Nmap Scripting Engine Documentation

Table of Contents

1. Introduction

Note

This PDF version of the NSE documentation was prepared for the presentation by Fyodor and David Fifield at the Black Hat Briefings Las Vegas 2010. While reading this will certainly help you master the Nmap Scripting Engine, we aim to make our talk useful, informative, and entertaining even for folks who haven't. You can read the latest version of this online at *http://nmap.org/book/nse.html*.

The Nmap Scripting Engine (NSE) is one of Nmap's most powerful and flexible features. It allows users to write (and share) simple scripts to automate a wide variety of networking tasks. Those scripts are then executed in parallel with the speed and efficiency you expect from Nmap. Users can rely on the growing and diverse set of scripts distributed with Nmap, or write their own to meet custom needs.

We designed NSE to be versatile, with the following tasks in mind:

Network discovery
: This is Nmap's bread and butter. Examples include looking up whois data based on the target domain, querying ARIN, RIPE, or APNIC for the target IP to determine ownership, performing identd lookups on open ports, SNMP queries, and listing available NFS/SMB/RPC shares and services.

More sophisticated version detection
: The Nmap version detection system (*http://nmap.org/book/vscan.html*) is able to recognize thousands of different services through its probe and regular expression signature based matching system, but it cannot recognize everything. For example, identifying the Skype v2 service requires two independent probes, which version detection isn't flexible enough to handle. Nmap could also recognize more SNMP services if it tried a few hundred different community names by brute force. Neither of these tasks are well suited to traditional Nmap version detection, but both are easily accomplished with NSE. For these reasons, version detection now calls NSE by default to handle some tricky services. This is described in Section 11, “Version Detection Using NSE” [37].

Vulnerability detection
: When a new vulnerability is discovered, you often want to scan your networks quickly to identify vulnerable systems before the bad guys do. While Nmap isn't a comprehensive vulnerability scanner, NSE is powerful enough to handle even demanding vulnerability checks. Many vulnerability detection scripts are already available and we plan to distribute more as they are written.

Backdoor detection

Many attackers and some automated worms leave backdoors to enable later reentry. Some of these can be detected by Nmap's regular expression based version detection. For example, within hours of the MyDoom worm hitting the Internet, Jay Moran posted an Nmap version detection probe and signature so that others could quickly scan their networks for MyDoom infections. NSE is needed to reliably detect more complex worms and backdoors.

Vulnerability exploitation

As a general scripting language, NSE can even be used to exploit vulnerabilities rather than just find them. The capability to add custom exploit scripts may be valuable for some people (particularly penetration testers), though we aren't planning to turn Nmap into an exploitation framework such as Metasploit[1].

These listed items were our initial goals, and we expect Nmap users to come up with even more inventive uses for NSE.

Scripts are written in the embedded Lua programming language[2]. The language itself is well documented in the books *Programming in Lua, Second Edition* and *Lua 5.1 Reference Manual*. The reference manual is also freely available online[3], as is the first edition of *Programming in Lua*[4]. Given the availability of these excellent general Lua programming references, this document only covers aspects and extensions specific to Nmap's scripting engine.

NSE is activated with the `-sC` option (or `--script` if you wish to specify a custom set of scripts) and results are integrated into Nmap normal and XML output. Two types of scripts are supported: service and host scripts. Service scripts relate to a certain open port (service) on the target host, and any results they produce are included next to that port in the Nmap output port table. Host scripts, on the other hand, run no more than once against each target IP and produce results below the port table. Example 1 shows a typical script scan. Service scripts producing output in this example are `ssh-hostkey`, which provides the system's RSA and DSA SSH keys, and `rpcinfo`, which queries portmapper to enumerate available services. The only host script producing output in this example is `smb-os-discovery`, which collects a variety of information from SMB servers. Nmap discovered all of this information in a third of a second.

[1] *http://www.metasploit.com*
[2] *http://www.lua.org/*
[3] *http://www.lua.org/manual/5.1/*
[4] *http://www.lua.org/pil/*

Example 1. Typical NSE output

```
# nmap -sC -p22,111,139 -T4 localhost

Starting Nmap ( http://nmap.org )
Interesting ports on flog (127.0.0.1):
PORT     STATE SERVICE
22/tcp   open  ssh
|  ssh-hostkey: 1024 b1:36:0d:3f:50:dc:13:96:b2:6e:34:39:0d:9b:1a:38 (DSA)
|_ 2048 77:d0:20:1c:44:1f:87:a0:30:aa:85:cf:e8:ca:4c:11 (RSA)
111/tcp  open  rpcbind
|  rpcinfo:
|  100000  2,3,4    111/udp  rpcbind
|  100024  1      56454/udp  status
|_ 100000  2,3,4    111/tcp  rpcbind
139/tcp  open  netbios-ssn

Host script results:
|  smb-os-discovery: Unix
|  LAN Manager: Samba 3.0.31-0.fc8
|_ Name: WORKGROUP

Nmap done: 1 IP address (1 host up) scanned in 0.33 seconds
```

2. Usage and Examples

While NSE has a complex implementation for efficiency, it is strikingly easy to use. Simply specify `-sC` to enable the most common scripts. Or specify the `--script` option to choose your own scripts to execute by providing categories, script file names, or the name of directories full of scripts you wish to execute. You can customize some scripts by providing arguments to them via the `--script-args` option. The two remaining options, `--script-trace` and `--script-updatedb`, are generally only used for script debugging and development. Script scanning is also included as part of the `-A` (aggressive scan) option.

Script scanning is normally done in combination with a port scan, because scripts may be run or not run depending on the port states found by the scan. With the `-sn` option it is possible to run a script scan without a port scan, only host discovery. In this case only host scripts will be eligible to run. To run a script scan with neither a host discovery nor a port scan, use the `-Pn -sn` options together with `-sC` or `--script`. Every host will be assumed up and still only host scripts will be run. This technique is useful for scripts like `whois.nse` that only use the remote system's address and don't require it to be up.

Scripts are not run in a sandbox and thus could accidentally or maliciously damage your system or invade your privacy. Never run scripts from third parties unless you trust the authors or have carefully audited the scripts yourself.

2.1. Script Categories

NSE scripts define a list of categories they belong to. Currently defined categories are `auth`, `default`, `discovery`, `external`, `fuzzer`, `intrusive`, `malware`, `safe`, `version`, and `vuln`. Category names are not case sensitive. The following list describes each category.

`auth`
: These scripts try to determine authentication credentials on the target system, often through a brute-force attack. Examples include `snmp-brute`, `http-auth`, and `ftp-anon`.

`default`
: These scripts are the default set and are run when using the `-sC` or `-A` options rather than listing scripts with `--script`. This category can also be specified explicitly like any other using `--script=default`. Many factors are considered in deciding whether a script should be run by default:

  Speed
  : A default scan must finish quickly, which excludes brute force authentication crackers, web spiders, and any other scripts which can take minutes or hours to scan a single service.

  Usefulness
  : Default scans need to produce valuable and actionable information. If even the script author has trouble explaining why an average networking or security professional would find the output valuable, the script should not run by default. The script may still be worth including in Nmap so that administrators can run for those occasions when they do need the extra information.

  Verbosity
  : Nmap output is used for a wide variety of purposes and needs to be readable and concise. A script which frequently produces pages full of output should not be added to the `default` category. When there is no important information to report, NSE scripts (particularly default ones) should return nothing. Checking for an obscure vulnerability may be OK by default as long as it only produces output when that vulnerability discovered.

  Reliability
  : Many scripts use heuristics and fuzzy signature matching to reach conclusions about the target host or service. Examples include `sniffer-detect` and `sql-injection`. If the script is often wrong, it doesn't belong in the `default` category where it may confuse or mislead casual users. Users who specify a script or category directly are generally more advanced and likely know how the script works or at least where to find its documentation.

  Intrusiveness
  : Some scripts are very intrusive because they use significant resources on the remote system, are likely to crash the system or service, or are likely to be perceived as an attack by the remote administrators. The more intrusive a script is, the less suitable it is for the `default` category. Default scripts are almost always in the `safe` category too, though we occasionally allow `intrusive` scripts by default when they are only mildly intrusive and score well in the other factors.

  Privacy
  : Some scripts, particularly those in the `external` category described later, divulge information to third parties by their very nature. For example, the `whois` script must divulge the target IP address to regional whois registries. We have also considered (and decided against) adding scripts which check target SSH and SSL key fingerprints against Internet weak key databases. The more privacy-invasive a script is, the less suitable it is for `default` category inclusion.

We don't have exact thresholds for each of these criteria, and many of them are subjective. All of these factors are considered together when making a decision whether to promote a script into the `default` category. A few default scripts are `identd-owners` (determines the username running remote services using identd), `http-auth` (obtains authentication scheme and realm of web sites requiring authentication), and `ftp-anon` (tests whether an FTP server allows anonymous access).

`discovery`

These scripts try to actively discover more about the network by querying public registries, SNMP-enabled devices, directory services, and the like. Examples include `html-title` (obtains the title of the root path of web sites), `smb-enum-shares` (enumerates Windows shares), and `snmp-sysdescr` (extracts system details via SNMP).

`external`

Scripts in this category may send data to a third-party database or other network resource. An example of this is `whois`, which makes a connection to whois servers to learn about the address of the target. There is always the possibility that operators of the third-party database will record anything you send to them, which in many cases will include your IP address and the address of the target. Most scripts involve traffic strictly between the scanning computer and the client; any that do not are placed in this category.

`fuzzer`

This category contains scripts which are designed to send server software unexpected or randomized fields in each packet. While this technique can useful for finding undiscovered bugs and vulnerabilities in software, it is both a slow process and bandwidth intensive. An example of a script in this category is `dns-fuzz`, which bombards a DNS server with slightly flawed domain requests until either the server crashes or a user specified time limit elapses.

`intrusive`

These are scripts that cannot be classified in the `safe` category because the risks are too high that they will crash the target system, use up significant resources on the target host (such as bandwidth or CPU time), or otherwise be perceived as malicious by the target's system administrators. Examples are `http-open-proxy` (which attempts to use the target server as an HTTP proxy) and `snmp-brute` (which tries to guess a device's SNMP community string by sending common values such as `public`, `private`, and `cisco`). Unless a script is in the special `version` category, it should be categorized as either `safe` or `intrusive`.

`malware`

These scripts test whether the target platform is infected by malware or backdoors. Examples include `smtp-strangeport`, which watches for SMTP servers running on unusual port numbers, and `auth-spoof`, which detects identd spoofing daemons which provide a fake answer before even receiving a query. Both of these behaviors are commonly associated with malware infections.

`safe`

Scripts which weren't designed to crash services, use large amounts of network bandwidth or other resources, or exploit security holes are categorized as `safe`. These are less likely to offend remote administrators, though (as with all other Nmap features) we cannot guarantee that they won't ever cause adverse reactions. Most of these perform general network discovery. Examples are `ssh-hostkey` (retrieves an SSH host key) and `html-title` (grabs the title from a web page). Scripts in the `version` category are not categorized by safety, but any other scripts which aren't in `safe` should be placed in `intrusive`.

`version`
: The scripts in this special category are an extension to the version detection feature and cannot be selected explicitly. They are selected to run only if version detection (`-sV`) was requested. Their output cannot be distinguished from version detection output and they do not produce service or host script results. Examples are `skypev2-version`, `pptp-version`, and `iax2-version`.

`vuln`
: These scripts check for specific known vulnerabilities and generally only report results if they are found. Examples include `realvnc-auth-bypass` and `xampp-default-auth`.

2.2. Command-line Arguments

These are the five command line arguments specific to script-scanning:

`-sC`
: Performs a script scan using the default set of scripts. It is equivalent to `--script=default`. Some of the scripts in this `default` category are considered intrusive and should not be run against a target network without permission.

`--script <filename>|<category>|<directory>|<expression>|all[,...]`
: Runs a script scan using the comma-separated list of filenames, script categories, and directories. Each element in the list may also be a Boolean expression describing a more complex set of scripts. Each element is interpreted first as an expression, then as a category, and finally as a file or directory name. The special argument `all` makes every script in Nmap's script database eligible to run. The `all` argument should be used with caution as NSE may contain dangerous scripts including exploits, brute force authentication crackers, and denial of service attacks.

  File and directory names may be relative or absolute. Absolute names are used directly. Relative paths are looked for in the following places until found:

  `--datadir`
  `$NMAPDIR`
  `~/.nmap` (not searched on Windows)
  `NMAPDATADIR`
  the current directory
  A `scripts` subdirectory is also tried in each of these.

  When a directory name is given, Nmap loads every file in the directory whose name ends with `.nse`. All other files are ignored and directories are not searched recursively. When a filename is given, it does not have to have the `.nse` extension; it will be added automatically if necessary.

  See Section 2.3, “Script Selection” [8] for examples and a full explanation of the `--script` option.

  Nmap scripts are stored in a `scripts` subdirectory of the Nmap data directory by default (see *http://nmap.org/book/data-files.html*). For efficiency, scripts are indexed in a database stored in `scripts/script.db`, which lists the category or categories in which each script belongs. The argument `all` will execute all scripts in the Nmap script database, but should be used cautiously since Nmap may contain exploits, denial of service attacks, and other dangerous scripts.

`--script-args <args>`
Provides arguments to the scripts. See Section 2.4, “Arguments to Scripts” [9] for a detailed explanation.

`--script-trace`
This option is similar to `--packet-trace`, but works at the application level rather than packet by packet. If this option is specified, all incoming and outgoing communication performed by scripts is printed. The displayed information includes the communication protocol, source and target addresses, and the transmitted data. If more than 5% of transmitted data is unprintable, hex dumps are given instead. Specifying `--packet-trace` enables script tracing too.

`--script-updatedb`
This option updates the script database found in `scripts/script.db` which is used by Nmap to determine the available default scripts and categories. It is only necessary to update the database if you have added or removed NSE scripts from the default `scripts` directory or if you have changed the categories of any script. This option is used by itself without arguments: **nmap --script-updatedb**.

Some other Nmap options have effects on script scans. The most prominent of these is `-sV`. A version scan automatically executes the scripts in the `version` category. The scripts in this category are slightly different than other scripts because their output blends in with the version scan results and they do not produce any script scan output.

Another option which affects the scripting engine is `-A`. The aggressive Nmap mode implies the `-sC` option.

2.3. Script Selection

The `--script` option takes a comma-separated list of categories, filenames, and directory names. Some simple examples of its use:

nmap --script default,safe
Loads all scripts in the `default` and `safe` categories.

nmap --script smb-os-discovery
Loads only the `smb-os-discovery.nse` script. Note that the `.nse` extension is optional.

nmap --script default,banner,/home/user/customscripts
Loads the script in the `default` category, the `banner.nse` script, and all `.nse` files in the directory `/home/user/customscripts`.

When referring to scripts from `script.db` by name, you can use a shell-style ‘`*`’ wildcard.

nmap --script "http-*"
Loads all scripts whose name starts with `http-`, such as `http-auth.nse` and `http-open-proxy.nse`. The argument to `--script` had to be in quotes to protect the wildcard from the shell.

More complicated script selection can be done using the `and`, `or`, and `not` operators to build Boolean expressions. The operators have the same precedence as in Lua: `not` is the highest, followed by `and` and then `or`. You can alter precedence by using parentheses. Because expressions contain space characters it is necessary to quote them.

nmap --script "not intrusive"
Loads every script except for those in the `intrusive` category.

nmap --script "default or safe"
This is functionally equivalent to **nmap --script "default,safe"**. It loads all scripts that are in the `default` category or the `safe` category or both.

nmap --script "default and safe"
Loads those scripts that are in *both* the `default` and `safe` categories.

nmap --script "(default or safe or intrusive) and not http-*"
Loads scripts in the `default`, `safe`, or `intrusive` categories, except for those whose names start with `http-`.

Names in a Boolean expression may be a category, a filename from `script.db`, or `all`. A name is any sequence of characters not containing ‘ ’, ‘`,`’, ‘`(`’, ‘`)`’, or ‘`;`’, except for the sequences `and`, `or`, and `not`, which are operators.

2.4. Arguments to Scripts

Arguments may be passed to NSE scripts using the `--script-args` option. The arguments describe a table of key-value pairs and possibly array values. The arguments are provided to scripts as a table in the registry called `nmap.registry.args`.

The syntax for script arguments is similar to Lua's table constructor syntax. Arguments are a comma-separated list of `name=value` pairs. Names and values may be strings not containing whitespace or the characters ‘`{`’, ‘`}`’, ‘`=`’, or ‘`,`’. To include one of these characters in a string, enclose the string in single or double quotes. Within a quoted string, ‘`\`’ escapes a quote. A backslash is only used to escape quotation marks in this special case; in all other cases a backslash is interpreted literally.

Values may also be tables enclosed in `{}`, just as in Lua. A table may contain simple string values, for example a list of proxy hosts; or more name-value pairs, including nested tables. Nested subtables are commonly used to pass arguments specific to one script, in a table named after the script. That is what is happening with the `whois` table in the example below.

Here is a typical Nmap invocation with script arguments:

```
nmap -sC --script-args 'user=foo,pass=",{}=bar",whois={whodb=nofollow+ripe},userdb=C:\Some\Path\To\Fi
```

Notice that the script arguments are surrounded in single quotes. This prevents the shell from interpreting the double quotes and doing automatic string concatenation. The command results in this Lua table:

```
{user="foo",pass=",{}=bar",whois={whodb="nofollow+ripe"},userdb="C:\\Some\\Path\\To\\F
```

You could then access the username `"foo"` inside your script with this statement:

```
local username = nmap.registry.args.user
```

The online NSE Documentation Portal at *http://nmap.org/nsedoc/* lists the arguments that each script accepts.

2.5. Complete Examples

nmap -sC example.com
A simple script scan using the default set of scripts.

nmap -sn -sC example.com
A script scan without a port scan; only host scripts are eligible to run.

nmap -Pn -sn -sC example.com
A script scan without host discovery or a port scan. All hosts are assumed up and only host scripts are eligible to run.

nmap --script smb-os-discovery --script-trace example.com
Execute a specific script with script tracing.

nmap --script snmp-sysdescr --script-args snmpcommunity=admin example.com
Run an individual script that takes a script argument.

nmap --script mycustomscripts,safe example.com
Execute all scripts in the `mycustomscripts` directory as well as all scripts in the `safe` category.

3. Script Format

NSE scripts consist of two–five descriptive fields along with either a port or host rule defining when the script should be executed and an action block containing the actual script instructions. Values can be assigned to the descriptive fields just as you would assign any other Lua variables. Their names must be lowercase as shown in this section.

3.1. `description` Field

The `description` field describes what a script is testing for and any important notes the user should be aware of. Depending on script complexity, the description may vary from a few sentences to a few paragraphs. The first paragraph should be a brief synopsis of the script function suitable for stand-alone presentation to the user. Further paragraphs may provide much more script detail.

3.2. `categories` Field

The `categories` field defines one or more categories to which a script belongs (see Section 2.1, “Script Categories” [4]). The categories are case-insensitive and may be specified in any order. They are listed in an array-style Lua table as in this example:

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

3.3. `author` Field

The `author` field contains the script authors' names and can also contain contact information (such as home page URLs). We no longer recommend including email addresses because spammers might scrape them

from the nsedoc web site. This optional field is not used by NSE, but gives script authors their due credit or blame.

3.4. `license` Field

Nmap is a community project and we welcome all sorts of code contributions, including NSE scripts. So if you write a valuable script, don't keep it to yourself! The optional `license` field helps ensure that we have legal permission to distribute all the scripts which come with Nmap. All of those scripts currently use the standard Nmap license (described in *http://nmap.org/book/man-legal.html*). They include the following line:

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

The Nmap license is similar to the GNU GPL. Script authors may use a BSD-style license (no advertising clause) instead if they prefer that.

3.5. `dependencies` Field

The `dependencies` field is an array containing the names of scripts that should run before this script. This is used when one script can make use of the results of another. For example, most of the `smb-*` scripts depend on `smb-brute`, because the accounts found by `smb-brute` may allow the other scripts to get more information.

When we say "depend on", we mean it in a loose sense. That is, a script will still run despite missing dependencies. Given the dependencies, the script will run after all the scripts listed in the dependencies array. This is an example of the `dependencies` table from `smb-os-discovery`:

```
dependencies = {"smb-brute"}
```

The dependencies table is optional. NSE will assume the script has no dependencies if the field is omitted.

Dependencies establish an internal ordering of scripts, assigning each one a number called a "runlevel"[5]. When running your scripts you will see the runlevel (along with the total number of runlevels) of each grouping of scripts run in NSE's output:

```
NSE: Script scanning 127.0.0.1.
NSE: Starting runlevel 1 (of 3) scan.
Initiating NSE at 17:38
Completed NSE at 17:38, 0.00s elapsed
NSE: Starting runlevel 2 (of 3) scan.
Initiating NSE at 17:38
Completed NSE at 17:38, 0.00s elapsed
NSE: Starting runlevel 3 (of 3) scan.
Initiating NSE at 17:38
Completed NSE at 17:38, 0.00s elapsed
NSE: Script Scanning completed.
```

[5]Up through Nmap version 5.10BETA2, dependencies didn't exist and script authors had to set a `runlevel` field manually.

3.6. Port and Host Rules

Nmap uses the script rules to determine whether a script should be run against a target. A script contains either a *port rule*, which governs which ports of a target the scripts may run against, or a *host rule*, which specifies that the script should be run only once against a target IP and only if the given conditions are met. A rule is a Lua function that returns either `true` or `false`. The script *action* is only performed if its rule evaluates to `true`. Host rules accept a host table as their argument and may test, for example, the IP address or hostname of the target. A port rule accepts both host and port tables as arguments for any TCP or UDP port in the `open`, `open|filtered`, or `unfiltered` port states. Port rules generally test factors such as the port number, port state, or listening service name in deciding whether to run against a port. Example rules are shown in Section 8.2, “The Rule” [25].

3.7. Action

The action is the heart of an NSE script. It contains all of the instructions to be executed when the script's port or host rule triggers. It is a Lua function which accepts the same arguments as the rule and can return either `nil` or a string. If a string is returned by a service script, the string and script's filename are printed in the Nmap port table output. A string returned by a host script is printed below the port table. No output is produced if the script returns `nil`. For an example of an NSE action refer to Section 8.3, “The Mechanism” [25].

4. Script Language

The core of the Nmap Scripting Engine is an embeddable Lua interpreter. Lua is a lightweight language designed for extensibility. It offers a powerful and well documented API for interfacing with other software such as Nmap.

The second part of the Nmap Scripting Engine is the NSE Library, which connects Lua and Nmap. This layer handles issues such as initialization of the Lua interpreter, scheduling of parallel script execution, script retrieval and more. It is also the heart of the NSE network I/O framework and the exception handling mechanism. It also includes utility libraries to make scripts more powerful and convenient. The utility library modules and extensions are described in Section 6, “NSE Libraries” [13].

4.1. Lua Base Language

The Nmap scripting language is an embedded Lua[6] interpreter which was extended with libraries for interfacing with Nmap. The Nmap API is in the Lua namespace `nmap`. This means that all calls to resources provided by Nmap have an `nmap` prefix. `nmap.new_socket()`, for example, returns a new socket wrapper object. The Nmap library layer also takes care of initializing the Lua context, scheduling parallel scripts and collecting the output produced by completed scripts.

During the planning stages, we considered several programming languages as the base for Nmap scripting. Another option was to implement a completely new programming language. Our criteria were strict: NSE had to be easy to use, small in size, compatible with the Nmap license, scalable, fast and parallelizable. Several previous efforts (by other projects) to design their own security auditing language from scratch

[6] *http://www.lua.org/*

resulted in awkward solutions, so we decided early not to follow that route. First the Guile Scheme interpreter was considered, but the preference drifted towards the Elk interpreter due to its more favorable license. But parallelizing Elk scripts would have been difficult. In addition, we expect that most Nmap users prefer procedural programming over functional languages such as Scheme. Larger interpreters such as Perl, Python, and Ruby are well-known and loved, but are difficult to embed efficiently. In the end, Lua excelled in all of our criteria. It is small, distributed under the liberal MIT open source license, has coroutines for efficient parallel script execution, was designed with embeddability in mind, has excellent documentation, and is actively developed by a large and committed community. Lua is now even embedded in other popular open source security tools including the Wireshark sniffer and Snort IDS.

5. NSE Scripts

This section (a long list of NSE scripts with brief summaries) is omitted since we already provide a better online interface to the information at the NSE Documentation Portal[7].

6. NSE Libraries

In addition to the significant built-in capabilities of Lua, we have written or integrated many extension libraries which make script writing more powerful and convenient. These libraries (sometimes called modules) are compiled if necessary and installed along with Nmap. They have their own directory, `nselib`, which is installed in the configured Nmap data directory. Scripts need only `require` [8] the default libraries in order to use them.

6.1. List of All Libraries

This list is just an overview to give an idea of what libraries are available. Developers will want to consult the complete documentation at *http://nmap.org/nsedoc/*.

afp
: This library was written by Patrik Karlsson <patrik@cqure.net> to facilitate communication with the Apple AFP Service. It is not feature complete and still missing several functions.

asn1
: ASN1 functions.

backdoor
: This config file is designed for adding a backdoor to the system. It has a few options by default, only one enabled by default. I suggest

base64
: Base64 encoding and decoding. Follows RFC 4648.

bin
: Pack and unpack binary data.

[7] *http://nmap.org/nsedoc/*
[8] *http://www.lua.org/manual/5.1/manual.html#pdf-require*

bit
Bitwise operations on integers.

citrixxml
This module was written by Patrik Karlsson and facilitates communication with the Citrix XML Service. It is not feature complete and is missing several functions and parameters.

comm
Common communication functions for network discovery tasks like banner grabbing and data exchange.

datafiles
Read and parse some of Nmap's data files: `nmap-protocols`, `nmap-rpc`, `nmap-services`, and `nmap-mac-prefixes`.

db2
DB2 Library supporting a very limited subset of operations

default
More verbose network scripts

dns
Simple DNS library supporting packet creation, encoding, decoding, and querying.

drive
This configuration file pulls info about a given harddrive

experimental
This is the configuration file for modules that aren't quite ready for prime time yet.

http
Client-side HTTP library.

imap
IMAP functions.

ipOps
Utility functions for manipulating and comparing IP addresses.

json
Library methods for handling Json data. It handles json encoding and decoding

ldap
Library methods for handling LDAP.

listop
Functional-style list operations.

match
Buffered network I/O helper functions.

mongodb
Library methods for handling MongoDB, creating and parsing packets

msrpc
By making heavy use of the 'smb' library, this library will call various MSRPC functions. The functions used here can be accessed over TCP ports 445 and 139, with an established session. A NULL session (the default) will work for some functions and operating systems (or configurations), but not for others.

msrpcperformance
This module is designed to parse the `PERF_DATA_BLOCK` structure, which is stored in the registry under HKEY_PERFORMANCE_DATA. By querying this structure, you can get a whole lot of information about what's going on.

msrpctypes
This module was written to marshall parameters for Microsoft RPC (MSRPC) calls. The values passed in and out are based on structs defined by the protocol, and documented by Samba developers. For detailed breakdowns of the types, take a look at Samba 4.0's .idl files.

mssql
MSSQL Library supporting a very limited subset of operations

mysql
Simple MySQL Library supporting a very limited subset of operations *http://forge.mysql.com/wiki/MySQL_Internals_ClientServer_Protocol*

netbios
Creates and parses NetBIOS traffic. The primary use for this is to send NetBIOS name requests.

network
This is the default configuration file. It simply runs some built-in Window programs to gather information about the remote system. It's intended to be simple, demonstrate some of the concepts, and not break/alte anything.

nmap
Interface with Nmap internals.

nsedebug
Converts an arbitrary data type into a string. Will recursively convert tables. This can be very useful for debugging.

openssl
OpenSSL bindings.

packet
Facilities for manipulating raw packets.

pcre
Perl Compatible Regular Expressions.

pgsql
PostgreSQL library supporting both version 2 and version 3 of the protocol The library currently contains the bare minimum to perform authentication Authentication is supported with or without SSL enabled and using the plain-text or MD5 authentication mechanisms

pop3
POP3 functions.

proxy
Functions for proxy testing

pwdump
This config file is designed for running password-dumping scripts. So far, it supports pwdump6 2.0.0 and fgdump.

rpc
RPC Library supporting a very limited subset of operations

shortport
Functions for building short portrules.

smb
Implements functionality related to Server Message Block (SMB, also known as CIFS) traffic, which is a Windows protocol.

smbauth
This module takes care of the authentication used in SMB (LM, NTLM, LMv2, NTLMv2). There is a lot to this functionality, so if you're interested in how it works, read on.

snmp
SNMP functions.

ssh1
Functions for the SSH-1 protocol.

ssh2
Functions for the SSH-2 protocol.

stdnse
Standard Nmap Scripting Engine functions.

strbuf
String buffer facilities.

strict
Strict Declared Global library.

tab
Arrange output into tables.

unpwdb
Username/password database library.

url
URI parsing, composition, and relative URL resolution.

6.2. Hacking NSE Libraries

Libraries often accidentally make use of globals variables when local scope was intended. Two or more scripts that make use of library functions which unintentionally use the same global variable will find that variable constantly rewritten. This is a serious bug that can cause NSE to stall or a correct script to spectacularly fail, and, because Lua uses global-by-default scope assignment when it encounters a variable, this is also a common bug.

Consider a global variable being used by two different scripts, within the library, to hold sockets or data. When one script is yielded after storing data in the variable, another script awakens only to replace that data. In contrast, a local variable would store the information on the stack of the running script separate from others.

To help correct this problem, NSE now uses an adapted library from the standard Lua distribution called `strict.lua`. The library will raise a runtime error on any access or modification of a global variable which was undeclared in the file scope. A global variable is considered declared if the library makes an assignment to the global name (even `nil`) in the file scope.

6.3. Adding C Modules to Nselib

A few of the modules included in nselib are written in C or C++ rather than Lua. Two examples are `bit` and `pcre`. We recommend that modules be written in Lua if possible, but C and C++ may be more appropriate if performance is critical or (as with the `pcre` and `openssl` modules) you are linking to an existing C library. This section describes how to write your own compiled extensions to nselib.

The Lua C API is described at length in *Programming in Lua*, *Second Edition*, so this is a short summary. C modules consist of functions that follow the protocol of the lua_CFunction[9] type. The functions are registered with Lua and assembled into a library by calling the `luaL_register` function. A special initialization function provides the interface between the module and the rest of the NSE code. By convention the initialization function is named in the form `luaopen_<module>`.

The smallest compiled module that comes with NSE is `bit`, and one of the most straightforward is `openssl`. These modules serve as good examples for a beginning module writer. The source code for `bit` is found in `nse_bit.cc` and `nse_bit.h`, while the `openssl` source is in `nse_openssl.cc` and `nse_openssl.h`. Most of the other compiled modules follow this `nse_<module name>.cc` naming convention.

Reviewing the `openssl` module shows that one of the functions in `nse_openssl.cc` is `l_md5`, which calculates an MD5 digest. Its function prototype is:

```
static int l_md5(lua_State *L);
```

[9] *http://www.lua.org/manual/5.1/manual.html#lua_CFunction*

The prototype shows that `l_md5` matches the lua_CFunction type. The function is static because it does not have to be visible to other compiled code. Only an address is required to register it with Lua. Later in the file, `l_md5` is entered into an array of type luaL_reg and associated with the name `md5`:

```
static const struct luaL_reg openssllib[] = {
  { "md5", l_md5 },
  { NULL, NULL }
};
```

This function will now be known as `md5` to NSE. Next the library is registered with a call to `luaL_register` inside the initialization function `luaopen_openssl`, as shown next. Some lines relating to the registration of OpenSSL BIGNUM types have been omitted:

```
LUALIB_API int luaopen_openssl(lua_State *L) {
  luaL_register(L, OPENSSLLIBNAME, openssllib);
  return 1;
}
```

The function `luaopen_openssl` is the only function in the file that is exposed in `nse_openssl.h`. `OPENSSLLIBNAME` is simply the string `"openssl"`.

After a compiled module is written, it must be added to NSE by including it in the list of standard libraries in `nse_main.cc`. Then the module's source file names must be added to `Makefile.in` in the appropriate places. For both these tasks you can simply follow the example of the other C modules. For the Windows build, the new source files must be added to the `mswin32/nmap.vcproj` project file using MS Visual Studio (see *http://nmap.org/book/inst-windows.html#inst-win-source*).

7. Nmap API

NSE scripts have access to several Nmap facilities for writing flexible and elegant scripts. The API provides target host details such as port states and version detection results. It also offers an interface to the Nsock library for efficient network I/O.

7.1. Information Passed to a Script

An effective Nmap scripting engine requires more than just a Lua interpreter. Users need easy access to the information Nmap has learned about the target hosts. This data is passed as arguments to the NSE script's `action` method. The arguments, `host` and `port`, are Lua tables which contain information on the target against which the script is executed. If a script matched a hostrule, it gets only the `host` table, and if it matched a portrule it gets both `host` and `port`. The following list describes each variable in these two tables.

`host`
: This table is passed as a parameter to the rule and action functions. It contains information on the operating system run by the host (if the `-O` switch was supplied), the IP address and the host name of the scanned target.

`host.os`
: The `os` entry in the host table is an array of strings. The strings (as many as eight) are the names of the operating systems the target is possibly running. Strings are only entered in this array if the target machine

is a perfect match for one or more OS database entries. If Nmap was run without the `-O` option, then `host.os` is `nil`.

`host.ip`
: Contains a string representation of the IP address of the target host. If the scan was run against a host name and the reverse DNS query returned more than one IP addresses then the same IP address is used as the one chosen for the scan.

`host.name`
: Contains the reverse DNS entry of the scanned target host represented as a string. If the host has no reverse DNS entry, the value of the field is an empty string.

`host.targetname`
: Contains the name of the host as specified on the command line. If the target given on the command line contains a netmask or is an IP address the value of the field is `nil`.

`host.directly_connected`
: A Boolean value indicating whether or not the target host is directly connected to (i.e. on the same network segment as) the host running Nmap.

`host.mac_addr`
: MAC address of the destination host (six-byte long binary string) or `nil`, if the host is not directly connected.

`host.mac_addr_next_hop`
: MAC address of the first hop in the route to the host, or `nil` if not available.

`host.mac_addr_src`
: Our own MAC address, which was used to connect to the host (either our network card's, or (with `--spoof-mac`) the spoofed address).

`host.interface`
: A string containing the interface name (dnet-style) through which packets to the host are sent.

`host.bin_ip`
: The target host's IPv4 address as a 32-bit binary value.

`host.bin_ip_src`
: Our host's (running Nmap) source IPv4 address as a 32-bit binary value.

`port`
: The port table is passed to an NSE service script (i.e. only those with a portrule rather than a hostrule) in the same fashion as the host table. It contains information about the port against which the script is running. While this table is not passed to host scripts, port states on the target can still be requested from Nmap using the `nmap.get_port_state()` and `nmap.get_ports()` calls.

`port.number`
: Contains the port number of the target port.

`port.protocol`
: Defines the protocol of the target port. Valid values are `"tcp"` and `"udp"`.

`port.service`
: Contains a string representation of the service running on `port.number` as detected by the Nmap service detection. If the `port.version` field is `nil`, Nmap has guessed the service based on the port number. Otherwise version detection was able to determine the listening service and this field is equal to `port.version.name`.

`port.version`
: This entry is a table which contains information retrieved by the Nmap version scanning engine. Some of the values (such as service name, service type confidence, and the RPC-related values) may be retrieved by Nmap even if a version scan was not performed. Values which were not determined default to `nil`. The meaning of each value is given in the following table:

Table 1. `port.version` values

Name	Description
`name`	Contains the service name Nmap decided on for the port.
`name_confidence`	Evaluates how confident Nmap is about the accuracy of `name`, from 1 (least confident) to 10.
`product`, `version`, `extrainfo`, `hostname`, `ostype`, `devicetype`	These five variables are the same as those described under `<version info>` in *http://nmap.org/book/vscan-fileformat.html#vscan-db-match*.
`service_tunnel`	Contains the string `"none"` or `"ssl"` based on whether or not Nmap used SSL tunneling to detect the service.
`service_fp`	The service fingerprint, if any, is provided in this value. This is described in *http://nmap.org/book/vscan-community.html*.
`rpc_status`	Contains a string value of `good_prog` if we were able to determine the program number of an RPC service listening on the port, `unknown` if the port appears to be RPC but we couldn't determine the program number, `not_rpc` if the port doesn't appear be RPC, or `untested` if we haven't checked for RPC status.
`rpc_program`, `rpc_lowver`, `rpc_highver`	The detected RPC program number and the range of version numbers supported by that program. These will be `nil` if `rpc_status` is anything other than `good_prog`.

`port.state`
: Contains information on the state of the port. Service scripts are only run against ports in the `open` or `open|filtered` states, so `port.state` generally contains one of those values. Other values might appear if the port table is a result of the `get_port_state` or `get_ports` functions. You can adjust the port state using the `nmap.set_port_state()` call. This is normally done when an `open|filtered` port is determined to be `open`.

7.2. Network I/O API

To allow for efficient and parallelizable network I/O, NSE provides an interface to Nsock, the Nmap socket library. The smart callback mechanism Nsock uses is fully transparent to NSE scripts. The main benefit of NSE's sockets is that they never block on I/O operations, allowing many scripts to be run in parallel. The

I/O parallelism is fully transparent to authors of NSE scripts. In NSE you can either program as if you were using a single non-blocking socket or you can program as if your connection is blocking. Even blocking I/O calls return once a specified timeout has been exceeded. Two flavors of Network I/O are supported: connect-style and raw packet.

Connect-style network I/O

This part of the network API should be suitable for most classical network uses: Users create a socket, connect it to a remote address, send and receive data and finally close the socket. Everything up to the Transport layer (which is either TCP, UDP or SSL) is handled by the library.

An NSE socket is created by calling `nmap.new_socket`, which returns a socket object. The socket object supports the usual `connect`, `send`, `receive`, and `close` methods. Additionally the functions `receive_bytes`, `receive_lines`, and `receive_buf` allow greater control over data reception. Example 2 shows the use of connect-style network operations. The `try` function is used for error handling, as described in Section 7.3, “Exception Handling” [22].

Example 2. Connect-style I/O

```
require("nmap")

local socket = nmap.new_socket()
socket:set_timeout(1000)
try = nmap.new_try(function() socket:close() end)
try(socket:connect(host.ip, port.number))
try(socket:send("login"))
response = try(socket:receive())
socket:close()
```

Raw packet network I/O

For those cases where the connection-oriented approach is too high-level, NSE provides script developers with the option of raw packet network I/O.

Raw packet reception is handled through a Libpcap wrapper inside the Nsock library. The steps are to open a capture device, register listeners with the device, and then process packets as they are received.

The `pcap_open` method creates a handle for raw socket reads from an ordinary socket object. This method takes a callback function, which computes a packet hash from a packet (including its headers). This hash can return any binary string, which is later compared to the strings registered with the `pcap_register` function. The packet hash callback will normally extract some portion of the packet, such as its source address.

The pcap reader is instructed to listen for certain packets using the `pcap_register` function. The function takes a binary string which is compared against the hash value of every packet received. Those packets whose hashes match any registered strings will be returned by the `pcap_receive` method. Register the empty string to receive all packets.

A script receives all packets for which a listener has been registered by calling the `pcap_receive` method. The method blocks until a packet is received or a timeout occurs.

The more general the packet hash computing function is kept, the more scripts may receive the packet and proceed with their execution. To handle packet capture inside your script you first have to create a socket with `nmap.new_socket` and later close the socket with `socket_object:close`—just like with the connection-based network I/O.

While receiving packets is important, sending them is certainly a key feature as well. To accomplish this, NSE provides access to sending at the IP and Ethernet layers. Raw packet writes do not use the same socket object as raw packet reads, so the `nmap.new_dnet` function is called to create the required object for sending. After this, a raw socket or Ethernet interface handle can be opened for use.

Once the dnet object is created, the function `ip_open` can be called to initialize the object for IP sending. `ip_send` sends the actual raw packet, which must start with the IPv4 header. The dnet object places no restrictions on which IP hosts may be sent to, so the same object may be used to send to many different hosts while it is open. To close the raw socket, call `ip_close`.

For sending at a lower level than IP, NSE provides functions for writing Ethernet frames. `ethernet_open` initializes the dnet object for sending by opening an Ethernet interface. The raw frame is sent with `ethernet_send`. To close the handle, call `ethernet_close`.

Sometimes the easiest ways to understand complex APIs is by example. The `ipidseq.nse` script included with Nmap uses raw IP packets to test hosts for suitability for Nmap's Idle Scan (`-sI`). The `sniffer-detect.nse` script also included with Nmap uses raw Ethernet frames in an attempt to detect promiscuous-mode machines on the network (those running sniffers).

7.3. Exception Handling

NSE provides an exception handling mechanism which is not present in the base Lua language. It is tailored specifically for network I/O operations, and follows a functional programming paradigm rather than an object oriented one. The `nmap.new_try` API method is used to create an exception handler. This method returns a function which takes a variable number of arguments that are assumed to be the return values of another function. If an exception is detected in the return values (the first return value is false), then the script execution is aborted and no output is produced. Optionally, you can pass a function to `new_try` which will be called if an exception is caught. The function would generally perform any required cleanup operations.

Example 3 shows cleanup exception handling at work. A new function named `catch` is defined to simply close the newly created socket in case of an error. It is then used to protect connection and communication attempts on that socket. If no catch function is specified, execution of the script aborts without further ado—open sockets will remain open until the next run of Lua's garbage collector. If the verbosity level is at least one or if the scan is performed in debugging mode a description of the uncaught error condition is printed on standard output. Note that it is currently not easily possible to group several statements in one try block.

Example 3. Exception handling example

```
local result, socket, try, catch

result = ""
socket = nmap.new_socket()
catch = function()
socket:close()
end
try = nmap.new_try(catch)

try(socket:connect(host.ip, port.number))
result = try(socket:receive_lines(1))
try(socket:send(result))
```

Writing a function which is treated properly by the try/catch mechanism is straightforward. The function should return multiple values. The first value should be a Boolean which is `true` upon successful completion of the function and `false` otherwise. If the function completed successfully, the try construct consumes the indicator value and returns the remaining values. If the function failed then the second returned value must be a string describing the error condition. Note that if the value is not `nil` or `false` it is treated as `true` so you can return your value in the normal case and return `nil, <error description>` if an error occurs.

7.4. The Registry

The registry is a Lua table (accessible as `nmap.registry`) with the special property that it is visible by all scripts and retains its state between script executions. The registry is transient—it is not stored between Nmap executions. Every script can read and write to the registry. Scripts commonly use it to save information for other instances of the same script. For example, the `whois` and `asn-query` scripts may query one IP address, but receive information which may apply to tens of thousands of IPs on that network. Saving the information in the registry may prevent other script threads from having to repeat the query.

The registry may also be used to hand information to completely different scripts. For example, the `snmp-brute` script saves a discovered community name in the registry where it may be used by other SNMP scripts. Script which use the results of another script must declare it using the `dependencies` variable to make sure that the earlier script runs first.

Because every script can write to the registry table, it is important to avoid conflicts by choosing keys wisely (uniquely).

8. Script Writing Tutorial

Suppose that you are convinced of the power of NSE. How do you go about writing your own script? Let's say that you want to extract information from an identification server to determine the owner of the process listening on a TCP port. This is not really the purpose of identd (it is meant for querying the owner of outgoing connections, not listening daemons), but many identd servers allow it anyway. Nmap used to have this functionality (called ident scan), but it was removed while transitioning to a new scan engine architecture. The protocol identd uses is pretty simple, but still too complicated to handle with Nmap's version detection language. First, you connect to the identification server and send a query of the form `<port-on-server>,`

<*port-on-client*> and terminated with a newline character. The server should then respond with a string containing the server port, client port, response type, and address information. The address information is omitted if there is an error. More details are available in RFC 1413, but this description is sufficient for our purposes. The protocol cannot be modeled in Nmap's version detection language for two reasons. The first is that you need to know both the local and the remote port of a connection. Version detection does not provide this data. The second, more severe obstacle, is that you need two open connections to the target—one to the identification server and one to the listening port you wish to query. Both obstacles are easily overcome with NSE.

The anatomy of a script is described in Section 3, “Script Format” [10]. In this section we will show how the described structure is utilized.

8.1. The Head

The head of the script is essentially its meta information. This includes the fields: `description`, `categories`, `dependencies`, `author`, and `license` as well as initial NSEDoc information such as usage, args, and output tags (see Section 9, “Writing Script Documentation (NSEDoc)” [27]).

The description field should contain a paragraph or more describing what the script does. If anything about the script results might confuse or mislead users, and you can't eliminate the issue by improving the script or results text, it should be documented in the `description`. If there are multiple paragraphs, the first is used as a short summary where necessary. Make sure that first paragraph can serve as a stand alone abstract. This description is short because it is such a simple script:

```
description = [[
Attempts to find the owner of an open TCP port by querying an auth
(identd - port 113) daemon which must also be open on the target system.
]]
```

Next comes NSEDoc information. This script is missing the common `@usage` and `@args` tags since it is so simple, but it does have an NSEDoc `@output` tag:

```
---
--@output
-- 21/tcp   open     ftp       ProFTPD 1.3.1
-- |_ auth-owners: nobody
-- 22/tcp   open     ssh       OpenSSH 4.3p2 Debian 9etch2 (protocol 2.0)
-- |_ auth-owners: root
-- 25/tcp   open     smtp      Postfix smtpd
-- |_ auth-owners: postfix
-- 80/tcp   open     http      Apache httpd 2.0.61 ((Unix) PHP/4.4.7 ...)
-- |_ auth-owners: dhapache
-- 113/tcp  open     auth?
-- |_ auth-owners: nobody
-- 587/tcp  open     submission Postfix smtpd
-- |_ auth-owners: postfix
-- 5666/tcp open     unknown
-- |_ auth-owners: root
```

Next come the `author`, `license`, and `categories` tags. This script belongs to the `safe` because we are not using the service for anything it was not intended for. Because this script is one that should run by default it is also in the `default` category. Here are the variables in context:

```
author = "Diman Todorov"

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

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

8.2. The Rule

The rule section is a Lua method which decides whether to skip or execute the script's action method against a particular service or host. This decision is usually based on the host and port information passed to the rule function. In the case of the identification script, it is slightly more complicated than that. To decide whether to run the identification script against a given port we need to know if there is an auth server running on the target machine. In other words, the script should be run only if the currently scanned TCP port is open and TCP port 113 is also open. For now we will rely on the fact that identification servers listen on TCP port 113. Unfortunately NSE only gives us information about the currently scanned port.

To find out if port 113 is open, we use the `nmap.get_port_state` function. If the auth port was not scanned, the `get_port_state` function returns `nil`. So we check that the table is not `nil`. We also check that both ports are in the `open` state. If this is the case, the action is executed, otherwise we skip the action.

```
portrule = function(host, port)
        local auth_port = { number=113, protocol="tcp" }
        local identd = nmap.get_port_state(host, auth_port)

        if
                identd ~= nil
                and identd.state == "open"
                and port.protocol == "tcp"
                and port.state == "open"
        then
                return true
        else
                return false
        end
end
```

8.3. The Mechanism

At last we implement the actual functionality! The script first connects to the port on which we expect to find the identification server, then it will connect to the port we want information about. Doing so involves first creating two socket options by calling `nmap.new_socket`. Next we define an error-handling `catch` function which closes those sockets if failure is detected. At this point we can safely use object methods such as `open`, `close`, `send` and `receive` to operate on the network socket. In this case we call `connect` to make the connections. NSE's exception handling mechanism is used to avoid excessive error-handling code.

We simply wrap the networking calls in a `try` call which will in turn call our `catch` function if anything goes wrong.

If the two connections succeed, we construct a query string and parse the response. If we received a satisfactory response, we return the retrieved information.

```
action = function(host, port)
        local owner = ""

        local client_ident = nmap.new_socket()
        local client_service = nmap.new_socket()

        local catch = function()
                client_ident:close()
                client_service:close()
        end

        local try = nmap.new_try(catch)

        try(client_ident:connect(host.ip, 113))
        try(client_service:connect(host.ip, port.number))

        local localip, localport, remoteip, remoteport =
                try(client_service:get_info())

        local request = port.number .. ", " .. localport .. "\n"

        try(client_ident:send(request))

        owner = try(client_ident:receive_lines(1))

        if string.match(owner, "ERROR") then
                owner = nil
        else
                owner = string.match(owner, "USERID : .+ : (.+)\n", 1)
        end

        try(client_ident:close())
        try(client_service:close())

        return owner
end
```

Note that because we know that the remote port is stored in `port.number`, we could have ignored the last two return values of `client_service:get_info()` like this:

```
local localip, localport = try(client_service:get_info())
```

In this example we exit quietly if the service responds with an error. This is done by assigning `nil` to the `owner` variable which will be returned. NSE scripts generally only return messages when they succeed, so they don't flood the user with pointless alerts.

9. Writing Script Documentation (NSEDoc)

Scripts are used by more than just their authors, so they require good documentation. NSE modules need documentation so developers can use them in their scripts. NSE's documentation system, described in this section, aims to meet both these needs. While reading this section, you may want to browse NSE's online documentation, which is generated using this system. It is at *http://nmap.org/nsedoc/*.

NSE uses a customized version of the LuaDoc[10] documentation system called NSEDoc. The documentation for scripts and modules is contained in their source code, as comments with a special form. Example 4 is an NSEDoc comment taken from the `stdnse.print_debug()` function.

Example 4. An NSEDoc comment for a function

```
--- Prints a formatted debug message if the current verbosity level is greater
-- than or equal to a given level.
--
-- This is a convenience wrapper around
-- <code>nmap.print_debug_unformatted()</code>. The first optional numeric
-- argument, <code>verbosity</code>, is used as the verbosity level necessary
-- to print the message (it defaults to 1 if omitted). All remaining arguments
-- are processed with Lua's <code>string.format()</code> function.
-- @param level Optional verbosity level.
-- @param fmt Format string.
-- @param ... Arguments to format.
```

Documentation comments start with three dashes: `---`. The body of the comment is the description of the following code. The first paragraph of the description should be a brief summary, with the following paragraphs providing more detail. Special tags starting with `@` mark off other parts of the documentation. In the above example you see `@param`, which is used to describe each parameter of a function. A complete list of the documentation tags is found in Section 9.1, “NSE Documentation Tags” [29].

Text enclosed in the HTML-like `<code>` and `</code>` tags will be rendered in a monospace font. This should be used for variable and function names, as well as multi-line code examples. When a sequence of lines start with the characters “`*  `”, they will be rendered as a bulleted list.

It is good practice to document every public function and table in a script or module. Additionally every script and module should have its own file-level documentation. A documentation comment at the beginning of a file (one that is not followed by a function or table definition) applies to the entire file. File-level documentation can and should be several paragraphs long, with all the high-level information useful to a developer using a module or a user running a script. Example 5 shows documentation for the `comm` module (with a few paragraphs removed to save space).

[10] *http://luadoc.luaforge.net/*

Example 5. An NSEDoc comment for a module

```
--- Common communication functions for network discovery tasks like
-- banner grabbing and data exchange.
--
-- These functions may be passed a table of options, but it's not required. The
-- keys for the options table are <code>"bytes"</code>, <code>"lines"</code>,
-- <code>"proto"</code>, and <code>"timeout"</code>. <code>"bytes"</code> sets
-- a minimum number of bytes to read. <code>"lines"</code> does the same for
-- lines. <code>"proto"</code> sets the protocol to communicate with,
-- defaulting to <code>"tcp"</code> if not provided. <code>"timeout"</code>
-- sets the socket timeout (see the socket function <code>set_timeout()</code>
-- for details).
-- @author Kris Katterjohn 04/2008
-- @copyright Same as Nmap--See http://nmap.org/book/man-legal.html
```

There are some special considerations for documenting scripts rather than functions and modules. In particular, scripts have special variables for some information which would otherwise belongs in @-tag comments (script variables are described in Section 3, “Script Format” [10]). In particular, a script's description belongs in the `description` variable rather than in a documentation comment, and the information that would go in `@author` and `@copyright` belong in the variables `author` and `license` instead. NSEDoc knows about these variables and will use them in preference to fields in the comments. Scripts should also have an `@output` tag showing sample output, as well as `@args` and `@usage` where appropriate. Example 6 shows proper form for script-level documentation, using a combination of documentation comments and NSE variables.

Example 6. An NSEDoc comment for a script

```
description = [[
Maps IP addresses to autonomous system (AS) numbers.

The script works by sending DNS TXT queries to a DNS server which in
turn queries a third-party service provided by Team Cymru
(team-cymru.org) using an in-addr.arpa style zone set up especially for
use by Nmap.
]]

---
-- @usage
-- nmap --script asn-query.nse [--script-args dns=<DNS server>] <target>
-- @args dns The address of a recursive nameserver to use (optional).
-- @output
-- Host script results:
-- |  AS Numbers:
-- |  BGP: 64.13.128.0/21 | Country: US
-- |    Origin AS: 10565 SVCOLO-AS - Silicon Valley Colocation, Inc.
-- |      Peer AS: 3561 6461
-- |  BGP: 64.13.128.0/18 | Country: US
-- |    Origin AS: 10565 SVCOLO-AS - Silicon Valley Colocation, Inc.
-- |_     Peer AS: 174 2914 6461

author = "jah, Michael"
license = "Same as Nmap--See http://nmap.org/book/man-legal.html"
categories = {"discovery", "external"}
```

Compiled NSE modules are also documented with NSEDoc, even though they have no Lua source code. Each compiled module has a file `<modulename>.luadoc` that is kept in the `nselib` directory alongside the Lua modules. This file lists and documents the functions and tables in the compiled module as though they were written in Lua. Only the name of each function is required, not its definition (not even `end`). You must use the `@name` and `@class` tags when documenting a table to assist the documentation parser in identifying it. There are several examples of this method of documentation in the Nmap source distribution (including `nmap.luadoc`, `bit.luadoc`, and `pcre.luadoc`).

9.1. NSE Documentation Tags

The following tags are understood by NSEDoc:

`@param`
: Describes a function parameter. The first word following `@param` is the name of the parameter being described. The tag should appear once for each parameter of a function.

`@see`
: Adds a cross-reference to another function or table.

`@return`
: Describes a return value of a function. `@return` may be used multiple times for multiple return values.

`@usage`
: Provides a usage example of a function, script, or module. In the case of a function, the example is Lua code; for a script it is an Nmap command line; and for a module it is usually a code sample. `@usage` may be given more than once. If it is omitted in a script, NSEDoc generates a default standardized usage example.

`@name`
: Defines a name for the function or table being documented. This tag is normally not necessary because NSEDoc infers names through code analysis.

`@class`
: Defines the "class" of the object being documented: `function`, `table`, or `module`. Like `@name`, this is normally inferred automatically.

`@field`
: In the documentation of a table, `@field` describes the value of a named field.

`@args`
: Describes a script argument, as used with the `--script-args` option (see Section 2.4, "Arguments to Scripts" [9]). The first word after `@args` is the name of the argument, and everything following that is the description. This tag is special to script-level comments.

`@output`
: This tag, which is exclusive to script-level comments, shows sample output from a script.

`@author`
: This tag, which may be given multiple times, lists the authors of an NSE module. For scripts, use the `author` variable instead.

`@copyright`
: This tag describes the copyright status of a module. For scripts, use the `license` variable instead.

10. Script Parallelism in NSE

Before now, we have only lightly touched on the steps NSE takes to allow multiple scripts to execute in parallel. Usually, the author need not concern himself with how any of this is implemented; however, there are a couple cases that warrant discussion that we will cover in this section. As a script writer, you may need to control how multiple scripts interact in a library; you may require multiple threads to work in parallel; or perhaps you need to serialize access to a remote resource.

The standard mechanism for parallel execution is a thread. A thread encapsulates execution flow and data of a script using the Lua `thread` or `coroutine`. A Lua thread allows us to yield the current script at arbitrary points to continue work on another script. Typically, these yield points are blocking calls to the NSE Socket library. The yield back to NSE is also transparent; the script is unaware of the transition and views each socket method as a blocking call.

Let's go over some common terminology. A *script* is analogous to a binary executable; it holds the information necessary to execute our script. A *thread* (a Lua coroutine) is analogous to a process; it runs a script against a host and possibly port. We sometimes abuse our terminology throughout the book by referring to a thread

as a running script. We are really saying the "instantiation of the script", in the same sense that a process is the instantiation of an executable.

NSE provides the bare-bone essentials you need to expand your degree of parallelism beyond the basic script thread: new independent threads, Mutexes, and Condition Variables. We will go into depth on each of these mechanisms in the following sections.

10.1. Worker Threads

There are several instances where a script needs finer control with respect to parallel execution beyond what is offered by default with a generic script. The common reason for this need is the inability for a script to read from multiple sockets concurrently. For example, an HTTP spidering script may want to have multiple Lua threads querying web server resources in parallel. To solve this problem, NSE offers the function `stdnse.new_thread` to create worker threads. These worker threads have all the power of independent scripts with the only restriction that they may not report Script Output.

Each worker thread launched by a script is given a main function and a variable number of arguments to be passed to the main function by NSE:

```
worker_thread, status_function = stdnse.new_thread(main, ...)
```

You are given back the Lua thread (coroutine) that uniquely identifies your worker thread and a status query function that queries the status of your new worker.

The status query function returns two values:

```
status, error_object = status_function()
```

The first return value, `status`, is simply the return value of `coroutine.status` on the worker thread coroutine (more precisely, the `base` coroutine, read more about `base` coroutine in the section called “The Base Thread” [36]). The second return value contains the error object thrown that ended the worker thread or `nil` if no error was thrown. This object is typically a string, like most Lua errors. However, recall that any Lua type can be an error object, even `nil`! You should inspect the error object, the second return value, only if the status of your worker is `"dead"`.

NSE discards all return values from the main function when the worker thread finishes execution. You should communicate with your worker through the use of `main` function parameters, upvalues, or function environments. You will see how to do this in Example 7.

Finally, when using worker threads you should always use condition variables and Mutexes to coordinate with your worker threads. Keep in mind that Nmap is single threaded so there are no (memory) issues in synchronization to worry about; however, there is resource contention. Your resources are usually network bandwidth, network sockets, etc. Condition variables are also useful if the work for any single thread is dynamic. For example, a web server spider script with a pool of workers will initially have a single root html document. Following the retrieval of the root document, the set of resources to be retrieved (the worker's work) will become very large (an html document adds many new hyperlinks (resources) to fetch).

Example 7. Worker Thread Example

```
local requests = {"/", "/index.html", --[[ long list of objects ]]}

function thread_main (host, port, responses, ...)
  local condvar = nmap.condvar(responses);
  local what = {n = select("#", ...), ...};
  local allReqs = nil;
  for i = 1, what.n do
    allReqs = http.pGet(host, port, what[i], nil, nil, allReqs);
  end
  local p = assert(http.pipeline(host, port, allReqs));
  for i, response in ipairs(p) do responses[#responses+1] = response end
  condvar "signal";
end

function many_requests (host, port)
  local threads = {};
  local responses = {};
  local condvar = nmap.condvar(responses);
  local i = 1;
  repeat
    local j = math.min(i+10, #requests);
    local co = stdnse.new_thread(thread_main, host, port, responses,
        unpack(requests, i, j));
    threads[co] = true;
    i = j+1;
  until i > #requests;
  repeat
    condvar "wait";
    for thread in pairs(threads) do
      if coroutine.status(thread) == "dead" then threads[thread] = nil end
    end
  until next(threads) == nil;
  return responses;
end
```

For brevity, this example omits typical behavior of a traditional web spider. The requests table is assumed to contain a number of objects (hundreds or thousands) to warrant the use of worker threads. Our example will dispatch a new thread with `11` relative Uniform Resource Identifiers (URI) to request, up to the length of the `requests` table. Worker threads are very cheap so we are not afraid to create a lot of them. After we dispatch this large number of threads, we wait on our Condition Variable until every thread has finished then finally return the responses table.

You may have noticed that we did not use the status function returned by `stdnse.new_thread`. You will typically use this for debugging or if your program must stop based on the error thrown by one of your worker threads. Our simple example did not require this but a fault tolerant library may.

10.2. Thread Mutexes

Recall from the beginning of this section that each script execution thread (e.g. `ftp-anon` running against an FTP server on a target host) yields to other scripts whenever it makes a call on network objects (sending

or receiving data). Some scripts require finer concurrency control over thread execution. An example is the `whois` script which queries whois servers for each target IP address. Because many concurrent queries often result in getting one's IP banned for abuse, and because a single query may return additional information for targets other threads are running against, it is useful to have other threads pause while one thread performs a query.

To solve this problem, NSE includes a `mutex` function which provides a mutex[11] (mutual exclusion object) usable by scripts. The Mutex allows for only one thread to be working on an object. Competing threads waiting to work on this object are put in the waiting queue until they can get a "lock" on the Mutex. A solution for the `whois` problem above is to have each thread block on a Mutex using a common string, thus ensuring that only one thread is querying whois servers at once. When finished querying the remote servers, the thread can store results in the NSE registry and unlock the Mutex. Other scripts waiting to query the remote server can then obtain a lock, check for usable results retrieved from previous queries, make their own queries, and unlock the Mutex. This is a good example of serializing access to a remote resource.

The first step in using a Mutex is to create one via a call to the `nmap` library:

```
mutexfn = nmap.mutex(object)
```

The `mutexfn` returned is a function which works as a Mutex for the `object` passed in. This object can be any Lua data type except `nil`, `booleans`, and `numbers`. The returned function allows you to lock, try to lock, and release the Mutex. Its first and only parameter must be one of the following:

`"lock"`
: Make a blocking lock on the Mutex. If the Mutex is busy (another thread has a lock on it), then the thread will yield and wait. The function returns with the Mutex locked.

`"trylock"`
: Makes a non-blocking lock on the Mutex. If the Mutex is busy then it immediately returns with a return value of `false`. Otherwise the Mutex locks the Mutex and returns `true`.

`"done"`
: Releases the Mutex and allows another thread to lock it. If the thread does not have a lock on the Mutex, an error will be raised.

`"running"`
: Returns the thread locked on the Mutex or `nil` if the Mutex is not locked. This should only be used for debugging as it interferes with garbage collection of finished threads.

NSE maintains a weak reference to the Mutex so other calls to `nmap.mutex` with the same object will return the same function (Mutex); however, if you discard your reference to the Mutex then it may be collected; and, subsequent calls to `nmap.mutex` with the object will return a different Mutex function! Thus you should save your Mutex to a (local) variable that persists for the entire time you require.

A simple example of using the API is provided in Example 8. For real-life examples, read the `asn-query.nse` and `whois.nse` scripts in the Nmap distribution.

[11] *http://en.wikipedia.org/wiki/Mutual_exclusion*

Example 8. Mutex manipulation

```
local mutex = nmap.mutex("My Script's Unique ID");
function action(host, port)
  mutex "lock";
  -- Do critical section work - only one thread at a time executes this.
  mutex "done";
  return script_output;
end
```

10.3. Condition Variables

Condition Variables arose out of a need to coordinate with worker threads created using the `stdnse.new_thread` function. A Condition Variable allows one or more threads to wait on an object and one or more threads to awaken one or all threads waiting on the object. Said differently, multiple threads may unconditionally `block` on the Condition Variable by *waiting*. Other threads may wake up one or all of the waiting threads via *signalling* the Condition Variable.

As an example, we may dispatch multiple worker threads that will produce results for us to use, like our earlier Example 7 [32]. Until all the workers finish, our master thread must sleep. Note that we cannot `poll` for results like in a traditional Operating System thread because NSE does not preempt Lua threads. Instead, we use a Condition Variable that the master thread *waits* on until awakened by a worker. The master will continually wait until all workers have terminated.

The first step in using a Condition Variable is to create one via a call to the `nmap` library:

```
condvarfn = nmap.condvar(object)
```

The semantics for Condition Variables are similar to Mutexes. The `condvarfn` returned is a function which works as a Condition Variable for the `object` passed in. This object can be any Lua data type except `nil`, `booleans`, and `numbers`. The returned function allows you to wait, signal, and broadcast on the Condition Variable. Its first and only parameter must be one of the following:

`"wait"`
: Wait on the Condition Variable. This adds your thread to the waiting queue for the Condition Variable. You will resume execution when another thread signals or broadcasts on the Condition Variable.

`"signal"`
: Signal the Condition Variable. A thread in the Condition Variable's waiting queue will be resumed.

`"broadcast"`
: Signal all threads in the Condition Variable's waiting queue.

Like with Mutexes, NSE maintains a weak reference to the Condition Variable so other calls to `nmap.condvar` with the same object will return the same function (Condition Variable); however, if you discard your reference to the Condition Variable then it may be collected; and, subsequent calls to `nmap.condvar` with the object will return a different Condition Variable function! Thus you should save your Condition Variable to a (local) variable that persists for the entire time you require.

When using Condition Variables, it is important to check the predicate before and after waiting. A predicate is a test on whether to continue doing work within your worker or master thread. For your worker threads, this will at the very least include a test to see if the master thread is still alive. You do not want to continue doing work when no thread will use your results. A typical test before waiting may be: check whether the master is still running, if not then quit; check that there is work to be done; if not then wait.

NSE does not guarantee spurious wakeups will not occur; that is, there is no guarantee your thread will not be awakened when no thread called `"signal"` or `"broadcast"` on the Condition Variable. The typical, but not only, reason for a spurious wakeup is the termination of a thread using a Condition Variable. This is an important guarantee NSE makes that allows you to avoid deadlock where a worker or master waits for a thread to wake them up that ended without signaling the Condition Variable.

10.4. Collaborative Multithreading

One of Lua's least known features is collaborative multithreading through *coroutines*. A coroutine provides an independent execution stack that is *resumable*. The standard `coroutine` provides access to the creation and manipulation of coroutines. Lua's online first edition of Programming in Lua[12] contains an excellent introduction to *coroutines*. We will provide an overview of the use of coroutines here for completeness but this is no replacement for reviewing PiL.

We have mentioned coroutines throughout this section as *threads*. This is the *type* (`thread`) of a coroutine in Lua. Users of NSE that have any parallel programming experience with Operating System threads may be confused by this. As a reminder, Nmap is single threaded. Lua threads provide the basis for parallel scripting but only one thread is ever running at a time.

A Lua `function` executes on top of a Lua `thread`. The thread maintains a stack of active functions, local variables, and the current instruction. We can switch between coroutines by explicitly *yielding* the running thread. The coroutine which *resumed* the yielded thread resumes operation. Example 9 shows a brief use of coroutines to print numbers.

Example 9. Basic Coroutine Use

```
local function main ()
  coroutine.yield(1)
  coroutine.yield(2)
  coroutine.yield(3)
end
local co = coroutine.create(main)
for i = 1, 3 do
  print(coroutine.resume(co))
end
--> true    1
--> true    2
--> true    3
```

What you should take from this example is the ability to transfer between flows of control extremely easily through the use of `coroutine.yield`. This is an extremely powerful concept that enables NSE to run scripts in parallel. All scripts are run as coroutines that yield whenever they make a blocking socket function

[12] *http://www.lua.org/pil/*

call. This enables NSE to run other scripts and later resume the blocked script when its I/O operation has completed.

As a script writer, there are times when coroutines are the best tool for a job. One common use in socket programming is to filter data. You may produce a function that generates all the links from an HTML document. An iterator using `string.gmatch` only catchs a single pattern. Because some complex matches may take many different Lua patterns, it is more appropriate to use a coroutine. Example 10 shows how to do this.

Example 10. Link Generator

```
function links (html_document)
  local function generate ()
    for m in string.gmatch(html_document, "url%((.-)%)") do
      coroutine.yield(m) -- css url
    end
    for m in string.gmatch(html_document, "href%s*=%s*\"(.-)\"") do
      coroutine.yield(m) -- anchor link
    end
    for m in string.gmatch(html_document, "src%s*=%s*\"(.-)\"") do
      coroutine.yield(m) -- img source
    end
  end
  return coroutine.wrap(generate)
end

function action (host, port)
  -- ... get HTML document and store in html_document local
  for link in links(html_document) do
    links[#links+1] = link; -- store it
  end
  -- ...
end
```

There are many other instances where coroutines may provide an easier solution to a problem. It takes experience from use to help identify those cases.

The Base Thread

Because scripts may use coroutines for their own multithreading, it is important to be able to identify an *owner* of a resource or to establish whether the script is still alive. NSE provides the function `stdnse.base` for this purpose.

Particularly when writing a library that attributes ownership of a cache or socket to a script, you may use the base thread to establish whether the script is still running. `coroutine.status` on the base thread will give the current state of the script. In cases where the script is `"dead"`, you will want to release the resource. Be careful with keeping references to these threads; NSE may discard a script even though it has not finished executing. The thread will still report a status of `"suspended"`. You should keep a weak reference to the thread in these cases so that it may be collected.

11. Version Detection Using NSE

The version detection system built into Nmap was designed to efficiently recognize the vast majority of protocols with a simple probe and pattern matching syntax. Some protocols require more complex communication than version detection can handle. A generalized scripting language as provided by NSE is perfect for these tough cases.

NSE's `version` category contains scripts that enhance standard version detection. Scripts in this category are run whenever you request version detection with `-sV`; you don't need to use `-sC` to run these. This cuts the other way too: if you use `-sC`, you won't get `version` scripts unless you also use `-sV`.

One protocol which we were unable to detect with normal version detection is Skype version 2. The protocol was likely designed to frustrate detection out of a fear that telecom-affiliated Internet service providers might consider Skype competition and interfere with the traffic. Yet we did find one way to detect it. If Skype receives an HTTP GET request, it pretends to be a web server and returns a 404 error. But for other requests, it sends back a chunk of random-looking data. Proper identification requires sending two probes and comparing the two responses—an ideal task for NSE. The simple NSE script which accomplishes this is shown in Example 11.

Example 11. A typical version detection script (Skype version 2 detection)

```
description = [[
Detects the Skype version 2 service.
]]
author = "Brandon Enright"
license = "Same as Nmap--See http://nmap.org/book/man-legal.html"
categories = {"version"}

require "comm"

portrule = function(host, port)
        return (port.number == 80 or port.number == 443 or
                port.service == nil or port.service == "" or
                port.service == "unknown")
               and port.protocol == "tcp" and port.state == "open"
               and port.service ~= "http" and port.service ~= "ssl/http"
end

action = function(host, port)
        local status, result = comm.exchange(host, port,
                "GET / HTTP/1.0\r\n\r\n", {bytes=26, proto=port.protocol})
        if (not status) then
                return
        end
        if (result ~= "HTTP/1.0 404 Not Found\r\n\r\n") then
                return
        end
        -- So far so good, now see if we get random data for another request
        status, result = comm.exchange(host, port,
                "random data\r\n\r\n", {bytes=15, proto=port.protocol})

        if (not status) then
                return
        end
        if string.match(result, "[^%s!-~].*[^%s!-~].*[^%s!-~]") then
                -- Detected
                port.version.name = "skype2"
                port.version.product = "Skype"
                nmap.set_port_version(host, port, "hardmatched")
                return
        end
        return
end
```

If the script detects Skype, it augments its `port` table with now-known `name` and `product` fields. It then sends this new information to Nmap by calling `nmap.set_port_version`. Several other version fields are available to be set if they are known, but in this case we only have the name and product. For the full list of version fields, refer to the `nmap.set_port_version` documentation.

Notice that this script does nothing unless it detects the protocol. A script shouldn't produce output (other than debug output) just to say it didn't learn anything.

12. Example Script: `finger.nse`

The finger script (`finger.nse`) is a perfect example of a short and simple NSE script.

First the information fields are assigned. A detailed description of what the script actually does goes in the `description` field.

```
description = [[
Attempts to get a list of usernames via the finger service.
]]
author = "Eddie Bell"
license = "Same as Nmap--See http://nmap.org/book/man-legal.html"
```

The `categories` field is a table containing all the categories the script belongs to—These are used for script selection with the `--script` option:

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

You can use the facilities provided by the nselib (Section 6, “NSE Libraries” [13]) with `require`. Here we want to use common communication functions and shorter port rules:

```
require "comm"
require "shortport"
```

We want to run the script against the finger service. So we test whether it is using the well-known finger port (`79/tcp`), or whether the service is named “finger” based on version detection results or in the port number's listing in `nmap-services`:

```
portrule = shortport.port_or_service(79, "finger")
```

First, the script uses `nmap.new_try` to create an exception handler that will quit the script in case of an error. Next, it passes control to `comm.exchange`, which handles the network transaction. Here we have asked to wait in the communication exchange until we receive at least 100 lines, wait at least 5 seconds, or until the remote side closes the connection. Any errors are handled by the `try` exception handler. The script returns a string if the call to `comm.exchange()` was successful.

```
action = function(host, port)
 local try = nmap.new_try()

 return try(comm.exchange(host, port, "\r\n",
         {lines=100, proto=port.protocol, timeout=5000}))
end
```

13. Implementation Details

Now it is time to explore the NSE implementation details in depth. Understanding how NSE works is useful for designing efficient scripts and libraries. The canonical reference to the NSE implementation is the source code, but this section provides an overview of key details. It should be valuable to folks trying to understand and extend the NSE source code, as well as to script authors who want to better-understand how their scripts are executed.

13.1. Initialization Phase

During its initialization stage, Nmap loads the Lua interpreter and its provided libraries. These libraries are fully documented in the Lua Reference Manual[13]. Here is a summary of the libraries, listed alphabetically by their namespace name:

`debug`
: The debug library provides a low-level API to the Lua interpreter, allowing you to access functions along the execution stack, retrieve function closures and object metatables, and more.

`io`
: The Input/Output library offers functions such as reading from files or from the output from programs you execute.

`math`
: Numbers in Lua usually correspond to the `double` C type, so the math library provides access to rounding functions, trigonometric functions, random number generation, and more.

`os`
: The Operating System library provides system facilities such as filesystem operations (including file renaming or removal and temporary file creation) and system environment access.

`package`
: Among the functions provided by Lua's package-lib is `require`, which is used to load nselib modules.

`string`
: The string library provides functions for manipulating Lua strings, including printf-style string formatting, pattern matching using Lua-style patterns, substring extraction, and more.

`table`
: The table manipulation library is essential for operating on Lua's central data structure (tables).

In addition to loading the libraries provided by Lua, the `nmap` namespace functions are loaded. The search paths are the same directories that Nmap searches for its data files, except that the `nselib` directory is appended to each. At this stage any provided script arguments are stored inside the registry.

The next phase of NSE initialization is loading the selected scripts, based on the defaults or arguments provided to the `--script` option. The `version` category scripts are loaded as well if version detection was enabled. NSE first tries to interpret each `--script` argument as a category. This is done with a Lua C function in `nse_init.cc` named `entry` based on data from the `script.db` script categorization database. If the category is found, those scripts are loaded. Otherwise Nmap tries to interpret `--script` arguments as files or directories. If no files or directories with a given name are found in Nmap's search path, an error is raised and the Script Engine aborts.

If a directory is specified, all of the `.nse` files inside it are loaded. Each loaded file is executed by Lua. If a *portrule* is present, it is saved in the *porttests* table with a portrule key and file closure value. Otherwise, if the script has a *hostrule*, it is saved in the *hosttests* table in the same manner.

[13] *http://www.lua.org/manual/5.1/manual.html*

13.2. Matching Scripts with Targets

After initialization is finished, the `hostrules` and `portrules` are evaluated for each host in the current target group. The rules of every chosen script is tested against every host and (in the case of service scripts) each `open` and `open|filtered` port on the hosts. The combination can grow quite large, so portrules should be kept as simple as possible. Save any heavy computation for the script's `action`.

Next, a Lua thread is created for each of the matching script-target combinations. Each thread is stored with pertinent information such as its dependencies, target, target port (if applicable), host and port tables (passed to the `action`), and the script type (service or host script). The `mainloop` function then processes each runlevel grouping of threads in order.

13.3. Script Execution

Nmap performs NSE script scanning in parallel by taking advantage of Nmap's Nsock parallel I/O library and the Lua coroutines [14] language feature. Coroutines offer collaborative multi-threading so that scripts can suspend themselves at defined points and allow other coroutines to execute. Network I/O, particularly waiting for responses from remote hosts, often involves long wait times, so this is when scripts yield to others. Key functions of the Nsock wrapper cause scripts to yield (pause). When Nsock finishes processing such a request, it makes a callback which causes the script to be pushed from the waiting queue back into the running queue so it can resume operations when its turn comes up again.

The `mainloop` function moves threads between the waiting and running queues as needed. A thread which yields is moved from the running queue into the waiting list. Running threads execute until they either yield, complete, or fail with an error. Threads are made ready to run (placed in the running queue) by calling `process_waiting2running`. This process of scheduling running threads and moving threads between queues continues until no threads exist in either queue.

[14] *http://www.lua.org/manual/5.1/manual.html#2.11*