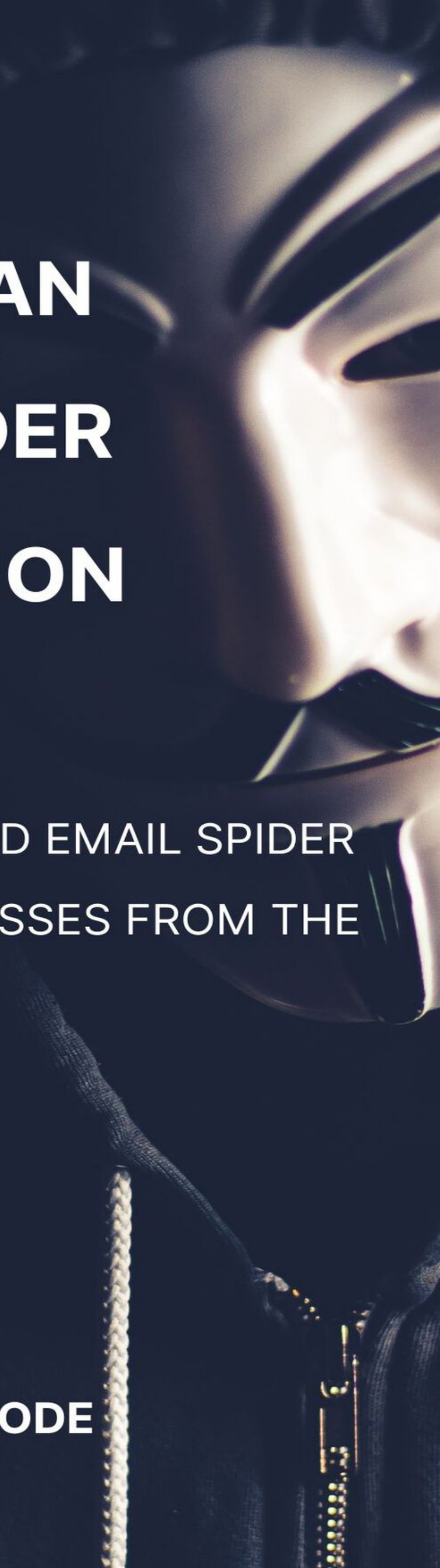


BUILDING AN EMAIL SPIDER WITH PYTHON

LEARN BUILDING AN ADVANCED EMAIL SPIDER THAT EXTRACTS EMAIL ADDRESSES FROM THE WEB

PYTHON CODE

About the Author

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

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

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

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

Preface

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

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

Each section in this book has been designed to be both informative and practical, offering you clear, step-by-step instructions on how to perform various forensic investigations techniques.

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

Welcome to the journey. Let's start scraping email addresses!

Notices and Disclaimers

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

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

Introduction

In the rapidly evolving landscape of digital communication, email remains an essential tool for business outreach, networking and information gathering. However, the manual process of collecting relevant email addresses is not only time-consuming but also limiting in scope and efficiency. This mini book aims to alleviate that issue by teaching you how to build an advanced email spider using Python, one of the most versatile programming languages.

We begin by exploring the foundational aspects of email extraction, starting with a simple script that gathers email addresses from a single web page. This serves as a stepping stone for what follows—an advanced email spider capable of crawling through multiple web pages, following links and collecting email data in an automated and efficient manner.

This mini book offers an in-depth look into sophisticated features that set our email spider apart. For instance, we integrate an integer parameter to ensure controlled crawling and prevent indefinite loops. Furthermore, to maximize Internet speed utilization, we employ a multi-threaded approach, allowing multiple instances of the email extractor to operate simultaneously.

Whether you’re new to Python programming or looking to enhance your existing skill set, this guide provides a comprehensive look at the steps and considerations involved in creating an advanced email spider. This isn’t merely a programming exercise; it’s an exploration of practical Python applications in the realm of web scraping and data extraction.

Building a Simple Email Extractor

An email extractor or harvester is a software that extracts email addresses from online and offline sources to generate an extensive list of addresses. Even though these extractors can serve multiple legitimate purposes, such as marketing campaigns or cold emailing, they are mainly used to send spamming and phishing emails, unfortunately.

Since the web is the primary source of information on the Internet, in this section, you will learn how to build such a tool in Python to extract email addresses from web pages using the `requests` and `requests-html` libraries. We will create a more advanced threaded email spider in the next section.

Because many websites load their data using JavaScript instead of directly rendering HTML code, I chose the `requests-html` library for this section as it supports JavaScript-driven websites.

Let’s install the `requests-html` library:

```
$ pip install requests-html
```

Open up a new file named `email_harvester.py` and import the following:

```
import  re
from  requests_html  import  HTMLSession
```

We need the [`re` module](#) here because we will extract emails from HTML content using regular expressions. If you're unsure what a regular expression is, it is a sequence of characters defining a search pattern (check [this Wikipedia article](#) for details).

I've grabbed the most used and accurate regular expression for email addresses from [this StackOverflow answer](#):

```
url  = "https://www.randomlists.com/email-addresses"
EMAIL_REGEX  = r """ (?:[ a-z0-9!#$%&'*+/=?^_`{|}~- ] + (?: \. [ a-
z0-9!#$%&'*+/=?^_`{|}~- ] + ) * | " (?:[ \x01 - \x08\x0b\x0c\x0e -
\x1f\x21\x23 - \x5b\x5d - \x7f ] | \\ [ \x01 - \x09\x0b\x0c\x0e -
\x7f ]) * " ) @ (?:(?:[ a-z0-9 ](?:[ a-z0-9- ] * [ a-z0-9 ]) ?\. ) + [
a-z0-9 ](?:[ a-z0-9- ] * [ a-z0-9 ]) ? | \[ (?:(?:( 2 ( 5 [ 0-5 ] | [ 0-4
][ 0-9 ]) | 1 [ 0-9 ][ 0-9 ] | [ 1-9 ] ? [ 0-9 ])) \. ) {3} (?:( 2 ( 5 [ 0-5
] | [ 0-4 ][ 0-9 ]) | 1 [ 0-9 ][ 0-9 ] | [ 1-9 ] ? [ 0-9 ]) | [ a-z0-9- ] *
[ a-z0-9 ] : (?:[ \x01 - \x08\x0b\x0c\x0e - \x1f\x21 - \x5a\x53 -
\x7f ] | \\ [ \x01 - \x09\x0b\x0c\x0e - \x7f ]) + ) \] ) """
```

It is very long, but this is the best so far that defines how email addresses are expressed in a general way.

`url` string is the URL we want to grab email addresses from. I'm using a website that generates random email addresses (which loads them using Javascript, by the way).

Let's initiate the HTML session, which is a consumable session for cookie persistence and connection pooling:

```
# initiate an HTTP session
session = HTMLSession()
```

Now let's send the GET request to the URL:

```
# get the HTTP Response
r = session.get(url)
```

If you're sure that the website you're grabbing email addresses from uses JavaScript to load most of the data, then you need to execute the below line of code:

```
# for JAVA-Script driven websites
r.html.render()
```

This will reload the website in Chromium and replace HTML content with an updated version, with Javascript executed. Of course, it'll take some time to do that. You must execute this only if the website loads its data using JavaScript.

Note: Executing the `render()` method the first time will automatically download Chromium for you, so it will take some time to do that.

Now that we have the HTML content and our email address regular expression, let's extract emails from the page:

```
for re_match in re.finditer(EMAIL_REGEX, r
.html.raw_html.decode()):
  print(re_match.group())
```

The `re.finditer()` method returns an iterator over all non-overlapping matches in the string. For each match, the iterator returns a `match` object, and we access the matched string (the email address) using the `group()` method.

The resulting HTML of the response object is located in the `r.html.raw_html`. Since it comes in the `bytes` type, `decode()` is necessary to convert it back to a string. There is also `r.html.html` that is equivalent to `raw_html` but in string form, so `decode()` won't be necessary. You're free to use any.

Here is the result of my execution:

```
$ python email_harvester.py
msherr@comcast.net
miyop@yahoo.ca
ardagna@yahoo.ca
tokuhirom@att.net
atmarks@comcast.net
isotopian@live.com
hoyer@msn.com
ozawa@yahoo.com
mchugh@outlook.com
sriha@outlook.com
monopole@sbcglobal.net
```

Excellent, with only a few lines of code, we can grab email addresses from any web page we want!

In the next section, we will extend this code to build a crawler that extracts all website URLs, run this same code on every page we find, and then save them to a text file.

Building an Advanced Email Spider

In this section, we will make a more advanced email harvester. The following are some of the main features that we will add to the program:

Instead of extracting emails from a single page, we add a crawler that visits every link on that page and parses emails.
To prevent the program from crawling indefinitely, we add an integer parameter to stop crawling when the number of crawled links reaches this parameter.
We run multiple email extractors simultaneously using threads to take advantage of the Internet speed.
When the crawler produces links to be visited for extracting emails, other threads will consume these links and visit them to search for email addresses.

As you may already have noticed, the program we will be building is based on the Producer-Consumer problem . If you're unsure what it is, it's a classical operating system problem used for multi-threading synchronization.

The producer produces something to add to a buffer, and the consumer consumes the item in the buffer that the producer makes. The producer and the consumer must be running on separate threads.

In our problem, the producer is the crawler: Going to a given URL, extracting all the links, and adding them to the buffer (i.e., a queue); these links are items for the email spider (the consumer) to consume.

The crawler then goes to the second link it finds during the first crawl and continues crawling until a certain number of crawls is reached.

We will have multiple consumers that read from this queue and extract email addresses, which are called email spiders and will be represented in a class.

Let’s get started. First, let’s install the required libraries:

```
$ pip install requests bs4 colorama
```

We will be using `BeautifulSoup` to parse links from HTML pages and colorama for printing in colors in the console.

Open up a new Python file called `advanced_email_spider.py` , and import the following:

```
import  re, argparse, threading, time, warnings, requests, colorama
from  urllib . parse  import  urlparse , urljoin
from  queue  import  Queue
warnings . filterwarnings ( "ignore" )
from  bs4  import  BeautifulSoup
# init the colorama module
colorama . init ()
# initialize some colors
```

```
GREEN  = colorama . Fore . GREEN
GRAY  = colorama . Fore . LIGHTBLACK_EX
RESET  = colorama . Fore . RESET
YELLOW  = colorama . Fore . YELLOW
RED  = colorama . Fore . RED
```

Nothing special here; we imported the necessary modules and defined the colors we will use for printing in the console.

Next, we define some variables that are necessary for the program:

```
EMAIL_REGEX  = r """ (?:[ a-z0-9!#$%&'*+=?^_`{|}~- ] + (?: \. [ a-
z0-9!#$%&'*+/=?^_`{|}~- ] + ) * | " (?:[ \x01 - \x08\x0b\x0c\x0e -
\x1f\x21\x23 - \x5b\x5d - \x7f ] | \\ [ \x01 - \x09\x0b\x0c\x0e -
\x7f ]) * " ) @ (?:(?:[ a-z0-9 ](?:[ a-z0-9- ] * [ a-z0-9 ]) ?\. ) + [
a-z0-9 ](?:[ a-z0-9- ] * [ a-z0-9 ]) ? | \[ (?:(?:( 2 ( 5 [ 0-5 ] | [ 0-4
][ 0-9 ]) | 1 [ 0-9 ][ 0-9 ] | [ 1-9 ] ? [ 0-9 ])) \. ) {3} (?:( 2 ( 5 [ 0-5
] | [ 0-4 ][ 0-9 ]) | 1 [ 0-9 ][ 0-9 ] | [ 1-9 ] ? [ 0-9 ]) | [ a-z0-9- ] *
[ a-z0-9 ] : (?:[ \x01 - \x08\x0b\x0c\x0e - \x1f\x21 - \x5a\x53 -
\x7f ] | \\ [ \x01 - \x09\x0b\x0c\x0e - \x7f ]) {2,12} ) \] ) """
# EMAIL_REGEX = r"[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-
9-]+(?:\.[a-zA-Z0-9-]{2,12})*"
# forbidden TLDs, feel free to add more extensions here to
prevent them identified as TLDs
FORBIDDEN_TLDS  = [
   "js" , "css" , "jpg" , "png" , "svg" , "webp" , "gz" , "zip" ,
"webm" , "mp3" ,
   "wav" , "mp4" , "gif" , "tar" , "gz" , "rar" , "gzip" , "tgz" ]
# a list of forbidden extensions in URLs, i.e 'gif' URLs won't
be requested
FORBIDDEN_EXTENSIONS  = [
   "js" , "css" , "jpg" , "png" , "svg" , "webp" , "gz" , "zip" ,
"webm" , "mp3" ,
   "wav" , "mp4" , "gif" , "tar" , "gz" , "rar" , "gzip" , "tgz" ]
# locks to assure mutex, one for output console & another for a
file
print_lock  = threading . Lock ()
file_lock  = threading . Lock ()
```

During the testing of the program, I found that many files are being parsed as email addresses, as they have the same shape as an email address. For instance, I found many files parsed as emails that look like this: `text@some-more-text.webp`.

As you may already know, the webp extension is for web images, not email addresses. Therefore, I made a list that excludes these extensions (`FORBIDDEN_TLDS`) being parsed as TLDs (Top Level Domains, e.g., .com, .net, etc.)

When crawling, the program also extracts URLs that are not text-based pages, such as a link to download a media file. Thus, I added a similar list for this and called it `FORBIDDEN_EXTENSIONS` to prevent crawling these non-text files.

Since there are multiple threads in our program, to assure mutual exclusion (mutex), I've added two locks, one for printing in the console and another for writing to the output file (that contains the resulting email addresses).

To simplify the locks, we need to ensure that threads will wait until other threads finish writing to the file to prevent data loss when multiple threads access the file and add data to it simultaneously.

Next, below are some utility functions to validate URLs and email addresses:

```
def  is_valid_email_address ( email ):
  """Verify whether `email` is a valid email address
  Args:
    email (str): The target email address.
  Returns: bool"""
  for  forbidden_tld  in  FORBIDDEN_TLDS :
    if  email .endswith( forbidden_tld ):
      # if the email ends with one of the forbidden TLDs,
return False
      return  False
  if  re . search ( r " \. . {1} $" , email ):
    # if the TLD has a length of 1, definitely not an email
    return  False
  elif  re . search ( r " \. . * \d + . * $" , email ):
    # TLD contain numbers, not an email either
```

```
        return  False
    # return true otherwise
    return  True

def  is_valid_url ( url ):
    """Checks whether `url` is a valid URL"""
    parsed  = urlparse ( url )
    return  bool ( parsed . netloc ) and  bool ( parsed . scheme )

def  is_text_url ( url ):
    """Returns False if the URL is one of the forbidden
extensions.
    True otherwise"""
    for  extension  in  FORBIDDEN_EXTENSIONS :
        if  url .endswith( extension ):
            return  False
    return  True
```

Even though we are extracting emails using a relatively good regular expression, I've added a second layer to verify email addresses and prevent the files I mentioned earlier from being parsed as email addresses. Also, some false addresses contain numbers in the TLD, and some have only one character; this function filters these out.

The `is_valid_url()` function checks whether a URL is valid; this is useful in the crawler. Whereas the `is_text_url()` checks whether the URL contains text-based content, such as raw text, HTML, etc., it is helpful to eliminate media-based URLs from the URLs to be visited.

Next, let’s now start with the crawler:

```
class  Crawler ( threading . Thread ):
    def  __init__ ( self , first_url , delay , crawl_external_urls
= False , max_crawl_urls = 30 ):
        # Call the Thread class's init function
        super (). __init__ ()
        self . first_url  = first_url
        self . delay  = delay
        # whether to crawl external urls than the domain specified
in the first url
```

```
     self . crawl_external_urls  = crawl_external_urls
     self . max_crawl_urls  = max_crawl_urls
     # a dictionary that stores visited urls along with their
HTML content
     self . visited_urls  = {}
     # domain name of the base URL without the protocol
     self . domain_name  = urlparse ( self . first_url ). netloc
     # simple debug message to see whether domain is extracted
successfully
     # print("Domain name:", self.domain_name)
     # initialize the set of links (unique links)
     self . internal_urls  = set ()
     self . external_urls  = set ()
     # initialize the queue that will be read by the email
spider
     self . urls_queue  = Queue ()
     # add the first URL to the queue
     self . urls_queue . put ( self . first_url )
     # a counter indicating the total number of URLs visited
     # used to stop crawling when reaching
`self.max_crawl_urls`
     self . total_urls_visited  = 0
```

Since the crawler will run in a separate thread, I've made it a class-based thread, which means inheriting the `Thread` class from the `threading` module, and overriding the `run()` method.

In the crawler constructor, we're defining some valuable attributes:

- `self.first_url` : The first URL to be visited by the crawler (which will be passed from the command-line arguments later on).
- `self.delay` : (in seconds) Helpful for not overloading web servers and preventing IP blocks.
- `self.crawl_external_urls` : Whether to crawl external URLs (relative to the first URL).
- `self.max_crawl_urls` : The maximum number of crawls.

We're also initializing handy object attributes:

- `self.visited_urls` : A dictionary that helps us store the visited URLs by the crawler along with their HTML response; it will become handy for the email spiders to prevent requesting the same page several times.
- `self.domain_name` : The domain name of the first URL visited by the crawler, helpful for determining extracted links to be external or internal links.
- `self.internal_urls` and `self.external_urls` : Sets for internal and external links, respectively.
- `self.urls_queue` : This is the producer-consumer buffer, a `Queue` object from the built-in Python's `queue` module. The crawler will add the URLs to this queue, and the email spiders will consume them (visit them and extract email addresses).
- `self.total_urls_visited` : This is a counter to indicate the total number of URLs visited by the crawler. It is used to stop crawling when reaching the `max_crawl_urls` parameter.

Next, let's make the method that, given a URL, extracts all the internal or external links, adds them to the sets mentioned above and the queue, and also return them:

```
  def  get_all_website_links ( self , url ):
     """Returns all URLs that is found on `url` in which it
belongs to the same website"""
     # all URLs of `url`
     urls  = set ()
     # make the HTTP request
     res  = requests . get ( url , verify = False , timeout = 10 )
     # construct the soup to parse HTML
     soup  = BeautifulSoup ( res . text , "html.parser" )
     # store the visited URL along with the HTML
     self . visited_urls [ url ] = res . text
     for  a_tag  in  soup . findAll ( "a" ):
       href  = a_tag .attrs.get( "href" )
       if  href  == ""  or  href  is  None :
         # href empty tag
         continue
       # join the URL if it's relative (not absolute link)
```

```
        href  = urljoin ( url , href )
        parsed_href  = urlparse ( href )
        # remove URL GET parameters, URL fragments, etc.
        href  = parsed_href . scheme  + "://"  + parsed_href .
netloc  + parsed_href . path
        if  not  is_valid_url ( href ):
          # not a valid URL
          continue
        if  href  in  self . internal_urls :
          # already in the set
          continue
        if  self . domain_name  not  in  href :
          # external link
          if  href  not  in  self . external_urls :
            # debug message to see external links when they're
found
            # print(f"{GRAY}[!] External link: {href}{RESET}")
            # external link, add to external URLs set
            self . external_urls . add ( href )
            if  self . crawl_external_urls :
              # if external links are allowed to extract
emails,
              # put them in the queue
              self . urls_queue . put ( href )
          continue
        # debug message to see internal links when they're
found
        # print(f"{GREEN}[*] Internal link: {href}{RESET}")
        # add the new URL to urls, queue and internal URLs
        urls . add ( href )
        self . urls_queue . put ( href )
        self . internal_urls . add ( href )
      return  urls
```

It is the primary method that the crawler will use to extract links from URLs. Notice that after making the request, we are storing the response HTML of the target URL in the `visited_urls` object attribute; we then add the extracted links to the queue and other sets.

You can check this online tutorial for more information about this function.

Next, we make our `crawl()` method:

```
    def crawl(self, url):
        """Crawls a web page and extracts all links.
        You'll find all links in `self.external_urls` and
`self.internal_urls` attributes."""
        # if the URL is not a text file, i.e not HTML, PDF, text,
etc.
        # then simply return and do not crawl, as it's unnecessary
download
        if not is_text_url(url):
            return
        # increment the number of URLs visited
        self.total_urls_visited += 1
        with print_lock:
            print(f"{YELLOW}[*] Crawling: {url}{RESET}")
        # extract all the links from the URL
        links = self.get_all_website_links(url)
        for link in links:
            # crawl each link extracted if max_crawl_urls is still
not reached
            if self.total_urls_visited > self.max_crawl_urls:
                break
            self.crawl(link)
            # simple delay for not overloading servers & cause it
to block our IP
            time.sleep(self.delay)
```

First, we check if it's a text URL. If not, we simply return and do not crawl the page, as it's unreadable and won't contain links.

Second, we use our `get_all_website_links()` method to get all the links and then recursively call the `crawl()` method on each one of the links until the `max_crawl_urls` is reached.

Next, let’s make the `run()` method that simply calls `crawl()`:

```
    def run(self):
     # the running thread will start crawling the first URL
passed
     self . crawl ( self . first_url )
```

Excellent, now we're done with the producer, let's dive into the `EmailSpider` class (i.e., consumer):

```
class  EmailSpider :
  def  __init__ ( self , crawler : Crawler , n_threads = 20 ,
output_file = "extracted-emails.txt" ):
     self . crawler  = crawler
     # the set that contain the extracted URLs
     self . extracted_emails  = set ()
     # the number of threads
     self . n_threads  = n_threads
     self . output_file  = output_file
```

The `EmailSpider` class will run multiple threads; therefore, we pass the crawler and the number of threads to spawn.

We also make the `extracted_emails` set containing our extracted email addresses.

Next, let's create the method that accepts the URL in the parameters and returns the list of extracted emails:

```
  def  get_emails_from_url ( self , url ):
     # if the url ends with an extension not in our interest,
     # return an empty set
     if  not  is_text_url ( url ):
       return  set ()
     # get the HTTP Response if the URL isn't visited by the
crawler
     if  url  not  in  self . crawler . visited_urls :
       try :
         with  print_lock :
           print ( f " { YELLOW } [*] Getting Emails from { url }{
RESET } " )
         r  = requests . get ( url , verify = False , timeout = 10 )
       except  Exception  as  e :
         with  print_lock :
           print ( e )
```

```
                return  set ()
            else :
                text  = r . text
        else :
            # if the URL is visited by the crawler already,
            # then get the response HTML directly, no need to
request again
            text  = self . crawler . visited_urls [ url ]
        emails  = set ()
        try :
            # we use finditer() to find multiple email addresses if
available
            for  re_match  in  re . finditer ( EMAIL_REGEX , text ):
                email  = re_match . group ()
                # if it's a valid email address, add it to our set
                if  is_valid_email_address ( email ):
                    emails . add ( email )
        except  Exception  as  e :
            with  print_lock :
                print ( e )
            return  set ()
        # return the emails set
        return  emails
```

The core of the above function is actually the code of the simple version of the email extractor we did earlier.

We have added a condition to check whether the crawler has already visited the URL. If so, we simply retrieve the HTML response and continue extracting the email addresses on the page.

If the crawler did not visit the URL, we make the HTTP request again with a `timeout` of 10 seconds and also set `verify` to `False` to not verify SSL, as it takes time. Feel free to edit the `timeout` based on your preferences and Internet conditions.

After the email is parsed using the regular expression, we double-check it using the previously defined `is_valid_email_address()` function to prevent some of the false positives I've encountered during the testing of the program.

Next, we make a wrapper method that gets the URL from the queue in the crawler object, extracts emails using the above method, and then writes them to the output file passed to the constructor of the `EmailSpider` class:

```
  def  scan_urls ( self ):
    while  True :
      # get the URL from the URLs queue
      url  = self . crawler . urls_queue . get ()
      # extract the emails from the response HTML
      emails  = self . get_emails_from_url ( url )
      for  email  in  emails :
        with  print_lock :
          print ( "[+] Got email:" , email , "from url:" , url
)
        if  email  not  in  self . extracted_emails :
          # if the email extracted is not in the extracted
emails set
          # add it to the set and print to the output file
as well
          with  file_lock :
            with  open ( self . output_file , "a" ) as  f :
              print ( email , file = f )
          self . extracted_emails . add ( email )
      # task done for that queue item
      self . crawler . urls_queue . task_done ()
```

Notice it's in an infinite `while` loop. Don't worry about that, as it'll run in a separate daemon thread, which means this thread will stop running once the main thread exits.

Let’s make the `run()` method of this class that spawns the threads calling the `scan_urls()` method:

```
  def  run ( self ):
    for  t  in  range ( self . n_threads ):
      # spawn self.n_threads to run self.scan_urls
      t  = threading . Thread ( target = self . scan_urls )
      # daemon thread
      t . daemon  = True
      t . start ()
```

```
     # wait for the queue to empty
     self . crawler . urls_queue . join ()
     print ( f "[+] A total of { len ( self . extracted_emails ) }
 emails were extracted & saved." )
```

We are spawning threads based on the specified number of threads passed to this object; these are daemon threads, meaning they will stop running once the main thread finish.

This `run()` method will run on the main thread. After spawning the threads, we wait for the queue to empty so the main thread will finish; hence, the daemon threads will stop running, and the program will close.

Next, I'm adding a simple statistics tracker (that is a daemon thread as well), which prints some statistics about the crawler and the currently active threads every five seconds:

```
def  track_stats ( crawler : Crawler ):
   # print some stats about the crawler & active threads every
5 seconds,
   # feel free to adjust this on your own needs
   while  is_running :
     with  print_lock :
       print ( f " { RED } [+] Queue size: { crawler . urls_queue .
qsize () }{ RESET } " )
       print ( f " { GRAY } [+] Total Extracted External links: {
len ( crawler . external_urls ) }{ RESET } " )
       print ( f " { GREEN } [+] Total Extracted Internal links: {
len ( crawler . internal_urls ) }{ RESET } " )
       print ( f "[*] Total threads running: { threading .
active_count () } " )
     time . sleep ( 5 )

def  start_stats_tracker ( crawler : Crawler ):
   # wrapping function to spawn the above function in a
separate daemon thread
   t  = threading . Thread ( target = track_stats , args =( crawler
,))
   t . daemon  = True
   t . start ()
```

Finally, let's use the `argparse` module to parse the command-line arguments and pass them accordingly to the classes we've built:

```
if  __name__  == "__main__" :
   parser  = argparse . ArgumentParser ( description = "Advanced
Email Spider" )
   parser . add_argument ( "url" , help = "URL to start crawling
from & extracting email addresses" )
   parser . add_argument ( "-m" , "--max-crawl-urls" ,
               help = "The maximum number of URLs to crawl,
default is 30." ,
               type = int , default = 30 )
   parser . add_argument ( "-t" , "--num-threads" ,
               help = "The number of threads that runs
extracting emails"  \
                 "from individual pages. Default is 10" ,
               type = int , default = 10 )
   parser . add_argument ( "--crawl-external-urls" ,
               help = "Whether to crawl external URLs that the
domain specified" ,
               action = "store_true" )
   parser . add_argument ( "--crawl-delay" ,
               help = "The crawl delay in seconds, useful for
not overloading web servers" ,
               type = float , default = 0.01 )
   # parse the command-line arguments
   args  = parser . parse_args ()
   url  = args .url
   # set the global variable indicating whether the program is
still running
   # helpful for the tracker to stop running whenever the main
thread stops
   is_running  = True
   # initialize the crawler and start crawling right away
   crawler  = Crawler ( url , max_crawl_urls = args
.max_crawl_urls, delay = args .crawl_delay,
              crawl_external_urls = args .crawl_external_urls)
   crawler . start ()
   # give the crawler some time to fill the queue
   time . sleep ( 5 )
```

```
  # start the statistics tracker, print some stats every 5
seconds
  start_stats_tracker ( crawler )
  # start the email spider that reads from the crawler's URLs
queue
  email_spider  = EmailSpider ( crawler , n_threads = args
.num_threads)
  email_spider . run ()
  # set the global variable so the tracker stops running
  is_running  = False
```

There are five main arguments passed from the command lines and are explained previously.

We start the program by initializing the `crawler` and starting the `crawler` thread. After that, we give it some time to produce some links (sleeping for five seconds seems an easy solution) into the queue. Then, we start our tracker and the email spider.

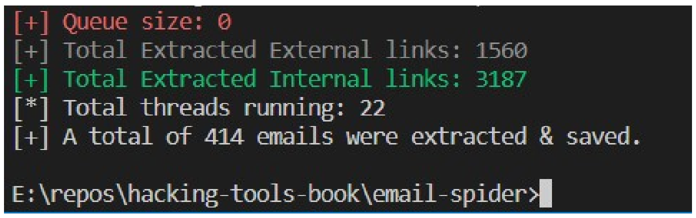
After the `run()` method of the email spider is returned, we set `is_running` to `False`, so the tracker exits out of the loop.

Running the Code

I have tried running the program from multiple places and with different parameters. Here's one of them:

```
$ python advanced_email_spider.py
https://en.wikipedia.org/wiki/Python_(programming_language) -m
10 -t 20 --crawl-external-urls --crawl-delay 0.1
```

I have instructed the spider to start from the Wikipedia page defining the Python programming language, only to crawl ten pages, to spawn 20 consumers, 0.1 seconds of delay between crawling, and to allow crawling external URLs than Wikipedia. Here's the output:



```
[+] Queue size: 0
[+] Total Extracted External links: 1560
[+] Total Extracted Internal links: 3187
[*] Total threads running: 22
[+] A total of 414 emails were extracted & saved.

E:\repos\hacking-tools-book\email-spider>
```

The program will print almost everything; the crawled URLs, the extracted emails, and the target URLs that the spider used to get emails. The tracker also prints every 5 seconds the valuable information you see above in colors.

After running for 10 minutes, and surprisingly, the program extracted 414 email addresses from more than 4700 URLs, most of them were Wikipedia pages that should not contain any email address.

Note that the crawler may produce a lot of links on the same domain name, which means the spiders will be overloading this server and, therefore, may block your IP address.

There are many ways to prevent that; the easiest is to spawn fewer threads on the spider, such as five, or add a delay on the spiders (because the current delay is only on the crawler).

Also, if the first URL you're passing to the program is slow to respond, you may not successfully crawl it, as the current program sleeps for 5 seconds before spawning the email harvester threads. If the consumers do not find any link in the queue, they will simply exit; therefore, you won't extract anything. Thus, you can increase the number of seconds when the server is slow.

Another problem is that other extensions are not text-based and are not in the `FORBIDDEN_EXTENSIONS` list. The spiders will download them, which may slow down your program and download unnecessary files.

I have been in a situation where the program hangs for several minutes (maybe even hours, depending on your Internet connection speed), downloading a 1GB+ file, which then turned out to be a ZIP file extracted somewhere by the crawler. After I experienced that, I decided to add this extension to the list. So, I invite you to add more extensions to this list to make the program more robust for such situations.

And that's it! You have now successfully built an email spider from scratch using Python! If you have learned anything from this program, you're definitely on a good path toward your goals!

Conclusion

In this mini book, we started by making a simple email extractor. Then, we added more complex code to the script to make it crawl websites and extract email addresses using Python threads.

Congratulations! You have finished the final mini book and hopefully the entire book series as well! You can always access the files of the entire series at this link or this GitHub repository.

In this book series, we have covered various topics in ethical hacking with Python. From information gathering scripts to building malware such as keyloggers and reverse shells. After that, we made offline and online password crackers. Then, we saw how to perform digital forensic investigations by extracting valuable metadata from files, passwords, and cookies from the Chrome browser and even hiding secret data in images. Following that, we explored the capabilities of the Scapy library and built many tools with it. Finally, we built an advanced email spider that crawls web pages and looks for email addresses.

After finishing the book series, I invite you to modify the code to suit your needs. For instance, you can use some useful functions and scripts covered in this book to build even more advanced programs that automate the thing you want to do.