25 most popular Python scripts for network security

KLEMENS NGUYEN

Preface

Creating Python scripts for network security and cybersecurity involves a wide array of tasks, from scanning networks to analyzing packets, from testing vulnerabilities to automating security checks. Below are 25 common Python script ideas that can serve as a starting point for various cybersecurity tasks. For each, I'll provide a brief description of what the script aims to achieve. Note that actual implementation would require a deeper understanding of Python programming, networking concepts, and cybersecurity principles.

Remember that, implementing these scripts requires a responsible approach, respecting legal boundaries and ethical standards. Many of the tasks, especially those involving penetration testing or vulnerability scanning, should only be performed on networks or systems where you have explicit permission to do so. Additionally, staying updated with the latest in cybersecurity practices and Python libraries (such as Scapy for packet manipulation, Requests for web requests, or BeautifulSoup for web scraping) will greatly enhance the effectiveness and efficiency of your scripts.

API Security Tester

Here is a basic Python script designed to test APIs for common security vulnerabilities such as SQL Injection, Cross-Site Scripting (XSS), and misconfigurations. This script will serve as an educational tool to understand how to approach testing APIs for vulnerabilities. It's crucial to remember that this script should only be used on APIs you have explicit permission to test, to adhere to ethical guidelines and legal boundaries.

➢ Prerequisites

To get the most out of this script, you should have:

- Python installed on your system.
- Basic understanding of HTTP requests and responses.
- Familiarity with common security vulnerabilities.
- `requests` library installed in Python for making HTTP requests (`pip install requests`).

➢ The script overview

The script will perform basic tests against a specified API endpoint. We'll check for:

- **SQL injection vulnerability**: by sending malicious SQL code in parameters and analyzing the response.
- **XSS vulnerability**: by injecting a script tag and checking if it's executed or reflected in the response.
- **Misconfiguration**: by checking for overly informative error messages or misconfigured headers.

➢ Python code

```
import requests

# Function to test for SQL Injection
def test_sql_injection(url):
    # Payload that might cause SQL error if improperly handled
    payload = "' OR '1'='1"
```

```
    params = {'input': payload}
    response = requests.get(url, params=params)

    if "SQL syntax" in response.text:
        print("Potential SQL Injection vulnerability found")
    else:
        print("No obvious SQL Injection vulnerability detected")

# Function to test for XSS
def test_xss(url):
    # Simple script tag payload
    payload = "<script>alert('XSS')</script>"
    params = {'input': payload}
    response = requests.get(url, params=params)

    if payload in response.text:
        print("Potential XSS vulnerability found")
    else:
        print("No obvious XSS vulnerability detected")

# Function to test for Misconfiguration
def test_misconfiguration(url):
    response = requests.get(url)

    # Check for insecure headers or too much information in error messages
    if 'X-Powered-By' in response.headers or 'Server' in response.headers:
        print("Potential misconfiguration found (informative headers)")
    else:
        print("No obvious misconfiguration detected")

# Main function to run our tests
def main():
    url = input("Enter the URL of the API endpoint to test: ")
```

```
    print("\nTesting for SQL Injection...")
    test_sql_injection(url)

    print("\nTesting for XSS...")
    test_xss(url)

    print("\nTesting for Misconfiguration...")
    test_misconfiguration(url)

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

➢ Explanation

- **Imports**: we use the `requests` library to make HTTP requests to the API.
- **SQL injection test**: we send a payload that could exploit SQL injection if the input is not properly sanitized by the backend. An error message containing "SQL syntax" might indicate a vulnerability.
- **XSS test**: we attempt to inject a basic `<script>` tag. If the payload appears unencoded in the response, it might be vulnerable to XSS.
- **Misconfiguration test**: we look for common misconfiguration signs, like revealing too much information in headers. These can provide attackers with useful hints to exploit other vulnerabilities.

➢ Usage and limitations

This script provides a basic starting point. Real-world applications require more comprehensive testing with a variety of payloads and techniques. Always ensure you have permission to test the target API. This script is for educational purposes. For more thorough testing, consider using professional tools and methodologies.

➢ Conclusion

This script demonstrates fundamental concepts in testing for some common web vulnerabilities. It's important to approach cybersecurity with an ethical mindset and continually update your skills to protect applications against evolving threats.

Automated phishing detector

Creating an „Automated phishing detector” in Python involves analyzing text for common indicators of phishing, such as suspicious URLs, the use of scare tactics, or impersonation attempts. This script will be a basic demonstration of how to analyze strings (which could be extracted from emails or web content) for phishing indicators. The script is educational and designed to be accessible to Python users of all levels. For a more comprehensive solution, integration with email APIs, web scraping tools, or more advanced natural language processing (NLP) techniques might be required.

➢ Prerequisites

- Python installed on your system.
- Basic understanding of Python programming.
- Familiarity with phishing tactics and indicators.
- `requests` library for making HTTP requests (`pip install requests`).

➢ Script overview

The script will:

- Analyze URLs to detect malicious or spoofed domains.
- Search for common phishing keywords within the text.
- Check if the text includes scare tactics or urgent requests for personal information.

➢ Python code

```
import re
import requests

# Function to analyze URLs for potentially malicious domains
def check_urls(text):
    # Regular expression to find URLs in the text
    urls = re.findall(r'(https?://\S+)', text)
```

```
    for url in urls:
        # Simple heuristic: Check if the URL uses a shortening service
        # More sophisticated checks could involve domain reputation APIs
        if "bit.ly" in url or "tinyurl.com" in url:
            print(f"Potential phishing URL detected: {url}")

# Function to search for phishing keywords
def check_keywords(text):
    # List of common phishing keywords
    keywords = ['confirm', 'account', 'urgent', 'verify', 'update', 'password']
    for keyword in keywords:
        if keyword in text.lower():
            print(f"Phishing keyword detected: {keyword}")

# Function to check for scare tactics
def check_scare_tactics(text):
    scare_tactics_phrases = [
        'immediately', 'as soon as possible', 'at risk', 'secure your account',
'suspicious activity'
    ]
    for phrase in scare_tactics_phrases:
        if phrase in text.lower():
            print(f"Potential use of scare tactics detected: {phrase}")

# Main function to run our phishing detector
def phishing_detector(text):
    print("Analyzing text for phishing indicators...")
    check_urls(text)
    check_keywords(text)
    check_scare_tactics(text)

# Example usage
```

```
if __name__ == "__main__":
    example_text = """
    Dear user, we've detected suspicious activity on your account. Please
verify your account immediately by visiting this link: https://bit.ly/fake-url
    """
    phishing_detector(example_text)
```

➢ Explanation

- **URL analysis**: the script uses a regular expression to find URLs within the text and checks if they use known URL shortening services, which can be a red flag for phishing attempts.
- **Keyword search**: common keywords associated with phishing emails, like "verify" or "urgent," are checked in the text.
- **Scare tactics detection**: phrases that aim to induce panic or urgency are identified, as they are often used in phishing to compel the recipient to act hastily.

➢ Limitations and considerations

- This script provides a basic level of detection based on simple heuristics. Real-world phishing attempts may use more sophisticated techniques.
- False positives and negatives are possible. Further refinement and more sophisticated analysis techniques would be needed for a production-level tool.
- Always respect privacy and legal considerations when analyzing email content or web pages.

➢ Conclusion

This script is a starting point for understanding how automated phishing detection can work. Developing a comprehensive solution requires a deeper dive into security practices, possibly incorporating machine learning models to improve accuracy.

Botnet detection system

Creating a botnet detection system that monitors network traffic to identify patterns indicative of botnet activity involves analyzing network packets for anomalies or patterns often associated with botnets, such as unusual volumes of outgoing requests, specific types of packet payloads, or communication with known malicious IP addresses. However, implementing a full-fledged system in a simple script and making it accessible to all levels of Python users is challenging due to the complexity of network traffic analysis and the need for access to network interfaces and possibly privileged system operations.

Nonetheless, I can guide you through a basic Python script using the `scapy` library, which is a powerful packet manipulation tool. This script will demonstrate how to capture and analyze packets in real-time for potential botnet activity indicators. This demonstration aims to provide a foundational understanding, from which you can explore more advanced topics.

➢ Prerequisites

- Python installed on your system.
- The `scapy` library installed (`pip install scapy`).
- Basic understanding of network protocols and Python programming.
- Permissions to capture network packets (might require root/administrator privileges).

➢ Script overview

The script will:

- Capture network packets in real-time.
- Analyze packets for simple indicators of botnet activity (e.g., high frequency of requests to known bad IPs, specific packet patterns).
- Log potential botnet activity for further investigation.

➢ Python Code

```
from scapy.all import sniff
from collections import Counter

# Known malicious IP addresses (for demonstration purposes; in a real scenario, this list should be dynamically updated)
MALICIOUS_IPS = ['123.45.67.89', '98.76.54.32']
# Counter to keep track of requests to potentially malicious IPs
ip_counter = Counter()

def packet_callback(packet):
    try:
        # For simplicity, we're only looking at IP layer
        if packet.haslayer('IP'):
            src_ip = packet['IP'].src
            dst_ip = packet['IP'].dst

            # Check if the destination IP is known to be malicious
            if dst_ip in MALICIOUS_IPS:
                ip_counter[dst_ip] += 1
                print(f"Detected packet to known malicious IP: {dst_ip}")

            # Example of a simple anomaly detection: too many requests to a single IP
            for ip, count in ip_counter.items():
                if count > 100:  # Threshold for alerting
                    print(f"Potential botnet activity: High volume of requests to {ip}")
    except Exception as e:
        print(f"Error processing packet: {e}")

# Start packet sniffing
def start_sniffing():
```

```
    print("Starting network packet sniffing...")
    sniff(prn=packet_callback, store=False)  # Using store=False to
minimize memory usage

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

➢ Explanation

- **Scapy sniffing**: we use `scapy`'s `sniff` function to capture packets in real-time, passing each captured packet to `packet_callback` for analysis.
- **Packet analysis**: the callback function checks if the packet is an IP packet, then examines the destination IP against a list of known malicious IPs, incrementing a counter for each observation.
- **Anomaly detection**: if the number of packets sent to any IP exceeds a predefined threshold, it's logged as potential botnet activity, suggesting an infected host might be part of a botnet.

➢ Limitations and considerations

- This script is a basic demonstration: real-world botnet detection requires analyzing a broader range of packet types, payloads, and behaviors, potentially incorporating machine learning for anomaly detection.
- Performance and scalability: processing every packet in Python can be CPU-intensive. For production systems, consider optimized or native solutions.
- Legal and ethical considerations: ensure you have the right to monitor and analyze the network traffic in question.

➢ Conclusion

This script introduces the basics of using Python for network traffic analysis to detect potential botnet activity. Expanding this into a comprehensive botnet detection system would involve deeper network analysis, dynamic threat intelligence, and perhaps integration with existing security infrastructure.

Credential leak monitor

Creating a credential leak monitor in Python to monitor public leaks and breaches for compromised credentials related to your domain or organization involves querying databases or APIs that track such leaks and breaches. A popular and accessible API for this purpose is "Have I Been Pwned" (HIBP), which allows you to check if an email or domain is part of any data breaches.

This script will demonstrate how to use the HIBP API to monitor for compromised credentials associated with a specific domain. It's important to note that using APIs such as HIBP responsibly and respecting privacy and ethical considerations are paramount. Always review and adhere to the API's terms of use.

➢ Prerequisites

- Python installed on your system.
- An API key for Have I Been Pwned (as of my last update, HIBP requires an API key for some of its services).
- The `requests` library installed (`pip install requests`).

➢ Script overview

The script will:

- Query the HIBP API for breach data related to a specific domain.
- Parse the response to identify specific breach instances.
- Print out details of each breach for further investigation.

➢ Python code

```
import requests

# Replace 'your_hibp_api_key' with your actual Have I Been Pwned API
key
HIBP_API_KEY = 'your_hibp_api_key'
# Replace 'example.com' with the domain you want to monitor
```

```
MONITORED_DOMAIN = 'example.com'

def check_domain_breaches(domain):
    url = f"https://haveibeenpwned.com/api/v3/breaches?domain={domain}"
    headers = {
        'hibp-api-key': HIBP_API_KEY,
        'User-Agent': 'Credential Leak Monitor Script'
    }
    response = requests.get(url, headers=headers)
    if response.status_code == 200:
        breaches = response.json()
        if breaches:
            print(f"Breaches found for domain {domain}:")
            for breach in breaches:
                print(f"- {breach['Name']}: {breach['Description']}")
        else:
            print(f"No breaches found for domain {domain}.")
    else:
        print("Error querying Have I Been Pwned API.")

if __name__ == "__main__":
    check_domain_breaches(MONITORED_DOMAIN)
```

➢ Explanation

- **API key and domain**: the script requires an API key from HIBP and the domain you wish to monitor for breaches.
- **Request to HIBP**: we make a GET request to the HIBP API's breach endpoint, passing the domain as a query parameter.
- **Parsing and reporting**: the script parses the JSON response from the API, listing each breach's name and description. This information can help in understanding the nature and scope of each breach.

➢ Limitations and considerations

- **API key and rate limits**: HIBP API keys are subject to rate limiting and other restrictions. Ensure your usage complies with HIBP's policies.
- **Privacy and sensitivity**: handling data related to breaches requires sensitivity and adherence to data protection regulations (e.g., GDPR, CCPA).
- **Comprehensive monitoring**: this script focuses on domain-related breaches. Comprehensive monitoring might also involve checking specific email addresses or other identifiers, depending on your needs and HIBP's capabilities.

➢ Conclusion

This script is a starting point for monitoring public leaks and breaches for compromised credentials associated with your domain. Expanding its capabilities and integrating it into your security operations can help in early detection of potential exposures, enhancing your organization's cybersecurity posture.

DDoS simulation tool

Creating a Distributed Denial of Service (DDoS) simulation tool is a delicate subject, as it involves generating high volumes of network traffic to test the resilience of networks or applications. It's crucial to stress that such tools must only be used in ethical, legal, and controlled environments. This means you should only target systems you own or have explicit permission to test, and you should always consider the impact of your testing on the network and the services.

Given these considerations, I'll guide you through creating a basic script that can generate a significant amount of HTTP requests to a target. This script is intended for educational purposes to understand the concept of generating high volumes of traffic and should not be used for any unauthorized testing.

➢ Prerequisites

- Python installed on your system.
- Understanding of network protocols and Python programming.
- The `requests` library for making HTTP requests (`pip install requests`).
- The `threading` library for concurrency, which is part of Python's standard library.

➢ Script overview

The script will:

- Perform HTTP GET requests to a specified URL.
- Use threading to simulate multiple sources of traffic.
- Be configurable to adjust the number of threads and requests.

➢ Python code

```
import requests
import threading
```

```
# Configuration: Target URL, Number of Threads, and Requests per Thread
TARGET_URL = 'http://example.com'
THREAD_COUNT = 10
REQUESTS_PER_THREAD = 100

# Function to perform a number of HTTP GET requests to the target URL
def send_requests():
    for _ in range(REQUESTS_PER_THREAD):
        try:
            response = requests.get(TARGET_URL)
            # Print status code to monitor responses (optional)
            print(f"Request sent, status code: {response.status_code}")
        except requests.exceptions.RequestException as e:
            print(f"Request failed: {e}")

# Function to start the threads that will send requests
def start_attack():
    threads = []
    for _ in range(THREAD_COUNT):
        thread = threading.Thread(target=send_requests)
        thread.start()
        threads.append(thread)

    for thread in threads:
        thread.join()

if __name__ == "__main__":
    print("Starting DDoS Simulation...")
    start_attack()
    print("DDoS Simulation Completed.")
```

➢ Explanation

- **Configuration**: the `TARGET_URL`, `THREAD_COUNT`, and `REQUESTS_PER_THREAD` variables allow you to configure the target and scale of the simulation.
- **`send_requests` function**: this function sends a configured number of HTTP GET requests to the target URL, simulating a single source of traffic.
- **`start_attack` function**: initializes and starts multiple threads (based on `THREAD_COUNT`) to simulate traffic from multiple sources. It waits for all threads to complete with `thread.join()`.

➢ Ethical and legal considerations

- **Authorization**: you must have explicit permission to test the target system.
- **Impact assessment**: understand the potential impact of the simulation on the target system and the surrounding network infrastructure.
- **Legality**: ensure that your actions are in compliance with local laws and regulations.

➢ Conclusion

This script provides a basic framework for simulating high volumes of HTTP requests to a target URL, which can be part of a DDoS resilience testing process. Expanding its functionality and integrating it into a comprehensive testing strategy requires careful planning, ethical consideration, and legal compliance.

DNS query tool

Creating a DNS query tool in Python is a great way to understand how domain names are resolved into IP addresses and to gather various pieces of information about domains. This tool can be useful for a wide range of users, from beginners to advanced programmers, by demonstrating the use of Python's socket and dns.resolver libraries. For this example, I'll focus on using the `dnspython` library, which provides a more extensive interface for DNS queries than the standard library's socket module.

➢ Prerequisites

- Python installed on your system.
- The `dnspython` library installed. You can install it using pip:

```
pip install dnspython
```

➢ Script overview

The script will:

- Perform a basic DNS lookup (a record) to resolve the IP address of a given domain name.
- Extend functionality to query different types of DNS records (e.g., MX, NS, TXT) for more comprehensive domain information.
- Provide clear output and error handling.

➢ Python code

```
import dns.resolver

# Function to perform DNS query
def dns_query(domain, record_type='A'):
    """
    Perform a DNS query to get specific record types for a given domain.

    :param domain: The domain name to query.
```

```
    :param record_type: The type of DNS record to retrieve (e.g., 'A', 'MX',
'NS').
    """
    try:
        # Perform the DNS query
        answers = dns.resolver.resolve(domain, record_type)
        print(f"DNS {record_type} records for {domain}:")
        for answer in answers:
            print(answer.to_text())
    except dns.resolver.NoAnswer:
        print(f"No {record_type} record found for {domain}.")
    except dns.resolver.NXDOMAIN:
        print(f"Domain {domain} does not exist.")
    except Exception as e:
        print(f"An error occurred: {e}")

if __name__ == "__main__":
    # Example usage
    domain_to_query = 'example.com'
    dns_query(domain_to_query, 'A')  # A record lookup
    dns_query(domain_to_query, 'MX') # MX record lookup
    dns_query(domain_to_query, 'NS') # NS record lookup
```

➢ Explanation

- **`dns_query` function**: this is the core function where the DNS query happens. It takes a domain name and the record type as parameters. The function uses the `dns.resolver.resolve` method to fetch the requested DNS record type for the domain.
- **record types**: the script can be easily adjusted to query different types of DNS records by changing the `record_type` parameter. Common types include 'A' (address record), 'MX'

(mail exchange record), 'NS' (name server record), and 'TXT' (text record).

- **error handling**: the script includes basic error handling to catch common issues like non-existent domains (`NXDOMAIN`), missing record types (`NoAnswer`), and other exceptions.

➢ Use cases

This DNS Query Tool can be used for a variety of purposes, including:

- Checking the IP address associated with a domain.
- Finding out the mail servers (MX records) of a domain.
- Identifying the name servers (NS records) managing a domain's DNS settings.
- Retrieving domain verification records (TXT records), often used for SPF, DKIM, and DMARC.

➢ Conclusion

This Python script provides a simple yet powerful way to perform DNS queries for different types of records. It can be a valuable tool for network administrators, security analysts, and anyone interested in domain information gathering. The `dnspython` library's extensive functionality makes it possible to extend this script further to suit more specialized needs or integrate it into larger projects.

Encryption/decryption tool

Creating an Encryption/decryption tool in Python that supports various algorithms requires a good understanding of cryptography. For this demonstration, we'll use the `cryptography` library, which provides a robust set of cryptographic functions for encryption and decryption. This example will focus on symmetric encryption using the Fernet module, which implements password-based encryption using the best practices. This choice makes the tool accessible to users of all experience levels while maintaining strong security standards.

➢ Prerequisites

- Python installed on your system.
- The `cryptography` library installed. You can install it using pip:

```
pip install cryptography
```

➢ Script overview

The script will:

- Generate a Fernet key for encryption and decryption.
- Encrypt a file with the generated key.
- Decrypt a file using the same key.
- Ensure the tool is user-friendly and includes error handling.

➢ Python code

```
from cryptography.fernet import Fernet
import os

# Function to generate a Fernet key
def generate_key():
    """
    Generates a Fernet key and saves it to a file.
    """
```

```
    key = Fernet.generate_key()
    with open('fernet_key.key', 'wb') as key_file:
        key_file.write(key)
    print("Key generated and saved to fernet_key.key")

# Function to load the Fernet key
def load_key():
    """
    Loads the Fernet key from a file.
    """
    return open('fernet_key.key', 'rb').read()

# Function to encrypt a file
def encrypt_file(file_name, key):
    """
    Encrypts a file using the provided Fernet key.

    :param file_name: The name of the file to encrypt.
    :param key: The Fernet key for encryption.
    """
    f = Fernet(key)
    with open(file_name, 'rb') as file:
        file_data = file.read()
    encrypted_data = f.encrypt(file_data)
    with open(file_name, 'wb') as file:
        file.write(encrypted_data)
    print(f"File {file_name} encrypted.")

# Function to decrypt a file
def decrypt_file(file_name, key):
    """
    Decrypts a file using the provided Fernet key.
```

```
    :param file_name: The name of the file to decrypt.
    :param key: The Fernet key for decryption.
    """
    f = Fernet(key)
    with open(file_name, 'rb') as file:
        encrypted_data = file.read()
    decrypted_data = f.decrypt(encrypted_data)
    with open(file_name, 'wb') as file:
        file.write(decrypted_data)
    print(f"File {file_name} decrypted.")

if __name__ == "__main__":
    # Example usage
    action = input("Do you want to (E)ncrypt or (D)ecrypt a file? ").upper()
    if action not in ['E', 'D']:
        print("Invalid option.")
    else:
        file_path = input("Enter the file path: ")
        if not os.path.exists(file_path):
            print("File does not exist.")
        else:
            if action == 'E':
                generate_key()
                key = load_key()
                encrypt_file(file_path, key)
            elif action == 'D':
                key = load_key()
                decrypt_file(file_path, key)
```

➢ Explanation

- **Key management**: the script generates a Fernet key and saves it to a file. This key is used for both encryption and decryption. It's crucial to keep this key secure and private.
- **Encryption and decryption functions**: these functions take a file name and a key as parameters. The file's content is read, then encrypted or decrypted, and finally written back to the same file. This replaces the original content with its encrypted or decrypted form.
- **User interaction**: the script prompts the user to choose between encryption and decryption, then asks for the file path. It includes basic error handling for invalid options and missing files.

➢ Security Considerations

- **Key security**: the security of encrypted data heavily relies on the secrecy of the key. If someone gains access to the key, they can decrypt the data.
- **Backup original files**: before encrypting files, it's a good practice to create backups in case of corruption or loss during the encryption process.

➢ Conclusion

This Python script provides a basic yet powerful tool for file encryption and decryption using the Fernet symmetric key cryptography. It demonstrates key generation, secure file encryption, and decryption processes. While the script is designed to be straightforward and accessible for users of all levels, it also incorporates fundamental security practices suitable for secure communication or storage tasks.

File integrity checker

Creating a file integrity checker in Python involves monitoring specific files or directories to detect any unauthorized changes, which can be an indicator of tampering or a security breach. This tool is crucial for maintaining the security and integrity of data within a system. The following script uses hashing to monitor file integrity, a common and effective approach for such tasks.

➢ Prerequisites

- Python installed on your system.
- Familiarity with basic file operations in Python.
- No additional Python packages are required for this example.

➢ Script overview

The script will:

- Calculate and store the hash of files in the specified directory.
- Periodically re-calculate hashes and compare them with the stored ones.
- Alert if any discrepancies are found, indicating a file has been altered.

➢ Python code

```
import hashlib
import os
import time

def hash_file(filepath):
    """
    Calculate the MD5 hash of a file.

    :param filepath: Path to the file to hash.
    :return: The hex digest of the file's hash.
    """
```

```
    hasher = hashlib.md5()
    with open(filepath, 'rb') as file:
        buf = file.read()
        hasher.update(buf)
    return hasher.hexdigest()

def monitor_directory(path, interval=60):
    """
    Monitor a directory for any changes in file integrity.

    :param path: The directory path to monitor.
    :param interval: Time interval (in seconds) between checks.
    """
    print(f"Monitoring {path} for changes...")
    file_hashes = {}

    # Initial hashing of directory contents
    for filename in os.listdir(path):
        filepath = os.path.join(path, filename)
        if os.path.isfile(filepath):
            file_hashes[filepath] = hash_file(filepath)

    while True:
        for filename in os.listdir(path):
            filepath = os.path.join(path, filename)
            if os.path.isfile(filepath):
                # Calculate the current hash
                current_hash = hash_file(filepath)
                if filepath in file_hashes:
                    if file_hashes[filepath] != current_hash:
                        print(f"File changed: {filename}")
                        file_hashes[filepath] = current_hash
```

```
                else:
                    print(f"New file detected: {filename}")
                    file_hashes[filepath] = current_hash

        time.sleep(interval)

if __name__ == "__main__":
    directory_to_monitor = input("Enter the directory path to monitor: ")
    monitor_interval = int(input("Enter the monitoring interval in seconds:
"))
    monitor_directory(directory_to_monitor, monitor_interval)
```

➢ Explanation

- **Hashing functionality**: the script uses MD5 for hashing file contents. While MD5 is not recommended for cryptographic purposes due to vulnerabilities, it is sufficient for basic integrity checks. For more critical applications, consider using SHA-256.
- **Monitoring logic**: initially, the script calculates the hash of every file in the specified directory and stores these hashes. It then enters a loop where it periodically recalculates the hashes and compares them to the stored values to detect any changes.
- **User interaction**: the user is prompted to enter the directory path and the monitoring interval. This allows for flexibility and user-defined settings based on their specific needs.

➢ Security considerations

- **Hash collision**: MD5 is susceptible to hash collisions. For more sensitive applications, use a more secure hashing algorithm like SHA-256.
- **Performance**: for directories with a large number of files or very large files, consider optimizing the file reading and hashing process to avoid performance issues.

➢ Conclusion

This script provides a foundational approach to monitoring file integrity using hashing in Python. It's designed to be accessible to users of all experience levels, offering a practical tool for detecting unauthorized file modifications. As with all security tools, it's crucial to adapt and update the implementation based on the specific security requirements and advances in technology.

Firewall configuration checker

Creating a firewall configuration checker in Python is a complex task due to the variability of firewall types, vendors, and configurations. However, I'll guide you through a basic Python script that analyzes a simplified representation of firewall rules. This script will check if these rules meet certain predefined security policies. The purpose of this tool is educational, to demonstrate how one might approach the task of analyzing firewall configurations for compliance with security policies.

➢ Prerequisites

- Python installed on your system.
- Basic understanding of firewall concepts and rule sets.
- This example assumes a simplified firewall rule set for demonstration.

➢ Script overview

The script will:

- Load a set of firewall rules from a file or hardcoded list.
- Define a set of security policies against which the rules will be checked.
- Analyze the rules to determine compliance with the security policies.
- Report rules that violate any of the predefined security policies.

➢ Python code

```
import json

def load_firewall_rules(filename):
    """
    Load firewall rules from a JSON file.

    :param filename: Path to the JSON file containing firewall rules.
    :return: A list of firewall rules.
```

```
    """
    with open(filename, 'r') as file:
        rules = json.load(file)
    return rules

def check_rule_compliance(rule, policies):
    """
    Check if a firewall rule complies with the given security policies.

    :param rule: A firewall rule represented as a dictionary.
    :param policies: A list of policies (functions) to check against.
    :return: True if the rule complies with all policies, False otherwise.
    """
    for policy in policies:
        if not policy(rule):
            return False
    return True

def policy_allow_https_only(rule):
    """
    Security policy: Allow only HTTPS traffic.

    :param rule: A firewall rule represented as a dictionary.
    :return: True if the rule allows only HTTPS traffic, False otherwise.
    """
    return rule.get('protocol') == 'TCP' and rule.get('port') == 443

def analyze_firewall_rules(rules, policies):
    """
    Analyze firewall rules against security policies.

    :param rules: A list of firewall rules.
    :param policies: A list of policies (functions) to check against.
```

```
    """
    for rule in rules:
        if not check_rule_compliance(rule, policies):
            print(f"Rule violation: {rule}")

if __name__ == "__main__":
    # Load firewall rules from a file (example: 'firewall_rules.json')
    rules = load_firewall_rules('firewall_rules.json')

    # Define security policies
    policies = [policy_allow_https_only]

    # Analyze rules against policies
    analyze_firewall_rules(rules, policies)
```

➢ Explanation

- **Loading rules**: this script assumes firewall rules are stored in a JSON file. Each rule is a dictionary with keys and values defining the rule properties (e.g., protocol, port).
- **Policy functions**: each policy is a function that takes a rule as input and returns `True` if the rule complies with the policy or `False` otherwise. This modular approach allows for easy addition of new policies.
- **Analysis**: the `analyze_firewall_rules` function checks each rule against all policies. If a rule violates any policy, it is reported as a violation.

➢A JSON file

A JSON file used to store firewall rules would typically contain an array of rule objects. Each rule object can have various fields representing the properties of a firewall rule, such as the action (allow or deny), protocol (e.g., TCP, UDP), source and destination IP addresses, source and destination ports, and possibly additional conditions or remarks.

Here's an example of what such a JSON file (`firewall_rules.json`) might look like:

```
[
    {
        "id": 1,
        "action": "allow",
        "protocol": "TCP",
        "src_ip": "any",
        "dest_ip": "192.168.1.10",
        "src_port": "any",
        "dest_port": "443",
        "description": "Allow HTTPS traffic to web server"
    },
    {
        "id": 2,
        "action": "allow",
        "protocol": "TCP",
        "src_ip": "any",
        "dest_ip": "192.168.1.11",
        "src_port": "any",
        "dest_port": "22",
        "description": "Allow SSH traffic to server"
    },
    {
        "id": 3,
        "action": "deny",
        "protocol": "TCP",
        "src_ip": "any",
        "dest_ip": "any",
        "src_port": "any",
        "dest_port": "23",
```

```
            "description": "Block Telnet access"
        }
    ]
```

➢ Explanation of the JSON structure:

- **`id`**: a unique identifier for each rule, making it easier to reference.
- **`action`**: specifies whether the traffic matching this rule should be allowed or denied.
- **`protocol`**: the network protocol to which this rule applies, such as TCP or UDP.
- **`src_ip`** and **`dest_ip`**: The source and destination IP addresses. "Any" indicates that any IP address is applicable.
- **`src_port`** and **`dest_port`**: the source and destination ports. "Any" for the port means that any port is applicable, while specific numbers (e.g., 443 for HTTPS) restrict the rule to those ports.
- **`description`**: a human-readable description of what the rule is intended to do or represent.

➢ Usage in the script:

In the context of the Python script I described earlier, this JSON file would be loaded to create a set of rules that the script then analyzes against defined security policies. For example, a security policy might check if there are any rules that allow traffic other than HTTPS (port 443 for TCP), ensuring that only secure web traffic is permitted, in line with best practices for minimizing attack surfaces.

This approach allows firewall configurations to be stored in a readable and easily editable format, facilitating automated analysis and potentially automated management and deployment of firewall rules.

➢ Security considerations

- **Complexity of firewall rules**: real-world firewall configurations can be much more complex. This script is a

starting point and might need significant expansion to handle real-world scenarios effectively.

- **Diverse environments**: different environments may have different security requirements. Customize the policies to fit the specific security posture and compliance requirements of the environment.

➢ Conclusion

This script offers a foundational approach to analyzing firewall configurations against security policies. While simplified, it introduces the concept of programmatically checking configuration compliance, a crucial aspect of cybersecurity. Expanding this script to handle more complex rule sets and diverse policies would be necessary for real-world applications.

Forensic image analyzer

Creating a forensic image analyzer involves processing disk images (e.g., ISO, dd images) to extract and analyze data for digital forensics purposes. This task can be complex due to the variety of file systems and the need to recover deleted files or hidden data. However, I'll guide you through a simplified Python script that demonstrates the basic concept of analyzing a disk image. This script will use the `pytsk3` library, a Python binding for the Sleuth Kit, allowing for low-level access to disk images and file systems for forensic analysis.

➢ Prerequisites

- Python environment set up.
- Installation of `pytsk3` and `pyewf` libraries if you are dealing with EWF images.
- Basic understanding of forensic analysis and disk image structures.

➢ Script overview

The script will:

- Open a disk image file.
- Traverse the file system found within the disk image.
- Print out basic information about each file (name, size, creation time, etc.).
- Highlight basic steps for finding potentially interesting files or artifacts.

➢ Python code

```
import pytsk3
import datetime

class ImageAnalyzer:
    def __init__(self, image_path):
        self.image_path = image_path
```

```
    def open_image(self):
        """
        Open the disk image using pytsk3.
        """
        try:
            # Open the disk image
            self.img = pytsk3.Img_Info(self.image_path)
        except IOError as e:
            print(f"Failed to open image: {e}")
            exit()

    def recursive_file_listing(self, directory, parent_path="/"):
        """
        Recursively list files and directories in the given directory object.
        """
        for entry in directory:
            if entry.info.name.name in [".", ".."]:
                continue

            file_path = f"{parent_path}{entry.info.name.name}"
            try:
                f_type = entry.info.meta.type
                if f_type == pytsk3.TSK_FS_META_TYPE_DIR:
                    sub_directory = entry.as_directory()
                    print(f"Directory: {file_path}")
                    self.recursive_file_listing(sub_directory,
parent_path=file_path + "/")
                elif f_type == pytsk3.TSK_FS_META_TYPE_REG:
                    size = entry.info.meta.size
                    print(f"File: {file_path}, Size: {size}")
```

```
                except AttributeError:
                    # This can happen with some system files
                    pass

        def analyze(self):
            """
            Analyze the disk image.
            """
            self.open_image()
            filesystem = pytsk3.FS_Info(self.img)
            directory = filesystem.open_dir(path="/")
            print("Starting analysis...")
            self.recursive_file_listing(directory)

    if __name__ == "__main__":
        image_path = "path_to_your_disk_image.dd"
        analyzer = ImageAnalyzer(image_path)
        analyzer.analyze()
```

➢ Explanation

- **Initialization**: the `ImageAnalyzer` class is initialized with the path to a disk image.
- **Opening the image**: `open_image` uses `pytsk3.Img_Info` to open and prepare the disk image for analysis.
- **Recursive file listing**: `recursive_file_listing` traverses the file system, printing information about directories and files. It uses recursion to handle directories within directories.
- **Analysis**: the `analyze` function starts the process, opening the file system and beginning the recursive file listing from the root directory.

➢ Usage notes

- This script is a basic demonstration. Real-world forensic analysis would require more detailed examination of file

contents, looking for deleted files, and potentially analyzing file signatures.
- You might need to handle different file systems, encrypted files, or disk images in formats like EWF (`pyewf` can be used for such cases).

➢ Security and privacy considerations

- Always ensure you have legal authorization to analyze the disk image.
- Be aware of the sensitivity of data within disk images.

➢ Conclusion

This script provides a starting point for forensic analysis of disk images. Expanding this to meet specific investigative needs can involve incorporating more sophisticated analysis techniques, handling a wider range of file systems, and automating the detection of known artifacts or indicators of compromise.

Forensic image analyzer - expanded

Expanding the forensic image analyzer script to enhance its capabilities for digital forensics investigations involves several key areas. These enhancements can include support for more file system types, extraction and analysis of deleted files, file signature analysis, integration with databases of known hashes for identifying suspicious or known files, and handling encrypted disk images. Below, I outline several ways to expand the script, making it a more powerful tool for forensic analysis.

➢ Support for additional file system types

The initial script is designed around basic file system traversal, assuming a straightforward structure. Real-world disk images may contain a variety of file systems (e.g., NTFS, FAT32, ext3/4).

Enhancement: use `pytsk3`'s ability to handle different file systems automatically, but ensure you test with disk images that have different file systems. For more exotic or less common file systems, additional handling or libraries might be necessary.

➢ Extraction and analysis of deleted files

Deleted files can be crucial in an investigation. `pytsk3` allows access to files flagged as deleted if the file system records haven't been overwritten.

Enhancement: modify the `recursive_file_listing` function to check if an entry is marked as deleted (using `entry.info.meta.flags` in `pytsk3`). For deleted files, you could print out additional information or attempt to recover the file contents.

➢ File signature analysis

File signatures (or magic numbers) can be used to identify the type of content a file contains, regardless of its extension. This can be useful for identifying files that have been intentionally mislabeled or hidden.

Enhancement: integrate a library or create a function to read the first few bytes of a file and compare it against known file signatures. The Python

`magic` library or a custom mapping of byte signatures to file types can be used for this purpose.

➢ Integration with hash databases

Comparing file hashes against known databases of malicious file hashes can help quickly identify potentially harmful files.

Enhancement: generate MD5, SHA-1, or SHA-256 hashes for files and compare these against a database or API service that tracks known malicious files (e.g., VirusTotal API).

➢ Handling encrypted disk images

Encrypted disk images present a significant challenge in forensic analysis. While decrypting these without the encryption key can be impractical or impossible, identifying the encryption type and any potential weaknesses is a start.

Enhancement: detect encrypted volumes and report their presence. For known encryption types, you might also provide recommendations for forensic approaches (e.g., searching for keys in unencrypted portions of the disk).

➢ Graphical User Interface (GUI)

A GUI can make the tool more accessible, especially for those less comfortable with command-line interfaces.

Enhancement: develop a simple GUI using a library like `Tkinter` or `PyQt`. This interface could allow users to select disk images through a file picker, display results in a more readable format, and provide options for saving reports.

➢ Automated reporting

In forensic analysis, creating detailed reports of findings is essential.

Enhancement: implement functionality to generate reports based on the analysis. This could include summaries of findings, lists of suspicious or deleted files found, and any potential malware identified by hash

comparison. Python's report generation libraries, such as `ReportLab`, could be used for PDF report generation.

➢ Python code snippet for a potential enhancement (file signature analysis)

Here's a basic example of how you might begin implementing file signature analysis:

```
import binascii

def get_file_signature(filepath):
    """
    Reads the first few bytes of a file to determine its signature.
    """
    signatures = {
        'ffd8ffe0': 'JPEG image',
        '89504e47': 'PNG image',
        # Add more signatures as needed
    }
    with open(filepath, 'rb') as file:
        header = file.read(4)  # Read the first 4 bytes
        signature = binascii.hexlify(header).decode('utf-8')
        for sig in signatures:
            if signature.startswith(sig):
                return signatures[sig]
    return "Unknown file type"

# In your existing file analysis loop
file_type = get_file_signature(file_path)
print(f"File: {file_path}, Type: {file_type}")
```

➢ Conclusion

Expanding the forensic image analyzer script requires a multi-faceted approach, addressing the diverse needs of digital forensic investigations. Each enhancement brings its challenges and requirements, but together,

they significantly increase the utility and power of the tool. Remember, with digital forensics, it's crucial to stay updated with the latest techniques and threats, continuously evolving your tools and methods.

Incident response toolkit

Creating an incident response toolkit involves assembling a collection of scripts and tools that can be quickly deployed during a security incident to perform various critical tasks such as system isolation, evidence collection, log analysis, and communication with team members. The toolkit is meant to be modular, allowing incident responders to select the appropriate tool for the situation at hand. Below is an example of a Python-based toolkit, focusing on two key areas: system isolation and evidence collection.

➢Incident response toolkit overview

The toolkit is designed to be extensible, with each script serving a specific purpose. It's structured around a simple command-line interface (CLI) that allows responders to easily select and execute different tools within the toolkit.

➢ System isolation script

This script isolates a system from the network to prevent an ongoing attack from spreading or external communication by the attacker. It's critical to have administrative privileges to execute these commands.

```
import subprocess
import sys

def isolate_system():
    """
    Attempts to isolate the system from the network.
    This example is for a Windows-based system. Adjust commands for
other OS.
    """
    try:
        # Disable network adapters
        subprocess.run(["powershell", "-Command", "Get-NetAdapter |
Disable-NetAdapter -Confirm:$false"], check=True)
```

```
            print("System isolated successfully.")
        except subprocess.CalledProcessError as e:
            print(f"Failed to isolate system: {e}")
            sys.exit(1)

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

➢ Evidence collection script

This script collects vital system information and logs for further analysis. It's designed to gather data quickly and comprehensively without needing manual intervention.

```
    import os
    import subprocess
    import datetime
    import zipfile

    def collect_evidence(output_dir="incident_response_evidence"):
        """
        Collects system evidence and stores it in the specified output directory.
        """
        timestamp = datetime.datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
        evidence_dir = f"{output_dir}_{timestamp}"
        os.makedirs(evidence_dir, exist_ok=True)

        # Collect system information
        system_info_cmd = "systeminfo"
        subprocess.run(system_info_cmd + f" > {evidence_dir}/system_info.txt", shell=True)

        # Collect logs (Example: Windows Security Logs)
        logs_cmd = 'wevtutil epl Security'
```

```
    subprocess.run(logs_cmd + f" {evidence_dir}/security_logs.evtx",
shell=True)

    # Zip evidence for easy transport
    with zipfile.ZipFile(f"{evidence_dir}.zip", 'w', zipfile.ZIP_DEFLATED)
as zipf:
        for root, dirs, files in os.walk(evidence_dir):
            for file in files:
                zipf.write(os.path.join(root, file),
os.path.relpath(os.path.join(root, file), os.path.join(evidence_dir, '..')))

    print(f"Evidence collected and stored in: {evidence_dir}.zip")

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

➢ Toolkit CLI wrapper (optional)

An optional CLI wrapper can be implemented to provide a unified interface for launching the various tools within the toolkit. This could be as simple as a Python script that uses `argparse` or a third-party library like `click` to handle command-line arguments and options.

➢ Adding more tools

The incident response toolkit can be expanded by adding more scripts for different tasks, such as:

- **Log analysis**: scripts to parse and analyze logs for suspicious activity.
- **Communication**: automated notifications to team members or systems.
- **Malware analysis**: quick scripts to quarantine and analyze potential malware samples.

➢ Conclusion

This Incident response toolkit is just a starting point. Each organization will have unique requirements and environments, so it's important to customize

and test your toolkit accordingly. Keep your tools updated and ensure that all incident response team members are familiar with their operation to ensure effective responses to security incidents.

Log analyzer

Creating a Log Analyzer in Python involves parsing log files to identify suspicious activities, which can include signs of unauthorized access, system errors that could indicate security vulnerabilities, or unexpected application behavior. This example demonstrates a basic but effective approach to log analysis, focusing on flexibility and ease of use across various experience levels. We'll aim to make it modular, allowing for easy updates or expansions, such as adding more complex patterns or integrating with other systems for alerting.

➢ Log analyzer overview

This script reads log files, searches for patterns indicative of suspicious activities, and reports these findings. It's designed to be run either as a scheduled task or manually, depending on the needs of the organization.

➢ Python code for log analyzer

```
import re
import sys

def analyze_log(file_path):
    """
    Analyzes a log file for suspicious activities based on predefined patterns.

    :param file_path: Path to the log file to be analyzed.
    """
    # Define suspicious patterns (These can be customized or expanded)
    patterns = {
        "unauthorized_access": "failed login",
        "sql_injection": "select.*from.*where",
        "xss_attack": "<script>.*</script>",
    }

    # Compile regex patterns for efficiency
```

```
    compiled_patterns = {key: re.compile(pattern, re.IGNORECASE) for key, pattern in patterns.items()}

    # Track found issues
    issues_found = {key: 0 for key in patterns}

    try:
        with open(file_path, 'r') as file:
            for line in file:
                for issue, pattern in compiled_patterns.items():
                    if pattern.search(line):
                        issues_found[issue] += 1
                        print(f"Suspicious activity detected [{issue}]: {line.strip()}")

        # Summary of analysis
        print("\nSummary of suspicious activities found:")
        for issue, count in issues_found.items():
            print(f"{issue}: {count}")

    except FileNotFoundError:
        print("Log file not found. Please check the file path and try again.")
        sys.exit(1)
    except Exception as e:
        print(f"An error occurred: {e}")
        sys.exit(1)

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python log_analyzer.py <path_to_log_file>")
        sys.exit(1)

    log_file_path = sys.argv[1]
```

```
analyze_log(log_file_path)
```

➢ Explanation

- **Pattern matching**: the script uses regular expressions (regex) to search for patterns in the log file that match common indicators of suspicious activity. These patterns are defined at the beginning of the script and can be customized based on the logs being analyzed and the specific threats of concern.
- **Efficiency**: by compiling the regex patterns before iterating through the log file, the script operates more efficiently, especially with large files.
- **Modularity**: the script is designed to be easily modified. New patterns can be added to the `patterns` dictionary without changing the core logic of the script.
- **User interaction**: the script expects the path to the log file as an argument, making it flexible to use with different files or in automated workflows. Error handling ensures the user is informed if the file doesn't exist or another issue occurs.

➢ Extending the script

To make the log analyzer even more powerful, consider the following enhancements:

- **Dynamic pattern loading**: allow the script to load patterns from an external file or database, making it easier to update the patterns without modifying the script.
- **Reporting**: integrate email or messaging system notifications to alert relevant stakeholders of detected issues.
- **Real-time analysis**: modify the script to analyze logs in real-time, using file system watchers or integrating directly with logging systems.

This script provides a solid foundation for log analysis, suitable for a wide range of Python users, from beginners to experienced professionals. It demonstrates key principles such as pattern matching, file handling, and basic error management, making it a valuable tool in cybersecurity efforts.

Malware analysis helper

In the field of cybersecurity, analyzing malware is a critical task to understand its behavior, impact, and origin. This guide introduces a Python script designed to automate basic static and dynamic analysis tasks for suspected malware files, making it accessible to a wide range of users, from novices to experienced analysts.

➢ Prerequisites

- **Python 3**: the script is written for Python 3.6 or newer.
- **Libraries**: `pefile` for static analysis, `subprocess` for dynamic analysis, and `virustotal-python` for online scanning.
- **VirusTotal API key**: for querying VirusTotal, you'll need an API key obtained by creating a free account on their website.

➢ Script overview

The script performs three main tasks:

- **Static analysis**: extracts basic information from the file, such as headers, sections, and embedded strings, which can indicate the file's purpose or origin.
- **Dynamic analysis**: executes the file in a controlled environment to observe its behavior, capturing system calls, file modifications, and network activity.
- **VirusTotal scanning**: submits the file to VirusTotal for scanning by multiple antivirus engines, providing a comprehensive report on its detection rates.

➢ Python code

```
import pefile

import subprocess

import sys

from virustotal_python import Virustotal

from pathlib import Path
```

```
# Replace 'YOUR_API_KEY' with your actual VirusTotal API key
VIRUSTOTAL_API_KEY = 'YOUR_API_KEY'

def static_analysis(file_path):
    """Performs static analysis on a PE file."""
    pe = pefile.PE(file_path)
    print("== Static Analysis Results ==")
    print(f"File Name: {Path(file_path).name}")
    print(f"Number of Sections: {len(pe.sections)}")
    # Add more static analysis as needed

def dynamic_analysis(file_path):
    """Performs dynamic analysis by executing the file."""
    print("== Dynamic Analysis ==\nWarning: This will execute the malware. Ensure this is run in a safe environment.")
    # Example: subprocess.run(["sandbox_executable", file_path], check=True)
    # Implement sandbox execution and monitoring

def scan_with_virustotal(file_path):
    """Scans the file with VirusTotal."""
    vtotal = Virustotal(API_KEY=VIRUSTOTAL_API_KEY)
    with open(file_path, "rb") as file:
        try:
            response = vtotal.file_scan(file)
            analysis_id = response["json_resp"]["data"]["id"]
            print(f"VirusTotal Analysis ID: {analysis_id}")
            # Fetch and display the report using the analysis ID
        except Exception as e:
            print(f"Error scanning with VirusTotal: {e}")

if __name__ == "__main__":
```

```
    if len(sys.argv) != 2:
        print("Usage: python malware_analysis_helper.py
<path_to_malware>")
        sys.exit(1)

    file_path = sys.argv[1]
    static_analysis(file_path)
    # Uncomment the next line if you wish to perform dynamic analysis
    # dynamic_analysis(file_path)
    scan_with_virustotal(file_path)
```

➢ Explanation

- **Static analysis**: uses `pefile` to inspect PE (Portable Executable) files, common for Windows executables and DLLs. This part of the script prints basic information about the file.
- **Dynamic analysis**: intended to be customized. The script provides a placeholder for executing the file in a controlled environment. Real-world use should include detailed monitoring tools and safety measures.
- **VirusTotal scanning**: uses the `virustotal-python` library to submit the file for scanning and retrieve the results. This requires an internet connection and a VirusTotal API key.

➢Usage and limitations

- Run the script from the command line, passing the path to the suspected malware file as an argument.
- The dynamic analysis section is deliberately simplistic and must be adapted for actual use, ideally within a secure sandbox environment.
- The script requires external dependencies and a VirusTotal API key.

➢ Conclusion

This malware analysis helper script provides a foundation for automating the initial stages of malware investigation. While effective for basic analysis, it should be expanded with more sophisticated static and dynamic analysis techniques for professional use. Always ensure that any dynamic analysis is performed in a secure, isolated environment to prevent unintended malware execution on critical systems.

Network scanner

Creating a network scanner in Python allows you to identify active devices on a network by sending ICMP packets (to ping the devices) or by scanning specific ports. This tool is invaluable for network administrators, security professionals, and anyone interested in network management or cybersecurity.

➢ Prerequisites

- **Python 3**: this script is compatible with Python 3.x.
- **Libraries**: `python-nmap` for port scanning, and `subprocess` for ICMP pinging. Install them via pip:

```
pip install python-nmap
```

- **Permissions**: running ICMP ping and certain port scans may require administrator or root privileges due to raw socket operations.

➢ Script overview

The script performs two main functions:

- **ICMP ping sweep**: checks for the presence of active devices by sending ICMP packets.
- **Port scanning**: identifies open ports on devices, indicating active services.

➢ Python code

```
import subprocess
import nmap
import sys

def icmp_ping_sweep(subnet):
    """Performs an ICMP ping sweep on the given subnet."""
    print(f"Starting ICMP ping sweep on {subnet}")
    for ip_end in range(1, 255):
```

```
            ip = f"{subnet}.{ip_end}"
            result = subprocess.run(['ping', '-c', '1', ip],
        stdout=subprocess.DEVNULL)
            if result.returncode == 0:
                print(f"Host {ip} is up")

        def port_scan(target, ports):
            """Scans the specified ports on the target device."""
            nm = nmap.PortScanner()
            print(f"Starting port scan on {target}")
            for port in ports:
                try:
                    nm.scan(target, str(port))
                    state = nm[target]['tcp'][port]['state']
                    print(f"Port {port} is {state}")
                except Exception as e:
                    print(f"Cannot scan port {port}: {e}")

        if __name__ == "__main__":
            if len(sys.argv) < 3:
                print("Usage: python network_scanner.py <subnet> <port,port,...>")
                sys.exit(1)

            subnet = sys.argv[1]
            ports = [int(port) for port in sys.argv[2].split(',')]

            icmp_ping_sweep(subnet)
            port_scan(subnet + ".1", ports)  # Example usage: scanning the gateway
```

➢ Explanation

- **ICMP ping sweep**: the `icmp_ping_sweep` function uses the `ping` command to check if devices in a subnet are reachable.

It iterates through IP addresses in the specified subnet, sending an ICMP packet to each.

- **Port scanning**: the `port_scan` function uses the `python-nmap` library to scan specified ports on a target device. It checks the state of each port (open, closed, filtered, etc.).
- **Command line interface**: the script accepts the subnet to scan and a list of ports as command-line arguments, offering flexibility for various scanning scenarios.

➢ Usage and limitations

- Run the script from the command line, providing the subnet to scan and a comma-separated list of ports. Example:

```
python network_scanner.py 192.168.1 22,80,443
```

- Accuracy and completeness of the scan depend on network security measures (firewalls, ICMP blocking, etc.).
- Scanning networks without permission is illegal and unethical.

➢ Conclusion

This network scanner script provides a basic framework for identifying active devices and open ports within a network. It demonstrates essential techniques in network scanning and can be expanded with more sophisticated scanning strategies or integrated into larger network management or security tools. Always ensure ethical use and obtain necessary permissions before conducting network scans.

Packet sniffer

A packet sniffer is a tool that captures network packets, allowing you to monitor network traffic in real-time and analyze these packets for various purposes, including network troubleshooting, traffic analysis, and detecting suspicious activities or anomalies.

➢ Prerequisites

- **Python 3**: ensure Python 3.6 or newer is installed.
- **Libraries**: `scapy` for packet capturing and analysis. Install it via pip:

```
pip install scapy
```

- **Permissions**: packet sniffing often requires root or administrative privileges to access network interfaces in promiscuous mode.

➢ Script overview

This script uses `scapy` to capture packets on a specified network interface. It demonstrates how to filter traffic, parse packet details, and identify potential anomalies or important information in the traffic.

➢ Python code

```
from scapy.all import sniff
from scapy.layers.http import HTTPRequest
from scapy.layers.inet import IP, TCP

def packet_callback(packet):
    """
    Callback function to process each captured packet.
    Filters for HTTP requests and prints out packet info.
    """
    if not packet.haslayer(HTTPRequest):
```

```
        # Filter for HTTP requests only
        return

    # Extract packet information
    ip_layer = packet.getlayer(IP)
    http_layer = packet.getlayer(HTTPRequest)

    print(f"[+] HTTP Request: {http_layer.Host.decode()}
{http_layer.Path.decode()}")
    print(f"    Source IP: {ip_layer.src} -> Destination IP: {ip_layer.dst}")

def start_sniffing(interface="eth0"):
    """
    Starts packet sniffing on a given network interface.
    """
    print(f"[*] Starting packet sniffing on {interface}...")
    sniff(iface=interface, prn=packet_callback, store=False, filter="tcp")

if __name__ == "__main__":
    # Customize the interface based on your network configuration
    network_interface = "eth0"
    start_sniffing(network_interface)
```

➢ Explanation

- **Scapy for packet capturing**: the script uses `scapy`, a powerful Python library for network packet manipulation and sniffing. It captures packets in real-time, filtering for TCP packets and specifically HTTP requests.
- **Callback function**: `packet_callback` is invoked by `scapy` for each packet captured that matches the filter criteria. It processes packets to extract and display HTTP request information and the source and destination IP addresses.
- **Network interface**: the `start_sniffing` function initializes packet sniffing on the specified network interface. Change

`eth0` to the appropriate interface on your system.

➢ Usage and limitations

- Run the script as root or with administrative privileges to allow packet capturing. Modify the `network_interface` variable as needed to match your system's configuration.
- The script is designed to capture HTTP traffic. For HTTPS or other protocols, adjustments are needed.
- Running a packet sniffer on a network without permission may be illegal or violate network policies.

➢ Conclusion

This packet sniffer script provides a foundation for building more complex network monitoring and analysis tools. While it demonstrates basic packet capturing and HTTP request logging, further development could include features like anomaly detection, logging to files, or real-time alerts for suspicious activities. Always use such tools responsibly and with respect to privacy and legal considerations.

Password strength tester

A password strength tester evaluates the complexity and robustness of passwords against defined criteria or common password policies. This tool is crucial for encouraging strong security practices by ensuring passwords are sufficiently complex to resist common attack methods like brute force or dictionary attacks.

➢ Prerequisites

- **Python 3**: the script is compatible with Python 3.x. No additional libraries are required for the basic functionality described here.

➢ Script overview

This script checks passwords against several common criteria:

- **Minimum length**: ensures the password meets a minimum length requirement.
- **Character diversity**: checks for the inclusion of uppercase and lowercase letters, digits, and special characters.
- **Common passwords**: optionally, checks against a list of common passwords that should be avoided.

➢ Python code

```
import re

def check_password_strength(password, common_passwords=None):
    """
    Evaluate the strength of a password against common criteria.

    :param password: The password string to test.
    :param common_passwords: An optional list of common passwords to
check against.
    :return: True if the password meets the criteria, False otherwise.
    """
```

```
    # Criteria
    min_length = 8
    contains_upper = re.search(r'[A-Z]', password) is not None
    contains_lower = re.search(r'[a-z]', password) is not None
    contains_digit = re.search(r'\d', password) is not None
    contains_special = re.search(r'[\W_]', password) is not None

    # Check against common passwords
    is_not_common = True
    if common_passwords and password in common_passwords:
        is_not_common = False

    # Final evaluation
    if (len(password) >= min_length and contains_upper and contains_lower
and
            contains_digit and contains_special and is_not_common):
        return True
    else:
        return False

if __name__ == "__main__":
    # Example usage
    common_passwords_list = ["password", "123456", "12345678"]
    test_password = input("Enter a password to test: ")
    if check_password_strength(test_password,
common_passwords=common_passwords_list):
        print("Password is strong.")
    else:
        print("Password is weak.")
```

➢ Explanation

- **Regular expressions**: the script uses Python's `re` module to check for the presence of uppercase letters, lowercase letters, digits, and special characters in the password.
- **Common passwords check**: if a list of common passwords is provided, the script checks to ensure the tested password is not in this list.
- **Criteria evaluation**: the password is evaluated against the set criteria, and the function returns `True` if the password meets all the strength requirements.

➢ Usage and limitations

- Run the script and input a password when prompted. For real-world applications, the list of common passwords should be expanded and updated regularly.
- The script checks basic criteria for password strength. More sophisticated checks might include patterns, sequences, or dictionary words.
- The common passwords list is hardcoded for demonstration purposes. In practice, this should be replaced with a comprehensive, regularly updated list.

➢ Conclusion

This password strength tester script provides a basic framework for evaluating password complexity. It can be easily expanded or integrated into user registration processes, encouraging stronger password creation practices. As cybersecurity threats evolve, enhancing and updating the criteria and methods for password strength testing is crucial for maintaining effective security measures.

Port scanner

A port scanner is an essential tool in network security and system administration, allowing you to identify open ports on a networked device. These open ports can reveal running services, which might indicate vulnerabilities or opportunities for further investigation.

➢ Prerequisites

- **Python 3**: ensure you have Python 3.x installed.
- **Libraries**: no external libraries are needed for a basic scanner. Advanced functionality may require additional libraries like `socket` for networking tasks.

➢ Script overview

This script performs a simple port scanning operation on a specified target (IP address or hostname) and a range of port numbers. It attempts to establish a TCP connection to each port and identifies whether the port is open.

➢ Python code

```
import socket
import sys

def scan_target(target, start_port, end_port):
    """
    Scan the specified target IP or hostname for open ports within the given range.

    :param target: Target IP address or hostname.
    :param start_port: Starting port number.
    :param end_port: Ending port number.
    """
    print(f"Scanning target {target} from port {start_port} to {end_port}")
```

```
    for port in range(start_port, end_port + 1):
        result = scan_port(target, port)
        if result:
            print(f"Port {port} is open")
        else:
            print(f"Port {port} is closed", end='\r')

def scan_port(ip, port):
    """
    Attempts to connect to the specified port on the given IP or hostname.

    :param ip: IP address or hostname to scan.
    :param port: Port number to test.
    :return: True if the port is open, False otherwise.
    """
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout(1)  # Timeout for the socket operation
        conn = sock.connect_ex((ip, port))  # Returns 0 if connection
succeeds
        sock.close()
        return conn == 0
    except socket.gaierror:
        print(f"Hostname {ip} could not be resolved.")
        sys.exit()
    except socket.error:
        print(f"Couldn't connect to server {ip}.")
        sys.exit()

if __name__ == "__main__":
    if len(sys.argv) < 4:
```

```
        print("Usage: python3 port_scanner.py <target> <start_port>
<end_port>")
        sys.exit(1)

    target_ip = sys.argv[1]
    start_port = int(sys.argv[2])
    end_port = int(sys.argv[3])

    scan_target(target_ip, start_port, end_port)
```

➢ Explanation

- **Socket programming**: the `socket` module is used to establish a low-level network connection. The `scan_port` function attempts to open a TCP connection to the specified port.
- **Target specification**: the user provides the target IP address or hostname along with the start and end ports as command-line arguments.
- **Result interpretation**: if a connection to a port is successfully established (`conn == 0`), the port is considered open. The script prints the status of each port as it scans.

➢ Usage and limitations

- Execute the script from the command line, specifying the target and port range. For example:

```
python3 port_scanner.py example.com 1 100
```

- The script performs a basic TCP connect scan, which can be detected and logged by firewalls or intrusion detection systems.
- Scanning a large number of ports or targets may be slow due to the sequential approach and socket timeout settings.

➢ Conclusion

This port scanner script provides a foundational approach to identifying open ports and potential services on a networked device. While effective for

basic scans, further development could include parallel scanning, integration with service identification databases, and stealthier scanning techniques. Always ensure ethical use of scanning tools, obtaining necessary permissions to avoid unauthorized access or network disruption.

Security policy compliance checker

A security policy compliance checker is a tool designed to automate the verification of system configurations against predefined security policies, standards, or benchmarks. It helps ensure that systems are configured securely and consistently, reducing the risk of vulnerabilities.

➢ Prerequisites

- **Python 3**: the script is developed for Python 3.x.
- **External libraries**: depending on the specifics of the policies being checked, external libraries may be required. For example, `paramiko` for SSH connections or `openpyxl` for reading Excel files containing policy definitions.
- **Access**: appropriate access to the target system(s) is necessary, whether through SSH for remote systems or local admin privileges for direct checks.

➢ Script overview

The script performs checks on a system to ensure compliance with a set of defined security policies. This example focuses on a simplified scenario where the compliance checks are based on the presence of specific files and the settings within configuration files, demonstrating how to structure such a tool.

➢ Python Code

```
import os

# Example security policies
SECURITY_POLICIES = {
    'ssh_config_secure': {
        'path': '/etc/ssh/sshd_config',
        'must_contain': ['PermitRootLogin no', 'PasswordAuthentication yes'],
        'must_not_contain': ['X11Forwarding yes'],
    },
```

```
    'firewall_enabled': {
        'check_command': 'ufw status',
        'expected_output': 'Status: active'
    }
}

def check_file_policy(policy):
    """
    Checks if a file complies with the must_contain and must_not_contain policies.
    """
    with open(policy['path'], 'r') as file:
        contents = file.read()
        for must in policy.get('must_contain', []):
            if must not in contents:
                return False
        for must_not in policy.get('must_not_contain', []):
            if must_not in contents:
                return False
    return True

def check_command_policy(policy):
    """
    Checks if the output of a command complies with the expected output defined in the policy.
    """
    import subprocess
    result = subprocess.run(policy['check_command'], shell=True, capture_output=True, text=True)
    return policy['expected_output'] in result.stdout
```

```
def check_compliance():
    """
    Runs through all security policies and checks compliance.
    """
    for name, policy in SECURITY_POLICIES.items():
        if 'path' in policy:
            result = check_file_policy(policy)
        elif 'check_command' in policy:
            result = check_command_policy(policy)
        else:
            result = False  # Unknown policy type

        print(f"Policy '{name}': {'Compliant' if result else 'Non-Compliant'}")

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

➢ Explanation

- **Policy definitions**: security policies are defined in a dictionary, specifying the checks to be performed. This includes file existence, content checks, and system command outputs.
- **File policy checks**: for file-based policies, the script reads the specified file and checks for the presence or absence of defined strings.
- **Command output checks**: for command-based policies, it executes the specified system command and verifies the output against expected results.
- **Modularity**: the script is designed to be easily extendable with new policies or different types of checks.

➢ Usage and limitations

- Run the script on the target system. Ensure Python 3 and any necessary permissions or access rights are in place.

- The example script is simplified and focuses on file presence and simple configuration checks. Real-world applications might require more complex checks, such as validating permissions, checksums, or integrating with APIs for centralized management.
- Security policies vary widely; thus, the script must be adapted to match specific organizational or industry standards.

➢ Conclusion

This security policy compliance checker script provides a foundational approach to automating the verification of system configurations against security policies. It demonstrates how Python can be utilized to streamline the compliance checking process, contributing to maintaining a secure and consistent system environment. Customization and expansion of the script will be necessary to cover the full range of specific policies relevant to different systems or organizational requirements.

SSH brute force attack

I must emphasize that using or developing tools for unauthorized access to systems is illegal and unethical. The information provided here is for educational purposes only, to understand how such attacks can be conducted and to reinforce the importance of strong security practices, including the use of secure passwords and two-factor authentication to protect against unauthorized access.

Given the sensitive nature of a "SSH brute force attack" tool, I will instead guide you on how to create a script for educational purposes that simulates a login attempt to an SSH server, focusing on how to secure systems against such attacks.

➢ Prerequisites

- **Python 3**: installed on your system.
- **Paramiko library**: a Python library for making SSH2 connections. Install it using pip:

```
pip install paramiko
```

➢ Script overview

The script will simulate login attempts to an SSH server by trying a combination of usernames and passwords. This example will only showcase the mechanism for attempting connections without actually performing brute-force attacks.

➢ Python code

```
import paramiko
import sys

def attempt_ssh_login(hostname, port, username, password):
    """
    Attempts to login to an SSH server with provided credentials.
    """
    try:
```

```
            client = paramiko.SSHClient()
            client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
            client.connect(hostname, port=port, username=username,
password=password, timeout=1)
            print(f"Success: Credentials '{username}:{password}' are valid.")
            client.close()
            return True
        except paramiko.AuthenticationException:
            print(f"Failed: Credentials '{username}:{password}' are invalid.")
            return False
        except Exception as e:
            print(f"Error: {e}")
            return False

    if __name__ == "__main__":
        if len(sys.argv) != 5:
            print("Usage: python ssh_login_simulator.py <hostname> <port>
<username> <password>")
            sys.exit(1)

        hostname, port, username, password = sys.argv[1], int(sys.argv[2]),
sys.argv[3], sys.argv[4]
        attempt_ssh_login(hostname, port, username, password)
```

➢ Explanation

- **Paramiko SSH client**: the script uses the `paramiko` library to handle the SSH connection. `paramiko.SSHClient` offers a high-level interface to connect and interact with SSH servers.
- **Connection attempt**: it tries to establish an SSH connection using the provided hostname, port, username, and password. On success, it prints a success message; on failure, it catches `paramiko.AuthenticationException` to indicate incorrect credentials.

- **Host Key Policy**: `client.set_missing_host_key_policy(paramiko.AutoAddPolicy())` is used here for simplicity, automatically adding the server's host key. In a production environment, you should manage host keys more securely to prevent Man-in-the-Middle attacks.

➢ Usage and limitations

- The script is executed with four arguments: hostname, port, username, and password. It's designed to simulate a single login attempt.
- As written, the script does not perform brute-force attacks. It demonstrates how to use `paramiko` for SSH connections.
- Using or modifying this script to attempt unauthorized access to systems can be illegal and unethical.

➢ Conclusion

While understanding the mechanics of SSH login attempts can be valuable for educational purposes and for developing security measures, it's crucial to use this knowledge responsibly. Implement strong security practices, such as using secure, complex passwords, employing two-factor authentication, and monitoring login attempts to protect against unauthorized access. Always adhere to ethical guidelines and legal requirements when working with security tools and techniques.

SSL/TLS scanner

Creating an SSL/TLS scanner in Python involves analyzing the SSL/TLS configurations of websites to identify potential vulnerabilities, misconfigurations, and the usage of outdated protocols or ciphers. This kind of tool is crucial for maintaining the security and integrity of data in transit.

➢ Prerequisites

- **Python 3**: ensure you have Python 3.x installed.
- **OpenSSL and libraries**: the script will use the `ssl` and `socket` modules, which are part of the Python Standard Library. For more advanced analysis, you might also explore external libraries, but this example will stick to standard modules for accessibility.

➢ Script overview

This script connects to a specified website using SSL/TLS and retrieves information about the certificate and the encryption details of the connection. It's a basic demonstration aimed at showing how you can start evaluating SSL/TLS security.

➢ Python code

```
import ssl
import socket
import sys

def ssl_tls_scanner(hostname, port=443):
    """
    Scans the SSL/TLS configuration of the specified hostname.

    :param hostname: The hostname of the website to scan.
    :param port: The port number, default is 443 for HTTPS.
    """
    context = ssl.create_default_context()
```

```
    try:
        # Establishing a connection to the host
        with socket.create_connection((hostname, port)) as sock:
            with context.wrap_socket(sock, server_hostname=hostname) as
ssock:
                print(f"SSL/TLS version: {ssock.version()}")
                print(f"Cipher suite: {ssock.cipher()}")
                print(f"Certificate: {ssock.getpeercert()}")
    except Exception as e:
        print(f"Error scanning {hostname}: {e}")

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python ssl_tls_scanner.py <hostname>")
        sys.exit(1)

    target_hostname = sys.argv[1]
    ssl_tls_scanner(target_hostname)
```

➢ Explanation

- **SSL context**: `ssl.create_default_context()` creates a new SSL context with secure default settings.
- **Socket connection**: establishes a TCP connection to the target hostname and port using `socket.create_connection()`.
- **SSL wrapping**: the TCP socket is wrapped with the SSL context, initiating the SSL/TLS handshake and establishing an encrypted connection.
- **Certificate and cipher details**: after establishing the connection, the script prints out the SSL/TLS version, the cipher suite used for the connection, and the server's certificate details.

➢ Usage and limitations

- Run the script from the command line with the target website's hostname as an argument. The script assumes HTTPS and default port 443, but you can modify it to accept a port number if needed.
- This basic scanner provides information on the SSL/TLS version, cipher suite, and certificate but does not perform in-depth vulnerability analysis or cipher suite strength evaluation.
- Advanced scanning, including testing for specific vulnerabilities like Heartbleed or checking for TLS downgrade attacks, would require more complex implementations or external tools.

➢ Conclusion

This SSL/TLS scanner script serves as a foundational tool for assessing the security configurations of SSL/TLS on websites. While it demonstrates the core concept of retrieving encryption details, extensive analysis for security auditing purposes would necessitate further development or the use of specialized software designed for comprehensive security assessments. Always ensure to use such tools ethically and with permission when scanning systems you do not own.

Subdomain finder

A subdomain finder is a tool that automates the discovery of subdomains for a given domain. This is crucial for security assessments, allowing penetration testers and system administrators to identify potential attack vectors and ensure that all parts of a domain are secure.

➢ Prerequisites

- **Python 3**: the script requires Python 3.x.
- **Requests library**: for sending HTTP requests to potential subdomains. Install it via pip:

```
pip install requests
```

- **Wordlist**: a list of common subdomain names to test against the target domain. This script assumes you have a text file named `subdomains.txt`.

➢ Script overview

This script attempts to find active subdomains for a specified domain by iterating over a list of potential subdomain names and checking for HTTP responses.

➢ Python code

```
import requests
import sys

def find_subdomains(domain, subdomains_file):
    """
    Enumerate subdomains by checking for HTTP responses.

    :param domain: The target domain to search subdomains for.
    :param subdomains_file: File containing a list of potential subdomain
names.
    """
```

```
    found_subdomains = []
    with open(subdomains_file, 'r') as file:
        for line in file:
            subdomain = line.strip()
            url = f"http://{subdomain}.{domain}"

            try:
                requests.get(url)
                print(f"Found: {url}")
                found_subdomains.append(url)
            except requests.ConnectionError:
                # Subdomain does not exist or is not reachable
                pass

    return found_subdomains

if __name__ == "__main__":
    if len(sys.argv) < 3:
        print("Usage: python subdomain_finder.py <domain>
<subdomains_file>")
        sys.exit(1)

    target_domain = sys.argv[1]
    subdomains_file_path = sys.argv[2]
    found = find_subdomains(target_domain, subdomains_file_path)

    if found:
        print("\nFound subdomains:")
        for subdomain in found:
            print(subdomain)
    else:
        print("No subdomains found.")
```

➢ Explanation

- **Subdomain enumeration**: the script reads from a provided list of common subdomain names (`subdomains_file`) and constructs URLs by appending each subdomain to the target domain.
- **HTTP requests**: for each constructed URL, an HTTP GET request is attempted. If the request does not raise a `ConnectionError`, the subdomain is considered found and is printed to the console.
- **Performance considerations**: the script performs synchronous requests, which could be slow for large wordlists. Consider implementing asynchronous requests or threading for performance improvements.

➢ Usage and limitations

- Run the script from the command line, providing the target domain and the path to your subdomains wordlist file as arguments.
- The script's effectiveness is heavily dependent on the quality and comprehensiveness of the subdomain wordlist.
- It only checks for the subdomain's existence via HTTP and does not perform deeper DNS record analysis.
- Some web configurations or security measures may block or mislead this type of scanning, affecting accuracy.

➢ Conclusion

This subdomain finder script provides a basic yet effective approach to discovering potential subdomains for a given domain, which is an essential step in comprehensive security assessments. While simple, it highlights the importance of subdomain enumeration in identifying and securing all parts of a domain. For more advanced or extensive use, consider integrating more sophisticated DNS querying capabilities or utilizing third-party APIs that offer subdomain discovery services. Always ensure ethical use of such tools, respecting privacy and legal restrictions.

Vulnerability scanner

A vulnerability scanner is a tool designed to automate the process of identifying known vulnerabilities in systems or applications by referencing databases such as the Common Vulnerabilities and Exposures (CVE) list. Such scanners are essential for maintaining security by enabling timely detection and remediation of vulnerabilities.

➢ Prerequisites

- **Python 3**: ensure Python 3.x is installed on your system.
- **Vulnerability data source**: access to a vulnerability database is required. For this script, we'll simulate access to CVE data. In practice, you might download CVE data from official sources or use an API provided by cybersecurity databases like the National Vulnerability Database (NVD).
- **Requests library**: for fetching data from online APIs (if applicable). Install via pip:

```
pip install requests
```

➢ Script overview

This script demonstrates how to scan a system or application by checking its version against known vulnerabilities listed in a simplified CVE database. We'll simulate the database as a local JSON file for this example, focusing on the process rather than live data retrieval.

➢ Python code

```
import json
import sys

# Simulated CVE database (JSON file format)
CVE_DATABASE_PATH = "cve_database.json"

def load_cve_database():
    """
    Load the CVE database from a local JSON file.
```

```
    """
    try:
        with open(CVE_DATABASE_PATH, 'r') as file:
            return json.load(file)
    except FileNotFoundError:
        print("CVE database file not found.")
        sys.exit(1)
    except json.JSONDecodeError:
        print("Error decoding the CVE database.")
        sys.exit(1)

def scan_for_vulnerabilities(software, version, cve_data):
    """
    Scan the specified software and version against the CVE database.

    :param software: The software name to scan.
    :param version: The version of the software.
    :param cve_data: The loaded CVE database.
    """
    vulnerabilities = []
    for cve in cve_data:
        if software in cve['software'] and version in cve['affected_versions']:
            vulnerabilities.append(cve['cve_id'])

    return vulnerabilities

if __name__ == "__main__":
    if len(sys.argv) < 3:
        print("Usage: python vulnerability_scanner.py <software>
<version>")
        sys.exit(1)
```

```
    target_software = sys.argv[1]
    target_version = sys.argv[2]
    cve_data = load_cve_database()

    found_vulnerabilities = scan_for_vulnerabilities(target_software,
target_version, cve_data)

    if found_vulnerabilities:
        print(f"Vulnerabilities found for {target_software} {target_version}:")
        for cve_id in found_vulnerabilities:
            print(cve_id)
    else:
        print(f"No known vulnerabilities found for {target_software}
{target_version}.")
```

➢ Explanation

- **CVE database loading**: the script starts by loading a CVE database from a local JSON file. This file should contain structured data about known vulnerabilities, including software names, affected versions, and CVE identifiers.
- **Scanning logic**: it then scans for vulnerabilities by comparing the target software and its version against entries in the CVE database. If matches are found, it lists the corresponding CVE identifiers.
- **Local database simulation**: this approach uses a static, local database for simplicity and demonstration purposes. Real-world applications might fetch data from live databases or APIs.

➢ Usage and limitations

- Execute the script by providing the software name and version as command-line arguments. Ensure the CVE database JSON file (`cve_database.json`) is correctly formatted and present in the same directory.

- The script's effectiveness depends on the completeness and currency of the CVE database. Real-time data retrieval from an official source is recommended for operational use.
- It does not account for complex versioning schemes or software configurations that might affect vulnerability exposure.

➢ Conclusion

This vulnerability scanner script concept demonstrates the foundational approach to automating vulnerability detection based on software versions. While the script is a simplified model, it highlights the importance of continuous vulnerability management as part of a comprehensive cybersecurity strategy. Expanding this script to integrate with live databases, support more complex version checks, and automate scans across multiple systems would significantly enhance its utility in real-world applications. Always ensure to use such tools responsibly and in compliance with legal and ethical standards.

Web scraper for security feeds

A web scraper for security feeds is designed to automate the collection of information from security websites or RSS feeds, providing timely updates on the latest vulnerabilities, threats, and security news. This tool is invaluable for cybersecurity professionals who need to stay informed about the latest developments.

➢ Prerequisites

- **Python 3**: ensure Python 3.x is installed.
- **Libraries**: `requests` for making HTTP requests and `BeautifulSoup` from `bs4` for parsing HTML or XML content. Install them via pip:

```
pip install requests beautifulsoup4
```

- **RSS feed or website URL**: know the target URL(s) you intend to scrape. Ensure you have the legal right to scrape the website and that it complies with the website's `robots.txt` policy.

➢ Script overview

This script demonstrates how to scrape an RSS feed for security news. RSS feeds are chosen for this example due to their structured format, which is generally easier and more consistent to parse than arbitrary HTML from websites.

➢Python code

```
import requests
from bs4 import BeautifulSoup

def scrape_security_feed(feed_url):
    """
    Scrapes a security RSS feed for the latest news and vulnerabilities.

    :param feed_url: The URL of the security RSS feed.
```

```
    """
    try:
        response = requests.get(feed_url)
        response.raise_for_status()  # Raise an error for bad responses

        soup = BeautifulSoup(response.content, 'xml')
        items = soup.find_all('item')

        for item in items[:5]:  # Limit to the first 5 entries for brevity
            title = item.find('title').text
            link = item.find('link').text
            description = item.find('description').text
            print(f"Title: {title}\nLink: {link}\nDescription: {description}\n{'-
'*40}")

    except requests.RequestException as e:
        print(f"Error fetching the RSS feed: {e}")

if __name__ == "__main__":
    # Example RSS feed URL
    rss_feed_url = "https://www.example.com/securityfeed"
    scrape_security_feed(rss_feed_url)
```

➢ Explanation

- **Fetching the feed**: the script uses `requests.get` to fetch the content of the RSS feed URL.
- **Parsing the feed**: `BeautifulSoup` parses the RSS feed content, using `'xml'` as the parser since RSS feeds are XML formatted.
- **Extracting information**: it iterates over each `<item>` (common in RSS feeds for individual news entries), extracting and printing the title, link, and description.
- **Error handling**: the script includes basic error handling to catch issues like network errors or invalid URLs.

➢ Usage and limitations

- Run the script with the RSS feed URL of your choice. The example provided in the script should be replaced with an actual security news RSS feed URL.
- The script is designed for RSS feeds. Scraping websites with dynamic content loaded via JavaScript may require tools like Selenium.
- Respect rate limiting and legal considerations when scraping content to avoid being blocked or violating terms of service.

➢ Conclusion

This web scraper for security feeds script provides a straightforward way to automate the collection of security news from RSS feeds, helping cybersecurity professionals stay informed about the latest threats and vulnerabilities. Expanding this script could involve adding features like automated email notifications, scraping multiple sources concurrently, or integrating with databases for tracking and analysis. As always, ethical and legal considerations should guide the use of web scraping tools.

Wireless network scanner

Creating a wireless network scanner in Python that identifies Wi-Fi networks and devices, assessing them for vulnerabilities, involves interacting with the system's wireless interfaces and possibly scanning the environment for Wi-Fi signals. This task is complex and can vary significantly across operating systems. For educational purposes, I'll guide you through a basic approach using Python on Linux systems, leveraging the `iwlist` command, which is commonly available on Linux distributions.

➢ Prerequisites

- **Linux operating system**: this script is designed for Linux due to the use of `iwlist`.
- **Python 3**: ensure Python 3.x is installed.
- **Wireless interface**: a wireless network interface that supports monitor mode is required for scanning Wi-Fi networks.
- **Permissions**: running network scanning operations typically requires root or sudo privileges.

➢Script overview

This script demonstrates how to invoke the `iwlist` command from Python to scan for nearby Wi-Fi networks, parse the output, and display information about each network. This example will focus on identifying network names (SSIDs) and their encryption types, which can hint at potential vulnerabilities (e.g., networks using outdated WEP encryption).

➢ Python code

```
import subprocess
import re
import sys

def scan_networks(interface):
    """
    Scans Wi-Fi networks using the specified wireless interface.

    :param interface: The wireless network interface to use for scanning.
```

```
    """
    try:
        # Run the iwlist command to scan for networks
        scan_output = subprocess.check_output(["iwlist", interface, "scan"],
text=True)

        # Regular expressions to find SSIDs and Encryption types
        ssid_re = re.compile(r"ESSID:\"(.+?)\"")
        encryption_re = re.compile(r"Encryption key:(on|off)")

        ssids = ssid_re.findall(scan_output)
        encryption_keys = encryption_re.findall(scan_output)

        networks = zip(ssids, encryption_keys)

        for ssid, encryption in networks:
            print(f"Network SSID: {ssid}, Encryption: {'Enabled' if encryption
== 'on' else 'Disabled'}")

    except subprocess.CalledProcessError:
        print(f"Failed to scan networks using interface {interface}. Ensure
you have the necessary permissions.")
        sys.exit(1)
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        sys.exit(1)

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: sudo python3 wireless_network_scanner.py
<interface>")
        sys.exit(1)
```

```
    wlan_interface = sys.argv[1]
    scan_networks(wlan_interface)
```

➢ Explanation

- **Subprocess call**: the script uses `subprocess.check_output` to execute `iwlist` with the scan option on the specified interface. This command requires root privileges, hence the need for `sudo`.
- **Parsing output**: it employs regular expressions to parse the SSIDs and encryption statuses from the `iwlist` output, displaying them to the user.
- **Security assessment**: while the script identifies networks and their encryption status, assessing them for vulnerabilities would involve further analysis, such as determining if outdated or weak encryption protocols are used.

➢ Usage and limitations

- Run the script with root privileges and the wireless interface name as an argument. Example: `sudo python3 wireless_network_scanner.py wlan0`.
- The script is tailored for Linux and might not work on other operating systems without modification.
- It provides basic network information. More sophisticated vulnerability assessment would require deeper analysis and possibly active testing, which should be done ethically and with permission.

➢ Conclusion

This wireless network scanner script offers a foundational approach to identifying nearby Wi-Fi networks and assessing basic security configurations. Expanding this tool could involve integrating more advanced scanning capabilities, automating vulnerability detection based on encryption types, and possibly interfacing with databases of known vulnerabilities. Always conduct network scanning and analysis responsibly, within ethical and legal boundaries.