicious'] = True
                    analysis['tampering_patterns'].add('internal_ip_exposure')
                    analysis['risk_level'] = 'critical'
                    analysis['reasons'].append(f"Internal IP address exposed: {ip}")

        # Check for DNS record types and modifications
        record_types = ['A', 'AAAA', 'MX', 'CNAME', 'TXT', 'NS', 'SOA', 'SRV', 'PTR']
        for record_type in record_types:
            # Look for record type followed by manipulation indicators
            pattern = r'\b' + record_type +
r'\s+(?:changed|modified|updated|deleted|removed|added)\b'
            if re.search(pattern, log_entry, re.IGNORECASE):
                analysis['is_suspicious'] = True
                analysis['tampering_patterns'].add('dns_record_modification')
                analysis['risk_level'] = 'high'
                analysis['reasons'].append(f"DNS {record_type} record modification detected")

        # Look for DNS amplification or reflection attack patterns
        if re.search(r'\b(?:amplification|reflection|flood|ddos)\b', log_entry, re.IGNORECASE)
and domain_info['has_domain']:
            analysis['is_suspicious'] = True
```

```
            analysis['tampering_patterns'].add('dns_amplification')
            analysis['risk_level'] = 'critical'
            analysis['reasons'].append(f"Possible DNS amplification attack signature")

        # Update risk level based on number of patterns
        if len(analysis['tampering_patterns']) >= 3:
            analysis['risk_level'] = 'critical'
        elif len(analysis['tampering_patterns']) == 2:
            analysis['risk_level'] = 'high' if analysis['risk_level'] != 'critical' else
'critical'

        return analysis

    @staticmethod
    def levenshtein_distance(s1: str, s2: str) -> int:
        """
        Calculate the Levenshtein distance between two strings.

        Args:
            s1: First string
            s2: Second string

        Returns:
            The Levenshtein distance
        """
        if len(s1) < len(s2):
            return DNSSubdomainBatchChecker.levenshtein_distance(s2, s1)

        if len(s2) == 0:
            return len(s1)

        previous_row = range(len(s2) + 1)
        for i, c1 in enumerate(s1):
            current_row = [i + 1]
            for j, c2 in enumerate(s2):
                insertions = previous_row[j + 1] + 1
                deletions = current_row[j] + 1
                substitutions = previous_row[j] + (c1 != c2)
                current_row.append(min(insertions, deletions, substitutions))
            previous_row = current_row
```

```
    return previous_row[-1]

def process_file_pair(self, message_file: str, hmac_file: str) -> Dict[str, Any]:
    """
    Process a single message/HMAC file pair.

    Args:
        message_file: Path to the message file
        hmac_file: Path to the HMAC file

    Returns:
        Dictionary with processing results
    """
    # Extract file number for identification
    match = re.search(r'message_(\d+)\.txt$', message_file)
    file_id = match.group(1) if match else os.path.basename(message_file)

    # Read files
    try:
        message_content = self.read_file(message_file)
        hmac_content = self.read_file(hmac_file)

        # Verify each line
        results = {
            'file_id': file_id,
            'message_file': message_file,
            'hmac_file': hmac_file,
            'total_lines': min(len(message_content), len(hmac_content)),
            'valid_lines': 0,
            'invalid_lines': 0,
            'suspicious_lines': 0,
            'invalid_entries': [],
            'tampering_summary': {
                'patterns': {},
                'risk_levels': {
                    'low': 0,
                    'medium': 0,
                    'high': 0,
                    'critical': 0
```

```
        }
    }
}

# Process lines
for i in range(min(len(message_content), len(hmac_content))):
    message = message_content[i]
    signature = hmac_content[i]

    # Skip empty lines
    if not message or not signature:
        continue

    # Verify HMAC
    is_valid = self.verify_hmac(message, signature)

    if is_valid:
        results['valid_lines'] += 1
    else:
        results['invalid_lines'] += 1

        # Generate correct signature
        correct_signature = self.calculate_hmac(message)

        # Check for tampering
        tampering_analysis = self.detect_tampering(message)

        invalid_entry = {
            'line_number': i + 1,
            'message': message,
            'provided_signature': signature,
            'correct_signature': correct_signature,
            'tampering_analysis': tampering_analysis
        }

        results['invalid_entries'].append(invalid_entry)

        # Update tampering statistics
        if tampering_analysis['is_suspicious']:
            results['suspicious_lines'] += 1
```

```
results['tampering_summary']['risk_levels'][tampering_analysis['risk_level']] += 1

                        # Count pattern occurrences
                        for pattern in tampering_analysis['tampering_patterns']:
                            if pattern not in results['tampering_summary']['patterns']:
                                results['tampering_summary']['patterns'][pattern] = 0
                            results['tampering_summary']['patterns'][pattern] += 1

            return results

        except Exception as e:
            print(f"Error processing file pair ({message_file}, {hmac_file}): {e}")
            return {
                'file_id': file_id,
                'message_file': message_file,
                'hmac_file': hmac_file,
                'error': str(e)
            }

    def process_directory(self, directory: str) -> Dict[str, Any]:
        """
        Process all matching file pairs in a directory.

        Args:
            directory: Directory containing message_*.txt and message_*.hmac files

        Returns:
            Dictionary with processing results for all files
        """
        # Find all matching file pairs
        file_pairs = self.find_file_pairs(directory)

        if not file_pairs:
            print(f"No matching message/HMAC file pairs found in {directory}")
            return {'error': 'No matching file pairs found'}

        # Process each file pair
        results = {
            'directory': directory,
```

```
            'total_files': len(file_pairs),
            'processed_files': 0,
            'files_with_errors': 0,
            'total_lines_processed': 0,
            'total_invalid_lines': 0,
            'total_suspicious_lines': 0,
            'file_results': [],
            'overall_tampering_summary': {
                'patterns': {},
                'risk_levels': {
                    'low': 0,
                    'medium': 0,
                    'high': 0,
                    'critical': 0
                }
            }
        }

        for message_file, hmac_file in file_pairs:
            print(f"Processing file pair: {os.path.basename(message_file)} and
{os.path.basename(hmac_file)}")

            # Process file pair
            file_result = self.process_file_pair(message_file, hmac_file)
            results['file_results'].append(file_result)

            # Update overall statistics
            if 'error' in file_result:
                results['files_with_errors'] += 1
            else:
                results['processed_files'] += 1
                results['total_lines_processed'] += file_result['total_lines']
                results['total_invalid_lines'] += file_result['invalid_lines']
                results['total_suspicious_lines'] += file_result['suspicious_lines']

                # Aggregate tampering patterns
                for pattern, count in file_result['tampering_summary']['patterns'].items():
                    if pattern not in results['overall_tampering_summary']['patterns']:
                        results['overall_tampering_summary']['patterns'][pattern] = 0
                    results['overall_tampering_summary']['patterns'][pattern] += count
```

```
            # Aggregate risk levels
            for level in ['low', 'medium', 'high', 'critical']:
                results['overall_tampering_summary']['risk_levels'][level] += \
                    file_result['tampering_summary']['risk_levels'][level]

    return results

def save_corrected_hmac_files(self, results: Dict[str, Any], output_dir: str) -> None:
    """
    Save corrected HMAC files for each processed file pair.

    Args:
        results: Overall processing results
        output_dir: Output directory
    """
    corrected_dir = os.path.join(output_dir, 'corrected_hmac_files')
    os.makedirs(corrected_dir, exist_ok=True)

    for file_result in results['file_results']:
        if 'error' in file_result:
            continue

        # Get original file content
        message_file = file_result['message_file']
        hmac_file = file_result['hmac_file']

        try:
            # Read original message file
            message_content = self.read_file(message_file)

            # Create corrected HMAC file
            corrected_hmac_path = os.path.join(corrected_dir, os.path.basename(hmac_file))

            with open(corrected_hmac_path, 'w') as f:
                for message in message_content:
                    if message:  # Skip empty lines
                        correct_signature = self.calculate_hmac(message)
                        f.write(f"{correct_signature}\n")
```

```
                print(f"Created corrected HMAC file: {corrected_hmac_path}")

            except Exception as e:
                print(f"Error creating corrected HMAC file for {os.path.basename(hmac_file)}:
{e}")

    def save_results(self, results: Dict[str, Any], output_dir: str) -> None:
        """
        Save processing results to output files.

        Args:
            results: Overall processing results
            output_dir: Output directory
        """
        os.makedirs(output_dir, exist_ok=True)

        # Save overall JSON results
        with open(os.path.join(output_dir, 'batch_results.json'), 'w') as f:
            # Convert sets to lists for JSON serialization
            serializable_results = json.dumps(results, indent=2, default=lambda x: list(x) if
isinstance(x, set) else x)
            f.write(serializable_results)

        # Save detailed report
        with open(os.path.join(output_dir, 'integrity_report.txt'), 'w') as f:
            f.write(f"DNS Subdomain Batch Integrity Report\n")
            f.write(f"==================================\n\n")
            f.write(f"Generated: {datetime.now().isoformat()}\n\n")

            f.write(f"Overall Summary:\n")
            f.write(f"---------------\n")
            f.write(f"Directory processed: {results['directory']}\n")
            f.write(f"Total file pairs: {results['total_files']}\n")
            f.write(f"Successfully processed: {results['processed_files']}\n")
            f.write(f"Files with errors: {results['files_with_errors']}\n")
            f.write(f"Total log lines processed: {results['total_lines_processed']}\n")
            f.write(f"Total invalid lines: {results['total_invalid_lines']}\n")
            f.write(f"Total suspicious lines: {results['total_suspicious_lines']}\n\n")

            # Risk level summary
```

```
            if results['total_suspicious_lines'] > 0:
                f.write(f"Risk Level Distribution:\n")
                for level in ['low', 'medium', 'high', 'critical']:
                    count = results['overall_tampering_summary']['risk_levels'][level]
                    indicator = '!' * (1 if level == 'low' else 2 if level == 'medium' else 3
if level == 'high' else 4)
                    f.write(f"  {indicator} {level.upper()}: {count}\n")

                f.write(f"\nTampering Patterns Detected:\n")
                for pattern, count in
sorted(results['overall_tampering_summary']['patterns'].items(),
                                           key=lambda x: x[1], reverse=True):
                    f.write(f"  - {pattern}: {count}\n")

            # Per-file summary
            f.write(f"\nPer-File Summary:\n")
            f.write(f"----------------\n")
            for file_result in results['file_results']:
                if 'error' in file_result:
                    f.write(f"File {file_result['file_id']}: ERROR -
{file_result['error']}\n")
                else:
                    integrity_status = "COMPROMISED" if file_result['invalid_lines'] > 0 else
"INTACT"
                    risk_level = "HIGH RISK" if
(file_result['tampering_summary']['risk_levels']['high'] > 0 or

file_result['tampering_summary']['risk_levels']['critical'] > 0) else \
                                  "MEDIUM RISK" if
file_result['tampering_summary']['risk_levels']['medium'] > 0 else \
                                  "LOW RISK" if file_result['suspicious_lines'] > 0 else "SAFE"

                    f.write(f"File {file_result['file_id']}: {integrity_status} -
{risk_level}\n")
                    f.write(f"  Message file:
{os.path.basename(file_result['message_file'])}\n")
                    f.write(f"  Lines: {file_result['total_lines']} total,
{file_result['invalid_lines']} invalid, {file_result['suspicious_lines']} suspicious\n")

                    if file_result['suspicious_lines'] > 0:
```

```
                        # Show the first few suspicious entries
                        suspicious_entries = [entry for entry in
file_result['invalid_entries']
                                              if entry['tampering_analysis']['is_suspicious']]

                        f.write(f"  Top suspicious entries ({min(3, len(suspicious_entries))}
of {len(suspicious_entries)}):\n")
                        for i, entry in enumerate(suspicious_entries[:3]):
                            f.write(f"    Line {entry['line_number']}:
{entry['message'][:50]}{'...' if len(entry['message']) > 50 else ''}\n")
                            f.write(f"      Risk:
{entry['tampering_analysis']['risk_level'].upper()}\n")
                            f.write(f"      Patterns: {',
'.join(entry['tampering_analysis']['tampering_patterns'])}\n")

                    f.write("\n")

        # Create a file with high-risk entries for immediate attention
        high_risk_entries = []
        for file_result in results['file_results']:
            if 'error' in file_result:
                continue

            file_id = file_result['file_id']
            for entry in file_result['invalid_entries']:
                if entry['tampering_analysis']['is_suspicious'] and \
                   entry['tampering_analysis']['risk_level'] in ['high', 'critical']:
                    entry_copy = entry.copy()
                    entry_copy['file_id'] = file_id
                    high_risk_entries.append(entry_copy)

        if high_risk_entries:
            with open(os.path.join(output_dir, 'high_risk_entries.txt'), 'w') as f:
                f.write(f"HIGH RISK DNS LOG ENTRIES - IMMEDIATE ATTENTION REQUIRED\n")
                f.write(f"=================================================\n\n")
                f.write(f"Generated: {datetime.now().isoformat()}\n")
                f.write(f"Total high-risk entries: {len(high_risk_entries)}\n\n")

                # Sort by risk level (critical first)
                high_risk_entries.sort(key=lambda x: 0 if
```

```
x['tampering_analysis']['risk_level'] == 'critical' else 1)

                for entry in high_risk_entries:
                    f.write(f"File {entry['file_id']}, Line {entry['line_number']} -
[{entry['tampering_analysis']['risk_level'].upper()}]\n")
                    f.write(f"  Message: {entry['message']}\n")
                    f.write(f"  Provided signature: {entry['provided_signature']}\n")
                    f.write(f"  Correct signature: {entry['correct_signature']}\n")
                    f.write(f"  Tampering patterns: {',
'.join(entry['tampering_analysis']['tampering_patterns'])}\n")
                    f.write(f"  Reasons:\n")
                    for reason in entry['tampering_analysis']['reasons']:
                        f.write(f"    - {reason}\n")

                    if entry['tampering_analysis']['possible_original']:
                        f.write(f"  Possible original:
{entry['tampering_analysis']['possible_original']}\n")

                    f.write("\n")

        # Save corrected HMAC files
        self.save_corrected_hmac_files(results, output_dir)

def main():
    """Main entry point for the script."""
    parser = argparse.ArgumentParser(description='DNS Subdomain Batch Integrity Checker')
    parser.add_argument('--directory', '-d', required=True, help='Directory containing log
files')
    parser.add_argument('--output', '-o', default='batch_output', help='Output directory
(default: batch_output)')
    parser.add_argument('--key', '-k', default=VALID_KEY, help=f'HMAC signing key (default:
{VALID_KEY})')

    args = parser.parse_args()

    checker = DNSSubdomainBatchChecker(key=args.key)

    try:
        start_time = datetime.now()
        print(f"Starting batch processing of DNS log files in {args.directory}")
```

```
        print(f"Started at: {start_time.isoformat()}")

        results = checker.process_directory(args.directory)

        if 'error' in results:
            print(f"Error: {results['error']}")
            sys.exit(1)

        # Save results
        checker.save_results(results, args.output)

        end_time = datetime.now()
        duration = end_time - start_time

        print(f"\nBatch processing completed!")
        print(f"Duration: {duration.total_seconds():.2f} seconds")
        print(f"Files processed: {results['processed_files']} of {results['total_files']}")
        print(f"Total lines checked: {results['total_lines_processed']}")
        print(f"Invalid lines detected: {results['total_invalid_lines']}")
        print(f"Suspicious lines detected: {results['total_suspicious_lines']}")
        print(f"Results saved to: {args.output}")

        if results['total_suspicious_lines'] > 0:
            print(f"\n⚠ WARNING: {results['total_suspicious_lines']} suspicious log entries
detected!")
            high_risk = results['overall_tampering_summary']['risk_levels']['high'] + \
                        results['overall_tampering_summary']['risk_levels']['critical']

            if high_risk > 0:
                print(f"🚨 CRITICAL: {high_risk} high or critical risk entries found!")
                print(f"Check {os.path.join(args.output, 'high_risk_entries.txt')} for
details")

    except Exception as e:
        print(f"Error: {e}")
        import traceback
        traceback.print_exc()
        sys.exit(1)

if __name__ == "__main__":
```

```
    main()
```

Scripts

Liber8tion Cracker

```
#!/usr/bin/env python3

import os
import argparse
import subprocess
import sys
import tempfile
import shutil

def run_hashcat(cmd, description):
    """Run a hashcat command with proper logging"""
    print(f"[+] {description}")
    print(f"[+] Command: {' '.join(cmd)}")
    try:
        subprocess.run(cmd, check=False)
    except Exception as e:
        print(f"[-] Error running hashcat: {e}")

def main():
    parser = argparse.ArgumentParser(description='Crack hashes using the Liber8ion Passphrase Standard')
    parser.add_argument('--hash-file', required=True, help='File containing hashes to crack')
    parser.add_argument('--hash-type', required=True, help='Hashcat hash type (e.g. 0 for MD5, 100 for SHA1)')
    parser.add_argument('--wordlist', default='/usr/share/wordlists/rockyou.txt', help='Dictionary wordlist')
    parser.add_argument('--output', default='cracked_passwords.txt', help='Output file for cracked passwords')
    args = parser.parse_args()

    # Create temporary directory
    temp_dir = tempfile.mkdtemp(prefix="liber8ion_")
    print(f"[+] Using temporary directory: {temp_dir}")
```

```
    # Path for the potfile
    potfile = os.path.join(temp_dir, "liber8ion.potfile")

    # Create a smaller dictionary with lowercase words
    print(f"[+] Creating optimized wordlist from {args.wordlist}...")
    lowercase_dict = os.path.join(temp_dir, "lowercase_dict.txt")
    uppercase_dict = os.path.join(temp_dir, "uppercase_dict.txt")

    try:
        # Take a reasonable subset to avoid memory issues
        with open(args.wordlist, 'r', encoding='latin-1', errors='ignore') as infile, \
             open(lowercase_dict, 'w') as lower_out, \
             open(uppercase_dict, 'w') as upper_out:
            for i, line in enumerate(infile):
                if i >= 100000:  # Limit to first 100k words
                    break
                word = line.strip()
                if word and len(word) >= 3 and len(word) <= 10:  # Filter reasonable word
lengths
                    lower_out.write(f"{word.lower()}\n")
                    upper_out.write(f"{word.upper()}\n")
    except Exception as e:
        print(f"[-] Error processing wordlist: {e}")
        sys.exit(1)

    # Create file with digits
    digits_dict = os.path.join(temp_dir, "digits.txt")
    with open(digits_dict, 'w') as f:
        for i in range(10):
            f.write(f"{i}\n")

    # Create special character dictionaries
    hyphen_dict = os.path.join(temp_dir, "hyphen.txt")
    with open(hyphen_dict, 'w') as f:
        f.write("-\n")

    special_chars_dict = os.path.join(temp_dir, "special_chars.txt")
    with open(special_chars_dict, 'w') as f:
        for c in "!@#$%^&*()-_=+[]{}|;:,.<>?/":
            f.write(f"{c}\n")
```

```
    # Create liber8 file
    liber8_dict = os.path.join(temp_dir, "liber8.txt")
    with open(liber8_dict, 'w') as f:
        f.write("liber8\n")

    # Generate specific pattern dictionaries for each type
    print("[+] Generating pattern dictionaries...")

    # For Type 1 (All lowercase, hyphen separator)
    type1_patterns = os.path.join(temp_dir, "type1_patterns.txt")
    try:
        with open(lowercase_dict, 'r') as word_file, open(type1_patterns, 'w') as out_file:
            words = [w.strip() for w in word_file.readlines()]
            for word in words[:1000]:  # Limit to first 1000 words for efficient processing
                out_file.write(f"{word}-liber8-\n")
    except Exception as e:
        print(f"[-] Error generating Type 1 patterns: {e}")

    # Generate all types of patterns with special characters
    # For Types 2, 3, and 4
    special_chars = "!@#$%^&*()-_=+[]{}|;:,.<>?/"

    # Type 2 (All lowercase, any special char)
    type2_patterns = os.path.join(temp_dir, "type2_patterns.txt")
    try:
        with open(lowercase_dict, 'r') as word_file, open(type2_patterns, 'w') as out_file:
            words = [w.strip() for w in word_file.readlines()]
            for word in words[:500]:  # Limit to 500 words
                for special_char in special_chars:
                    out_file.write(f"{word}{special_char}liber8{special_char}\n")
    except Exception as e:
        print(f"[-] Error generating Type 2 patterns: {e}")

    # Type 3 lower patterns (lowercase first word, any special char)
    type3_lower_patterns = os.path.join(temp_dir, "type3_lower_patterns.txt")
    try:
        with open(lowercase_dict, 'r') as word_file, open(type3_lower_patterns, 'w') as
out_file:
            words = [w.strip() for w in word_file.readlines()]
```

```
            for word in words[:500]:  # Limit to 500 words
                for special_char in special_chars:
                    out_file.write(f"{word}{special_char}liber8{special_char}\n")
    except Exception as e:
        print(f"[-] Error generating Type 3 lower patterns: {e}")

    # Type 3 upper patterns (uppercase first word, any special char)
    type3_upper_patterns = os.path.join(temp_dir, "type3_upper_patterns.txt")
    try:
        with open(uppercase_dict, 'r') as word_file, open(type3_upper_patterns, 'w') as
out_file:
            words = [w.strip() for w in word_file.readlines()]
            for word in words[:500]:  # Limit to 500 words
                for special_char in special_chars:
                    out_file.write(f"{word}{special_char}liber8{special_char}\n")
    except Exception as e:
        print(f"[-] Error generating Type 3 upper patterns: {e}")

    # Type 4 digit patterns - with digits at end of first word
    type4_first_digit_patterns = os.path.join(temp_dir, "type4_first_digit_patterns.txt")
    try:
        with open(lowercase_dict, 'r') as word_file, open(type4_first_digit_patterns, 'w') as
out_file:
            words = [w.strip() for w in word_file.readlines()]
            for word in words[:300]:  # Limit words
                for digit in range(10):
                    for special_char in special_chars[:5]:  # Limit special chars
                        out_file.write(f"{word}{digit}{special_char}liber8{special_char}\n")
    except Exception as e:
        print(f"[-] Error generating Type 4 first word digit patterns: {e}")

    print("\n[+] Starting hash cracking with Liber8ion Passphrase Standard...")

    # Type 1: word1-liber8-word2 (all lowercase, hyphen separators)
    print("\n[+] Cracking Type 1 passphrases...")
    cmd = [
        "hashcat", "-a0", f"-m{args.hash_type}", args.hash_file,
        type1_patterns, lowercase_dict,
        "--potfile-path", potfile
    ]
```

```
    run_hashcat(cmd, "Trying Type 1 patterns: word1-liber8-word2 (all lowercase)")

    # Type 2: word1<special>liber8<special>word2 (all lowercase)
    print("\n[+] Cracking Type 2 passphrases...")
    cmd = [
        "hashcat", "-a0", f"-m{args.hash_type}", args.hash_file,
        type2_patterns, lowercase_dict,
        "--potfile-path", potfile
    ]
    run_hashcat(cmd, "Trying Type 2 patterns: word1<special>liber8<special>word2 (all
lowercase)")

    # Type 3: Each word all lowercase OR all uppercase
    print("\n[+] Cracking Type 3 passphrases - lowercase first word...")
    cmd = [
        "hashcat", "-a0", f"-m{args.hash_type}", args.hash_file,
        type3_lower_patterns, lowercase_dict,
        "--potfile-path", potfile
    ]
    run_hashcat(cmd, "Trying Type 3 patterns: lower<special>liber8<special>lower")

    cmd = [
        "hashcat", "-a0", f"-m{args.hash_type}", args.hash_file,
        type3_lower_patterns, uppercase_dict,
        "--potfile-path", potfile
    ]
    run_hashcat(cmd, "Trying Type 3 patterns: lower<special>liber8<special>UPPER")

    print("\n[+] Cracking Type 3 passphrases - uppercase first word...")
    cmd = [
        "hashcat", "-a0", f"-m{args.hash_type}", args.hash_file,
        type3_upper_patterns, lowercase_dict,
        "--potfile-path", potfile
    ]
    run_hashcat(cmd, "Trying Type 3 patterns: UPPER<special>liber8<special>lower")

    cmd = [
        "hashcat", "-a0", f"-m{args.hash_type}", args.hash_file,
        type3_upper_patterns, uppercase_dict,
        "--potfile-path", potfile
```

```
]
run_hashcat(cmd, "Trying Type 3 patterns: UPPER<special>liber8<special>UPPER")

# Type 4: One word with digit appended
print("\n[+] Cracking Type 4 passphrases - first word with digit...")
cmd = [
    "hashcat", "-a0", f"-m{args.hash_type}", args.hash_file,
    type4_first_digit_patterns, lowercase_dict,
    "--potfile-path", potfile
]
run_hashcat(cmd, "Trying Type 4 patterns: word1+digit<special>liber8<special>word2")

cmd = [
    "hashcat", "-a0", f"-m{args.hash_type}", args.hash_file,
    type4_first_digit_patterns, uppercase_dict,
    "--potfile-path", potfile
]
run_hashcat(cmd, "Trying Type 4 patterns: word1+digit<special>liber8<special>WORD2")

# Type 4 with second word with digit
# For this, we'll use the Type 3 patterns but with a rule to append a digit
print("\n[+] Cracking Type 4 passphrases - second word with digit...")

# Create a digit append rule file
append_digit_rule = os.path.join(temp_dir, "append_digit.rule")
with open(append_digit_rule, 'w') as f:
    for i in range(10):
        f.write(f"${i}\n")

# For lowercase second word with digit
cmd = [
    "hashcat", "-a0", f"-m{args.hash_type}", args.hash_file,
    type3_lower_patterns, lowercase_dict,
    "-r", append_digit_rule,
    "--potfile-path", potfile
]
run_hashcat(cmd, "Trying Type 4 patterns: word1<special>liber8<special>word2+digit")

# For uppercase second word with digit
cmd = [
```

```
    "hashcat", "-a0", f"-m{args.hash_type}", args.hash_file,
    type3_lower_patterns, uppercase_dict,
    "-r", append_digit_rule,
    "--potfile-path", potfile
]
run_hashcat(cmd, "Trying Type 4 patterns: word1<special>liber8<special>WORD2+digit")

# Same for uppercase first words
cmd = [
    "hashcat", "-a0", f"-m{args.hash_type}", args.hash_file,
    type3_upper_patterns, lowercase_dict,
    "-r", append_digit_rule,
    "--potfile-path", potfile
]
run_hashcat(cmd, "Trying Type 4 patterns: WORD1<special>liber8<special>word2+digit")

cmd = [
    "hashcat", "-a0", f"-m{args.hash_type}", args.hash_file,
    type3_upper_patterns, uppercase_dict,
    "-r", append_digit_rule,
    "--potfile-path", potfile
]
run_hashcat(cmd, "Trying Type 4 patterns: WORD1<special>liber8<special>WORD2+digit")

# Process results to the output file
print(f"\n[+] Processing results to {args.output}...")
with open(potfile, 'r') as pot, open(args.output, 'w') as out:
    for line in pot:
        if ':' in line:
            hash_val, plaintext = line.strip().split(':', 1)
            out.write(f"{hash_val}:{plaintext}\n")

print(f"\n[+] Cracking complete! Results saved to {args.output}")
print(f"[+] To show your cracked passwords: cat {args.output}")

# Ask if user wants to remove temp files
response = input(f"\n[?] Remove temporary files in {temp_dir}? (y/n): ")
if response.lower() == 'y':
    try:
        shutil.rmtree(temp_dir)
```

```
            print(f"[+] Temporary directory {temp_dir} removed")
        except Exception as e:
            print(f"[-] Error removing temporary directory: {e}")
    else:
        print(f"[+] Temporary files kept in {temp_dir}")

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

Scripts

PDF to Hashcat

```
#!/usr/bin/env python

# Copyright (c) 2013 Shane Quigley, < shane at softwareontheside.info >

# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:

# The above copyright notice and this permission notice shall be included in all
# copies or substantial portions of the Software.

# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.

# modified to only output hash for hashcat by philsmd, 2015

import re
import sys
import os
from xml.dom import minidom

PY3 = sys.version_info[0] == 3

class PdfParser:
    def __init__(self, file_name):
        self.file_name = file_name
```

```
        f = open(file_name, 'rb')
        self.encrypted = f.read()
        f.close()
        self.process = True
        psr = re.compile(b'PDF-\d\.\d')
        try:
            self.pdf_spec = psr.findall(self.encrypted)[0]
        except IndexError:
            sys.stderr.write("%s is not a PDF file!\n" % file_name)
            self.process = False

    def parse(self):
        if not self.process:
            return

        try:
            trailer = self.get_trailer()
        except RuntimeError:
            e = sys.exc_info()[1]
            sys.stderr.write("%s : %s\n" % (self.file_name, str(e)))
            return
        # print >> sys.stderr, trailer
        object_id = self.get_object_id(b'Encrypt', trailer)
        # print >> sys.stderr, object_id
        if(len(object_id) == 0):
            raise RuntimeError("Could not find object id")
        encryption_dictionary = self.get_encryption_dictionary(object_id)
        # print >> sys.stderr, encryption_dictionary
        dr = re.compile(b'\d+')
        vr = re.compile(b'\/V \d')
        rr = re.compile(b'\/R \d')
        try:
            v = dr.findall(vr.findall(encryption_dictionary)[0])[0]
        except IndexError:
            raise RuntimeError("Could not find /V")
        r = dr.findall(rr.findall(encryption_dictionary)[0])[0]
        lr = re.compile(b'\/Length \d+')
        longest = 0
        # According to the docs:
        # Length : (Optional; PDF 1.4; only if V is 2 or 3). Default value: 40
```

```
        length = b'40'
        for le in lr.findall(encryption_dictionary):
            if(int(dr.findall(le)[0]) > longest):
                longest = int(dr.findall(le)[0])
                length = dr.findall(le)[0]
        pr = re.compile(b'\/P -?\d+')
        try:
            p = pr.findall(encryption_dictionary)[0]
        except IndexError:
                # print >> sys.stderr, "** dict:", encryption_dictionary
                raise RuntimeError("Could not find /P")
        pr = re.compile(b'-?\d+')
        p = pr.findall(p)[0]
        meta = '1' if self.is_meta_data_encrypted(encryption_dictionary) else '0'
        idr = re.compile(b'\/ID\s*\[\s*<\w+>\s*<\w+>\s*\]')
        try:
            i_d = idr.findall(trailer)[0] # id key word
        except IndexError:
            # some pdf files use () instead of <>
            idr = re.compile(b'\/ID\s*\[\s*\(\w+\)\s*\(\w+\)\s*\]')
            try:
                i_d = idr.findall(trailer)[0] # id key word
            except IndexError:
                # print >> sys.stderr, "** idr:", idr
                # print >> sys.stderr, "** trailer:", trailer
                raise RuntimeError("Could not find /ID tag")
                return
        idr = re.compile(b'<\w+>')
        try:
            i_d = idr.findall(trailer)[0]
        except IndexError:
            idr = re.compile(b'\(\w+\)')
            i_d = idr.findall(trailer)[0]
        i_d = i_d.replace(b'<',b'')
        i_d = i_d.replace(b'>',b'')
        i_d = i_d.lower()
        passwords = self.get_passwords_for_JtR(encryption_dictionary)
        output =
'$pdf$'+v.decode('ascii')+'*'+r.decode('ascii')+'*'+length.decode('ascii')+'*'
        output += p.decode('ascii')+'*'+meta+'*'
```

```
    output += str(int(len(i_d)/2))+'*'+i_d.decode('ascii')+'*'+passwords
    sys.stdout.write("%s\n" % output.encode('UTF-8'))

def get_passwords_for_JtR(self, encryption_dictionary):
    output = ""
    letters = [b"U", b"O"]
    if(b"1.7" in self.pdf_spec):
        letters = [b"U", b"O", b"UE", b"OE"]
    for let in letters:
        pr_str = b'\/' + let + b'\s*\([^)]+\)'
        pr = re.compile(pr_str)
        pas = pr.findall(encryption_dictionary)
        if(len(pas) > 0):
            pas = pr.findall(encryption_dictionary)[0]
            # because regexs in python suck <=== LOL
            while(pas[-2] == b'\\'):
                pr_str += b'[^)]+\)'
                pr = re.compile(pr_str)
                # print >> sys.stderr, "pr_str:", pr_str
                # print >> sys.stderr, encryption_dictionary
                try:
                    pas = pr.findall(encryption_dictionary)[0]
                except IndexError:
                    break
            output += self.get_password_from_byte_string(pas)+"*"
        else:
            pr = re.compile(let + b'\s*<\w+>')
            pas = pr.findall(encryption_dictionary)
            if not pas:
                continue
            pas = pas[0]
            pr = re.compile(b'<\w+>')
            pas = pr.findall(pas)[0]
            pas = pas.replace(b"<",b"")
            pas = pas.replace(b">",b"")
            if PY3:
                output += str(int(len(pas)/2))+'*'+str(pas.lower(),'ascii')+'*'
            else:
                output += str(int(len(pas)/2))+'*'+pas.lower()+'*'
    return output[:-1]
```

```
def is_meta_data_encrypted(self, encryption_dictionary):
    mr = re.compile(b'\/EncryptMetadata\s\w+')
    if(len(mr.findall(encryption_dictionary)) > 0):
        wr = re.compile(b'\w+')
        is_encrypted = wr.findall(mr.findall(encryption_dictionary)[0])[-1]
        if(is_encrypted == b"false"):
            return False
        else:
            return True
    else:
        return True

def parse_meta_data(self, trailer):
    root_object_id = self.get_object_id(b'Root', trailer)
    root_object = self.get_pdf_object(root_object_id)
    object_id = self.get_object_id(b'Metadata', root_object)
    xmp_metadata_object = self.get_pdf_object(object_id)
    return self.get_xmp_values(xmp_metadata_object)

def get_xmp_values(self, xmp_metadata_object):
    xmp_metadata_object = xmp_metadata_object.partition(b"stream")[2]
    xmp_metadata_object = xmp_metadata_object.partition(b"endstream")[0]
    try:
        xml_metadata = minidom.parseString(xmp_metadata_object)
    except:
        return ""
    values = []
    values.append(self.get_dc_value("title", xml_metadata))
    values.append(self.get_dc_value("creator", xml_metadata))
    values.append(self.get_dc_value("description", xml_metadata))
    values.append(self.get_dc_value("subject", xml_metadata))
    created_year = xml_metadata.getElementsByTagName("xmp:CreateDate")
    if(len(created_year) > 0):
        created_year = created_year[0].firstChild.data[0:4]
        values.append(str(created_year))
    return " ".join(values).replace(":", "")

def get_dc_value(self, value, xml_metadata):
    output = xml_metadata.getElementsByTagName("dc:"+value)
```

```
        if(len(output) > 0):
            output = output[0]
            output = output.getElementsByTagName("rdf:li")[0]
            if(output.firstChild):
                output = output.firstChild.data
                return output
        return ""

    def get_encryption_dictionary(self, object_id):
        encryption_dictionary = self.get_pdf_object(object_id)
        for o in encryption_dictionary.split(b"endobj"):
            if(object_id+b" obj" in o):
                encryption_dictionary = o
        return encryption_dictionary

    def get_object_id(self, name , trailer):
        oir = re.compile(b'\/' + name + b'\s\d+\s\d\sR')
        try:
            object_id = oir.findall(trailer)[0]
        except IndexError:
            # print >> sys.stderr, " ** get_object_id: name \"", name, "\", trailer ", trailer
            return ""
        oir = re.compile(b'\d+ \d')
        object_id = oir.findall(object_id)[0]
        return object_id

    def get_pdf_object(self, object_id):
        output = object_id+b" obj" + \
            self.encrypted.partition(b"\r"+object_id+b" obj")[2]
        if(output == object_id+b" obj"):
            output = object_id+b" obj" + \
            self.encrypted.partition(b"\n"+object_id+b" obj")[2]
        output = output.partition(b"endobj")[0] + b"endobj"
        # print >> sys.stderr, output
        return output

    def get_trailer(self):
        trailer = self.get_data_between(b"trailer", b">>", b"/ID")
        if(trailer == b""):
            trailer = self.get_data_between(b"DecodeParms", b"stream", b"")
```

```
        if(trailer == ""):
            raise RuntimeError("Can't find trailer")
    if(trailer != "" and trailer.find(b"Encrypt") == -1):
        # print >> sys.stderr, trailer
        raise RuntimeError("File not encrypted")
    return trailer

def get_data_between(self, s1, s2, tag):
    output = b""
    inside_first = False
    lines = re.split(b'\n|\r', self.encrypted)
    for line in lines:
        inside_first = inside_first or line.find(s1) != -1
        if(inside_first):
            output += line
            if(line.find(s2) != -1):
                if(tag == b"" or output.find(tag) != -1):
                    break
                else:
                    output = b""
                    inside_first = False
    return output

def get_hex_byte(self, o_or_u, i):
    if PY3:
        return hex(o_or_u[i]).replace('0x', '')
    else:
        return hex(ord(o_or_u[i])).replace('0x', '')

def get_password_from_byte_string(self, o_or_u):
    pas = ""
    escape_seq = False
    escapes = 0
    excluded_indexes = [0, 1, 2]
    #For UE & OE in 1.7 spec
    if not PY3:
        if(o_or_u[2] != '('):
            excluded_indexes.append(3)
    else:
        if(o_or_u[2] != 40):
```

```
                excluded_indexes.append(3)
        for i in range(len(o_or_u)):
            if(i not in excluded_indexes):
                if(len(self.get_hex_byte(o_or_u, i)) == 1 \
                   and o_or_u[i] != "\\"[0]):
                    pas += "0"  # need to be 2 digit hex numbers
                is_back_slash = True
                if not PY3:
                    is_back_slash = o_or_u[i] != "\\"[0]
                else:
                    is_back_slash = o_or_u[i] != 92
                if(is_back_slash or escape_seq):
                    if(escape_seq):
                        if not PY3:
                            esc = "\\"+o_or_u[i]
                        else:
                            esc = "\\"+chr(o_or_u[i])
                        esc = self.unescape(esc)
                        if(len(hex(ord(esc[0])).replace('0x', '')) == 1):
                            pas += "0"
                        pas += hex(ord(esc[0])).replace('0x', '')
                        escape_seq = False
                    else:
                        pas += self.get_hex_byte(o_or_u, i)
                else:
                    escape_seq = True
                    escapes += 1
        output = len(o_or_u)-(len(excluded_indexes)+1)-escapes
        return str(output)+'*'+pas[:-2]

    def unescape(self, esc):
        escape_seq_map = {'\\n':"\n", '\\s':"\s", '\\e':"\e",
                '\\r':"\r", '\\t':"\t", '\\v':"\v", '\\f':"\f",
                '\\b':"\b", '\\a':"\a", "\\)":")",
                "\\(":"(", "\\\\":"\\" }

        return escape_seq_map[esc]

if __name__ == "__main__":
    if len(sys.argv) < 2:
```

```
        sys.stderr.write("Usage: %s <PDF file(s)>\n" % \
                         os.path.basename(sys.argv[0]))
        sys.exit(-1)
    for j in range(1, len(sys.argv)):
        if not PY3:
            filename = sys.argv[j].decode('UTF-8')
        else:
            filename = sys.argv[j]
        # sys.stderr.write("Analyzing %s\n" % sys.argv[j].decode('UTF-8'))
        parser = PdfParser(filename)
        try:
            parser.parse()
        except RuntimeError:
            e = sys.exc_info()[1]
            sys.stderr.write("%s : %s\n" % (filename, str(e)))
```

Scripts

PDF to John

```
#!/usr/bin/env python3

# This software is Copyright (c) 2023 Benjamin Dornel <benjamindornel@gmail.com>
# and it is hereby released to the general public under the following terms:
# Redistribution and use in source and binary forms, with or without
# modification, are permitted.



import argparse
import logging

try:
    from pyhanko.pdf_utils.misc import PdfReadError
    from pyhanko.pdf_utils.reader import PdfFileReader
except ImportError:
    print("pyhanko is missing, run 'pip install --user pyhanko==0.20.1' to install it!")
    exit(1)

logger = logging.getLogger(__name__)



class SecurityRevision:
    """Represents Standard Security Handler Revisions
    and the corresponding key length for the /O and /U entries

    In Revision 5, the /O and /U entries were extended to 48 bytes,
    with three logical parts -- a 32 byte verification hash,
    an 8 byte validation salt, and an 8 byte key salt."""

    revisions = {
        2: 32,  # RC4_BASIC
        3: 32,  # RC4_EXTENDED
        4: 32,  # RC4_OR_AES128
        5: 48,  # AES_R5_256
```

```
        6: 48,  # AES_256
    }

    @classmethod
    def get_key_length(cls, revision):
        """
        Get the key length for a given revision,
        defaults to 48 if no revision is specified.
        """
        return cls.revisions.get(revision, 48)


class PdfHashExtractor:
    """
    Extracts hash and encryption information from a PDF file

    Attributes:
    - `file_name`: PDF file path.
    - `strict`: Boolean that controls whether an error is raised, if a PDF
        has problems e.g. Multiple definitions in encryption dictionary
        for a specific key. Defaults to `False`.
    - `algorithm`: Encryption algorithm used by the standard security handler
    - `length`: The length of the encryption key, in bits. Defaults to 40.
    - `permissions`: User access permissions
    - `revision`: Revision of the standard security handler
    """

    def __init__(self, file_name: str, strict: bool = False):
        self.file_name = file_name

        with open(file_name, "rb") as doc:
            self.pdf = PdfFileReader(doc, strict=strict)
            self.encrypt_dict = self.pdf._get_encryption_params()

            if not self.encrypt_dict:
                raise RuntimeError("File not encrypted")

            self.algorithm: int = self.encrypt_dict.get("/V")
            self.length: int = self.encrypt_dict.get("/Length", 40)
            self.permissions: int = self.encrypt_dict["/P"]
```

```
        self.revision: int = self.encrypt_dict["/R"]

    @property
    def document_id(self) -> bytes:
        return self.pdf.document_id[0]

    @property
    def encrypt_metadata(self) -> str:
        """
        Get a string representation of whether metadata is encrypted.

        Returns "1" if metadata is encrypted, "0" otherwise.
        """
        return str(int(self.pdf.security_handler.encrypt_metadata))

    def parse(self) -> str:
        """
        Parse PDF encryption information into a formatted string for John
        """
        passwords = self.get_passwords()
        fields = [
            f"$pdf${self.algorithm}",
            self.revision,
            self.length,
            self.permissions,
            self.encrypt_metadata,
            len(self.document_id),
            self.document_id.hex(),
            passwords,
        ]
        return "*".join(map(str, fields))

    def get_passwords(self) -> str:
        """
        Creates a string consisting of the hexidecimal string of the
        /U, /O, /UE and /OE entries and their corresponding byte string length
        """
        passwords = []
        keys = ("udata", "odata", "oeseed", "ueseed")
        max_key_length = SecurityRevision.get_key_length(self.revision)
```

```
        for key in keys:
            if data := getattr(self.pdf.security_handler, key):
                data: bytes = data[:max_key_length]
                passwords.extend([str(len(data)), data.hex()])

        return "*".join(passwords)


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="PDF Hash Extractor")
    parser.add_argument(
        "pdf_files", nargs="+", help="PDF file(s) to extract information from"
    )
    parser.add_argument(
        "-d", "--debug", action="store_true", help="Print the encryption dictionary"
    )
    args = parser.parse_args()

    for filename in args.pdf_files:
        try:
            extractor = PdfHashExtractor(filename)
            pdf_hash = extractor.parse()
            print(pdf_hash)

            if args.debug:
                if extractor.encrypt_dict:
                    print("Encryption Dictionary:")
                    for key, value in extractor.encrypt_dict.items():
                        print(f"{key}: {value}")
                else:
                    print("No encryption dictionary found in the PDF.")

        except PdfReadError as error:
            logger.error("%s : %s", filename, error, exc_info=True)
```

Scripts

Steg

```
#!/usr/bin/env python3
"""
Steganography Extraction Tool

This script extracts hidden data from images using various steganography techniques.
Usage: python steg_extract.py <image_file>
"""

import sys
import os
import numpy as np
from PIL import Image
import binascii
import re
import zlib
import struct
from bitstring import BitArray

def extract_lsb(image_path, bit_depth=1):
    """Extract data hidden using LSB (Least Significant Bit) steganography."""
    try:
        img = Image.open(image_path)
        pixels = np.array(img)

        # Flatten the pixel array and extract LSBs
        flat_pixels = pixels.flatten()

        # Get the least significant bits
        bits = ""
        for pixel in flat_pixels:
            # Extract the specified number of least significant bits
            for i in range(bit_depth):
                bits += str((pixel >> i) & 1)
```

```
        # Convert bits to bytes
        bytes_data = BitArray(bin=bits).bytes

        # Try to find printable text
        printable_data = ""
        for i in range(len(bytes_data)):
            char = bytes_data[i:i+1]
            if 32 <= ord(char) <= 126 or ord(char) in (10, 13, 9):  # Printable ASCII or
newline/tab
                printable_data += char.decode('ascii', errors='ignore')
            else:
                printable_data += '.'

        return {
            'raw_bits': bits[:100] + "...",  # First 100 bits
            'raw_bytes': binascii.hexlify(bytes_data[:50]).decode('ascii') + "...",  # First
50 bytes
            'possible_text': printable_data[:1000]  # First 1000 printable chars
        }
    except Exception as e:
        return {'error': f"LSB extraction failed: {str(e)}"}

def extract_metadata(image_path):
    """Extract metadata from the image that might contain hidden information."""
    try:
        img = Image.open(image_path)
        metadata = {}

        # Extract EXIF data if available
        if hasattr(img, '_getexif') and img._getexif():
            metadata['exif'] = str(img._getexif())

        # Extract other metadata
        metadata['format'] = img.format
        metadata['mode'] = img.mode
        metadata['info'] = str(img.info)

        return metadata
    except Exception as e:
        return {'error': f"Metadata extraction failed: {str(e)}"}
```

```
def extract_color_plane(image_path):
    """Extract data from color planes separately to find potential hidden information."""
    try:
        img = Image.open(image_path)
        if img.mode != 'RGB' and img.mode != 'RGBA':
            return {'error': "Not an RGB/RGBA image"}

        planes = {}
        pixels = np.array(img)

        # Extract red, green, blue planes
        if img.mode == 'RGB' or img.mode == 'RGBA':
            planes['red'] = pixels[:,:,0]
            planes['green'] = pixels[:,:,1]
            planes['blue'] = pixels[:,:,2]

        # Check for unusual patterns in each plane
        results = {}
        for plane_name, plane_data in planes.items():
            # Look for unusual distributions (e.g., even/odd patterns)
            even_count = np.sum(plane_data % 2 == 0)
            odd_count = np.sum(plane_data % 2 == 1)

            # If there's a significant imbalance, it might indicate steganography
            results[f"{plane_name}_analysis"] = {
                'even_pixels': even_count,
                'odd_pixels': odd_count,
                'imbalance': abs(even_count - odd_count) / (even_count + odd_count)
            }

            # Extract LSB from this color plane only
            bits = "".join([str(p & 1) for p in plane_data.flatten()])
            results[f"{plane_name}_lsb_sample"] = bits[:100] + "..."

        return results
    except Exception as e:
        return {'error': f"Color plane extraction failed: {str(e)}"}

def extract_hidden_files(image_path):
```

```
    """Look for embedded files using common signatures/headers."""
    try:
        with open(image_path, 'rb') as f:
            data = f.read()

        # Common file signatures to look for
        file_signatures = {
            b'\x50\x4B\x03\x04': 'ZIP',
            b'\x52\x61\x72\x21\x1A\x07': 'RAR',
            b'\x25\x50\x44\x46': 'PDF',
            b'\xFF\xD8\xFF': 'JPG',
            b'\x89\x50\x4E\x47': 'PNG',
            b'\x47\x49\x46\x38': 'GIF',
            b'\x7F\x45\x4C\x46': 'ELF',
            b'\xD0\xCF\x11\xE0': 'MS Office',
            b'\x50\x4B\x05\x06': 'ZIP (empty)',
            b'\x1F\x8B\x08': 'GZIP',
            b'\x42\x5A\x68': 'BZ2',
            b'\x75\x73\x74\x61\x72': 'TAR',
            b'\x49\x44\x33': 'MP3',
            b'\x4D\x5A': 'EXE',
        }

        found_files = []
        for signature, filetype in file_signatures.items():
            # Find all occurrences of the signature
            offsets = [m.start() for m in re.finditer(re.escape(signature), data)]
            for offset in offsets:
                found_files.append({
                    'type': filetype,
                    'offset': offset,
                    'signature': binascii.hexlify(signature).decode('ascii')
                })

        return found_files
    except Exception as e:
        return {'error': f"Hidden file extraction failed: {str(e)}"}

def extract_parity_steganography(image_path):
    """Check for parity-based steganography."""
```

```
    try:
        img = Image.open(image_path)
        pixels = np.array(img)

        # Count the parity of pixels in each row and column
        row_parity = np.sum(pixels.sum(axis=2) % 2, axis=1) % 2
        col_parity = np.sum(pixels.sum(axis=2) % 2, axis=0) % 2

        # Convert to binary strings (potentially hidden messages)
        row_message = "".join([str(int(bit)) for bit in row_parity])
        col_message = "".join([str(int(bit)) for bit in col_parity])

        return {
            'row_parity_bits': row_message,
            'col_parity_bits': col_message
        }
    except Exception as e:
        return {'error': f"Parity steganography extraction failed: {str(e)}"}

def extract_hidden_text(image_path):
    """Extract text from the image using several methods."""
    try:
        with open(image_path, 'rb') as f:
            data = f.read()

        # Look for ASCII/UTF-8 text patterns
        possible_strings = []
        ascii_regex = rb'[ -~\r\n\t]{8,}'  # 8+ printable ASCII chars
        for match in re.finditer(ascii_regex, data):
            possible_strings.append(match.group(0).decode('ascii', errors='ignore'))

        return {
            'possible_strings': possible_strings[:20]  # Return first 20 found strings
        }
    except Exception as e:
        return {'error': f"Text extraction failed: {str(e)}"}

def analyze_bit_distribution(image_path):
    """Analyze bit distribution for statistical anomalies."""
    try:
```

```
        img = Image.open(image_path)
        pixels = np.array(img)

        # Analyze distribution of each bit position
        bit_counts = []
        for bit_pos in range(8):
            mask = 1 << bit_pos
            bit_count = np.sum((pixels & mask) > 0)
            bit_counts.append(bit_count)

        total_bits = pixels.size * 8
        bit_frequencies = [count / total_bits for count in bit_counts]

        # Calculate deviation from expected 0.5 frequency
        deviations = [abs(freq - 0.5) for freq in bit_frequencies]

        return {
            'bit_frequencies': bit_frequencies,
            'deviations': deviations,
            'analysis': "High deviation in LSBs may indicate steganography"
        }
    except Exception as e:
        return {'error': f"Bit distribution analysis failed: {str(e)}"}

def extract_stegano_data(image_path):
    """Main function to extract steganographic data from an image."""
    results = {
        'filename': os.path.basename(image_path),
        'filesize': os.path.getsize(image_path)
    }

    # Run all extraction methods
    results['lsb_extraction'] = extract_lsb(image_path)
    results['lsb_extraction_2bit'] = extract_lsb(image_path, bit_depth=2)
    results['metadata'] = extract_metadata(image_path)
    results['color_planes'] = extract_color_plane(image_path)
    results['hidden_files'] = extract_hidden_files(image_path)
    results['parity_data'] = extract_parity_steganography(image_path)
    results['text_data'] = extract_hidden_text(image_path)
    results['bit_distribution'] = analyze_bit_distribution(image_path)
```

```
    return results

def save_extracted_data(results, original_image_path):
    """Save extracted data to files."""
    base_name = os.path.splitext(os.path.basename(original_image_path))[0]
    output_dir = f"{base_name}_extracted"

    # Create directory if it doesn't exist
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    # Save main results as text
    with open(f"{output_dir}/results.txt", 'w') as f:
        f.write(f"Steganography Extraction Results for {results['filename']}\n")
        f.write(f"File size: {results['filesize']} bytes\n\n")

        # Write metadata
        f.write("=== METADATA ===\n")
        for k, v in results['metadata'].items():
            f.write(f"{k}: {v}\n")

        # Write LSB extraction results
        f.write("\n=== LSB EXTRACTION ===\n")
        f.write(f"LSB sample: {results['lsb_extraction']['raw_bits']}\n")
        f.write(f"Bytes: {results['lsb_extraction']['raw_bytes']}\n")
        f.write(f"Possible text:\n{results['lsb_extraction']['possible_text']}\n")

        # Write 2-bit LSB extraction
        f.write("\n=== 2-BIT LSB EXTRACTION ===\n")
        f.write(f"LSB sample: {results['lsb_extraction_2bit']['raw_bits']}\n")
        f.write(f"Bytes: {results['lsb_extraction_2bit']['raw_bytes']}\n")
        f.write(f"Possible text:\n{results['lsb_extraction_2bit']['possible_text']}\n")

        # Write color plane analysis
        f.write("\n=== COLOR PLANE ANALYSIS ===\n")
        for k, v in results['color_planes'].items():
            f.write(f"{k}: {v}\n")

        # Write hidden files
```

```
        f.write("\n=== POSSIBLE HIDDEN FILES ===\n")
        for file_info in results['hidden_files']:
            f.write(f"Type: {file_info['type']}, Offset: {file_info['offset']}, Signature:
{file_info['signature']}\n")

        # Write parity data
        f.write("\n=== PARITY STEGANOGRAPHY ===\n")
        f.write(f"Row parity: {results['parity_data']['row_parity_bits']}\n")
        f.write(f"Column parity: {results['parity_data']['col_parity_bits']}\n")

        # Write found text strings
        f.write("\n=== POSSIBLE HIDDEN TEXT ===\n")
        for s in results['text_data']['possible_strings']:
            f.write(f"{s}\n")
            f.write("---\n")

        # Write bit distribution analysis
        f.write("\n=== BIT DISTRIBUTION ANALYSIS ===\n")
        f.write("Bit position frequencies (0-7, LSB to MSB):\n")
        for i, freq in enumerate(results['bit_distribution']['bit_frequencies']):
            f.write(f"Bit {i}: {freq:.4f} (deviation:
{results['bit_distribution']['deviations'][i]:.4f})\n")

    # If we found potential embedded files, try to extract them
    if results['hidden_files']:
        with open(original_image_path, 'rb') as f:
            data = f.read()

        for i, file_info in enumerate(results['hidden_files']):
            # Create a name for the extracted file
            ext = file_info['type'].lower().split()[0]  # Use the first word of the type as
extension
            output_file = f"{output_dir}/extracted_file_{i}.{ext}"

            # Get start position from offset
            start_pos = file_info['offset']

            # Write the data to a file, up to 10MB maximum
            with open(output_file, 'wb') as out_f:
                out_f.write(data[start_pos:start_pos + 10*1024*1024])
```

```
    return output_dir

def main():
    if len(sys.argv) != 2:
        print(f"Usage: {sys.argv[0]} <image_file>")
        sys.exit(1)

    image_path = sys.argv[1]
    if not os.path.exists(image_path):
        print(f"Error: File '{image_path}' not found.")
        sys.exit(1)

    print(f"Analyzing {image_path} for steganographic data...")
    results = extract_stegano_data(image_path)

    # Save results to files
    output_dir = save_extracted_data(results, image_path)
    print(f"Analysis complete. Results saved to {output_dir}/")

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

Scripts

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()
```