Nmapthon

Release 1.5.0

Dec 09, 2021

Contents

CHAPTER 1

Getting started

Nmapthon is a Python module that allows you to interact with the Nmap tool and even extend its capabilities with Python functions. With this module you will be able to:

- Execute Nmap scans and easily retrieve all the results.
- Execute Nmap scans asynchronously.
- Merge different scanner objects, allowing to easily build multiprocessed, multithreaded and distributed applications.
- Register Python functions as if they were NSE scripts.

1.1 Installation

The module requires an updated version of Nmap installed on the system. To install Nmapthon, simply use the *pip* package manager:

```
# If your pip command corresponds to Python 3
pip install nmapthon

# If you use pip3 instead
pip3 install nmapthon
```

Warning: Python 2 is not supported.

CHAPTER 2

NmapScanner

2.1 Instantiation

The `NmapScanner` class takes one positional parameter and `**kwargs` parameters for instantiation:

```
NmapScanner(targets, ports=None, arguments=None, engine=None)
```

- `targets`: Can be specified as a `str` or a `list`. A string may contain any number of targets separated by commas, and a list may contain any number of targets as separate elements. Nmapthon considers a target any of the following:
  - A domain or URL.
  - A resolvable hostname (like a NetBIOS hostname).
  - A single IP Address (like `"192.168.2.45"`).
  - A full IP Address range (like `"192.168.0.2-192.168.0.17"`).
  - A partial IP Address range (like `"192.167-168.0.1-20"`).
  - An IP address with a netmask, which will include all the IP address inside the mask but without the network address and the broadcast address (like `"192.168.0.0/24"`).

For `kwargs`:

- `ports`: Can be specified as a `str` or a `list`. A list may contain any number of target ports separated by commas, and a list may contain any number of target ports as separate elements. Nmapthon considers a target port any of the following:
  - A single port as `str` or `int` type (like `22` or `"80"`).
  - A port range (like `"22-80"`).
- `arguments`: String containing every nmap parameter that we want to execute. For example `'-sV -Pn'`.

Note: No `-d` or `-v` options allowed (That means no debugging or verbosity). The `-p` parameter is not allowed either, ports must be specified on instantiation or by the `ports` setter as explained above.

- `engine`: Specify a `PyNSEEngine` object. Refer to this section *PyNSEEngine* to learn more about it.

Note that every instantiation parameter can be set as `None`, including the `targets`, but at least those need to be set before running the scan. Each of these instantiation parameters have their properties and setters, which means that you can interact with them after instantiation the scanner itself:

- `<scanner_instance>.targets`: Property and setter for the `NmapScanner` targets.
- `<scanner_instance>.ports`: Property and setter for the `NmapScanner` ports.
- `<scanner_instance>.arguments`: Property and setter for the `NmapScanner` arguments.
- `<scanner_instance>.engine`: Property and setter for the `NmapScanner` NSE engine.

2.1.1 Simple example

```
import nmapthon as nm

# This instantiates a scanner for localhost and Service Detection on default ports
scanner = nm.NmapScanner('127.0.0.1', arguments='-sV')

# This one scans 255 hosts at maximum speed and with script launching, OS detection␣
↪and Service Detection
scanner = nm.NmapScanner(['192.168.1.1', '192.168.1.11', '192.168.2.0/24'], arguments=
↪'-A -T4')

# This one scans localhost and another IP range for the first 200 ports and 443.
scanner = nm.NmapScanner('127.0.0.1, 10.10.0.0/16', ports=['1-200', 443])
```

2.2 Running the scan

After instantiating the scanner, the `run()` method will execute it. The program will block until the nmap process finishes, and after that, the `NmapScanner` instance will contain all the information from the scan.

2.2.1 Example

```
import nmapthon as nm

example_scanner = nm.NmapScanner(target='127.0.0.1', arguments='-sS')

# Execute the scan
try:
    example.scanner.run()
except nm.NmapScanError as e:
    print('Catching all scan errors!: {}'.format(e))

# Now the 'example_scanner' object contains all the information from the scan.
```

Please head to the next sections to know how to manage all the information gathered from the scan.

2.2.2 Errors

When executing the `run()` method, several type of errors can pop, but all of them are raised by the same Exception: `NmapScanError`. The situations when this Exception could come out are:

- No targets to scan are specified.
- When nmapthon cannot parse the nmap output, due to any type of nmap error that interrupted the execution. In this case, the `NmapScanError` will print the nmap error.
- When no output from nmap is given.

2.3 Getting simple scan information

After calling the `run()` method, the `NmapScanner` instance will have several properties to access scan information, only if no errors are raised. These properties are:

- `start_timestamp`: Get the timestamp from when the scan started.
- `start_time`: Get the human-readable date and hour from when the scan started.
- `exit_status`: Nmap application exit status.
- `args`: All arguments used in the scan, **but this args are printed by nmap**.
- `summary`: Scan summary.
- `version`: Nmap's version.
- `end_timestamp`: Get the timestamp from when the scan finished.
- `end_time`: Get the human-readable date and hour from when the scan finished.
- `finished`: Boolean flag that tells if the scan has finished.
- `tolerant_errors`: String with errors that happened during the Nmap execution but let the scan finish.

Important: If any of this properties is accessed before calling the `run()` method, they will return `None`.

2.3.1 Example

```
import nmapthon as nm

scanner = nm.NmapScanner('192.168.1.0/24', ports='1-1024', arguments='-sS')
scanner.run()

# If program reaches this point, we can get the properties.
print("Started at: {}".format(scanner.start_time))
print("Used {} nmap version.".format(scanner.version))
print("The tolerant errors were:\n{}".format(scanner.tolerant_errors))
# You can keep calling any of this properties
```

2.4 Hosts and ports

After running the scan, we can execute two primary methods to obtain the hosts from the scan:

- `scanned_hosts()`: Returns a list of scanned hosts.
- `non_scanned_hosts()`: Returns a list with all the hosts that where specified on `targets` but did not appear on the nmap output, which means that they were not scanned.

To get the **hostnames** associated with a particular host:

- `hostnames(host:str)`: Returns a list with all the hostnames from a host.

Having the scanned hosts, we can get their state, reason and scanned protocols:

- `state(host:str)`: Returns the state of a given host.
- `reason(host:str)`: Returns the reason why the host has a certain state.
- `all_protocols(host:str)`: **Yields** every protocol scanned for a given host.

For a given host and protocol, we can also get the scanned and non scanned ports, plus their state:

- `scanned_ports(host:str, protocol:str)`: Return a list of scanned ports for a given host and protocol.
- `non_scanned_ports(host:str, protocol:str`: Return a list of non scanned ports for a given host and protocol.
- `port_state(host:str, protocol:str, port:str,int)`: Return the state and reason `tuple` from a port.

Note: If scanning domains, their information would not be under the domain name itself, but under an IP Address, which is the IP address of the host gathered by nmap after resolving the domain.

2.4.1 Host information example

```
import nmapthon as nm

sc = nm.NmapScanner(['127.0.0.1', '192.168.1.99'], ports=[1,101], arguments='-sT')
sc.run()

# Loop through protocols, for every scanned host and get other information
for host in sc.scanned_hosts():
    # Get state, reason and hostnames
    print("Host: {}\tState: {}\tReason: {}".format(host, sc.state(host), sc.
↪reason(host))
    print("Hostname: {}".format(','.join(sc.hostnames(host))))
    # Get scanned protocols
    for proto in sc.all_protocols(host):
        # Get scanned ports
        for port in sc.scanned_ports(host, proto):
            state, reason = sc.port_state(host, proto, port)
            print("Port: {0:<7}State:{1:<9}Reason:{2}".format(port, state, reason))
```

2.5 Services

If service detection was performed (for example with `'-sV'` or `'-A'`), we can gather the service information for a given host, protocol and port:

- `service(host:str, protocol:str, port:str,int)`: Get a Service instance representing the gathered information from the service, if no service information was found it returns `None`.
- `standard_service_info(host:str, protocol:str, port:str,int)`: Returns the service name and service information. The service information is a string formed by the service product, version and extrainfo. If there is no info about a particular service, two *None* values will be returned. If nmap has found the name of the service, but it doesnt know anything about the service information itself, this method will return the name and an empty string (`''`).

2.5.1 Service object

Executing the function `service(host:str, protocol:str, port:int,str)` will return `None` if there is no known service, or it will return a `Service` object in any other case. A `Service` object has 4 simple properties:

- `name`: Return the name of the service.
- `product`: Return the product running on that service.
- `version`: Return the version of the product.
- `extrainfo`: Return extra information about the product.

We can also get all CPEs associated with that service:

- `all_cpes()`: Return a list containing all the CPEs from a service.

Get all the scripts information that were launched against that particular service:

- `all_scripts()`: **Yields** every script name and output from every script that was launched against that service.

Service instances can be used as list objects, which allows scripts management, for example:

- `service_instance[script_name]`: Return the output from a given script name.
- `service_instance[script_name] = script_output`: Add a script name with an associated output.
- `del service_instance[script_name]`: Delete every script related information for a given script name.
- `'my_script' in service_instance`: Check if a given script is inside the instance.

2.5.2 Service object example

```
import nmapthon as nm

scanner = nm.NmapScanner('192.168.1.0/24', ports='22,53,443', arguments='-A -T4')
scanner.run()

# for every host scanned
for host in scanner.scanned_hosts():
    # for every protocol scanned for each host
    for proto in scanner.all_protocols(host):
        # for each scanned port
        for port in scanner.scanned_ports(host, proto):
            # Get service object
            service = scanner.service(host, proto, port)
            if service is not None:
```

(continues on next page)

(continued from previous page)

```
                print("Service name: {}".format(service.name))
                print("Service product: {}".format(service.product))
                for cpe in service.all_cpes():
                    print("CPE: {}".format(cpe))
                for name, output in service.all_scripts():
                    print("Script: {}\nOutput: {}".format(name, output))
                # You could also do print(str(service))
                # You could also know if 'ssh-keys' script was launched and print the␣
↪output
                if 'ssh-keys' in service:
                    print("{}".format(service['ssh-keys']))
```

2.5.3 Service standard info example

```
import nmapthon as nm

scanner = nm.NmapScanner('192.168.1.0/24', ports='22,53,443', arguments='-sV -T4')
scanner.run()

# for every host scanned
for host in scanner.scanned_hosts():
    # for every protocol scanned for each host
    for proto in scanner.all_protocols(host):
        # for each scanned port
        for port in scanner.scanned_ports(host, proto):
            # Get service information
            service, service_info = scanner.standard_service_info(host, proto, port)
            if service is not None:
                print("Service: {}\tInfo: {}".format(service, service_info))
```

2.6 Scripts

Nmapthon supports two types of scripts:

- NSE scripts. Which are LUA scripts that can be execute through the `--script` argument from the nmap tool.
- PyNSE scripts. Which are python functions that are registered as "NSE scripts". See *PyNSEEngine* to learn how they work.

Whatever type of script you are executing, each script has a name. In case of NSE scripts, the script name will be the argument(s) passed to the `--script` argument like `--script ssl-cert`. On the other hand, PyNSE scripts have a mandatory `name` parameter.

When retrieving a script output, it needs to be referenced by its name. Nmapthon has several ways of retrieving those scripts:

- `host_script(host:str, script_name:str)`: Returns the host script output for a given script name. If the target does not have any information about that script, it will raise a `NmapScanError`.
- `port_script(host:str, proto:str, port:(str,int), script_name:str)`: Returns the port script output for a given script name, associated with a protocol and a port. If the target does not have any information about that script, it will raise a `NmapScanError`.

- `host_scripts(host:str, script_name:str=None)`: Yields a tuple with `(script_name, script_output)` for every host script from a particular host. If `script_name` is specified, then it will only yield scripts whose names **contain** that string.
- `port_scripts(host:str, proto:str, port:(str,int), script_name:str=None)`: Yields a tuple with `(script_name, script_output)` for every port script from a particular host, port and protocol. If `script_name` is specified, then it will only yield scripts whose names **contain** that string.

Note: `host_script()` and `port_script()` functions must raise a `NmapScanError` to indicate "missing" scripts. The `None` return value is not possible, since a PyNSE script may return a None value if the user defines it to do so, and may confuse the real script output with the "missing script" situation.

Note: Apart from that, we can get the scripts from a `Service` instance, as explained in the previous page.

2.6.1 Example

```
import nmapthon as nm

sc = nm.NmapScanner('10.10.10-15.2-254', ports=[443, 80, 53], arguments='-sV --
↪script=ssl-cert,dns-brute')
sc.run()
for i in sc.scanned_hosts():
    for port in sc.scanned_ports(i, 'tcp'):
        for n, o in sc.port_scripts(i, 'tcp', port):
            print('Name: {}\nOutput: {}'.format(n, o))

# Check unique script output
print('{}'.format(sc.port_script('10.10.10.4', 'tcp', 443, 'ssl-cert')))

# Check unique script from service
service_example = sc.service('10.10.10.4', 'tcp', 443)
if service_example is not None:
    print('{}'.format(service_example['ssl-cert']))
```

2.7 OS Detection

If OS detection was performed (for example, by using `'-O'` or `'-A'`), you can get the OS matches with their accuracy and the OS fingerprint:

- `os_matches(host:str)`: **Yields** every OS name with it's corresponding accuracy for a given host.
- `os_fingerprint(host:str)`: Returns the OS fingerprint for a given host. If no fingerprint was found or performed, it will return `None`.
- `most_accurate_os(host:str)`: Returns a list with the most accurate OSs. **The list is needed because there might not be only one OS match with the highest accuracy, but several.**

2.7.1 OS Detection example

```
import nmapthon as nm

scanner = nm.NmapScanner('127.0.0.1', arguments='-O --osscan-guess')
scanner.run()

# Notice that '127.0.0.1' can be used without expecting an NmapScanError
# localhost should always respond.
for os_match, acc in scanner.os_matches('127.0.0.1'):
    print('OS Match: {}\tAccuracy:{}%'.format(os_match, acc))

fingerprint = scanner.os_fingerprint('127.0.0.1')
if fingerprint is not None:
    print('Fingerprint: {}'.format(fingerprint))

for most_acc_os in scanner.most_accurate_os('127.0.0.1'):
    print('Most accurate OS: {}'.format(most_acc_os))
```

2.8 Traceroute

We can get every hop information from executing a traceroute to a particular host:

- `trace_info(host:str)`: **Yields** one `TraceHop` instance per traceroute hop.

2.8.1 TraceHop object

A `TraceHop` instance has four basic properties to access its information:

- `ttl`: Time-To-Live. IP layer field.
- `ip_addr`: IP Address of the node.
- `rtt`: Round Trip Time.
- `domain_name`: Domain name of the node.

Note: If any of the traceroute hop information is unknown, the corresponding property will return `None`.

2.8.2 Traceroute example

```
import nmapthon as nm

scanner = nm.NmapScanner('85.65.234.12', arguments='--traceroute')
scanner.run()

if '85.65.234.12' in scanner.scanned_hosts():
    for tracehop_instance in scanner.trace_info('85.65.234.12'):
        print('TTL: {}\tIP address: {}'.format(tracehop_instance.ttl, tracehop_
↪instance.ip_addr))
```

2.9 Merging NmapScanner objects

There may be situations where several `NmapScanner` instances may be instantiated separately, so a `merge()` is available to merge scans. It must be called after the instance finishes the scan, and it accepts any number of other `NmapScanner` instances plus additional `**kwargs`:

- `merge_tcp=True`: Flag to allow TCP merging
- `merge_udp=True`: Flag to allow UDP merging
- `merge_scripts=True`: Flag to merge host scripts. TCP/UDP port scripts are merged if their respective flag is `True`.
- `merge_trace=True`: Merge Traceroute information.
- `merge_os=True`: Merge OS information.
- `merge_non_scanned=True`: Merge IPs that could not be scanned.

2.9.1 `merge()` deep inspect

The `merge()` method acts differently depending on a main condition, which is: "Does the instance that's calling the method have the target X?". Depending on the answer:

- If the target is not in the caller scanner, all the information from the target is copied depending on the `**kwargs` flags values.
- If the target is on the caller scanner, the information is copied depending on the flags, particularly:
  - TCP/UDP ports are copied if they where not scanned on the caller scan, but if the caller already has information about them, it's not overwritten.
  - OS information, as well as Host scripts are checked one by one, only adding them if the caller does not have information of a particular OS/script.
  - Traceroute is only added while no Traceroute information is in the caller scanner.

2.9.2 Example 1: Dividing TCP and UDP scans

```
import nmapthon as nm

# Run a TCP scan synchronously and a UDP async to the same target
main_scanner = nm.NmapScanner('10.10.10.2', ports=[22, 80, 443], arguments='-sV -sS -n
↪')
udp_scanner = nm.AsyncNmapScanner('10.10.10.2', ports=[21, 53], arguments='-sU -n',␣
↪mute_error=True)

# Launch the UDP first
udp_scanner.run()

# Launch the TCP
try:
    main_scanner.run()
except nm.NmapScanError as e:
    print('Error while scanning TCP ports:\n{}'.format(e))

# Wait until UDP ends
```

(continues on next page)

(continued from previous page)

```
udp_scanner.wait()

if udp_scanner.finished_successfully():
    # Merge the scans (Do not need to set all flags to False since there is no␣
↪information on the UDP scanner,
    # but just to show the usage thay are set to False here
    main_scanner.merge(udp_scanner, merge_os=False, merge_scripts=False, merge_
↪tcp=False, merge_trace=False)
```

2.9.3 Example 2: Multi-threading/processing scans

```
import nmapthon as nm
import multiprocessing

def read_ips(ips_file):
    with open(ips_file) as f:
        return [x.strip() for x in f.readlines()]

def worker(n, ip, return_dict):
    sc = nm.NmapScanner(ip, ports=[1-1000], arguments='-sT -sV -T4 -n')
    try:
        sc.run()
    except nm.NmapScanner as e:
        raise e
    return_dict[n] = sc


if __name__ == '__main__':
    # Create share dict to store scans
    manager = multiprocessing.Manager()
    return_dict = manager.dict()
    jobs = []
    # Read IPS from file
    ips = read_ips('my_ips_file.txt')
    for i in range(len(ips)):
        p = multiprocessing.Process(target=worker, args=(i, ips[i], return_dict))
        jobs.append(p)
        p.start()

    # Freeze application until all apps finish
    for proc in jobs:
        proc.join()

    # Take the first scanner as caller
    main_scan = return_dict[0]
    # Pass the rest of the scans as arguments for merging
    main_scan.merge(*list(return_dict.values())[1:])

    # Now you can use the main_scan as a single scanner with all the information
    for host in main_scan:
        # Continue normally
```

2.10 Import XML

You can build an `NmapScanner` object from an existing Nmap XML file. To do so, just execute the `from_xml(file)` constructor:

- `NmapScanner.from_xml(file:str)`: Returns a `NmapScanner` instance from a valid Nmap XML output file.

Note: Note that `non_scanned_targets()` and `non_scanned_ports(target:str, proto:str)` will both return empty values, since Nmapthon uses the <instance>.targets and <instance>.ports setters, respectively, to process which targets and ports are not scanned.

2.10.1 Example

```
import nmapthon as nm

scanner = nm.NmapScanner.from_xml('/path/to/nmap.xml')
# Of course, you do NOT call the run() method

for i in scanner.scanned_hosts():
    for proto in scanner.all_protocols(i):
        print('Continue normally....')
```

Class for executing and parsing nmap scans. Provides a flexible target management and an easy way to retrieve the results. Head into the next page to see how it works.

CHAPTER 3

AsyncNmapScanner

3.1 Instantiation

Instantiating `AsyncNmapScanner` has the same `**kwargs` as the `NmapScanner` class (*Instantiation*), but this one has an optional extra `kwargs` parameter:

- `mute_errors`: A boolean type parameter, `False` by default. If set to `True`, the scanner won't show fatal errors when executing.
- `wrapper`: Wrapper class for executing the background scan. By default, this value is `threading.Thread`, but you can specify `multiprocessing.Process` if needed.

3.1.1 Example

```
import nmapthon as nm

async_scanner = nm.AsyncNmapScanner('10.126.65.0/23', ports='21,22,100-200',␣
↪arguments='-sV -n -T4')

# Async Scanner with error muting
async_scanner = nm.AsyncNmapScanner('192.168.1.30', arguments='-A -T4', mute_
↪errors=True)
```

3.2 Running the scan

`AsyncNmapScanner` also has the `run()` method, which will start executing the scan in background. You can use several methods to get the scan state and block the application:

- `is_running()`: Returns `True` if the scanner is running, `False` if not.
- `wait()`: Blocks the program execution until the scan finishes.

- `finished_succesfully()`: Returns `True` if the scan finished with no fatal errors. `False` if not.

If `mute_errors=True` is used, you can get the Exception raised when muted in case it did not finish successfully:

- `fatal_errors()`: Returns list of `NmapScanError` with the information from the Exceptions raised that was muted. If no `mute_errors=True` was set, it will return `None`, but you will have anyways an `NmapScanError` raised on your program.

3.2.1 Example 1

```
import nmapthon as nm
import time

scanner = nm.AsyncNmapScanner('192.168.1.2', ports=range(1,10001), arguments='-sS -sU
↪')
scanner.run()

# Do something while it executes
while scanner.is_running():
    print("I print because I can :)")
    time.sleep(1)

# Check if it was not successful
if not scanner.finished_succesfully():
    print("Uh oh! Something went wrong!")
```

3.2.2 Example 2

```
import nmapthon as nm

scanner = nm.AsyncNmapScanner('192.168.1.2', ports=range(1,10001), arguments='-sS -sU
↪')
scanner.run()

# Do something and block execution until finishes
for i in range(1, 1000000):
    print("Im printing a lot of lines!")
scanner.wait()

# Check if it was not successful
if not scanner.finished_succesfully():
    print("Uh oh! Something went wrong!\nPopped error:\n{}".format(scanner.fatal_
↪error()))
```

Class for executing Asynchronous Nmap scans.

CHAPTER 4

PyNSEEngine

4.1 Host scripts

Host scripts are functions that are execute once per host, if they respond to the scan.

To register a host script, decorate the functions with `@<engine_instance>.host_script('name')`. The function is defined as follows:

`host_script(name:str, targets='*', args=None):`

- `name`: Name that will be used on the `NmapScanner` instance to reference the script output.
- `targets`: Specify the targets that will be affected by the function. `'*'` means all of them. Targets can be specified as an `str` or a `list` type, the same way as targets are specified during the `NmapScanner` *Instantiation*.
- `args`: If the function has arguments, pass them as a `tuple` or `list` of arguments.

The information gathered from each of the registered host function is stored as a normal host script inside the `NmapScanner` instance. To access them, use the `host_scripts(host:str)` function.

Note that the data that will be stored inside the instance will be whatever the decorated function returns

4.1.1 Example 1

```
import nmapthon as nm

engine = nm.engine.PyNSEEngine()

# This function will only execute when the gateway (192.168.0.1) responds to the scan.
@engine.host_script('custom_script', targets='192.168.0.1')
def custom_gateway_scan():
    return 'I could return any type of information here'

sc = nm.NmapScanner('192.168.0.0/24', arguments='-sS -n -T5', engine=engine)
```

(continues on next page)

(continued from previous page)

```
sc.run()

# If the gateway responds to the scan, it will have an assigned host script
for name, output in sc.host_scripts('192.168.0.1'):
    print('{}: {}'.format(name, output))
    # Prints 'custom_script: I could return any type of information here!'
```

4.1.2 Example 2

```
import nmapthon as nm

engine = nm.engine.PyNSEEngine()

# Pass the function parameters with the decorator
@engine.host_script('param_testing', args=('Nmapthon',))
def func_with_params(my_arg):
    return 'Testing {}!'.format(my_arg)

sc = nm.NmapScanner('127.0.0.1', engine=engine)
sc.run()

# Localhost should always respond
print('{}'.format(sc.host_script('127.0.0.1', 'param_testing')))
```

4.2 Port scripts

Port scripts are those that execute when a particular port responds to the nmap scan.

To register a port script, decorate the functions with `@<engine_instance>.port_script(...)`. The function is defined as follows:

`port_script(name:str, port:(str,int,list), targets='*', proto='*', states=None, args=None):`

- `name`: Name that will be used on the `NmapScanner` instance to reference the script output.
- `ports`: Single port or list of ports that, when found with the given states, will make the engine execute the function. They are specified the same way as ports are specified during the `NmapScanner` *Instantiation*.
- `targets`: Specify the targets that will be affected by the function. `'*'` means all of them. Targets can be specified as an `str` or a `list` type, the same way as targets are specified during the `NmapScanner` *Instantiation*.
- `proto`: Transport layer protocol from the port. Default is `*' which means anyone, but can also be `'tcp'` or `'udp'`.
- `states`: Port states when the function will be triggered. Default is `None`, which means only `'open'` state, but can be a `list` containing any of the following values: `'open'`, `'filtered'` and `'closed'`.
- `args`: If the function has arguments, pass them as a `tuple` or `list` of arguments.

The information gathered from each of the registered port function is stored inside a `Service` object from that particular port. If there `NmapScanner` has already generated a `service` instance for that port, the script will be added to it.

Note that the data that will be stored inside the instance will be whatever the decorated function returns

4.2.1 Example

```
import nmapthon as nm

engine = nm.engine.PyNSEEngine()

# Create a custom SSH enum function
@engine.port_script('custom_ssh_enum', 22, proto='tcp', states=['open', 'filtered'],␣
↪args=('path/to/wordlist',))
def ssh_enum_function(wordlist):
    return 'My SSH enum with the wordlist: {}'.format(wordlist)

sc = nm.NmapScanner('127.0.0.1', ports='22', arguments='-sV -Pn -sS -n',␣
↪engine=engine)
sc.run()

# If the gateway responds to the scan, it will have an assigned port script
print(sc.port_script('127.0.0.1', 'tcp', 22, 'custom_ssh_enum'))
```

4.3 Accessing values

When the engine functions are executed, you may want to access some execution time values that are being handled by the `NmapScanner` object at that point. For that purpose you can use the following `PyNSEEngine` instance properties:

- `current_target`: Returns the target being processed when the function is executed
- `current_port`*: Returns the port being processed when the function is executed.
- `current_proto`*: Returns the transport layer protocol being processed when the function is executed.
- `current_state`*: Returns the state of the port being processed when the function is executed.

** These properties are only suitable if the function is decorated as a ‘‘@port_script‘‘.*

Important: Any of the above properties will return `None` if they are not handled by the appropriate decorator. i.e. `current_port` returns `None` if the function is decorated by `host_script`.

4.3.1 Example

```
import nmapthon as nm
import socket

engine = nm.engine.PyNSEEngine()

@engine.port_script('smtp_banner', 25, states=['open', 'filtered'])
def get_smtp_banner():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    # Each time the port script executes, it will execute with the current target and␣
↪port
    s.connect((engine.current_target, engine.current_port))
    banner = s.recv(1024)[4:]
    s.close()
```

(continues on next page)

(continued from previous page)

```
    return banner

sc = nm.NmapScanner('127.0.0.1', arguments='-sV',  engine=engine)
smtp_service = sc.service('127.0.0.1', 'tcp', 25)
if smtp_service is not None:
    print('Here is your SMTP banner: {}'.format(smtp_service['smtp_banner']))
```

Any `NmapScanner` object may receive a `PyNSEEngine` object, where several functions can be registered as host or port scripts.

Functions are registered by using **decorators**.

Note: Although the engine registers functions, they are referenced as "scripts". That's because the PyNSEEngine emulates a Python extension of the Nmap Scripting Engine (NSE).

A simple example:

```
import nmapthon as nm

engine = nm.engine.PyNSEEngine()

@engine.host_script('my_script')
def example():
    print('My own function as a script!')
    return None

sc = nm.NmapScanner('127.0.0.1', engine=engine)
```

Important: All the functions registered in the `PyNSEEngine` will be executed once the scan finishes, as opposed to the NSE scripts that are executed when the corresponding host/port is found open

---

CHAPTER 5

Errors

The nmapthon error hierarchy is the following:

```
| NmapScanError
|__ InvalidPortError
|__ MalformedIpAddressError
|__ InvalidArgumentError
| EngineError
```

Any error related with the Nmap scanner will be raised under an `NmapScanError` or any child error, while any error related to registering Python functions into the `PyNSEEngine` will raise an `EngineError`. All the Exception classes are imported automatically when `import nmapthon` executes, but you can also find them under `nmapthon.exceptions`.