Introduction to Computer Security Module – G6077

-

Nmap scripts and Lua Scripting language

Introduction Nmap Scripting Engine

* Need for a specific scripting language
* Inner working of an Nmap script
* Finding and running Nmap script

Writing Nmap Scripts

* Structure of an Nmap script
* Headers of Nmap script
* Rules of Nmap script
* Actions of Nmap script

Contents

[Introduction Nmap Scripting Engine 3](#_Toc22309166)

[Why there was a Need for Nmap scripting? 3](#_Toc22309167)

[The inner working of the NSE 3](#_Toc22309168)

[Finding Nmap scripts 4](#_Toc22309169)

[Running Nmap scripts 4](#_Toc22309170)

[Task 5](#_Toc22309171)

[Task 5](#_Toc22309172)

[Task 5](#_Toc22309173)

[Task 5](#_Toc22309174)

[Task 5](#_Toc22309175)

[Task 5](#_Toc22309176)

[Writing Nmap Scripts 6](#_Toc22309177)

[Structure of an Nmap script 6](#_Toc22309178)

[Defining an Nmap script – script headers 6](#_Toc22309179)

[Task 7](#_Toc22309180)

[Task 10](#_Toc22309181)

[Task 10](#_Toc22309182)

[--script-trace 10](#_Toc22309183)

# Introduction Nmap Scripting Engine

Although being able to conduct port scans is an integral part of using the Nmap suite of tools, the developers of Nmap created a very powerful engine that's built into the tool: the **Nmap Scripting Engine** (**NSE**). This section introduces the NSE, and covers all the topics needed to use reliably-written scripts in the Nmap script repository, in order to conduct reconnaissance scans that include much more than just what ports are open and which services are listening.

## Why there was a Need for Nmap scripting?

New vulnerabilities and scanning techniques were being pioneered on a very frequent basis, and full Nmap releases couldn't keep up with the things that security professionals needed to assess. Every time a new vulnerability came out, security professionals (and malicious hackers!) would scan for vulnerable services with Nmap, but could only test whether software versions were vulnerable by using manual analysis: clearly, not a very efficient use of time.

Google Summer of Code is a software development internship/association project, during which students are selected and put on open source software teams to build new features into existing projects. In a Google Summer of Code developers, an arbitrary scripting framework was created that allows users to trigger additional checks based on certain open ports or services. This means, for example, that if you're looking for a specific file on all web servers—robots.txt, for example—you can easily create a script that can check for it on all HTTP and HTTPS services. The NSE (and the inclusion of Nmap scripts in default installations of Nmap) truly revolutionized the versatility of the tool suite.

## The inner working of the NSE

The NSE is a framework that runs code written in the programming language Lua with specific flags that the engine can parse. Lua is a lightweight, fast, and interpreted programming language—one that has the most fame for scripting user interfaces for computer games such as World of Warcraft—that has a similar syntax to other contemporary interpreted languages. An example of code in Lua is listed below. The script shows an Nmap script that identifies information about Bitcoins (written by Patrik Karlsson). Don't worry if you don't understand it yet.



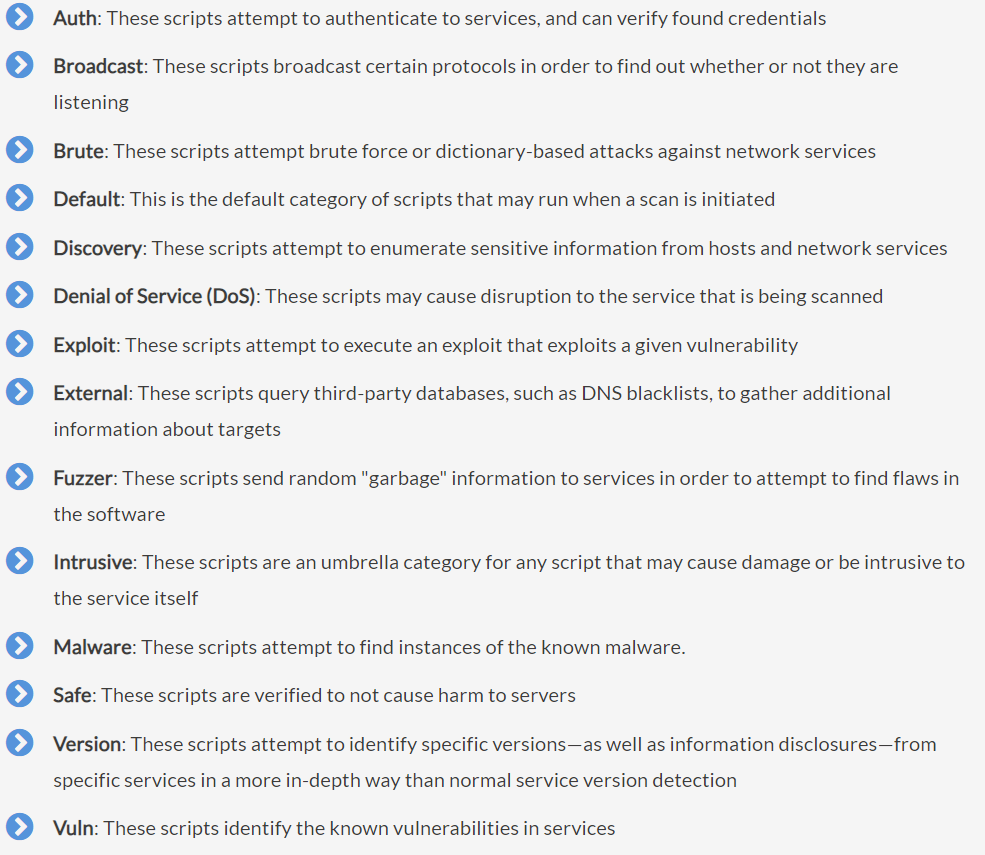
You can see that the code used to generate a relatively complex Nmap script looks very simple. This is the whole point of the NSE! Where security engineers and system administrators used to have to export Nmap results, find the information they are looking for and then use third-party tools to assist them; they are now able to either find a script that serves their purposes, or write a simple one themselves. Many penetration testers can leverage the Nmap scripting language to even weaponize the tool for security exploits.

## Finding Nmap scripts

Many Nmap scripts come with Nmap, and are already prepackaged on your system. Still, though, it can be difficult to determine which scripts you'd like to run for each particular scan—or assessment—that you may be on. Fortunately, the NSE documentation portal is one of the most in-depth and well-documented aspects of the entire Nmap project.

NSEdoc: <https://nmap.org/nsedoc/>

Scripts are broken down into several categories, each with their own specific use case. It is worth noting that sometimes these scripts can be in several categories, depending on the full functionality of the script. The categories and their definitions are as follows:



It's important to know which categories you want to run, since several of these categories—specifically DoS, exploit, and intrusive—can be dangerous to run against weak or production systems. The inclusion of these Nmap scripts in security assessments can easily increase the utility of Nmap in a very significant way.

## Running Nmap scripts

Running Nmap scripts is easy—and some, the "default" category, will even run on their own as a part of a normal scan. Some scripts are designed to simply give additional information about a target, while others will go so far as to actively exploit it (the "exploit" category) or even take it offline (the "DoS" category).

### Task

The first step to run an Nmap script that's part of the actually NSEDoc repository is to verify that the script is stored locally. Unlike the Nmap tool itself, the Nmap script repository is frequently updated—so it's in your best interest to always verify that you have the most updated version. You can update the NSE scripts by running Nmap with the flag --script-updatedb, which updates the script database.

### Task

Run the following script and examine its output. In lab3, you also run a command with some flags that was giving the same output.

nmap scanme.nmap.org --script default

You can see in the preceding screenshot that running default scripts here clearly flagged several findings immediately. If you were to run the same scan with -vv to have double-verbose mode enabled on the scan, you would also be able to see the number of scripts loaded against the given target

### Task

Run the following script and check if the script find http-xssed. What is http-xssed?

nmap scanme.nmap.org --script "http-xssed"

### Task

Run the following script and check if the script find http-xssed. What title value is retrieved?

nmap scanme.nmap.org --script "http-title"

After running above then run the following

nmap scanme.nmap.org --script "http-traceroute"

### Task

If you notice in the last three scripts that you run with three different --script values all have a common string which is http-. We can use a wildcard for this. So instead of three different commands, we will write just one with the wildcard. The following command may take a while to retrieve information, it took about 17 minutes on my machine

nmap scanme.nmap.org --script "http-\*"

You can see that launching a scan with the "http-\*" wildcard script name works in loading every script, but there are a few errors coming up. Certain scripts take parameters, so if loading many scripts, it's important to understand which ones are being loaded. The scripts that could accurately fire at HTTP ports will still launch, but those that require additional information would fail (and not return any useful information). To provide additional information to Nmap scripts, you can provide arguments with the --script-args flag.

### Task

This is optional task, you can try it if it works for you.

From your previous command, find a script that has failed due to not providing arguments. Check information about that particular script on NSE documents. Check if you can provide the arguments and then run only that particular script.

# Writing Nmap Scripts

Because of the versatile and extremely customized nature of writing Nmap scripts, there are several different ways to produce a script that performs various functions—and also many pros and cons to write your own script. While creating an Nmap script from scratch may not always be the fastest way to get things done (as there is almost always a script that already exists for whatever purpose you may need), there are certain situations during which leveraging the powerful built-in functions of the Nmap scripting engine leads to exactly the right circumstances to write your own script.

In this section, we will look at:

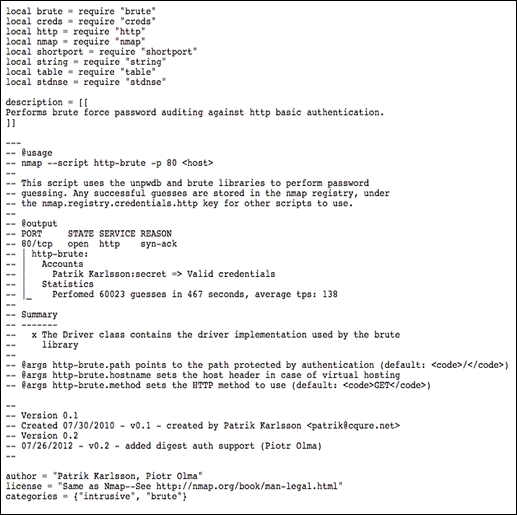
* Structure of an Nmap script
* Writing the Nmap script’s head
* Creating a rule
* Defining script’s action
* Debugging Nmap scripts

### Structure of an Nmap script

* **The head**: This section of an Nmap script includes documentation and categorization for the script so that Nmap and the NSE database can successfully categorize the script into the appropriate areas.
* **The rule**: This section of the script defines exactly where and how an Nmap script is executed. Because the script is leveraging the data of the Nmap scan as it runs, certain elements can trigger the script to run. This is effectively a trigger that evaluates whether or not the script should execute.
* **The action**: Lastly, this section of an Nmap script is where (you guessed it!) the action takes place. This is the part of the script that is doing a lot of the processing, after the head has defined the script and the rule has triggered the action.

### Defining an Nmap script – script headers

Each Nmap script must be created with certain required variables defined at the beginning of the script. Any Nmap prerequisites that are needed for successful execution, definitions of how the script is categorized (for example, whether or not it is intrusive, safe, contains an exploit, and so on), and the license are also among the things necessary in the header.



The preceding script illustrates the various sections necessary for an Nmap script, each of which are critical to the successful execution of the program. Let's walk through these elements in order to determine what the author of the script is doing.

First, several variables (defined by the local prefix) are defined. Several requirements are made in order to ensure that each of the Nmap elements are included appropriately.

Next, a longer variable—the description—is created. This is a multiline Lua variable, which is encapsulated in [[ and ]] brackets. This area is supposed to include a basic description of the Nmap script so that when run programmatically, it is possible to choose the correct script.

Below the description variable is a fully commented text block that defines the usage of the script. In Lua, the -- preamble comments out that line of code, making it not run when the script is executed. You can easily see how the @usage block is formatted—simply showing how the script should be run, and any arguments it may accept—and how the @output block is formatted below. These blocks show how to run the script correctly, how to pass arguments on the command line (if any are required), and what output you should expect from the script in question.

Below the commented out block are several other variable definitions that Nmap parses. Specifically, the author block (which is how you would like to be credited for the script), the license block (which is generally listed as the same as Nmap for distribution purposes, but can be specified in certain ways if you want to protect certain elements of your script), and the categories array (which lists the categories that the script should fall into). You want to make sure that, for example, if your script is intrusive, you label it as such.

### Task

Write a simple, easy-to-follow Nmap script that uses Nmap's built-in functionality (combined with the power of the NSE) to determine whether a web server has a robots.txt

Let us do this systematically; the steps are head, rule and action

Head:

For our script, we only need a few required includes, which make our header relatively short. Let's create our head section, looking something like the following (of course, feel free to modify your script however you like!):

Open leafnodes, which is a text editor. You will find this in Applications/Accessories. Copy and paste the following code. Save the file as robots.nse in a location, from where you can easily run it. Note the extension of the file, it is not txt. Examine the code carefully, look at the keywords, scope, brackets, operators and variable names.

local http = require "http"

local nmap = require "nmap"

description = [[

Checks to see if robots.txt exists on a web server.

]]

author = "Nmap Essentials readers"

license = "Same as Nmap--See http://nmap.org/book/man-legal.html"

categories = {"default", "discovery", "safe"}

We need the HTTP module in order to perform an HTTP GET request to robots.txt in question (on an open port 80), and of course we need the Nmap include in order to leverage the Nmap scripting engine.

Rule:

The rule or portrule section of an Nmap script determines when the action should take place. It's important to define this clearly so that we are confident that our script will run every time we need it to (based on port number and version). There are two ways to accomplish this type of rule: standard portrule documentation, and a helper library built in the NSE called **shortport**.

Defining a rule is actually very simple, depending on what we're looking for. In the case of our robots.txt detection script (aptly named robots.nse), we just want to trigger on port 80 to see if robots.txt exists.

If we were writing a production script, rather than a proof of concept, it would probably be a good idea to use shortport's port or service functionality to trigger on port 80, or any web server that Nmap detects through its underlying functionality. However, in our case, we can simply define something much easier to digest:

portrule = function(host, port)

return port.state == "open"

end

As you can see, this is a very minimal portrule that will return true when port.state is open. These are built-in Nmap functionalities, and when the script is running, each port is checked against the portrule.

While our portrule is intentionally very easy to understand, many production scripts have very complicated portrules that are designed to trigger different elements of analysis, based on specific version and configuration settings. To learn more about advanced portrule and the shortport library, you can read the full overviews at the **Nmap Scripting Engine Documentation** (**NSEDoc**) portal.

Action:

After we define the portrule, the only step left is to define the action that executes when the portrule returns true. In our case, we want to check whether robots.txt exists on the web server we're scanning.

In order to determine whether the server exists, there's a little bit about the Hypertext Transfer Protocol (HTTP) that we need to learn. First of all, the way to request a page is through an HTTP GET request. For example, if we wanted to go to <http://google.com/images>, our browser would send a request containing GET /images to the server at Google.com.

If the status of the GET request is OK, the web server returns the status code 200. If there is a server-side error, a 500 error will return. If the page is moved, an error in the range of 300 will return. Lastly (for our purposes), if there is an authorization error or file-not-found error, the server will return 403 or 404 respectively.

In order to define our action function, we need to perform the following steps:

1. Request the robots.txt page.
2. Find out whether it's there or not.

The following action segments define this request:

action = function(host, port)

local robots = http.get(host, port, "/robots.txt")

if robots.status == 200 then

return "robots.txt status 200"

else

return "robots.txt status: " .. robots.status

end

end

As you can see in the preceding code snippet, this is a very simple action section. Let's walk through the process step by step:

1. First, we define the action function that takes the parameter's host and port. These are automatically passed to the action block once the portrule triggers.
2. Next, we define a local variable (called robots), which is the HTTP result of the NSE's http.get request. In this instance, we're performing a GET to the host and port that we're currently scanning and making a request to /robots.txt.
3. Once we receive the HTTP data, we can easily make an if statement to determine whether the status is a 200 OK response or something else. We could have combined this with a shorter if statement (rather than an if/else), but it's useful to see how to have multiple possibilities for output.
4. If the output is not 200, we go to the else statement and see what the status is. For example, if the status is 404, we know that it simply doesn't exist; if we get a 500 server error or a 403 not authorized, however, it might be worth looking into a greater depth:

### Task

Now you have got a complete your own script

Run the script on scanme target and check port 80.

Note the HTTP status code.

### Task

Run the script on dshaw.net and check port 80.

Note HTTP status code.

Why the status code of both results are different?

### --script-trace

One last set of flags that can be very useful to write, understand, and debug Nmap scripts are the --script-trace and -d (debug) flags. The --script-trace flag shows the information on the wire about all the different requests that the script is making on its own, which is very useful to determine what exactly is happening.

You can see in the preceding screenshot that while there may be a little bit of information overload, you can see exactly what the Nmap script is doing by using the --script-trace flag. The -d flag, to debug, works similarly: if you're writing a script and you encounter errors, try debugging it with the -d flag. You'd be surprised at the great things you can learn!

# Reference/Acknowledgement

Acknowledgement and Reference: Information in this particular lab has been adopted from book Nmap essentials. The book is not available in Library now. It will be available soon for further reading.