Course Name: CYB 339

Professor: Ish Morales

Date: 1/26/2021

Examiner Name: Michael Meade

Introduction

Automating the process of scanning a target can be a helpful task for people who's job it is to protect the networks. It allows the network administrator to scan the network and get an idea of what is happening on the networks such as what services are running, what ports are opening, etc. This tool will bundle a bunch of scans together that will convert the results into a HTML report that could be edited to add more details. The HTML file could then be, converted into a PDF. The PDF could kept as historic history of what the network found, it could be given to the higher ups which might take action if a server is found that is misconfigured or not active.

Objective

The objective of this project is scanning the IP with NMAP and subdomain3. After scanning the target, the code will take the results and create an HTML file that describes the results in HTML tables. The code can perform a couple different scans such as DNS, a top port scan, a service scan.

The DNS scan will get any hostnames and Ips that are associated with the domain or IP. The service scan will try to get information on any services that the IP might have.

The top port scan will scan for common ports to see if any ports are opened on the machine. The code is also able to perform subdomain3 scans. The code includes a wrapper for the subdomain3 code. After the subdomain3 scan is done, the code will move the results of the subdomain3 scan into the "scans/ip/". The code will read the CSV and add the data to a HTML table.

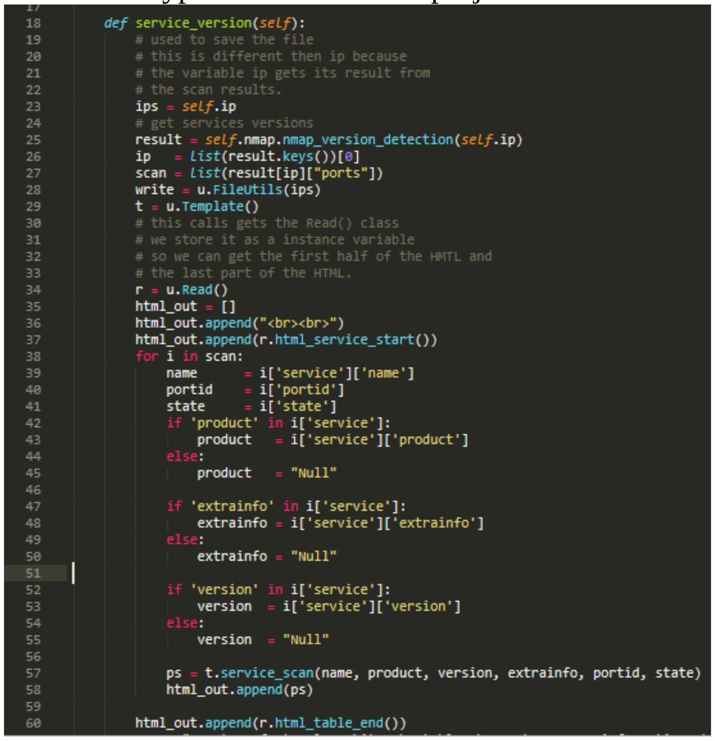
This tool purpose was to make it easy for the CCDC team to scan a target and create a report that can be turned into during the competition. The project will compile a report in HTML which the team members of the CCDC team could edit the HTML file adding more information about the scans. The team members will need to convert the HTML into a PDF to turn in to the judges.

To create a valid HTML file the code has HTML templates that the program will use to make a HTML file piece by piece. First it will call a method that will add the HTML header into the array. Next will call a method that will add code to start the HTML table. Then the code will call a method that was designed for each scan that will convert the json into a HTML row which it will loop through each of the results and add them to the array.

Another feature of the program is the ability to directory scan the target. It will take the IP of the host and use the list file to check to see if the website responds with a status code of 200. If the website does respond with a status code of 200, the program will add the url and the web directory path to the file. I did not put the results of the directory scan into a HTML file because it would probably be long. The results are stored in a text file in the website's directory.

Implementation

The nmaps.py file is the file that should be ran. It takes two instance variables; the IP instance variable and it also creates and store the NMAP module class instance. It will use the Scan class instance to call methods that will run the nmap scan. Inside the Scan class there is a method for each of the types of scan that the project can run.



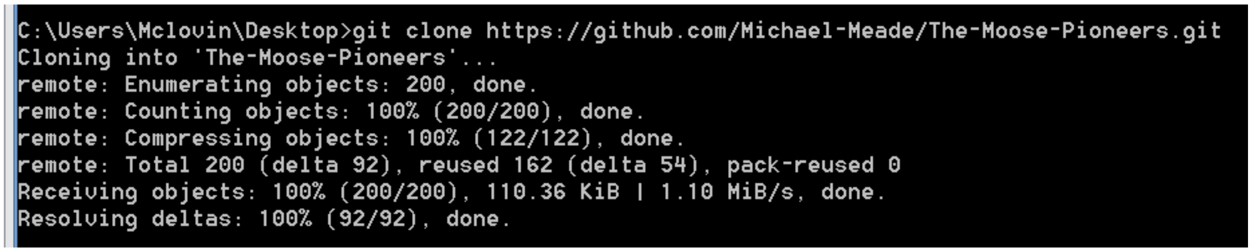
```python
    def service_version(self):
        # used to save the file
        # this is different then ip because
        # the variable ip gets its result from
        # the scan results.
        ips = self.ip
        # get services versions
        result = self.nmap.nmap_version_detection(self.ip)
        ip   = list(result.keys())[0]
        scan = list(result[ip]["ports"])
        write = u.FileUtils(ips)
        t = u.Template()
        # this calls gets the Read() class
        # we store it as a instance variable
        # so we can get the first half of the HMTL and
        # the last part of the HTML.
        r = u.Read()
        html_out = []
        html_out.append("<br><br>")
        html_out.append(r.html_service_start())
        for i in scan:
            name      = i['service']['name']
            portid    = i['portid']
            state     = i['state']
            if 'product' in i['service']:
                product   = i['service']['product']
            else:
                product   = "Null"

            if 'extrainfo' in i['service']:
                extrainfo = i['service']['extrainfo']
            else:
                extrainfo = "Null"

            if 'version' in i['service']:
                version  = i['service']['version']
            else:
                version  = "Null"

            ps = t.service_scan(name, product, version, extrainfo, portid, state)
            html_out.append(ps)

        html_out.append(r.html_table_end())
```

Figure 1, shows the service_scan method

Program Execution and Results

Running the nmaps.py script will perform all the NMAP scan as well as the subdomain3 class. To brute force the web directory, the dir_scanner.py file needs to be ran. The script takes one argument which is python dir_scanner.py -H https://utica.edu.

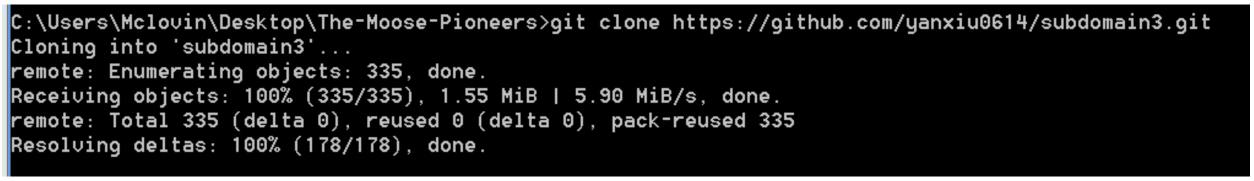
To install the package the user first has run the following command to git clone the program on their system.



```
C:\Users\Mclovin\Desktop>git clone https://github.com/Michael-Meade/The-Moose-Pioneers.git
Cloning into 'The-Moose-Pioneers'...
remote: Enumerating objects: 200, done.
remote: Counting objects: 100% (200/200), done.
remote: Compressing objects: 100% (122/122), done.
remote: Total 200 (delta 92), reused 162 (delta 54), pack-reused 0
Receiving objects: 100% (200/200), 110.36 KiB | 1.10 MiB/s, done.
Resolving deltas: 100% (92/92), done.
```

Figure 1, git cloning the program.

Figure 1 shows the user git cloning the program. Git must be already installed to use the git clone command. Users could also download the zip folder; they would have to unzip the file.
Git was used to keep track of what was done, it also allows the programmer to revert back any changes that is done without loosing any progress. Git also allows other users to access, add content to it because it is publicly available to anyone.



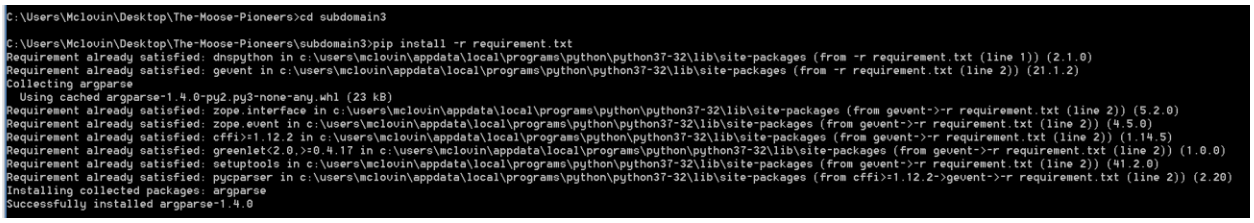
```
C:\Users\Mclovin\Desktop\The-Moose-Pioneers>git clone https://github.com/yanxiu0614/subdomain3.git
Cloning into 'subdomain3'...
remote: Enumerating objects: 335, done.
Receiving objects: 100% (335/335), 1.55 MiB | 5.90 MiB/s, done.
remote: Total 335 (delta 0), reused 0 (delta 0), pack-reused 335
Resolving deltas: 100% (178/178), done.
```

Figure 2, shows the git cloning of the subdomain script

The program that was created for this project also uses the subdomain3 package. Figure 2 shows the user git cloning the subdomain package. The subdomain3 folder has to cloned within the same directory that the The-Moose-Pioneers is located in. Nmap has to already be installed on the system to work. After you have installed nmap, you have to install the modules needed to use NMAP in python. Enter the following command.

`“pip install python3-nmap”`

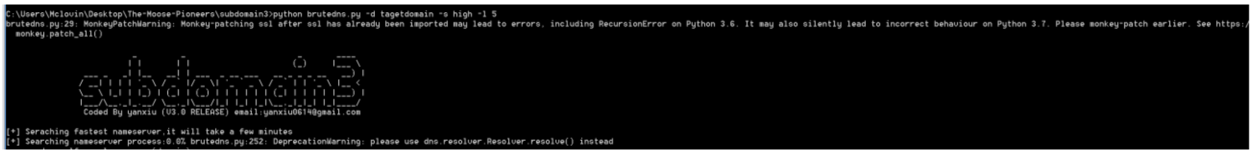
Now that we have git cloned Subdomain3 we must install the other dependences that are needed to run subdomain3.



```
C:\Users\Mclovin\Desktop\The-Moose-Pioneers>cd subdomain3

C:\Users\Mclovin\Desktop\The-Moose-Pioneers\subdomain3>pip install -r requirement.txt
Requirement already satisfied: dnspython in c:\users\mclovin\appdata\local\programs\python\python37-32\lib\site-packages (from -r requirement.txt (line 1)) (2.1.0)
Requirement already satisfied: gevent in c:\users\mclovin\appdata\local\programs\python\python37-32\lib\site-packages (from -r requirement.txt (line 2)) (21.1.2)
Collecting argparse
  Using cached argparse-1.4.0-py2.py3-none-any.whl (23 kB)
Requirement already satisfied: zope.interface in c:\users\mclovin\appdata\local\programs\python\python37-32\lib\site-packages (from gevent->-r requirement.txt (line 2)) (5.2.0)
Requirement already satisfied: zope.event in c:\users\mclovin\appdata\local\programs\python\python37-32\lib\site-packages (from gevent->-r requirement.txt (line 2)) (4.5.0)
Requirement already satisfied: cffi>=1.12.2 in c:\users\mclovin\appdata\local\programs\python\python37-32\lib\site-packages (from gevent->-r requirement.txt (line 2)) (1.14.5)
Requirement already satisfied: greenlet<2.0,>=0.4.17 in c:\users\mclovin\appdata\local\programs\python\python37-32\lib\site-packages (from gevent->-r requirement.txt (line 2)) (1.0.0)
Requirement already satisfied: setuptools in c:\users\mclovin\appdata\local\programs\python\python37-32\lib\site-packages (from gevent->-r requirement.txt (line 2)) (41.2.0)
Requirement already satisfied: pycparser in c:\users\mclovin\appdata\local\programs\python\python37-32\lib\site-packages (from cffi>=1.12.2->gevent->-r requirement.txt (line 2)) (2.20)
Installing collected packages: argparse
Successfully installed argparse-1.4.0
```

Figure 3 shows the installing of Subdomain3’s requirements.txt file.

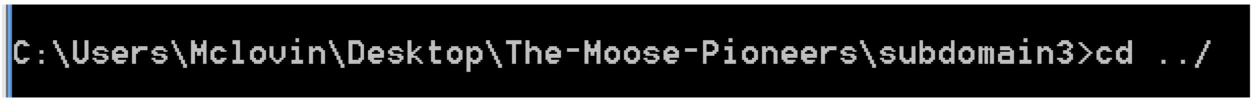
Figure 3 shows that first the user used the change directory command to change the directory to inside the subdomin3 directory. Next, we entered the following command, “pip install -r requirements.txt” this will install all the dependencies that are needed to use subdomain3.



```
C:\Users\Mclovin\Desktop\The-Moose-Pioneers\subdomain3>python brutedns.py -d tagetdomain -s high -l 5
brutedns.py:29: MonkeyPatchWarning: Monkey-patching ssl after ssl has already been imported may lead to errors, including RecursionError on Python 3.6. It may also silently lead to incorrect behaviour on Python 3.7. Please monkey-patch earlier. See https:/
  monkey.patch_all()

           Coded By yanxiu (V3.0 RELEASE) email:yanxiu0614@gmail.com

[+] Seraching fastest nameserver,it will take a few minutes
[+] Searching nameserver process:0.0% brutedns.py:252: DeprecationWarning: please use dns.resolver.Resolver.resolve() instead
```

Figure 4, checking to make sure that subdomain3 is installed.

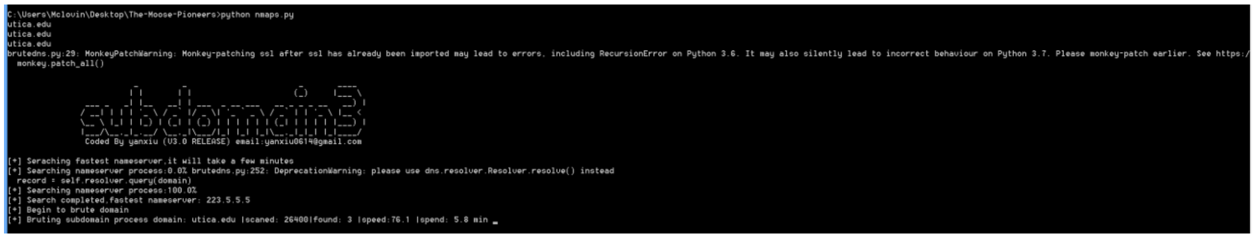
Before running the nmaps.py file it might be a good idea to make sure that Subdomain3 is correctly installed. Figure 4 shows the user running the command, “python2/3 brutedns.py -d tagetdomain -s high -l 5”. Figure 4 shows what the Subdomain3 script should display if Subdomain3 dependencies are installed.



```
C:\Users\Mclovin\Desktop\The-Moose-Pioneers\subdomain3>cd ../
```

Figure 5 shows the user going back a directory.

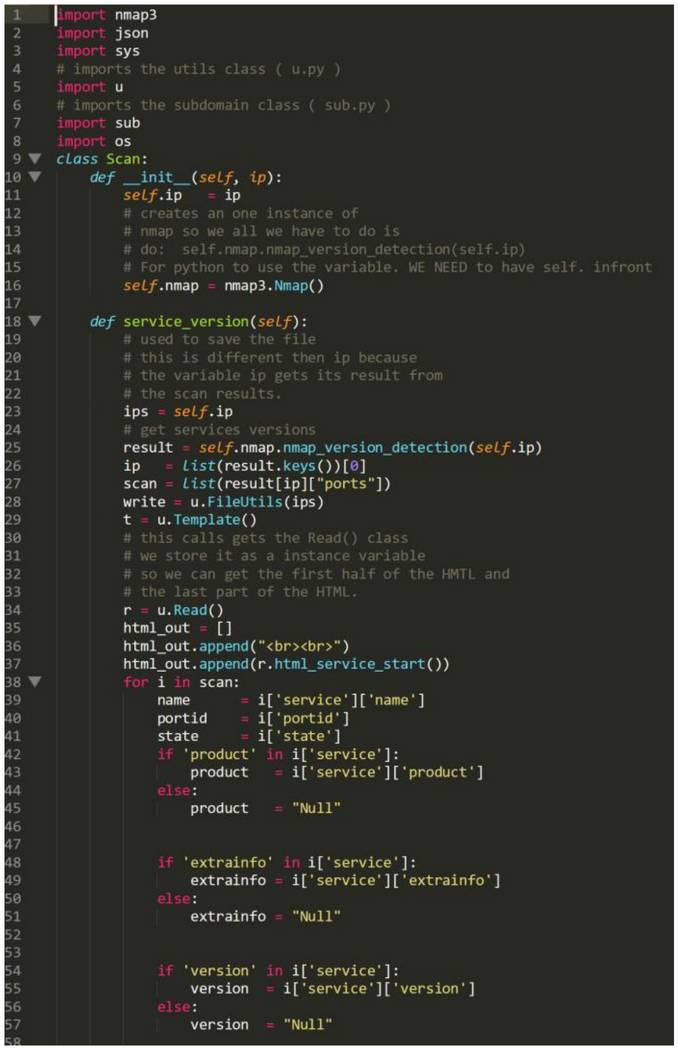
Now we must go back a directory back to the main directory. To do this we will need to enter the following command, “cd ../”



```
C:\Users\Mclovin\Desktop\The-Moose-Pioneers>python nmaps.py
utica.edu
utica.edu
utica.edu
brutedns.py:29: MonkeyPatchWarning: Monkey-patching ssl after ssl has already been imported may lead to errors, including RecursionError on Python 3.6. It may also silently lead to incorrect behaviour on Python 3.7. Please monkey-patch earlier. See https:/
  monkey.patch_all()

                    Coded By yanxiu (V3.0 RELEASE) email:yanxiu0614@gmail.com

[+] Seraching fastest nameserver,it will take a few minutes
[+] Searching nameserver process:0.0% brutedns.py:252: DeprecationWarning: please use dns.resolver.Resolver.resolve() instead
  record = self.resolver.query(domain)
[+] Searching nameserver process:100.0%
[+] Search completed,fastest nameserver: 223.5.5.5
[+] Begin to brute domain
[+] Bruting subdomain process domain: utica.edu |scaned: 26400|found: 3 |speed:76.1 |spend: 5.8 min _
```

Figure 6 shows the tool being ran

Figure 6 shows the tool being ran. The first few lines are from the NMAP scan. The last text that is printed is the subdomain3 being ran on the target.



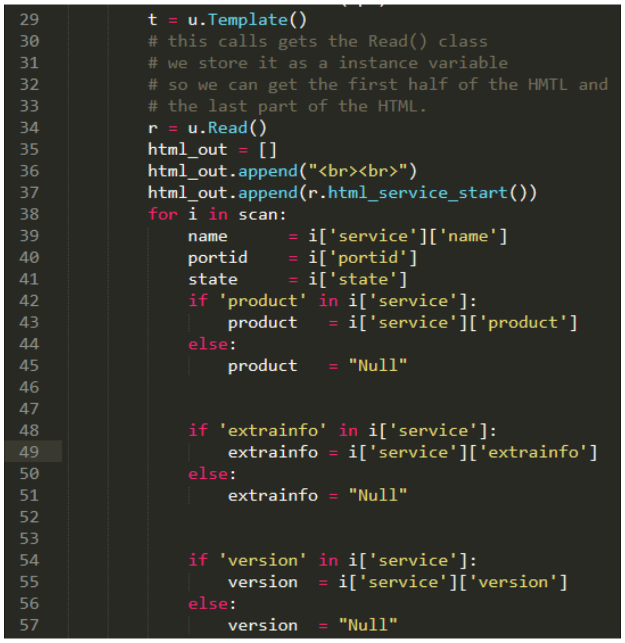
```
1   import nmap3
2   import json
3   import sys
4   # imports the utils class ( u.py )
5   import u
6   # imports the subdomain class ( sub.py )
7   import sub
8   import os
9   class Scan:
10      def __init__(self, ip):
11          self.ip   = ip
12          # creates an one instance of
13          # nmap so we all we have to do is
14          # do:  self.nmap.nmap_version_detection(self.ip)
15          # For python to use the variable. WE NEED to have self. infront
16          self.nmap = nmap3.Nmap()
17
18      def service_version(self):
19          # used to save the file
20          # this is different then ip because
21          # the variable ip gets its result from
22          # the scan results.
23          ips = self.ip
24          # get services versions
25          result = self.nmap.nmap_version_detection(self.ip)
26          ip   = list(result.keys())[0]
27          scan = list(result[ip]["ports"])
28          write = u.FileUtils(ips)
29          t = u.Template()
30          # this calls gets the Read() class
31          # we store it as a instance variable
32          # so we can get the first half of the HMTL and
33          # the last part of the HTML.
34          r = u.Read()
35          html_out = []
36          html_out.append("<br><br>")
37          html_out.append(r.html_service_start())
38          for i in scan:
39              name      = i['service']['name']
40              portid    = i['portid']
41              state     = i['state']
42              if 'product' in i['service']:
43                  product   = i['service']['product']
44              else:
45                  product   = "Null"
46
47
48              if 'extrainfo' in i['service']:
49                  extrainfo = i['service']['extrainfo']
50              else:
51                  extrainfo = "Null"
52
53
54              if 'version' in i['service']:
55                  version  = i['service']['version']
56              else:
57                  version  = "Null"
58
```

Figure 7. Variables

The python code in Figure 7, shows what the Scan class looks like. This is the class that will define all the methods that we use later to do the NMAP scanning. It takes two instances variables, ip and we also create a variable named ip nmap. We will use the ip variable to keep track of the ip that we want to scan. With the nmap instance variable we call and store the class as the variable nmap The nmap variable will be called by each method so we can run the NMAP scan. Line 18 is where we define the method, service_scan. This method will scan the IP and gather up information about the services that are running on the IP. Line 26 of figure 7 is where we use the instance variable “nmap” to access the nmap_version_detection method, which will run code that will use NMAP’s -sV argument. This type of scan will get information about the services that are running on the host.

We store the results of the scan as the variable results. Next, we use Python’s list method to convert the results into a list. We then use Python’s keys() method to get all the keys of the list. NMAP gives us the results in JSON, with the first key of the hash is always the IP of the host that was scanned. This is needed later so we can access other keys and values of the JSON. The code will store the value of the first key as the variable ip since its value is the IP of the host . On line 27, we do something like what we did on line 26 but instead we use we used the ip variable to access the other part of the JSON. We also get the values of the ports key. This key will give us all the results of the scan. We store the results as the variable scan.

On the top of the script, we imported a few classes and some modules. One of the classes we imported is the FileUtils() class. This class makes it easy to save information as a couple of different file types. On line 29 we also call the template class. Each scan method has a different method that will turn the input into HTML rows. When the scan is finish, the code will loop through all the results and turn it into a HTML row. The method will return the HTML row code which it will append into the output file. Which will create a valid HTML page of the results.

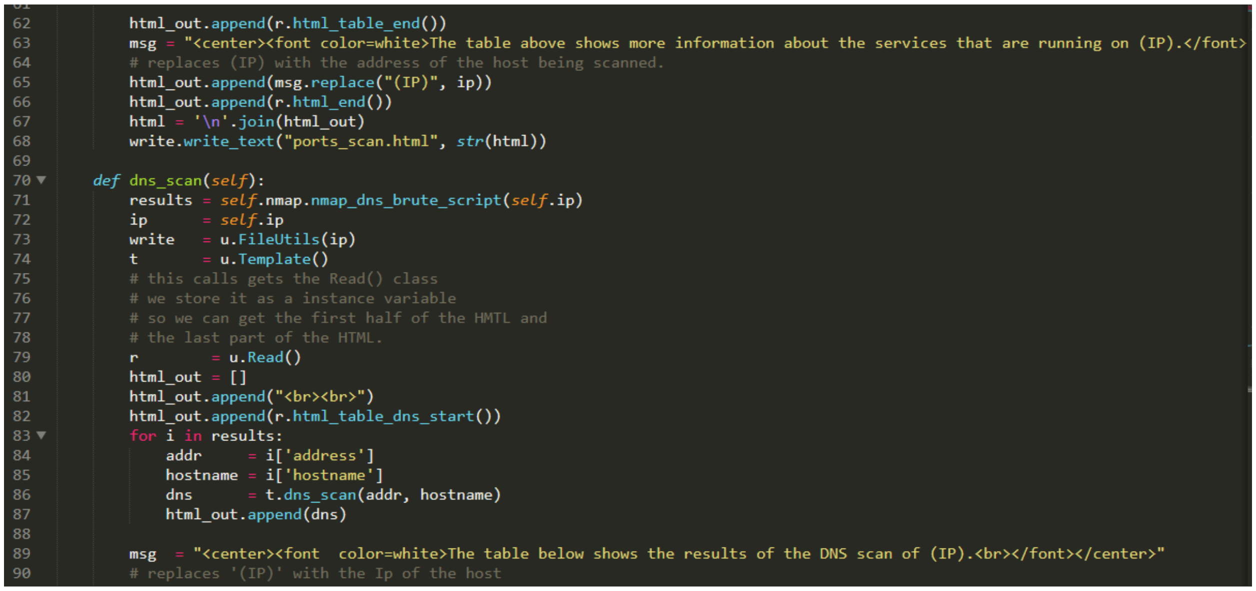


```
29          t = u.Template()
30          # this calls gets the Read() class
31          # we store it as a instance variable
32          # so we can get the first half of the HMTL and
33          # the last part of the HTML.
34          r = u.Read()
35          html_out = []
36          html_out.append("<br><br>")
37          html_out.append(r.html_service_start())
38          for i in scan:
39              name       = i['service']['name']
40              portid     = i['portid']
41              state      = i['state']
42              if 'product' in i['service']:
43                  product   = i['service']['product']
44              else:
45                  product   = "Null"
46
47
48              if 'extrainfo' in i['service']:
49                  extrainfo = i['service']['extrainfo']
50              else:
51                  extrainfo = "Null"
52
53
54              if 'version' in i['service']:
55                  version  = i['service']['version']
56              else:
57                  version  = "Null"
```

Figure 8, shows what Figure 7 output looks like

In Figure 8 on line 34 we start an instance of the Read() class which can be accessed through the r variable. On the next line we create an empty array named html_out. The array will hold all the html that will added to the file. On line 36 we add “<br><br>” to the html_out. On the next line we call the html_service_start() method that is inside the Read class. The html_service_start method will read the file located in templates/html_template_start.html file and append it to the array we created at line 35. The file contains the header and stuff that is needed to create a valid HTML file.

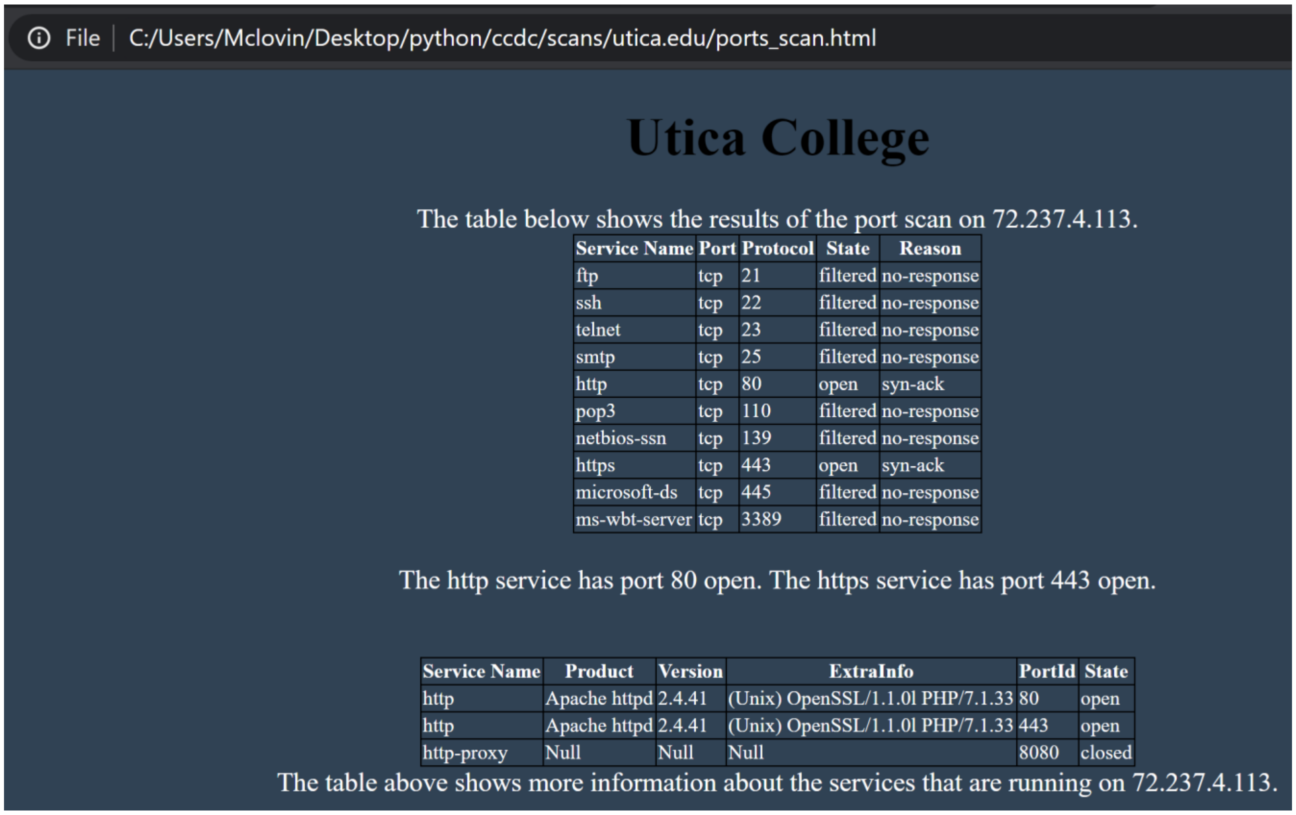
Line 38 of Figure 8, will take the scans variable which is where the results of the scan is located and loops through the JSON, storing the results as the variable i. On the next line we access the JSON so we can get the name of the service that was found. We store this value as the variable named name. We do the same thing with the variables portid and state. On line 42 we check to see if the JSON has the key to products. If it does not have the key, then we give it the value of Null. We do the same thing for extrainfo and version. This just makes sure that the HTML does not look empty.



```
62          html_out.append(r.html_table_end())
63          msg = "<center><font color=white>The table above shows more information about the services that are running on (IP).</font>
64          # replaces (IP) with the address of the host being scanned.
65          html_out.append(msg.replace("(IP)", ip))
66          html_out.append(r.html_end())
67          html = '\n'.join(html_out)
68          write.write_text("ports_scan.html", str(html))
69
70      def dns_scan(self):
71          results = self.nmap.nmap_dns_brute_script(self.ip)
72          ip      = self.ip
73          write   = u.FileUtils(ip)
74          t       = u.Template()
75          # this calls gets the Read() class
76          # we store it as a instance variable
77          # so we can get the first half of the HMTL and
78          # the last part of the HTML.
79          r        = u.Read()
80          html_out = []
81          html_out.append("<br><br>")
82          html_out.append(r.html_table_dns_start())
83          for i in results:
84              addr     = i['address']
85              hostname = i['hostname']
86              dns      = t.dns_scan(addr, hostname)
87              html_out.append(dns)
88
89          msg  = "<center><font  color=white>The table below shows the results of the DNS scan of (IP).<br></font></center>"
90          # replaces '(IP)' with the Ip of the host
```

Figure 9 shows last part of the server_version.

On line 62, we use the r variable to call the html_table_end method so we can end the HTML table. On line 63 is where we create a variable that has a basic message stating that we found an open service and the IP and the name of the service. On the next line we use Python's replace method to replace the occurrence of "(IP)" with the IP of the host that was scanned. We add the replace message to the array, html_out. Lastly, we close up the HTML file by using the r variable to access the html_out() method. This will close the html File. Next, we use the join method to make the array not an array anymore. We store this as the variable html. On line 68 we use the FileUtils class to run the write_text method, which will save the contents of the html variable to file.


File | C:/Users/Mclovin/Desktop/python/ccdc/scans/utica.edu/ports_scan.html

Utica College

The table below shows the results of the port scan on 72.237.4.113.

Service Name	Port	Protocol	State	Reason
ftp	tcp	21	filtered	no-response
ssh	tcp	22	filtered	no-response
telnet	tcp	23	filtered	no-response
smtp	tcp	25	filtered	no-response
http	tcp	80	open	syn-ack
pop3	tcp	110	filtered	no-response
netbios-ssn	tcp	139	filtered	no-response
https	tcp	443	open	syn-ack
microsoft-ds	tcp	445	filtered	no-response
ms-wbt-server	tcp	3389	filtered	no-response

The http service has port 80 open. The https service has port 443 open.

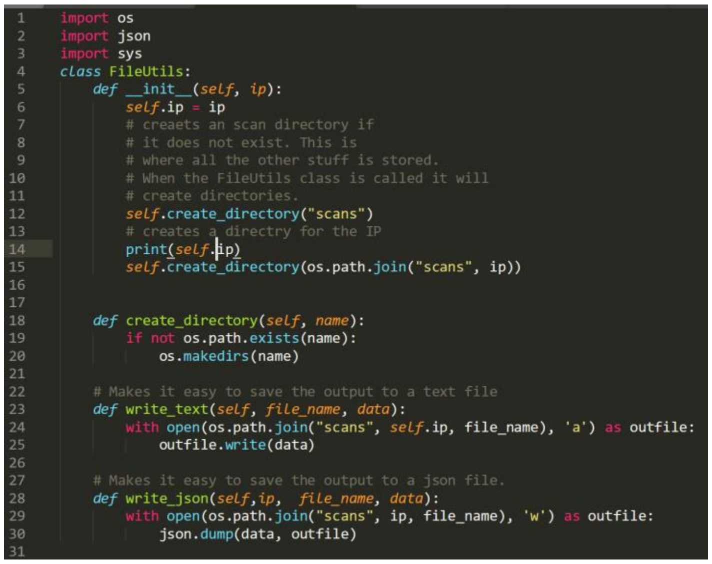
Service Name	Product	Version	ExtraInfo	PortId	State
http	Apache httpd	2.4.41	(Unix) OpenSSL/1.1.0l PHP/7.1.33	80	open
http	Apache httpd	2.4.41	(Unix) OpenSSL/1.1.0l PHP/7.1.33	443	open
http-proxy	Null	Null	Null	8080	closed

The table above shows more information about the services that are running on 72.237.4.113.

Figure 10, shows the HTML tables of the results

Figure 10 shows the results of scan in the HTML table we created in Figure 9. Figure 9 shows the status of all the ports that was detected. Below the table is a short message identifying what open ports were found and the name of the service. Figure 10 shows that when scanning Utica.edu it found two open ports, port 80 & port 443 and the type of service.

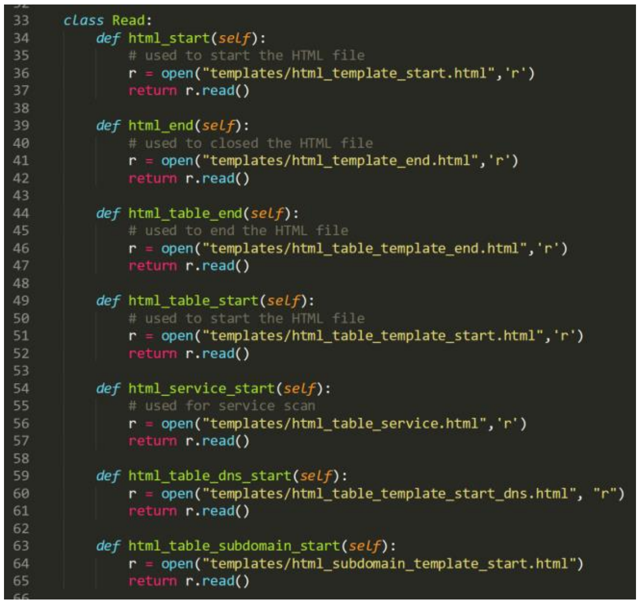
The next table found in Figure 10 shows the results of the service scan. The table shows the service type, the product, the version, extra information if there is any, the port id and the state of the port.



```python
import os
import json
import sys
class FileUtils:
    def __init__(self, ip):
        self.ip = ip
        # creaets an scan directory if
        # it does not exist. This is
        # where all the other stuff is stored.
        # When the FileUtils class is called it will
        # create directories.
        self.create_directory("scans")
        # creates a directry for the IP
        print(self.ip)
        self.create_directory(os.path.join("scans", ip))


    def create_directory(self, name):
        if not os.path.exists(name):
            os.makedirs(name)

    # Makes it easy to save the output to a text file
    def write_text(self, file_name, data):
        with open(os.path.join("scans", self.ip, file_name), 'a') as outfile:
            outfile.write(data)

    # Makes it easy to save the output to a json file.
    def write_json(self,ip,  file_name, data):
        with open(os.path.join("scans", ip, file_name), 'w') as outfile:
            json.dump(data, outfile)
```

Figure 11 shows the FileUtils class.

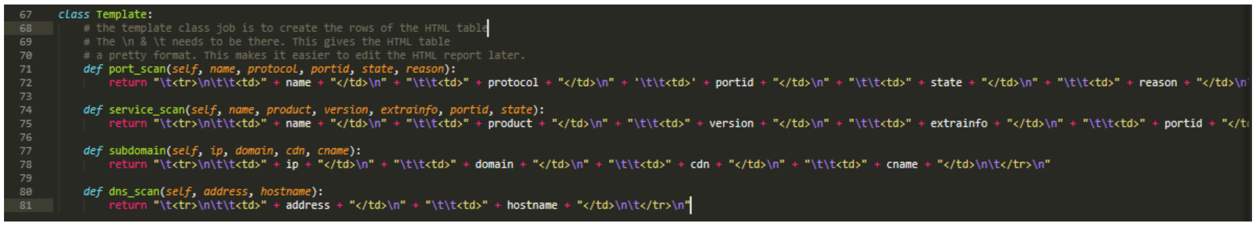
Figure 11 shows the FileUtils class that is in the u.py file. The u,py file has a couple of different classes in it. Figure 5 shows the FileUtils class which allows us to easily export any information. The script is able save output in JSON or text file. Another feature of this class is that when the class is called it will automagically create a directory named scans if it does not exist. The script will also create a new directory in the scan directory with the name of the IP or site that was scanned. This is to ensure that the scripts will never run into any errors of the directory not existing.



```python
class Read:
    def html_start(self):
        # used to start the HTML file
        r = open("templates/html_template_start.html",'r')
        return r.read()

    def html_end(self):
        # used to closed the HTML file
        r = open("templates/html_template_end.html",'r')
        return r.read()

    def html_table_end(self):
        # used to end the HTML file
        r = open("templates/html_table_template_end.html",'r')
        return r.read()

    def html_table_start(self):
        # used to start the HTML file
        r = open("templates/html_table_template_start.html",'r')
        return r.read()

    def html_service_start(self):
        # used for service scan
        r = open("templates/html_table_service.html",'r')
        return r.read()

    def html_table_dns_start(self):
        r = open("templates/html_table_template_start_dns.html", "r")
        return r.read()

    def html_table_subdomain_start(self):
        r = open("templates/html_subdomain_template_start.html")
        return r.read()
```

Figure 12, shows the Read class

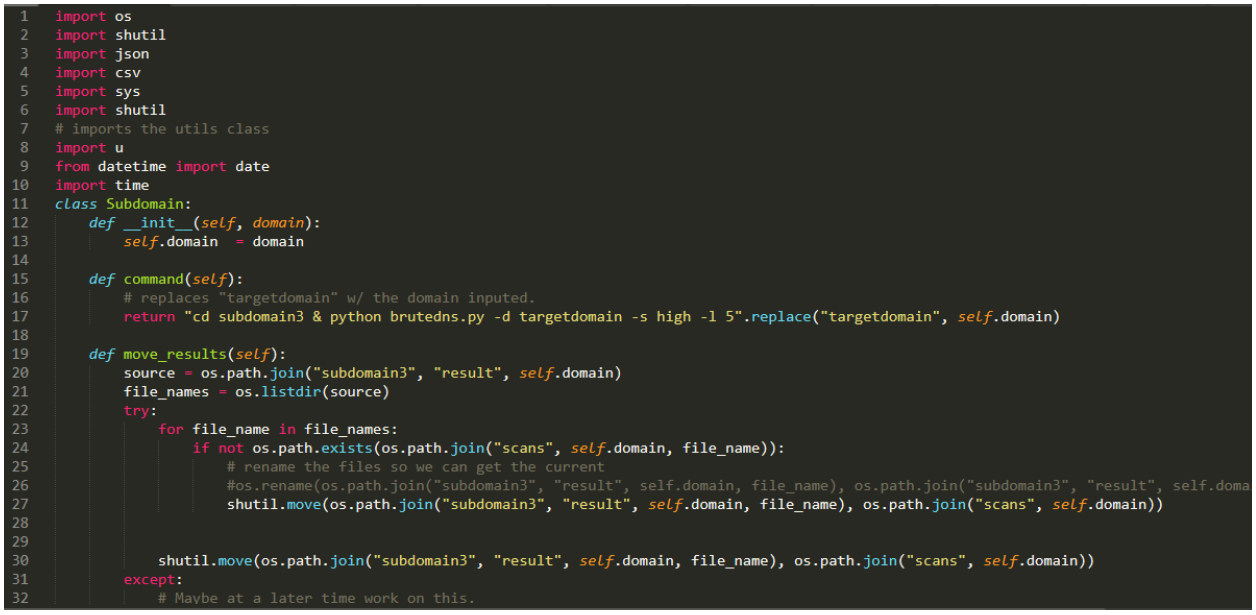
Figure 12 shows Read class which is still apart of the u.py file. This class allows us to create a valid HTML file, table. Each method in this class does something different like add the start of a HTML file, end a HTML table, start a HTML table, end a HMTL table. Earlier in Figure 9 is where we used these methods to systematically call each of the methods to make a valid html file.



```
class Template:
    # the template class job is to create the rows of the HTML table
    # The \n & \t needs to be there. This gives the HTML table
    # a pretty format. This makes it easier to edit the HTML report later.
    def port_scan(self, name, protocol, portid, state, reason):
        return "\t<tr>\n\t\t<td>" + name + "</td>\n" + "\t\t<td>" + protocol + "</td>\n" + '\t\t<td>' + portid + "</td>\n" + "\t\t<td>" + state + "</td>\n" + "\t\t<td>" + reason + "</td>\n

    def service_scan(self, name, product, version, extrainfo, portid, state):
        return "\t<tr>\n\t\t<td>" + name + "</td>\n" + "\t\t<td>" + product + "</td>\n" + "\t\t<td>" + version + "</td>\n" + "\t\t<td>" + extrainfo + "</td>\n" + "\t\t<td>" + portid + "</t

    def subdomain(self, ip, domain, cdn, cname):
        return "\t<tr>\n\t\t<td>" + ip + "</td>\n" + "\t\t<td>" + domain + "</td>\n" + "\t\t<td>" + cdn + "</td>\n" + "\t\t<td>" + cname + "</td>\n\t</tr>\n"

    def dns_scan(self, address, hostname):
        return "\t<tr>\n\t\t<td>" + address + "</td>\n" + "\t\t<td>" + hostname + "</td>\n\t</tr>\n"
```

Figure 13, shows how the script will create each line of the HTML Table.

Each type of scan has their own method inside this class. The method purpose is to turn the data into a HTML row. The rows is added to the html_out array.

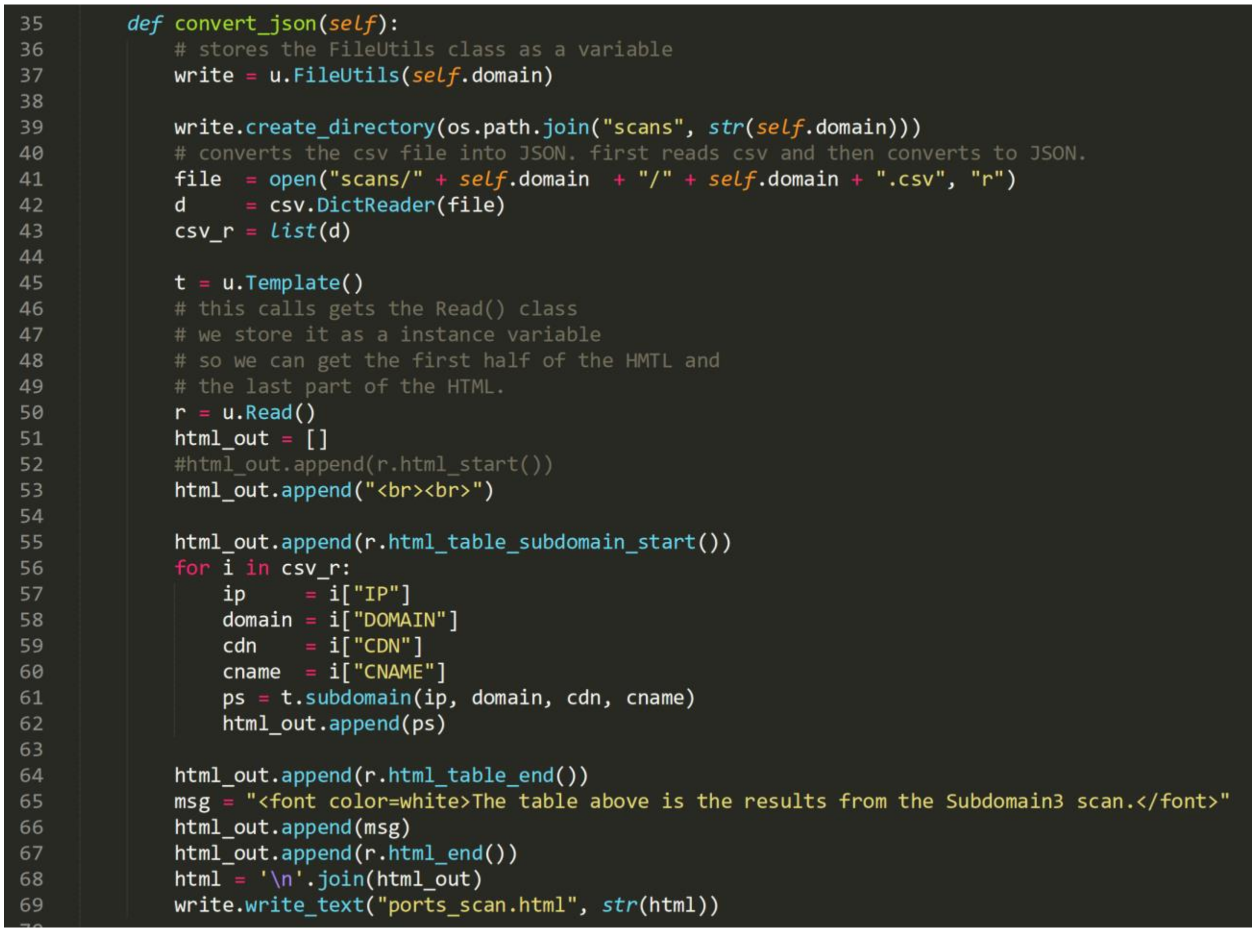


```
import os
import shutil
import json
import csv
import sys
import shutil
# imports the utils class
import u
from datetime import date
import time
class Subdomain:
    def __init__(self, domain):
        self.domain  = domain

    def command(self):
        # replaces "targetdomain" w/ the domain inputed.
        return "cd subdomain3 & python brutedns.py -d targetdomain -s high -l 5".replace("targetdomain", self.domain)

    def move_results(self):
        source = os.path.join("subdomain3", "result", self.domain)
        file_names = os.listdir(source)
        try:
            for file_name in file_names:
                if not os.path.exists(os.path.join("scans", self.domain, file_name)):
                    # rename the files so we can get the current
                    #os.rename(os.path.join("subdomain3", "result", self.domain, file_name), os.path.join("subdomain3", "result", self.doma
                    shutil.move(os.path.join("subdomain3", "result", self.domain, file_name), os.path.join("scans", self.domain))


            shutil.move(os.path.join("subdomain3", "result", self.domain, file_name), os.path.join("scans", self.domain))
        except:
            # Maybe at a later time work on this.
```

Figure 14, The subdomain3 class.

Figure 14 shows the subdomain class. This class is a wrapper for the Subdomain script. Subdomain3 should be installed the main directory. The first thing the class does is imports a few modules and the u file. The class takes one instance variable named domain. The first method named, command purpose is storing the command that will run subdomain script. The method uses Python’s replace method to replace the text “targetdomain” with the value of instance variable of domain.

The next method in the class is the move results. This method sole purpose is to move the results of the Subdomain scan from the directory that Subdomain uses to the one that our program uses.

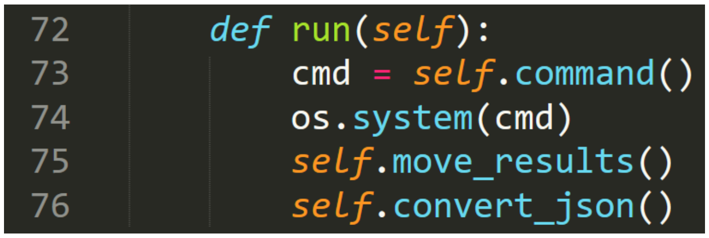


```
35      def convert_json(self):
36          # stores the FileUtils class as a variable
37          write = u.FileUtils(self.domain)
38
39          write.create_directory(os.path.join("scans", str(self.domain)))
40          # converts the csv file into JSON. first reads csv and then converts to JSON.
41          file  = open("scans/" + self.domain  + "/" + self.domain + ".csv", "r")
42          d     = csv.DictReader(file)
43          csv_r = list(d)
44
45          t = u.Template()
46          # this calls gets the Read() class
47          # we store it as a instance variable
48          # so we can get the first half of the HMTL and
49          # the last part of the HTML.
50          r = u.Read()
51          html_out = []
52          #html_out.append(r.html_start())
53          html_out.append("<br><br>")
54
55          html_out.append(r.html_table_subdomain_start())
56          for i in csv_r:
57              ip     = i["IP"]
58              domain = i["DOMAIN"]
59              cdn    = i["CDN"]
60              cname  = i["CNAME"]
61              ps = t.subdomain(ip, domain, cdn, cname)
62              html_out.append(ps)
63
64          html_out.append(r.html_table_end())
65          msg = "<font color=white>The table above is the results from the Subdomain3 scan.</font>"
66          html_out.append(msg)
67          html_out.append(r.html_end())
68          html = '\n'.join(html_out)
69          write.write_text("ports_scan.html", str(html))
```

Figure 15 shows the Subdomain's convert_json method.

Figure 15 shows convert_json method that belongs to the Subdomain class. Subdomain3 will output the results of the scan into a CSV file. This method converts the outputted CSV files into a JSON file. After the method converts the results into JSON, the code will call do the same thing it did in nmaps.py and use the Template() class and the Read() class to access the HTML templates and so that we can turn the results that we got from the JSON into HTML rows that we use to create the subdomain HTML table.

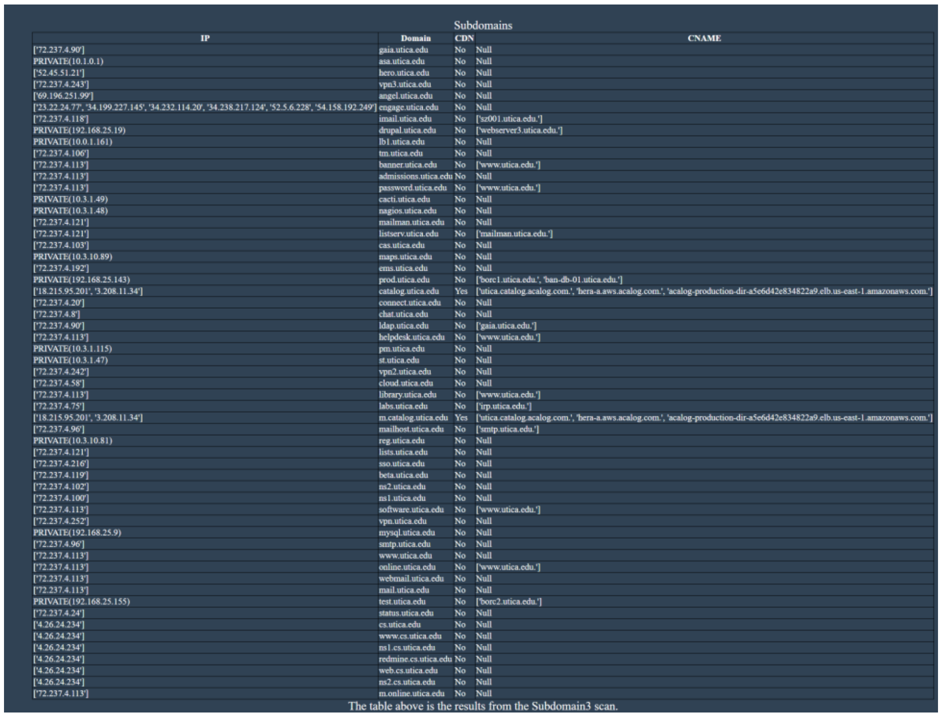
On line 64 of Figure 15 shows that the code calls the html_table_end method from the Read class to close the HTML table. Next, we create a variable named msg. The variable msg includes HTML that will center and turn the text color to white. The message included the msg variable is used to state that the table above is from the results of the Subdomain3 scan On line 66 is where we add the msg to the html_out array. On line 67 we call the r variable which allows us to access the html_end() method. Which will close the HTML file. On line 68 is where we turn the array that contains all the html table into the HTML file by using Python's join method to make each line is on a new line which it stores the output as the variable html. The last thing that the method does is save the information into the scans/website_name/ directory.



```python
    def run(self):
        cmd = self.command()
        os.system(cmd)
        self.move_results()
        self.convert_json()
```

Figure 16 shows the run method from the subs.py file.

Figure 16 shows the run method. This is the method that should be called to use the Subdomain class. First the code will access the commands() method. This method stores the command used to run the subdomain3 code as the variable cmd. Next the code uses Python's system method to run the command that will run the Subdomain3 code. After the scan is finished, the code will call the self.move_results() method which will move the results from the Subdomain folder to the folder for the other scans. Then the code calls the self.convert_json method which converts the results into JSON, convert the JSON into HTML and saves the results in the scan/website_name/scan_results.html.



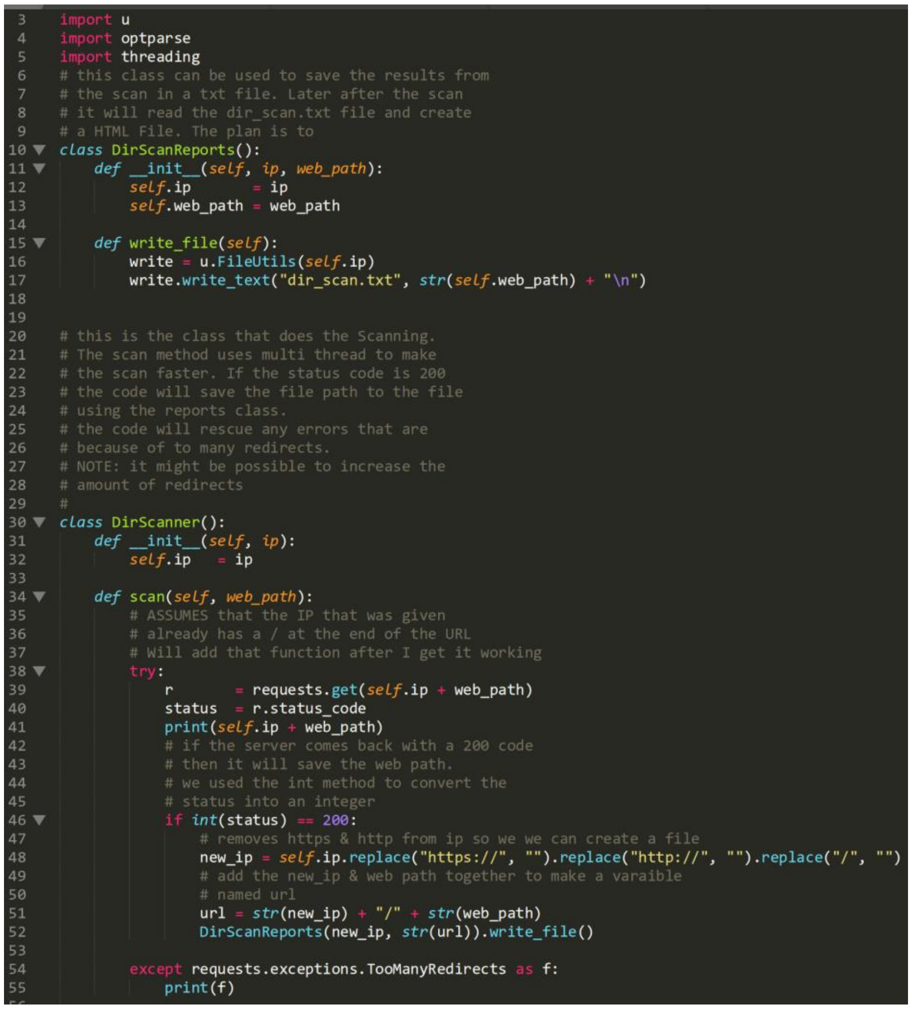
Subdomains

IP	Domain	CDN	CNAME
['72.237.4.90']	gaia.utica.edu	No	Null
PRIVATE(10.1.0.1)	asa.utica.edu	No	Null
['52.45.51.21']	hero.utica.edu	No	Null
['72.237.4.243']	vpn3.utica.edu	No	Null
['69.196.251.99']	angel.utica.edu	No	Null
['23.22.24.77', '34.199.227.145', '34.232.114.20', '34.238.217.124', '52.5.6.228', '54.158.192.249']	engage.utica.edu	No	Null
['72.237.4.118']	imail.utica.edu	No	['sz001.utica.edu.']
PRIVATE(192.168.25.19)	drupal.utica.edu	No	['webserver3.utica.edu.']
PRIVATE(10.0.1.161)	lb1.utica.edu	No	Null
['72.237.4.106']	tm.utica.edu	No	Null
['72.237.4.113']	banner.utica.edu	No	['www.utica.edu.']
['72.237.4.113']	admissions.utica.edu	No	Null
['72.237.4.113']	password.utica.edu	No	['www.utica.edu.']
PRIVATE(10.3.1.49)	cacti.utica.edu	No	Null
PRIVATE(10.3.1.48)	nagios.utica.edu	No	Null
['72.237.4.121']	mailman.utica.edu	No	Null
['72.237.4.121']	listserv.utica.edu	No	['mailman.utica.edu.']
['72.237.4.103']	cas.utica.edu	No	Null
PRIVATE(10.3.10.89)	maps.utica.edu	No	Null
['72.237.4.192']	ems.utica.edu	No	Null
PRIVATE(192.168.25.143)	prod.utica.edu	No	['borc1.utica.edu.', 'ban-db-01.utica.edu.']
['18.215.95.201', '3.208.11.34']	catalog.utica.edu	Yes	['utica.catalog.acalog.com.', 'hera-a.aws.acalog.com.', 'acalog-production-dir-a5e6d42e834822a9.elb.us-east-1.amazonaws.com.']
['72.237.4.20']	connect.utica.edu	No	Null
['72.237.4.8']	chat.utica.edu	No	Null
['72.237.4.90']	ldap.utica.edu	No	['gaia.utica.edu.']
['72.237.4.113']	helpdesk.utica.edu	No	['www.utica.edu.']
PRIVATE(10.3.1.115)	pm.utica.edu	No	Null
PRIVATE(10.3.1.47)	st.utica.edu	No	Null
['72.237.4.242']	vpn2.utica.edu	No	Null
['72.237.4.58']	cloud.utica.edu	No	Null
['72.237.4.113']	library.utica.edu	No	['www.utica.edu.']
['72.237.4.75']	labs.utica.edu	No	['irp.utica.edu.']
['18.215.95.201', '3.208.11.34']	m.catalog.utica.edu	Yes	['utica.catalog.acalog.com.', 'hera-a.aws.acalog.com.', 'acalog-production-dir-a5e6d42e834822a9.elb.us-east-1.amazonaws.com.']
['72.237.4.96']	mailhost.utica.edu	No	['smtp.utica.edu.']
PRIVATE(10.3.10.81)	reg.utica.edu	No	Null
['72.237.4.121']	lists.utica.edu	No	Null
['72.237.4.216']	sso.utica.edu	No	Null
['72.237.4.119']	beta.utica.edu	No	Null
['72.237.4.102']	ns2.utica.edu	No	Null
['72.237.4.100']	ns1.utica.edu	No	Null
['72.237.4.113']	software.utica.edu	No	['www.utica.edu.']
['72.237.4.252']	vpn.utica.edu	No	Null
PRIVATE(192.168.25.9)	mysql.utica.edu	No	Null
['72.237.4.96']	smtp.utica.edu	No	Null
['72.237.4.113']	www.utica.edu	No	Null
['72.237.4.113']	online.utica.edu	No	['www.utica.edu.']
['72.237.4.113']	webmail.utica.edu	No	Null
['72.237.4.113']	mail.utica.edu	No	Null
PRIVATE(192.168.25.155)	test.utica.edu	No	['borc2.utica.edu.']
['72.237.4.24']	status.utica.edu	No	Null
['4.26.24.234']	cs.utica.edu	No	Null
['4.26.24.234']	www.cs.utica.edu	No	Null
['4.26.24.234']	ns1.cs.utica.edu	No	Null
['4.26.24.234']	redmine.cs.utica.edu	No	Null
['4.26.24.234']	web.cs.utica.edu	No	Null
['4.26.24.234']	ns2.cs.utica.edu	No	Null
['72.237.4.113']	m.online.utica.edu	No	Null

The table above is the results from the Subdomain3 scan.

Figure 17, shows the subdomain results

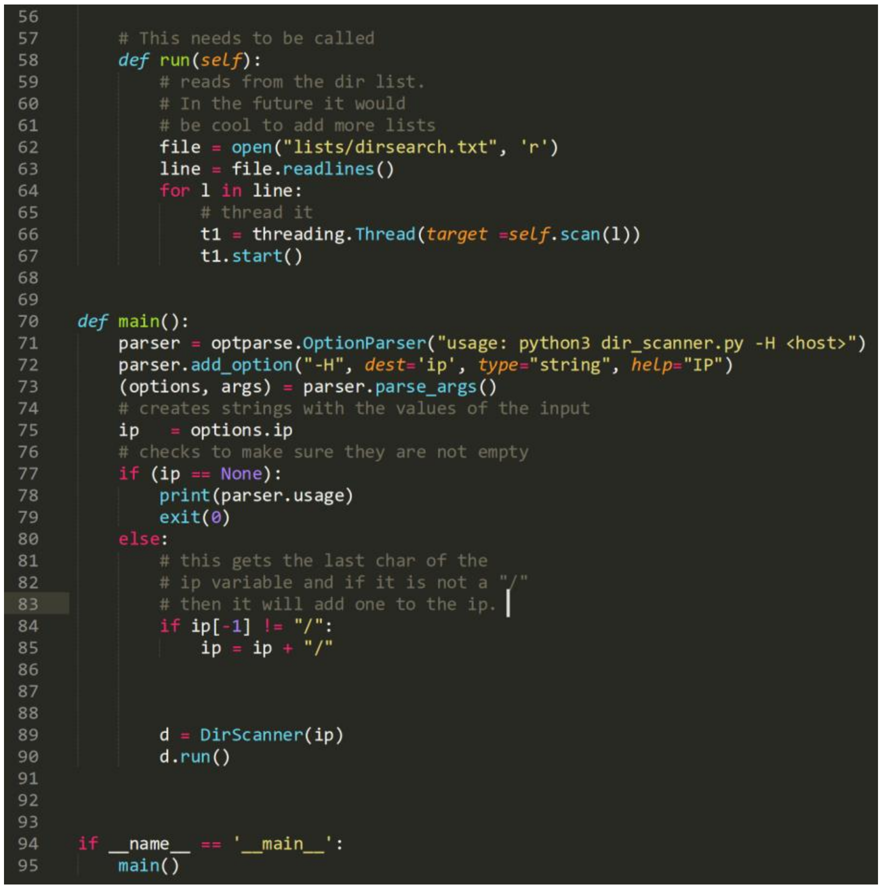
Figure 17 shows all the subdomains we found for Utica.edu. The table includes data like the IP of the subdomain, the URL domain of the subdomains, if the subdomain is a CDN and the cname of the detected subdomain.



```
3   import u
4   import optparse
5   import threading
6   # this class can be used to save the results from
7   # the scan in a txt file. Later after the scan
8   # it will read the dir_scan.txt file and create
9   # a HTML File. The plan is to
10  class DirScanReports():
11      def __init__(self, ip, web_path):
12          self.ip       = ip
13          self.web_path = web_path
14
15      def write_file(self):
16          write = u.FileUtils(self.ip)
17          write.write_text("dir_scan.txt", str(self.web_path) + "\n")
18
19
20  # this is the class that does the Scanning.
21  # The scan method uses multi thread to make
22  # the scan faster. If the status code is 200
23  # the code will save the file path to the file
24  # using the reports class.
25  # the code will rescue any errors that are
26  # because of to many redirects.
27  # NOTE: it might be possible to increase the
28  # amount of redirects
29  #
30  class DirScanner():
31      def __init__(self, ip):
32          self.ip   = ip
33
34      def scan(self, web_path):
35          # ASSUMES that the IP that was given
36          # already has a / at the end of the URL
37          # Will add that function after I get it working
38          try:
39              r       = requests.get(self.ip + web_path)
40              status  = r.status_code
41              print(self.ip + web_path)
42              # if the server comes back with a 200 code
43              # then it will save the web path.
44              # we used the int method to convert the
45              # status into an integer
46              if int(status) == 200:
47                  # removes https & http from ip so we we can create a file
48                  new_ip = self.ip.replace("https://", "").replace("http://", "").replace("/", "")
49                  # add the new_ip & web path together to make a varaible
50                  # named url
51                  url = str(new_ip) + "/" + str(web_path)
52                  DirScanReports(new_ip, str(url)).write_file()
53
54          except requests.exceptions.TooManyRedirects as f:
55              print(f)
```

Figure 18, shows the DirScan class

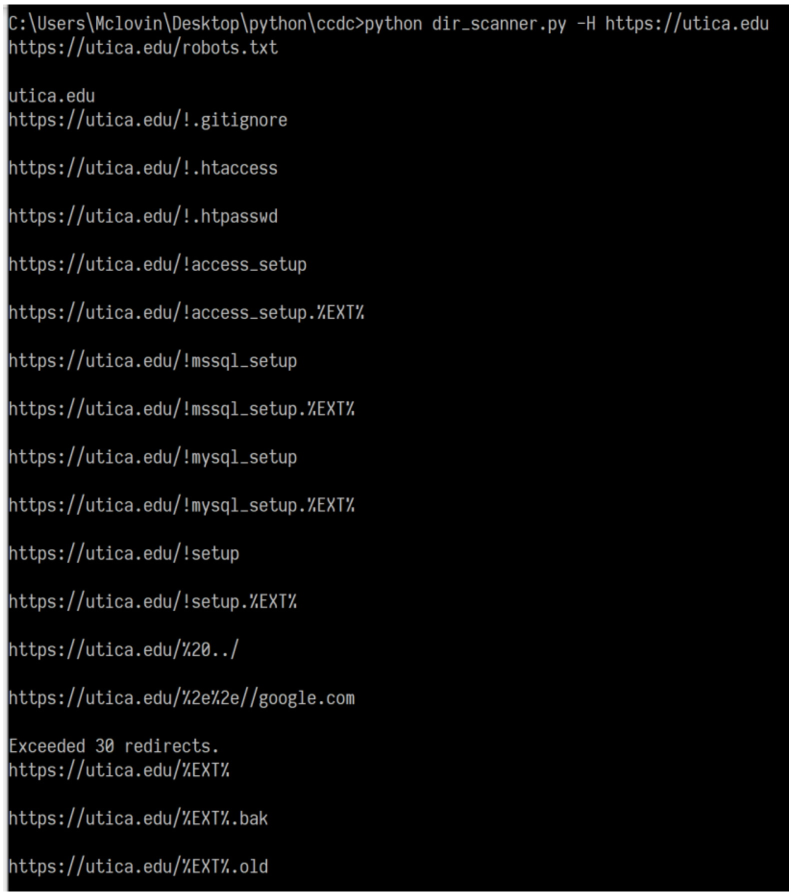
Figure 18 shows the DirScan class, which is in dir_scanner.py. The class takes on instance variable which is the self.ip. The ip instance is used to store the URL that we want to scan. The first method in the class is named scan method. This method takes one argument named web_path. Later, the class we loop through each line of a file that contains a web path. The web_path argument is used to create the URL that we will scan. The DirScan classes uses Python request module to send the requests to the site. On line 19 we create the URL and send the request to the site. After that is done the code will store the status code as the variable named status. On line 16 is where we call the status variable and convert it into an integer which it checks to see if it is equal to 200. A 200-status code means that the Url request successfully went through which means that the url exists. Note there might be some a false alert because a site could respond with a 200 status, but the page actually just displays an error or something that says that the content was not found. If the site responds with a 200 code the code will replace any strings that matches “https://” . “http://” and “/”.
On line 52 the code calls the DirScanReports class. The DirScanReports class will save the url to a file named dir_scan.txt.



```
56
57      # This needs to be called
58      def run(self):
59          # reads from the dir list.
60          # In the future it would
61          # be cool to add more lists
62          file = open("lists/dirsearch.txt", 'r')
63          line = file.readlines()
64          for l in line:
65              # thread it
66              t1 = threading.Thread(target =self.scan(l))
67              t1.start()
68
69
70  def main():
71      parser = optparse.OptionParser("usage: python3 dir_scanner.py -H <host>")
72      parser.add_option("-H", dest='ip', type="string", help="IP")
73      (options, args) = parser.parse_args()
74      # creates strings with the values of the input
75      ip    = options.ip
76      # checks to make sure they are not empty
77      if (ip == None):
78          print(parser.usage)
79          exit(0)
80      else:
81          # this gets the last char of the
82          # ip variable and if it is not a "/"
83          # then it will add one to the ip.
84          if ip[-1] != "/":
85              ip = ip + "/"
86
87
88
89          d = DirScanner(ip)
90          d.run()
91
92
93
94  if __name__ == '__main__':
95      main()
```

Figure 19 shows the end part of the dir_scanner file.

Figure 19 shows the end of the dir_scanner file. The run method of the DirScan class firsts opens the file located at “lists/dirsearch.txt” and will loop through each line of the file. On line 66 the script uses Multithread which makes the scanner able to scan faster. On line 64 is where we start the multithread. The main() method is not apart of the class but is ran when the file is ran due to the code on line 94. The script uses optparse to take the input so that we can scan the domain. On line 77, the code will check to see if the ip variable has a value of None. If it does have the value of None then the code will display the usage. On line 84 is where the code gets the last character of the variable named ip and check if the last character is “/”. If the last character does not have “/” as the last character, then the code will add it to the end of the URL. Line 89 is where we call the DirScan class and store it as the variable d. On the next line we use the variable d to access the run() method used by the DirScan class.



```
C:\Users\Mclovin\Desktop\python\ccdc>python dir_scanner.py -H https://utica.edu
https://utica.edu/robots.txt

utica.edu
https://utica.edu/!.gitignore

https://utica.edu/!.htaccess

https://utica.edu/!.htpasswd

https://utica.edu/!access_setup

https://utica.edu/!access_setup.%EXT%

https://utica.edu/!mssql_setup

https://utica.edu/!mssql_setup.%EXT%

https://utica.edu/!mysql_setup

https://utica.edu/!mysql_setup.%EXT%

https://utica.edu/!setup

https://utica.edu/!setup.%EXT%

https://utica.edu/%20../

https://utica.edu/%2e%2e//google.com

Exceeded 30 redirects.
https://utica.edu/%EXT%

https://utica.edu/%EXT%.bak

https://utica.edu/%EXT%.old
```

Figure 20 shows the dir scanner in action.

Figure 20 shows the results of the dir scanner directory. It will save the results in the "/scan/web_path/dir_scan.txt" if the server responds with a 200 code.

Conclusion

Being able to perform automated scans on a ip or ips can be a proactive way of finding vulnerabilities that may lay in the network or even old servers that are not in use anymore before someone malicious found the security vulnerabilities and exploited the service. Some of the things I learned while working on the project is how JSON work. Another thing I had to read up on is how multi-threading works.

Work cited

Json - JSON encoder and decoder¶. (n.d.). Retrieved March 15, 2021, from https://docs.python.org/3/library/json.html

Rathod, M. (2021, March 11). Multithreading & multiprocessing in Python3. Retrieved March 15, 2021, from https://medium.com/mindful-engineering/multithreading-multiprocessing-in-python3-f6314ab5e23f