Basic Auth Web Scanner Tool

Nicholas Park

Table of Contents

1. Introduction

Introduction

This is a program used to test web servers with basic auth credentials to find out which ones could be vulnerable to the credentials “root:root”.

Once results are in it provides different formats of results, some of which could be used for further pentesting purposes.

Hope you have fun with this tool!

Requirements

- Python version at least 3.10 or higher installed (due to match-case statement used)
- Pip install libraries (run requirements.py script using “python requirements.py” for automated installation)
- Libraries to be installed using pip: asyncio, aiohttp, ipaddress, resource

How to Run

Run the command “python webscanner.py” in the terminal where the file is located. Options will be shown on the interactive interface once the program is loaded.

Please refer to Test Case for an example running case.

2. Logic

Explanation

When sending basic auth requests and verifying our success, the high level core logic is to act differently based on what kind of response we receive from the web server. The best way for myself was to use response status codes because of these examples:

200 OK: We can now check additionally if our request was indeed okay because of correct credentials or because of how the web server is configured (it might accept any requests regardless of credentials)

4xx Client Side Error: Most of these cases will result in true negatives which makes it easier for us to confirm. For example, 401 will let us know that credentials were wrong, 403/404 being site access problems, etc. There are a few outliers like 405 which are explained a bit more in Accuracy.

5xx Server Side Error: Also relatively easy to conclude since it is an error on the server. So we can add these results to their own list in case the user wants to try them out later if the server is back up, etc.

So to summarize, we handle our results based on response status codes and what they mean. Of course, this isn't perfect in itself so I had to add a few more validations and verifications. Details are also in Accuracy.

Scalability

There are three ways this code provides accommodations for scalability

The first is asynchronous execution where requests are sent concurrently. Compared to sequential execution, this means that overall scanning time is reduced because requests are executed and handled concurrently and thus this can help with scalability.

Furthermore, I added the option to modify the number of ports to scan. This is significant because as a programmer of this tool I do not know how many hosts the user wants to scan. Therefore, if they would like to scan a few hosts they can instead increase the number of ports to scan to really drill at their handful of hosts or otherwise select a few ports if there are a large number of hosts to be tested.

Finally, for some of the handling cases of responses, I added a user controlled switch to choose between in-depth scanning mode and simple mode. This again allows the user to choose whether they would like to focus on accuracy or speed depending on how many hosts they are testing.

Ports

Similar to what was mentioned in Scalability, I have added the option for users to add or remove ports should they want to test for alternative ports on each host.

Output

My goal was to create two types of outputs: one that can be used easily for other programs and another that is a more detailed report.

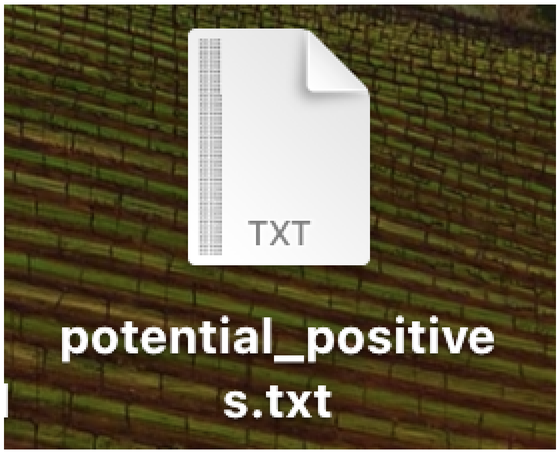
Text Files:

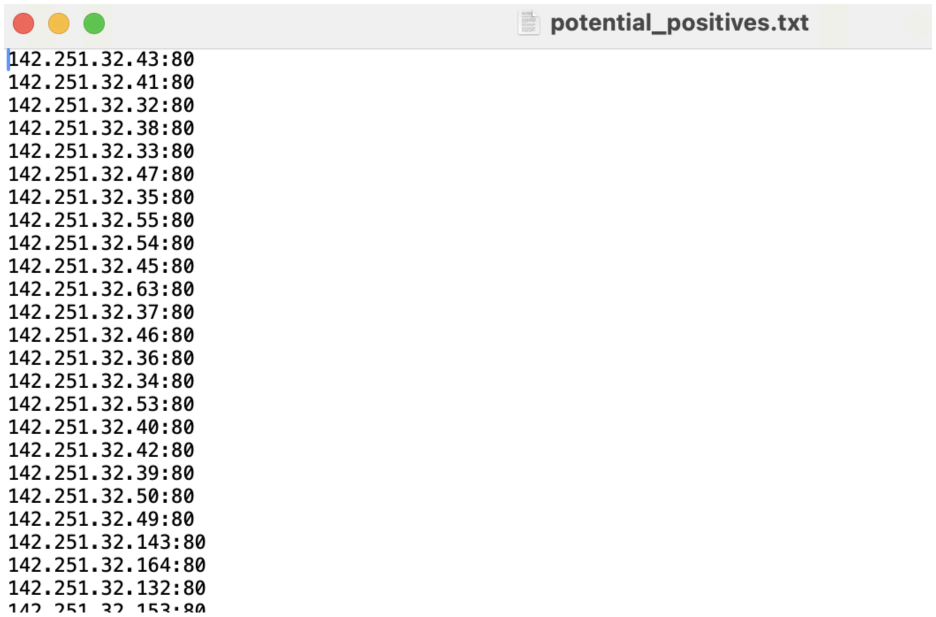
```
self.true_positives = {}
self.true_negatives = {}
self.potential_positives = {}
self.server_error = {}
```

These are dictionaries that have key value pairs of “ip:port” and holds values that correspond to their names. For example:

- self.true_positives will hold ip:port combinations that were indeed true positives
- self.true_negatives for true negatives
- self.potential_positives for not confirmed but potentially positive results
- self.server_error for results that ended in “5xx” status code (server based errors)

Then in one of my functions, create_output_files(), I convert all of these dictionary key-value pairs into each line within a text file and create the text files within the same directory as the webscanner.py file. They look something like this:





```
142.251.32.43:80
142.251.32.41:80
142.251.32.32:80
142.251.32.38:80
142.251.32.33:80
142.251.32.47:80
142.251.32.35:80
142.251.32.55:80
142.251.32.54:80
142.251.32.45:80
142.251.32.63:80
142.251.32.37:80
142.251.32.46:80
142.251.32.36:80
142.251.32.34:80
142.251.32.53:80
142.251.32.40:80
142.251.32.42:80
142.251.32.39:80
142.251.32.50:80
142.251.32.49:80
142.251.32.143:80
142.251.32.164:80
142.251.32.132:80
142.251.32.153:80
```

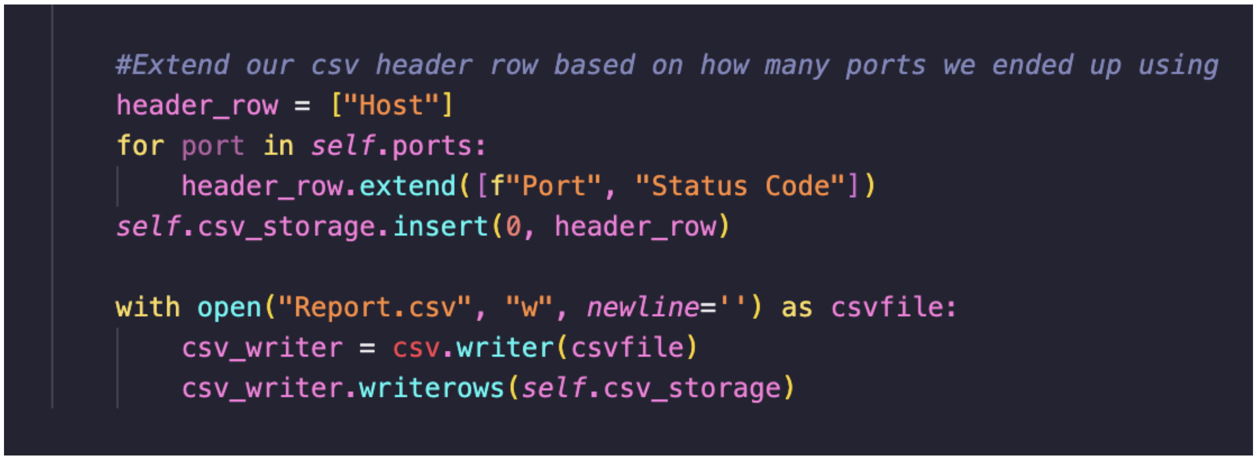
Easy to parse when using as inputs for other programs. Simple and effective.

Report

For this one, I built a nested list storage:

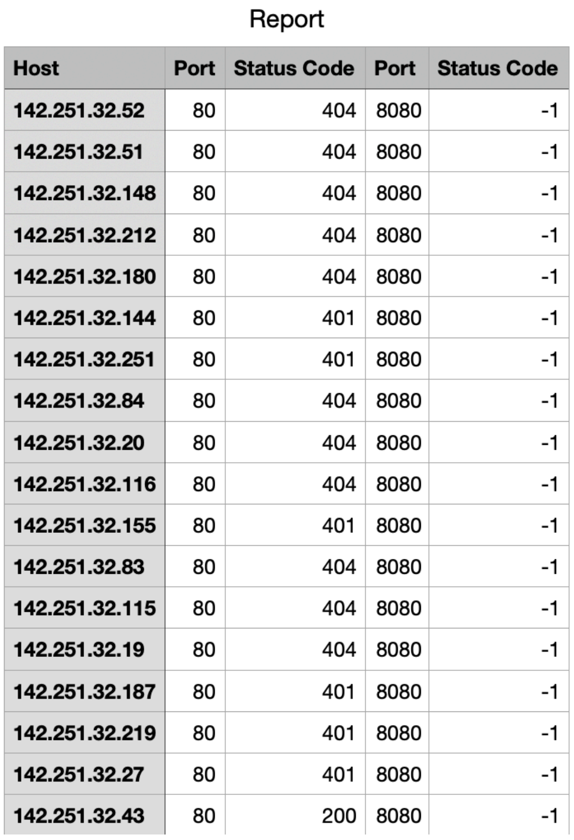
```
self.csv_storage = []
```

Which is used to hold lists called “csv_input”. These are generated for each host and the values are appended when the handle_requests() function is called and each port on each host appends the port and status code into this list. Once each host is covered that list will be appended to “self.csv_storage” which goes through a conversion to a .csv file output:



```
#Extend our csv header row based on how many ports we ended up using
header_row = ["Host"]
for port in self.ports:
    header_row.extend([f"Port", "Status Code"])
self.csv_storage.insert(0, header_row)

with open("Report.csv", "w", newline='') as csvfile:
    csv_writer = csv.writer(csvfile)
    csv_writer.writerows(self.csv_storage)
```

Which will similarly create the output in the same directory as where the script is. And finally it looks something like this:



Report

Host	Port	Status Code	Port	Status Code
142.251.32.52	80	404	8080	-1
142.251.32.51	80	404	8080	-1
142.251.32.148	80	404	8080	-1
142.251.32.212	80	404	8080	-1
142.251.32.180	80	404	8080	-1
142.251.32.144	80	401	8080	-1
142.251.32.251	80	401	8080	-1
142.251.32.84	80	404	8080	-1
142.251.32.20	80	404	8080	-1
142.251.32.116	80	404	8080	-1
142.251.32.155	80	401	8080	-1
142.251.32.83	80	404	8080	-1
142.251.32.115	80	404	8080	-1
142.251.32.19	80	404	8080	-1
142.251.32.187	80	401	8080	-1
142.251.32.219	80	401	8080	-1
142.251.32.27	80	401	8080	-1
142.251.32.43	80	200	8080	-1

Accuracy

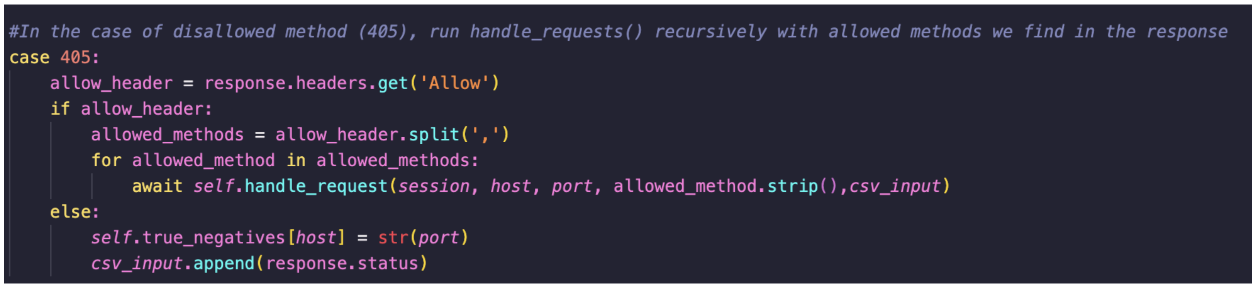
Easily the most difficult aspect of this project. Spent hours trying to find the one solution that fits all but I realized that it is near impossible to find a method that will produce perfect true positives.
However, we can get close to perfect. Here are a combination of methods I used to improve accuracy.

To start off, I give the option of in-depth scanning for the users. What does it do? Well it has a confidence value that goes up the more there is evidence indicating a successful login.For example, when a 200 status code is received, this is what the code does in an in-depth search:

1. Check for any login indicating words in response.text such as “Welcome”, “User Account”, etc.
2. Check for any headers indicating successful login such as “Set-Cookie”, “Session-Token”, etc.
3. Send a second request with credentials “root:root1”. If this produces a 401 response we can confirm for sure that our root:root credential was a true positive.

If any of these provide positive indications, we increase our confidence value by 1. And if that value is above a certain threshold we can add that to the “potential positives” group. Again, we want to be super careful about what we put in the true positives category which is why I made this code a bit more conservative in its estimates.

Furthermore, here is another cool snippet:



```
#In the case of disallowed method (405), run handle_requests() recursively with allowed methods we find in the response
case 405:
    allow_header = response.headers.get('Allow')
    if allow_header:
        allowed_methods = allow_header.split(',')
        for allowed_method in allowed_methods:
            await self.handle_request(session, host, port, allowed_method.strip(),csv_input)
    else:
        self.true_negatives[host] = str(port)
        csv_input.append(response.status)
```

The context to this is that we send our basic auth credentials using the GET request since it is most commonly used. However, if we receive a 405 response saying that GET is not allowed, we recursively call the handle_requests() function to try for every method that is allowed. Just to go for that extra bit of accuracy 😉

3. Q&A

Q1: Why use CLI rather than command line execution? Latter seems easier to maintain long term?

- For the time being, I wanted to make my program more user friendly. Thanks to having a CLI, the program is a lot more self-intuitive than having to use additional flags when running the program.
- However, I will probably convert to command line execution in the future (check out future plans for more info)

Q2: For in-depth scan, why use confidence levels?

- Because the validations that are in place, such as looking for certain words in response.text or headers relating to a login, are not always correct.
- For example, I can search for words like “Successful login” or “Welcome” in response.text but not all login systems have that in their response. Same with headers: “Set-Cookie” for instance could be used for session management rather than authentication.
- So overall, while it is true that these indicators are a good signal, they are not perfect. Hence why I use the confidence level in the in-depth scan to determine whether we should further investigate by adding these cases into the “potential positives” dictionary.

Q3: Why is response status code 403 treated like a true negative? Technically our request did not fail because of the authentication?

- Technically true, but the fact that the resource is forbidden means that we would probably need to hack our way into the source. So in terms of the “Basic Auth” context I would say that this result should be in the true negatives bracket.

Q4: Why are server error status codes (5xx) in their own text file output?

- Because the error is on the server side, there is still a chance that we can try again later if the server is healthy. The reason I did not put these values into “potential_positives” is because “potential_positives” is mainly for outputs that we were able to test and can half-confidently assume that they could be positives. In the 5xx server code cases we have not tested them so I mix those results.

4. Future Plans?

1. Change execution into command line. This includes converting original CLI interface and also adding -h (help option) to explain flags and example usages
2. In general, create an option for users to choose their own credentials to execute. Right now it is hardcoded to test for root:root but later give some more flexibility
3. Add a semaphore to limit request sending rates if there is too much overhead in the system due to our program

5. Error Handling

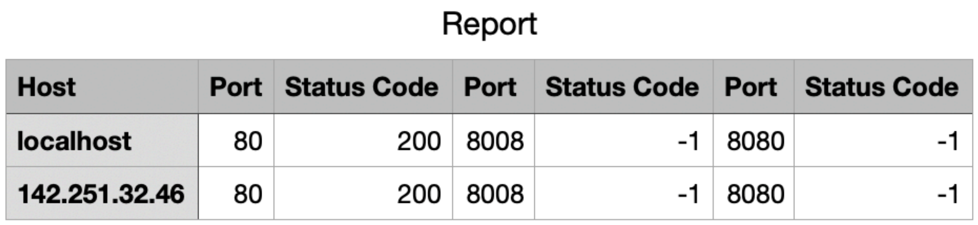
Try to scan with no hosts:

```
Main Menu:
1. Load hosts from File
2. Manual Input
3. Load hosts from Subnet
4. Settings
5. Start Scanning
6. Exit
Enter your choice: 5
Do you want to scan in depth? (y/n): y
No hosts loaded. Please load hosts before scanning.
```

Try to scan with no ports:

```
Main Menu:
1. Load hosts from File
2. Manual Input
3. Load hosts from Subnet
4. Settings
5. Start Scanning
6. Exit
Enter your choice: 5
Do you want to scan in depth? (y/n): y
No ports loaded. Please load ports before scanning.
```

Try to scan with both hostnames and ipv4 addresses in host array:



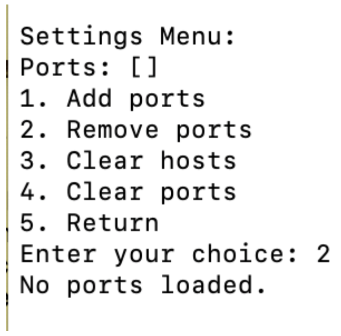
Report

Host	Port	Status Code	Port	Status Code	Port	Status Code
localhost	80	200	8008	-1	8080	-1
142.251.32.46	80	200	8008	-1	8080	-1

Try loading non .txt files:

```
Main Menu:
1. Load hosts from File
2. Manual Input
3. Load hosts from Subnet
4. Settings
5. Start Scanning
6. Exit
Enter your choice: 1
Enter filename: Report.csv
Invalid file format. Only text files (.txt) are supported.
```

Try removing port when none exists:

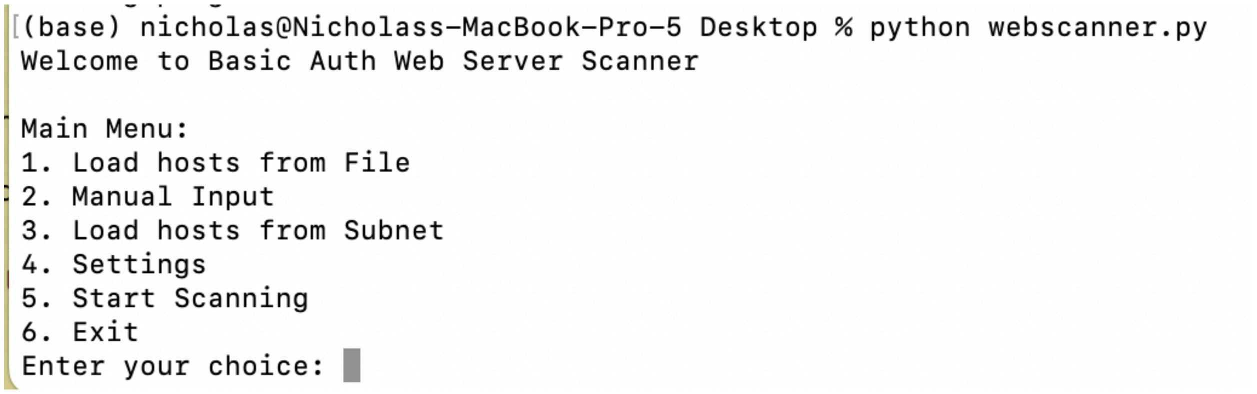


```
Settings Menu:
Ports: []
1. Add ports
2. Remove ports
3. Clear hosts
4. Clear ports
5. Return
Enter your choice: 2
No ports loaded.
```

6. Test Case

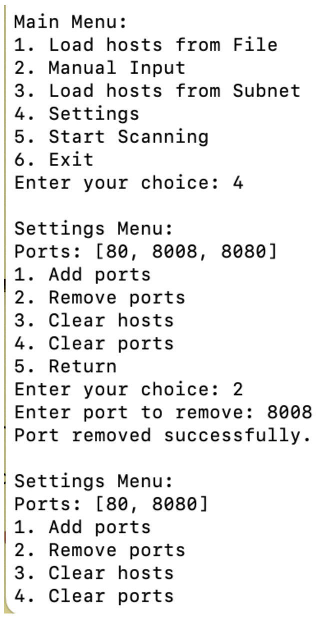
Testing one of Google’s IP domains (142.251.32.0/24)

1. Start the program



```
(base) nicholas@Nicholass-MacBook-Pro-5 Desktop % python webscanner.py
Welcome to Basic Auth Web Server Scanner

Main Menu:
1. Load hosts from File
2. Manual Input
3. Load hosts from Subnet
4. Settings
5. Start Scanning
6. Exit
Enter your choice: 
```

2. Let’s modify settings first. Navigate to settings and remove one of the default ports

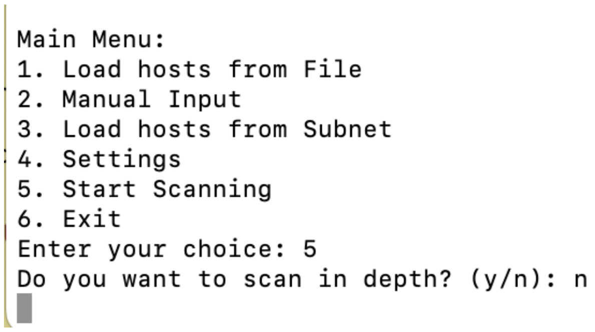


```
Main Menu:
1. Load hosts from File
2. Manual Input
3. Load hosts from Subnet
4. Settings
5. Start Scanning
6. Exit
Enter your choice: 4

Settings Menu:
Ports: [80, 8008, 8080]
1. Add ports
2. Remove ports
3. Clear hosts
4. Clear ports
5. Return
Enter your choice: 2
Enter port to remove: 8008
Port removed successfully.

Settings Menu:
Ports: [80, 8080]
1. Add ports
2. Remove ports
3. Clear hosts
4. Clear ports
```

3. Return to main menu and load hosts via subnet range

```
Settings Menu:
Ports: [80, 8080]
1. Add ports
2. Remove ports
3. Clear hosts
4. Clear ports
5. Return
Enter your choice: 5

Main Menu:
1. Load hosts from File
2. Manual Input
3. Load hosts from Subnet
4. Settings
5. Start Scanning
6. Exit
Enter your choice: 3
Enter subnet range (e.g., '10.0.0.0/24'): 142.251.32.0/24
```

4. Start scanning and let’s go for simple scanning in this example

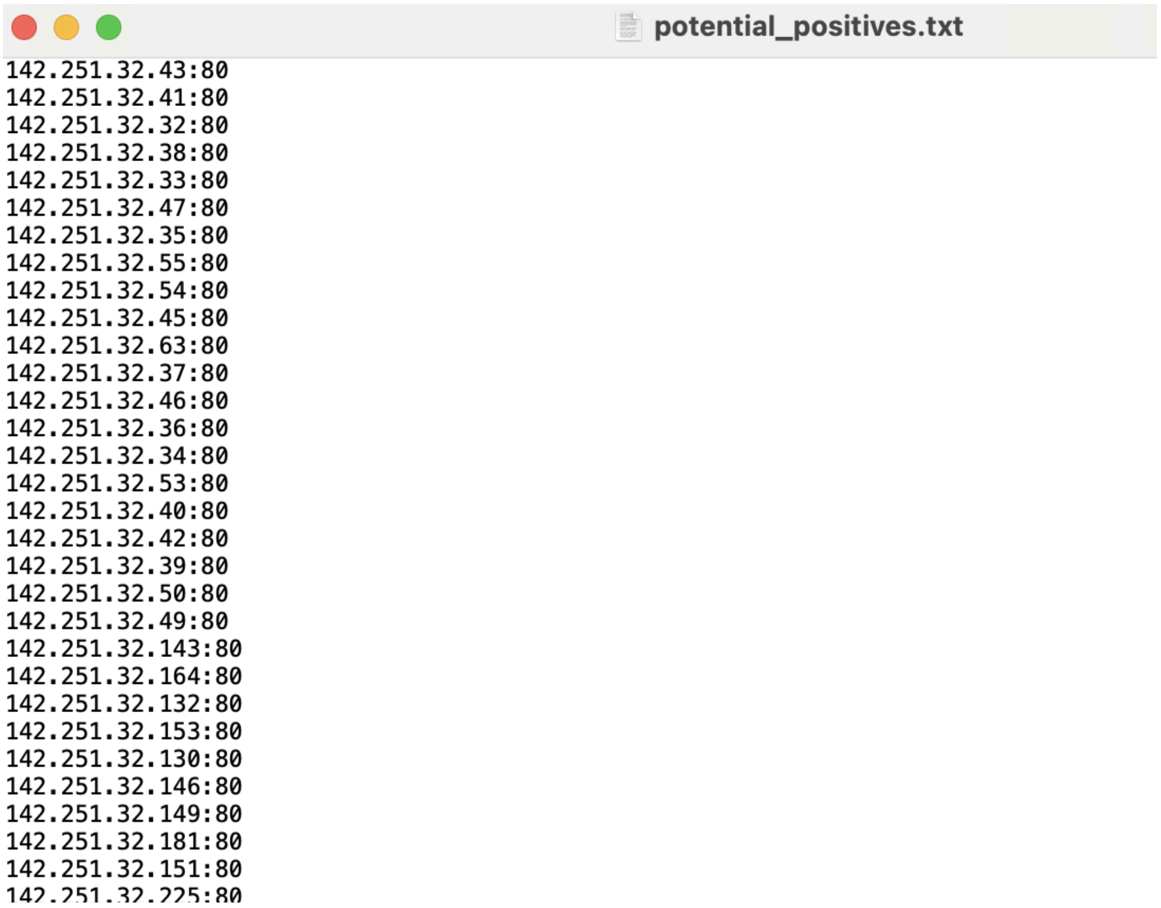


```
Main Menu:
1. Load hosts from File
2. Manual Input
3. Load hosts from Subnet
4. Settings
5. Start Scanning
6. Exit
Enter your choice: 5
Do you want to scan in depth? (y/n): n
```

5. Let’s wait until it’s done

```
Main Menu:
1. Load hosts from File
2. Manual Input
3. Load hosts from Subnet
4. Settings
5. Start Scanning
6. Exit
Enter your choice: 5
Do you want to scan in depth? (y/n): n
Done scanning!
Creating Output...
Output Created!
```

6. Check results (outputs)


potential_positives.txt

```
142.251.32.43:80
142.251.32.41:80
142.251.32.32:80
142.251.32.38:80
142.251.32.33:80
142.251.32.47:80
142.251.32.35:80
142.251.32.55:80
142.251.32.54:80
142.251.32.45:80
142.251.32.63:80
142.251.32.37:80
142.251.32.46:80
142.251.32.36:80
142.251.32.34:80
142.251.32.53:80
142.251.32.40:80
142.251.32.42:80
142.251.32.39:80
142.251.32.50:80
142.251.32.49:80
142.251.32.143:80
142.251.32.164:80
142.251.32.132:80
142.251.32.153:80
142.251.32.130:80
142.251.32.146:80
142.251.32.149:80
142.251.32.181:80
142.251.32.151:80
142.251.32.225:80
```

Report

Host	Port	Status Code	Port	Status Code
142.251.32.52	80	404	8080	-1
142.251.32.51	80	404	8080	-1
142.251.32.148	80	404	8080	-1
142.251.32.212	80	404	8080	-1
142.251.32.180	80	404	8080	-1
142.251.32.144	80	401	8080	-1
142.251.32.251	80	401	8080	-1
142.251.32.84	80	404	8080	-1
142.251.32.20	80	404	8080	-1
142.251.32.116	80	404	8080	-1
142.251.32.155	80	401	8080	-1
142.251.32.83	80	404	8080	-1
142.251.32.115	80	404	8080	-1
142.251.32.19	80	404	8080	-1
142.251.32.187	80	401	8080	-1
142.251.32.219	80	401	8080	-1
142.251.32.27	80	401	8080	-1
142.251.32.43	80	200	8080	-1
142.251.32.41	80	200	8080	-1
142.251.32.38	80	200	8080	-1
142.251.32.32	80	200	8080	-1
142.251.32.16	80	401	8080	-1