Nmap NSE development for offense and defense

Paulino Calderon (@calderpwn)
&
Tom Sellers (@tomsellers)

Contact information

Paulino Calderon
@calderpwn
http://calderonpale.com
paulino@calderonpale.com

Tom Sellers
@tomsellers
https://fadedlab.wordpress.com/
tom@fadedcode.net

Agenda

Introduction to the Nmap Scripting Engine

- Brief history
- NSE script categories
- Installing NSE scripts
- Basic NSE usage
- Advanced NSE usage

Agenda

Lua

- Introduction
- Quick Reference Guide for Lua
  - Data types
  - Common data structures
  - Conditional statements
  - Flow control statements
  - String handling
  - Captures and patterns
  - Coroutines
  - I/O operations

Agenda

NSE development

- Format and required/optional tags
- Execution rules
- Code standards & how to submit contributions
- Creating version detection scripts and signatures
- Writing brute force scripts using the brute library
- Writing spidering scripts for web applications
- Receiving and transmitting network data in NSE
- Exploiting and reporting security vulnerabilities in NSE
- Creating your NSE libraries

Introduction to the Nmap Scripting Engine

Nmap Scripting Engine

- The Nmap Scripting Engine extends the capabilities of Nmap.
- Enhances Nmap's output and behavior (e.g. adding targets, additional checks, etc)
- Over +500 official scripts so far.
- Official documentation: https://nmap.org/nsedoc/

NSE categories

AUTH - These scripts deal with authentication credentials (or bypassing them) on the target system
BROADCAST - Scripts in this category typically do discovery of hosts not listed on the command line by broadcasting on the local network
BRUTE - These scripts use brute force attacks to guess authentication credentials of a remote server.
DEFAULT - These scripts are the default set and are run when using the -sC or -A options
DISCOVERY - These scripts try to actively discover more about the network by querying public registries, SNMP-enabled devices, directory services, and the like.
DOS - Scripts in this category may cause a denial of service.
EXPLOIT - These scripts aim to actively exploit some vulnerability.
EXTERNAL - Scripts in this category may send data to a third-party database or other network resource.

FUZZER - This category contains scripts which are designed to send server software unexpected or randomized fields in each packet.
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.
MALWARE - These scripts test whether the target platform is infected by malware or backdoors.
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.
VERSION - The scripts in this special category are an extension to the version detection feature and cannot be selected explicitly.
VULN - These scripts check for specific known vulnerabilities and generally only report results if they are found.

Installing NSE scripts

Place the script in your local scripts folder and update the database with --script-updatedb:

- **#nmap --script-updatedb**

Script vault

Scripts don’t always make it to the official repository, check the script vault for some really cool contributions.

https://secwiki.org/w/Nmap/Script_Vault

NSE usage: Script selection

Select scripts with --script:

- **nmap --script http-title <target>**
- **nmap -p80 --script http-iis-short-names-brute <target>**

You may select multiple scripts by separating them with a comma:

- **nmap --script http-title,http-methods <target>**

NSE usage: Script selection

You may select scripts by NSE category:

- **nmap --script exploit <target>**
- **nmap -p80 --script discovery,intrusive <target>**

You may select all scripts inside a local folder:

- **nmap --script /usr/local/scripts/ <target>**

NSE Usage: Advanced script selection

You may use expressions to select scripts:

- **nmap --script "not(intrusive or dos or exploit)" -sV <target>**
- **nmap --script "http-* and not(http-slowloris or http-wordpress-* or http-drupal-* or http-enum or http-form-fuzzer or http-iis-short-name-brute)" <target>**
- **nmap --script "http-* and not(exploit)" <target>**

Note how you may combine names and NSE categories in expressions.

NSE Usage: Relative paths

If you use relative URLs Nmap will try to find scripts automatically from the following folders:

- --datadir
- $NMAPDIR
- ~/.nmap
- %HOMEPATH%\AppData\Roaming\nmap
- The directory containing the Nmap executable
- The directory containing the Nmap executable followed by “ ../share/nmap”
- NMAPDATADIR

NSE Usage: Force execution of scripts

Execution rules of scripts may be overwritten to force the execution of scripts:

- **nmap --script +<script selection> <target>**
- **nmap --script +http-title -p1212 <target>**

NSE Usage: Script arguments

Use --script-args to set script arguments:

- **nmap --script <script> --script-args <script args> <target>**
- **nmap -sV --script http-title --script-args http.useragent="Mozilla 123" <target>**

NSE Usage: Script arguments

You may use script argument short alias to avoid typing the script name before each argument:

- **$nmap -p80 --script http-trace --script-args http-trace.path=/path/ <target>**
- **$nmap -p80 --script http-trace --script-args path=/path/ <target>**

NSE Usage: Script argument file

Store script arguments in a file and use --script-args-file:

- **$nmap -p80 --script http-trace --script-args-file nmap-args.txt <target>**

NSE Usage: Debugging the execution of scripts

Use -d[1-9] to enable debugging and set the debugging level accordingly:

- **nmap -d --script http-iis-short-name-brute <target>**
- **nmap -d2 --script http-iis-short-name-brute <target>**

Reading XML results with xmlstartlet

Xmlstarlet allow us to navigate XML files, works great for parsing Nmap results:

- **xmlstarlet el -a myscan.xml**
- **xmlstarlet el -v myscan.xml**
- **xmlstarlet sel -t -m "//host[ports/port/state/@state='open']" -v "address [@addrtype='ipv4']/@addr" -n myscan.xml**

Lua

Introduction to Lua

- Powerful language yet small.
- A basic Lua engine, including parser/compiler/interpreter, but excluding standard libraries, weighs in at under 100kb. (*)
- Used by many other products including Wireshark, Photoshop, Snort, World of Warcraft, NetBSD drivers and many other commercial products...

Introduction to Lua

Similar to other scripting languages as Python or Ruby:

```
$cat hello.lua
#!/usr/bin/lua
print("Name?")
name = io.read()
print("Hello " .. name)
```

Quick reference guide for Lua

We compiled a quick reference guide to get you started right away with Lua. See QuickRefGuideLua.pdf.

In this guide you will find everything you need to know about the programming language syntax, flow control statements, data types, I/O operations, patterns, captures, co-routines and much more.

Developing scripts

Development resources

- Development mailing list: **http://seclists.org/nmap-dev/**
- Code standards: **https://secwiki.org/w/Nmap/Code_Standards**
- Vim rules: **https://raw.githubusercontent.com/cldrn/nmap-nse-scripts/master/.vimrc**
- Graphical IDE: **https://github.com/s4n7h0/Halcyon**

Script structure

```
description = [[
]]

---
-- @usage
--
-- @output
--
-- @xmloutput
--
-- @args
--
---

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

portrule =

action = function(host, port)

end
```

NSE libraries

There are more than 110 NSE libraries available. To include a library in your NSE script you declare the library as follows:

local http = require ”http”
local stdnse = require “stdnse”
local bin = require “bin”

Required script fields

The script fields available are:

- **description**: Describes what a script is testing for and any important notes the user should be aware of.
- **categories**: One or more categories to which a script belongs.
- **author**: Author of the script.
- **license**: Usually Nmap's license.
- **dependencies**: Array containing names of scripts that should run before the script (Optional)

Script types

The scripts can be classified depending on their execution rules:

- Prerule scripts
- Host scripts
- Service scripts
- Portrule scripts

Execution rules

NSE scripts must declare a host or port rule as the execution rule of the script.

At least one of the following functions must be defined:

- **prerule()**
- **hostrule(host)**
- **portrule(host, port)**
- **postrule()**

Shortport execution rules

There are useful aliases in shortport.http:

http = shortport.port_or_service({80, 443, 631, 7080, 8080, 8088, 5800, 3872, 8180, 8000},
{"http", "https", "ipp", "http-alt", "vnc-http", "oem-agent"})

To execute if a HTTP service is found:
portrule = shortport.http

prerule

```
prerule =  function()
  return nmap.is_privileged() and
        (stdnse.get_script_args("targets-sniffer.iface") or
nmap.get_interface())
```

hostrule

This hostrule will make the script execute all the time:

hostrule = function() return true end

action

Contains all the functions to be executed when the script's prerule, portrule, hostrule or postrule triggers. It receives as parameters a host and port table which hold the target information.

Host table

The host table is a regular Lua table with the following fields:

- **host.os**: This is the table containing OS matches (only available with OS detection)
- **host.ip**: This is the IP address of the target
- **host.name**: This is the reverse DNS name of the target (if available)
- **host.targetname**: This is the hostname specified in the command line
- **host.directly_connected**: This is a Boolean that indicates whether the target is on the same network segment
- **host.mac_addr**: This is the Mac address of the target
- **host.mac_addr_next_hop**: This is the Mac address of the first hop to the target

Host table

- **host.mac_addr_src**: This is the Mac address of our client
- **host.interface_mtu**: This is the MTU value of your network interface
- **host.bin_ip**: This is the target IP address as a 4-byte and 16-byte string for IPv4 and Ipv6, respectively
- **host.bin_ip_src**: This is our client's IP address as a 4-byte and 16-byte string for IPv4 and Ipv6, respectively
- **host.times**: This is the timing data of the target
- **host.traceroute**: This is only available with --traceroute

Port table

- **port.number**: This is the number of the target port.
- **port.protocol**: This is the protocol of the target port. It could be tcp or udp.
- **port.service**: This is the service name detected via port matching or with service detection (-sV).
- **port.version**: This is the table containing the version information discovered by the service detection scan. The table contains fields such as name, name_confidence, product, version, extrainfo, hostname, ostype, devicetype, service_tunnel, service_ftp, and cpe code.
- **port.state**: This returns information about the state of the port.

Script output

The return value of the action function may be a:

- **Table of name-value pairs**
- **string**
- **nil**

NSE may use structured or unstructured output depending on the return value.

Script output

By default, Nmap returns the normal output if no output option is declared.

The available output mode options are:

- **Normal output (-oN)**
- **XML output (-oX)**
- **Grepable output (-oG)**
- **Script kiddie (-oS)**

Structured output

NSE scripts can implement structured output by returning one of the following values:

- **A Lua table.**
- **A Lua table and a string.**
- **A Lua table with a __tostring() metamethod.**

Structured output

The easiest way of making your scripts support structured output is by using the NSE function **stdnse.output_table()** to create a Lua table that maintains the order in which the elements are inserted to construct the output table.

```
local output_tab = stdnse.output_table()

...

output_tab.path = “/eaea/”

output_tab.version = extracted_version

return output_path
```

Using NSE data files

If a script requires information stored in a database file, it should be placed on the folder nselib/data.

You may verify a file is readable in Nmap's search paths with the function **nmap.fetchfile(filename)**.

Reading script arguments

To read script arguments it is recommended to use the NSE function stdnse.get_script_args like this:

local path = stdnse.get_script_args(“http-custom.argument”)

Or to make it even more portable:

local path = stdnse.get_script_args(SCRIPT_NAME..”.argument”)

NSE Registry

nmap.registry

Lua table used to store variables shared between all scripts during a scan. We read/write information the same way as we do in regular Lua tables:

table.insert(nmap.registry.credentials.http, { username = username, password = password})

nmap.registry.whois.local_assignments_file_expiry = 1

local allowed_age = nmap.registry.whois.local_assignments_file_expiry

Debugging NSE scripts

To debug NSE scripts we have two options:

- Use --script-trace to print incoming/outgoing communication performed by scripts.
- Use -d[0-9] to increase/decrease the debugging messages.

Printing debugging and verbosity messages

Use stdnse.verbose() and stdnse.debug() to print verbosity and debugging messages according to the specified level.

stdnse.verbose(2, “Prints message when verbosity level is higher than 1”)

stdnse.verbose2(“Also prints message when verbosity level is higher than 1”)

stdnse.debug(2, “Prints debug message when debugging level is higher than 1”)

stdnse.debug(2, “Also prints debug message when debugging level is higher than 1”)

Network communications

For the classic connect-style type (connect, send data and receive data) we have two options:

- Use NSE sockets. (Create, connect, send, receive, close)
- Use the NSE function comm.exchange().

NSE sockets

NSE sockets are defined in the library nmap and they support operations such as connect(), send(), receive() and close().

local s = nmap.new_socket()

local status, err = s:connect(host, port)

local status, err = s:send(data)

local status, err = s:receive()

s:close()

NSE library “comm”

```
local function ping(host, port)
  local status, data = comm.exchange(host, port, "PING")
  if not status then
    stdnse.debug1("Failed to send PING command:%s", data)
    return nil
  end
  if data and data:match("PONG") then
    stdnse.debug1("PONG response received")
    return true
  end
  return nil
end
```

Manipulating binary data

In NSE users may use the library bin to encode/decode binary data. This library only contains the functions **pack()** and **unpack()**.

The pack() method is used to obtain a binary packed string formatted by the character operators and with operator repetitions formatting the given parameters.

local login_pkt = bin.pack("cAcAcAx", 0x6, "/login", 0x0b, "=name=".. username, 0x2c, "=response=00"..chksum)

Extracting information from binary data

Similarly, the **bin.unpack()** method can be used to extract values from binary data strings:

local pos, len = bin.unpack(">S", data)

The **bin.unpack()** method's first return value is the position at which unpacking was stopped to allow subsequent calls to the method.

The unpack() method's arguments are as follows:

- format: Format string
- data: Input binary data string
- init: Starting position within the string

Supported operators

- H hex string
- B bit string
- x null byte
- z zero-terminated string
- p string preceded by a 1-byte integer length
- P string preceded by a 2-byte integer length
- a string preceded by a 4-byte integer length
- A string
- f float
- d double

Supported operators

- n Lua number
- c char (1-byte integer)
- C unsigned char (1-byte unsigned integer)
- s short integer (2-byte integer)
- S unsigned short integer (2-byte unsigned integer)
- i integer (4-byte integer)
- I unsigned integer (4-byte unsigned integer)
- l long integer (8-byte integer)
- L unsigned long integer (8-byte unsigned integer)
- < little endian modifier
- > big endian modifier
- = native endian modifier

NSE parallelism

The stdnse NSE library supports the creation of NSE threads that can run inside your script's Lua thread, and performs network operations in parallel.

The **stdnse.new_thread()** function creates a new NSE thread. This function takes as the first parameter the function to execute in the new thread and, optionally, the arguments needed for the worker thread's main function.

To create an NSE worker, you must load the stdnse library and invoke the **stdnse.new_thread()** function:

stdnse.new_thread(func, arg1, arg2, arg3, …)

Version scripts

- NSE version scripts are used to obtain service information when the operations were too complex for the service detection engine.
- Probes that are simple enough should be written as a standard Nmap service detection signature without using NSE.

Updating port version information

To update port version information we use the NSE function nmap.set_port_version(host, port) like this:

```
local clamchk = check_clam(host, port)
 if clamchk then
   stdnse.debug1("ClamAV daemon found")
   port.version.name = "clam"
   port.version.product = "ClamAV"
   nmap.set_port_version(host, port)
 end
```

Version detection signatures

```
# This is the NULL probe that just compares any banners given to us
##############################NEXT PROBE##############################
Probe TCP NULL q||
# Wait for at least 5 seconds for data.  Otherwise an Nmap default is used.
totalwaitms 5000
# Windows 2003
match ftp m/^220[ -]Microsoft FTP Service\r\n/ p/Microsoft ftpd/
match ftp m/^220 ProFTPD (\d\S+) Server/ p/ProFTPD/ v/$1/
softmatch ftp m/^220 [-.\w ]+ftp.*\r\n$/i
match ident m|^flock\(\) on closed filehandle .*midentd| p/midentd/ i/broken/
match imap m|^\* OK Welcome to Binc IMAP v(\d[-.\w]+)| p/Binc IMAPd/ v$1/
softmatch imap m/^\* OK [-.\w ]+imap[-.\w ]+\r\n$/i
match lucent-fwadm m|^0001;2$| p/Lucent Secure Management Server/
match meetingmaker m/^\xc1,$/ p/Meeting Maker calendaring/
# lopster 1.2.0.1 on Linux 1.1
match napster m|^1$| p/Lopster Napster P2P client/
```

Creating your own version detection signatures

Each line must contain one of the following directives:

- Exclude
- Probe
- match
- softmatch
- ports and sslports
- totalwaitms
- tcpwrappedms
- rarity
- fallback

HTTP scripts

The NSE library http implements the HTTP client protocol. The library offers a generic interface **generic_request** but common HTTP methods like **get**, **post** and **head** are supported.

Other benefits of using this library:

- Transparent HTTP(S) support
- HTTP pipelining
- Customizable options

HTTP Spidering

NSE has a library designed to help users create scripts that require spidering a web server called httpspider. It consists of the following classes:

- Options
- LinkExtractor
- URL
- UrlQueue
- Crawler

httpspider: Example usage

```
crawler.options.withinhost = function(url)
    if crawler:iswithinhost(url)
    and not crawler:isresource(url, "js")
    and not crawler:isresource(url, "css") then
        return true
    end
  end
```

Reporting vulnerabilities

NSE provides the **vulns** library a mechanism for reporting vulnerabilities in a consistent manner. The information about the vulnerabilities is stored in the Nmap registry.

Sample vulnerability output

```
3310/tcp open  clam    ClamAV 0.99.2 (21714)
| clamav-exec:
|   VULNERABLE:
|   ClamAV Remote Command Execution
|     State: VULNERABLE
|       ClamAV 0.99.2, and possibly other previous versions, allow the execution of the
|       clamav commands SCAN and SHUTDOWN without authentication. The command 'SCAN'
|       may be used to enumerate system files and the command 'SHUTDOWN' shut downs the
|       service. This vulnerability was discovered by Alejandro Hernandez (nitr0us).
|
|     Disclosure date: 2016-06-8
|     Extra information:
|       SCAN command is enabled.
|     References:
|       https://bugzilla.clamav.net/show_bug.cgi?id=11585
|_      https://twitter.com/nitr0usmx/status/740673507684679680
```

Vulnerability components

Vulnerabilities are stored in a table with the following fields:

title (required)	dates
state (required)	check_results
IDS	exploit_results
risk_factor	extra_info
scores	references
description	

Sample vulnerability structure

```
local vulns = require "vulns"
local vuln_report = vulns.Report:new(SCRIPT_NAME, host, port)
local vuln = {
  title = 'ZOMG vulnerability',
  state = vulns.STATE.NOT_VULN,
  description = [[ Useful description of vulnerability  ]],
  IDS = {CVE = 'CVE-2016-0000'},
  references = {
    'http://www.openwall.com/lists/oss-security/2014/09/24/10'
  },
  dates = {
    disclosure = {year = '2016', month = '00', day = '01'},
  },
}

vuln.state = vulns.STATE.EXPLOIT
return vuln_report:make_output(vuln)
```

Creating a brute force script

- The NSE library brute provides a framework to write brute force scripts. One of the benefits of using this framework is transparent parallelism.
- A GSoC student is working on an algorithm to automatically adjusts to network conditions.
- The NSE library brute defines the following classes:
  - **Account**
  - **Engine**
  - **Options**
  - **Error**

Brute force script - Engine instance

```
local status, result, engine

engine = brute.Engine:new(Driver, host, port, options)

engine:setMaxThreads(thread_num)

engine.options.script_name = SCRIPT_NAME

status, result = engine:start()
```

Brute force script - Driver class

The brute engine will create instances of the Driver class for each login attempt.

The methods that need to be defined in this class are:

- **Driver:login**
- **Driver:connect**
- **Driver:disconnect**

Brute force script - Driver class

```
Driver = {

    new = function(self, host, port, options) ... end,

    login = function(self) ... end

    connect = function(self) ... end

    disconnect = function(self) ... end

}
```

Account objects

The Account class is used to represent the valid accounts found in the target during execution. Each account stored using this class will have a state.

The available states are:

- **OPEN**
- **DISABLED**
- **LOCKED**

You will find yourself working with this object when implementing the Driver class. An instance of this class must be used as the return value of the Driver:login() function.

Managing user credentials found during scans

- In versions before 6.x, the credentials found by NSE were stored in the Nmap registry.
- The NSE library **creds** was created to provide an interface to easily read and write user credentials stored in this registry.
- Each account is linked to a state, similar to the brute.Account class, so it allows users to filter by account state.

Adding credentials to the registry

```
local creds = require “creds”

local c = creds.Credentials:new(SCRIPT_NAME, self.host, self.port )

c:add(username, password, creds.State.VALID )
```

Creating your own NSE library

To create new NSE library in Lua:

1. Create your Lua library file file.lua inside your nselib directory.
2. In you Lua library declare its name with stdnse.module(“modulename”, stdnse.seeall) and return _ENV.
3. Import your NSE library in your script using the function require “modulename”.

testlib.lua

```
local stdnse = require "stdnse"
_ENV = stdnse.module("testlib", stdnse.seeall)

function hello(world)
  return string.format("Hello '%s'", world)
end

return _ENV;
```

test.nse

```
description = "This is my first NSE script."
author = "me"
license = "Same as Nmap--See http://nmap.org/book/man-legal.html";
categories = { "safe" }

local shortport = require 'shortport'
local testlib = require "testlib"

portrule = function() return true end

action = function( host, port )
  return testlib.hello("foo")
end
```

NSE modules in C/C++

Overall, the steps to get a C library to communicate with NSE are as follows:

1. Place your source and header files for the library inside Nmap's root directory
2. Add entries to the source, header, and object file for the new library in the Makefile.in file
3. Link the new library from the nse_main.cc file

Project contributions

If you want to contribute to the project but don't know where to start/what is needed:

- Check the list of issues at Github

https://github.com/nmap/nmap/issues?q=is%3Aopen+label%3ANSE+is%3Aissue

- Check the Script Idea wiki page:

https://secwiki.org/w/Nmap_Script_Ideas

- Write whatever you think is useful.

Update this presentation

The most up to date version of this presentation, VMs and appendixes will always be available at the following URL:

https://drive.google.com/open?id=0BzPR8exG0kt6ZTBFTkxQOWdwdXM