Network Host Discovery and Service Detection Tools

Marin Maržić

Zagreb, Croatia
July 2013.
CONTENTS

1. Introduction ........................................... 1

2. Host Discovery in Nmap ................................ 2
   2.1. Host Discovery ...................................... 2
   2.2. Introduction to Host Discovery in Nmap ............... 2
   2.3. No Port Scan (\texttt{-sn}) .......................... 3
   2.4. No Ping (\texttt{-Pn}) ............................... 3
   2.5. List Scan (\texttt{-SL}) ............................. 3
      2.5.1. Examples .................................... 4
   2.6. TCP SYN Ping (\texttt{-PS}) .......................... 4
      2.6.1. Examples .................................... 4
   2.7. TCP ACK Ping (\texttt{-PA}) .......................... 6
      2.7.1. Examples .................................... 7
   2.8. UDP Ping (\texttt{-PU}) ............................... 8
      2.8.1. UDP Payloads: The \texttt{nmap-payloads} File .... 8
      2.8.2. Examples .................................... 9
   2.9. SCTP INIT Ping (\texttt{-PY}) .......................... 11
      2.9.1. Examples .................................... 11
   2.10. ICMP Ping Types (\texttt{-PE}, \texttt{-PP}, \texttt{-PM}) ........ 14
      2.10.1. Examples ................................... 14
   2.11. IP Protocol Ping (\texttt{-PO}) ........................ 16
      2.11.1. Examples ................................... 16
   2.12. ARP Ping (\texttt{-PR}) ............................... 18
      2.12.1. Examples ................................... 18

3. Service Detection in Nmap .............................. 19
   3.1. Service Detection ................................... 19
   3.2. Introduction to Service Detection in Nmap ............. 19
   3.3. The \texttt{nmap-service-probes} Approach ............. 20
      3.3.1. The \texttt{nmap-service-probes} File Format ....... 21
      3.3.2. The Process ................................... 25
   3.4. The Nmap Scripting Engine (NSE) Approach ............ 26
      3.4.1. Script Categories .............................. 28
      3.4.2. Script Format ................................ 30
      3.4.3. Information Passed to a Script .................. 33
      3.4.4. The Registry ................................ 36
      3.4.5. Mutexes ..................................... 36
      3.4.6. Exception Handling ............................. 37
4. Developing Extensions to Nmap’s Service Detection

4.1. Introduction ................................................................. 38
4.2. Murmur ................................................................. 39
4.3. Ventrilo ................................................................. 41
4.4. TeamSpeak 2 ............................................................. 47
    4.4.1. TCP: The TCPQuery Interface (Admin Telnetd) Port ......... 47
    4.4.2. TCP: The Web Admin Interface (HTTP) Port .................. 48
    4.4.3. UDP: The Voice and Control Port (The TeamSpeak 2 Protocol) . 48
4.5. TeamSpeak 3 ............................................................. 51
    4.5.1. TCP: The ServerQuery Interface (Admin Telnetd) Port ........ 52
    4.5.2. UDP: The Voice and Control Port (The TeamSpeak 3 Protocol) . 53
4.6. The All-Seeing Eye Protocol ........................................... 54
4.7. Freelancer ............................................................... 57

5. Summary ................................................................. 60

Bibliography ................................................................. 61

A. SCTP INIT Ping (~PY) Patch ........................................... 64
1. Introduction

Network host discovery and service detection are at the core of any network exploration endeavor. Being able to tell the status of machines on a network is an obvious requirement for continuous network maintenance. Assessing which hosts are active and, more importantly, knowing which applications and services they are running is critical to network security. System and network administrators have been known to continue using deprecated software versions, which is in itself a cause for vigil. On top of that, the ordinary user is unpredictable and liable to run almost anything on a corporate computer, potentially exposing the network to unexpected vectors of attack. As a network grows in scale and complexity, the likelihood of such unchecked vulnerabilities existing increases. Therefore, it is wise to regularly scan for out of date and otherwise vulnerable or unknown applications and services, so that potential security holes can quickly be detected and made subject to further inspection.

It is commonplace today that an organization should have servers and various network segments spread out across the globe, but maintained and audited over the Internet from a few fixed locations. To do such crucial operations in a timely manner and offset the high latencies caused by long distance communication, developing efficient host discovery and service detection methods is necessary. Time is rarely an abundant resource, and that in itself provides a constant motivation to further improve on these methods. Furthermore, the never-ending influx of new applications and service versions ensures that service detection will always need patching in order to keep up.

Whether it is a network administrator performing regular inventory, a security expert conducting an audit, or an enthusiast probing to satisfy their curiosity, they must choose appropriate tools for the job at hand. Nmap was chosen for this thesis as the token network exploration and security auditing utility. It is free and open-source, has an active developer and user community, and offers powerful and relatively simple ways of developing extensions to its service detection. It also features different host discovery methods and enables for forging custom probes. Several additions to Nmap’s service detection were developed as a part of this thesis. Nmap Versions 6.25 and 6.26SVN r30898 (fixed SCTP INIT ping patch) were used for most of the development, and while creating examples of usage. A comprehensive description of Nmap’s features can be found in Nmap Network Scanning [1], and some of it has been updated in the online version [2] to reflect recent changes. That work has been used throughout the thesis as the base source when addressing Nmap functionality, and will generally not be explicitly cited further.

Chapter two first introduces some basic terms and issues that arise when dealing with network host discovery, and then continues to discuss it through Nmap’s host discovery techniques and examples depicting their use in some common scenarios. As with host discovery, service detection is first introduced and then addressed in greater detail through Nmap’s implementation in chapter three. Chapter four holds descriptions and usage examples of developed additions to Nmap’s service detection, as well as descriptions of the service protocols involved.

The focus of the thesis will be almost exclusively on IPv4 networks, most often on TCP, UDP, and SCTP, so services will generally be assumed to run behind port numbers. Some familiarity with IPv4, TCP, UDP, SCTP, and general networking concepts is assumed.
2. Host Discovery in Nmap

2.1. Host Discovery

Network host discovery (or simply host discovery) is the process of determining whether an IP address is registered to an active host. A host will be considered active if it implements any network service that is responsive when given the proper input, even if it is only a functional IP, TCP or UDP stack. An inactive host cannot be coerced into responding, making it effectively nonexistent within the context of the network. Host discovery techniques involve sending probes that are known to commonly cause hosts to respond, e.g. an ICMP echo request (as sent by the ubiquitous ping program) or an ARP request on a local area network. If a host responds in any way to any probe it receives, it can be marked as active. However, if such techniques do not procure a reply, it does not necessarily mean the host is inactive. The right probes may just not have been sent.

If there is a service process running and listening on a port number, the port is said to be open. In the case of TCP an open port means that the remote host responds with a SYN/ACK packet upon receiving a SYN packet. For UDP an open port means that the service running behind the port is responsive when sent the proper payload. If there is no service running on a port number, the port is said to be closed. Attempting to communicate with a closed port via TCP will usually cause the remote host to reply with a RST packet. Sending to a closed UDP port will generally cause the remote host to reply with an ICMP port unreachable error packet. If the remote host is simply ignoring (i.e. dropping, blocking, not responding to) packets sent to a certain port, the port is said to be filtered. The port is also said to be filtered, or politely filtered, if the host does not directly respond to the what it was sent, but instead of ignoring received packets it replies with an ICMP port or protocol unreachable error.

Using host discovery to reduce a set of IP ranges into a list of active hosts before conducting extensive port scans and service detection can greatly reduce the total scan duration, because only hosts determined to be active are interrogated further. Such filtering may exclude potentially interesting active hosts that remain undiscovered by the chosen host discovery probes. Yet running a full port and service scan on the entire IP range may take far too long. That is why choosing the appropriate host discovery probes is crucial in achieving an acceptable scan duration, while still discovering the right active hosts.

2.2. Introduction to Host Discovery in Nmap

A more comprehensive look at these and other Nmap features can be found in Nmap Network Scanning [1] and the online Nmap Reference Guide [2]. These are the sources from which most of the information in this chapter was gathered (some taken verbatim or with minor alterations).

By default, Nmap only performs heavy probing such as port scans and version detection against hosts that are found to be active during the host discovery phase. It is possible to perform only host discovery (the \(-sN\) option), or to skip it altogether and run scans against all specified targets (the \(-Pn\) option).
Nmap allows for customizing the set of host discovery probes used when determining if the chosen target hosts are active. If no specific host discovery technique has been specified, a default set of probes is used (an ICMP echo request, TCP SYN to port 443, TCP ACK to port 80, and an ICMP timestamp request). If multiple probes are specified, they will be sent in parallel.

Using most host discovery techniques and port scan types usually requires a privileged (root) user account on Unix systems as Nmap needs to send and receive raw packets. A connect system call is automatically initiated as a workaround for unprivileged users attempting to perform TCP or ICMP based host discovery. In that case only SYN packets are sent to ports 80 and 443 on the target hosts, instead of the previously mentioned default set of probes. If connect returns with a quick success or an ECONNREFUSED failure, the underlying TCP stack must have received a SYN/ACK or RST and the host is marked active. If connect times out, the host is marked as inactive. Using an administrator account on Windows is recommended, although Nmap sometimes works for unprivileged users on that platform when WinPcap\(^1\) has already been loaded into the OS. If a privileged user tries to scan targets on a local area network, ARP requests are used unless the --disable-arp-ping or --send-ip options are specified.

Following are descriptions of host discovery techniques and related options employed by Nmap, as well as examples of some common usage scenarios. Nmap’s output in the examples has been cut for brevity.

2.3. No Port Scan (\(-sn\))

Selecting this option puts Nmap in a pure host discovery mode and is often known as a ping scan or ping sweep, as it only prints out confirmed active hosts without performing further port scanning. It allows for light reconnaissance of a target network without attracting much attention. It is also more reliable than pinging the broadcast address because many hosts do not reply to broadcast queries. System administrators often find this option valuable as it can be used to count available machines on a network or monitor server availability.

In previous versions of Nmap, the option \(-sn\) was known as \(-sP\).

2.4. No Ping (\(-Pn\))

Selecting this option causes Nmap to skip the host discovery stage altogether and to port scan all targets as though they were found to be active. For machines on a local area network, ARP scanning will still be performed (unless --disable-arp-ping is specified) because Nmap needs MAC addresses to further scan target hosts.

In previous versions of Nmap, the option \(-Pn\) was known as \(-P0\) and \(-PN\).

2.5. List Scan (\(-sL\))

The list scan is not an actual host discovery technique in that selecting it simply lists each of the specified target hosts, without sending any packets their way. Options for higher level functionality such as port scanning, OS detection, or ping scanning cannot be combined

---

\(^1\)A tool Nmap relies on for link-layer network access in Windows environments. Allows applications to capture and transmit network packets bypassing the protocol stack.
with this option. By default, reverse-DNS resolution is performed on the hosts, which may provide useful information. The total number of listed IP addresses (target hosts) is reported at the end. The list scan can serve as a good sanity check by helping ensure that the proper targets have been chosen.

### 2.5.1. Examples

The list scan is useful for checking exactly which targets have been selected, when more complicated target hosts and networks are specified.

```bash
```

Starting Nmap 6.25 ( http://nmap.org )
Nmap scan report for www.google.com (173.194.35.48)
Nmap scan report for www.nmap.org (173.255.243.189)
Nmap scan report for 192.168.1.10
Nmap scan report for 192.168.1.20
Nmap scan report for 192.168.2.10
Nmap scan report for 192.168.2.20
Nmap scan report for 192.168.3.10
Nmap scan report for 192.168.3.20
Nmap scan report for 192.168.4.0
Nmap scan report for 192.168.4.1
Nmap scan report for 192.168.4.2
Nmap scan report for 192.168.4.3
Nmap done: 12 IP addresses (0 hosts up) scanned in 0.04 seconds
```

### 2.6. TCP SYN Ping (–PS)

An empty TCP packet with the SYN flag set is sent as a host discovery probe. If specified by an unprivileged user on a Unix system, the `connect` workaround will be used instead. The default destination port is 80. Alternate ports can be specified as a parameter (e.g. `-PS22` or `-PS22-25,80,113,1050,35000`).

The SYN flag suggests to the target host that there is an attempt to establish a connection. If the destination port is closed, a RST packet will be sent back. If the port happens to be open, the target will take the second step of a TCP three-way-handshake by responding with a SYN/ACK TCP packet. The machine running Nmap then breaks the process by responding with a RST, rather than sending an ACK packet which would complete the three-way-handshake and establish a full connection. The RST packet is sent by the kernel of the machine running Nmap in response to the unexpected SYN/ACK, not by Nmap itself. Receiving either the RST or SYN/ACK response would indicate that the host is active.

#### 2.6.1. Examples

**TCP SYN Ping of Open Port 80**

A SYN ping of port 80 on one of Google’s web servers discovers that it is active, as expected.

```bash
# nmap -d --packet-trace -n -sn -PS www.google.com
```

Starting Nmap 6.25 ( http://nmap.org )
Initiating Ping Scan
Scanning www.google.com (173.194.35.52) [1 port]  
SENT (0.0149s) TCP 192.168.1.20:33305 > 173.194.35.52:80 S  
RCVD (0.0607s) TCP 173.194.35.52:80 > 192.168.1.20:33305 SA  
We got a TCP ping packet back from 173.194.35.52 port 80  
Completed Ping Scan, 0.05s elapsed (1 total hosts)

Nmap scan report for www.google.com (173.194.35.52)  
Host is up, received syn-ack (0.046s latency).  
Nmap done: 1 IP address (1 host up) scanned in 0.06 seconds  
Raw packets sent: 1 (44B) | Rcvd: 1 (44B)

TCP SYN Ping of Closed Port 80

A SYN ping of port 80 on an IRC server discovers the host when a RST response is received. This IRC server does not seem to be hosting a web site and was found to be active by a probe sent to the closed port 80.

# nmap -d --packet-trace -n -sn -PS quakenet.underworld.no

Starting Nmap 6.25 ( http://nmap.org )  
Initiating Ping Scan  
Scanning quakenet.underworld.no (158.38.8.251) [1 port]  
SENT (0.0146s) TCP 192.168.1.20:58641 > 158.38.8.251:80 S  
RCVD (0.0880s) TCP 158.38.8.251:80 > 192.168.1.20:58641 RA  
We got a TCP ping packet back from 158.38.8.251 port 80  
Completed Ping Scan, 0.07s elapsed (1 total hosts)

Nmap scan report for quakenet.underworld.no (158.38.8.251)  
Host is up, received reset (0.073s latency).  
Nmap done: 1 IP address (1 host up) scanned in 0.09 seconds  
Raw packets sent: 1 (44B) | Rcvd: 1 (40B)

TCP SYN Ping of Filtered Port 80

A SYN ping of port 80 on another IRC server gets no response. Nmap is not able to determine that the host is active.
# nmap -d --packet-trace -n -sn -PS quakenet.xs4all.nl

Starting Nmap 6.25 ( http://nmap.org )
Initiating Ping Scan
Scanning quakenet.xs4all.nl (194.109.129.222) [1 port]
SENT (0.0718s) TCP 192.168.1.20:40502 > 194.109.129.222:80 S
SENT (1.0728s) TCP 192.168.1.20:40503 > 194.109.129.222:80 S
Completed Ping Scan, 2.00s elapsed (1 total hosts)

Nmap scan report for quakenet.xs4all.nl (194.109.129.222)
[host down, received no-response]
Nmap done: 1 IP address (0 hosts up) scanned in 2.08 seconds
Raw packets sent: 2 (88B) | Rcvd: 0 (0B)

A SYN ping of port 6667 (IRC server port) shows that the host is, in fact, active.

# nmap -d --packet-trace -n -sn -PS6667 quakenet.xs4all.nl

Starting Nmap 6.25 ( http://nmap.org )
Initiating Ping Scan
Scanning quakenet.xs4all.nl (194.109.129.222) [1 port]
SENT (0.0134s) TCP 192.168.1.20:63982 > 194.109.129.222:6667 S
RCVD (0.0476s) TCP 194.109.129.222:6667 > 192.168.1.20:63982 SA
We got a TCP ping packet back from 194.109.129.222 port 6667
Completed Ping Scan, 0.04s elapsed (1 total hosts)

Nmap scan report for quakenet.xs4all.nl (194.109.129.222)
Host is up, received syn-ack (0.034s latency).
Nmap done: 1 IP address (1 host up) scanned in 0.05 seconds
Raw packets sent: 1 (44B) | Rcvd: 1 (44B)

2.7. **TCP ACK Ping (–PA)**

An empty TCP packet with the ACK flag set is sent as a host discovery probe. If specified by an unprivileged user on a Unix system, the connect workaround will be used instead (effectively sending a SYN packet instead of an ACK packet). The default destination port is 80. Alternate ports can be specified as a parameter (e.g. –PA22 or –PA22–25,80,113,1050,35000).

The ACK packet purports to be acknowledging data over an established TCP connection, but no such connection exists. Remote hosts should always respond with a RST packet, disclosing their existence in the process.

Stateless firewalls will often be configured to block incoming TCP connections (SYN packets) except for those destined for public services. When such a firewall is in place, SYN probes will likely be filtered when sent to closed ports, but the ACK probe would go through
those rules. Firewalls with stateful rules can be configured to block unexpected packets, in which case an ACK probe will probably fail, but the SYN probe might go through. Properly configured firewalls that combine these approaches are commonplace now. Regardless, using both SYN and ACK ping probes maximizes the chances of bypassing firewalls and discovering an active host.

2.7.1. Examples

TCP ACK Ping of Open or Closed Port 80

An ACK ping of port 80 on a LAN router makes a discovery. The port is actually open and hosts an HTTP management service for the router, but the results would be the same with a closed port. The host replies with a RST packet and reveals itself.

```
# nmap --disable-arp-ping -d --packet-trace -n -sn -PA 192.168.1.1
```

Starting Nmap 6.25 ( http://nmap.org )
Initiating Ping Scan
Scanning 192.168.1.1 [1 port]
SENT (0.0020s) TCP 192.168.1.20:64116 > 192.168.1.1:80 A
RCVD (0.0046s) TCP 192.168.1.1:80 > 192.168.1.20:64116 R
We got a TCP ping packet back from 192.168.1.1 port 80
Completed Ping Scan, 0.00s elapsed (1 total hosts)

Nmap scan report for 192.168.1.1
Host is up, received reset (0.0027s latency).
MAC Address: 38:C8:5C:00:DE:AD (Cisco Spvtg)
Nmap done: 1 IP address (1 host up) scanned in 0.01 seconds
Raw packets sent: 1 (40B) | Rcvd: 1 (40B)

TCP ACK Ping of Filtered Port 80

An ACK ping of port 80 on one of Google’s web servers produces no results. Google must be filtering invalid TCP traffic on this port.

```
# nmap -d --packet-trace -n -sn -PA www.google.com
```

Starting Nmap 6.25 ( http://nmap.org )
Initiating Ping Scan
Scanning www.google.com (173.194.35.50) [1 port]
SENT (0.0142s) TCP 192.168.1.20:53269 > 173.194.35.50:80 A
SENT (1.0151s) TCP 192.168.1.20:53270 > 173.194.35.50:80 A
Completed Ping Scan, 2.00s elapsed (1 total hosts)

Nmap scan report for www.google.com (173.194.35.50)
2.8. UDP Ping (\texttt{-PU})

A UDP packet is sent as a host discovery probe. It cannot be used by an unprivileged user on a Unix system. The default destination port is 40125. Alternate ports can be specified as a parameter (e.g. \texttt{-PU31337} or \texttt{-PU22-25,80,113,1050,35000}).

For most ports the packet will be empty. A protocol-specific payload that is more likely to get a response will be sent to ports specified in the \texttt{nmap-payloads} file. The \texttt{--data-length} option can be used to send a fixed-length random payload to every port or to disable payloads (\texttt{--data-length 0}).

The UDP probe should elicit an ICMP port unreachable packet when sent to a closed port, which signifies to Nmap that the machine is active. A lack of a response or the arrival of other ICMP errors such as host/network unreachable or TTL exceeded is indicative of a down or unreachable host. If an open port is reached, most services ignore the empty packet, hence the default probe port 40125 which is highly unlikely to be in use. If a service happens to respond to a UDP packet, the host will also be marked as active. The primary advantage of the UDP ping technique is that it bypasses firewalls and filters that only screen TCP.

2.8.1. UDP Payloads: The \texttt{nmap-payloads} File

UDP scanning is difficult because most services do not send a reply to an empty probe, making it impossible to distinguish open and filtered ports. For some ports, Nmap knows a payload that tends to elicit a response and is safe to send instead of an empty probe. Such payloads are stored in the \texttt{nmap-payloads} file.

Each entry in the \texttt{nmap-payloads} file begins with a protocol name (only udp is supported), followed by a comma-separated list of ports. After that comes the payload data as one or more C-style quoted strings that will be concatenated. Comments preceding each entry typically say what the probe means and what kind of response is expected.

Example entry from \texttt{nmap-payloads}:

\begin{verbatim}
# DNSStatusRequest
udp 53 "\x00\x00\x10\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
\end{verbatim}

This example shows that Nmap will send a specific UDP probe payload that is more likely to get a response than an empty probe when probing the UDP port 53.
2.8.2. Examples

**UDP Ping of Open Port 53 (nmap-payloads probe)**

A UDP ping of port 53 finds an active Yahoo DNS server. The sent UDP probe has a specific payload (iplen=40) chosen to elicit responses from DNS servers, as they are often found on port number 53. The payload is read from the nmap-payloads file (described in "2.8.1 UDP Payloads: The nmap-payloads File").

```
# nmap -d --packet-trace -n -sn -PU53 ns1.yahoo.com
```

Starting Nmap 6.25 ( http://nmap.org )
Initiating Ping Scan
Scanning ns1.yahoo.com (68.180.131.16) [1 port]
SENT (0.0235s) UDP 192.168.1.20:48922 > 68.180.131.16:53 iplen=40
RCVD (0.1560s) UDP 68.180.131.16:53 > 192.168.1.20:48922
In response to UDP-ping, we got UDP packet back from 68.180.131.16:53
Completed Ping Scan, 0.13s elapsed (1 total hosts)

Nmap scan report for ns1.yahoo.com (68.180.131.16)
Host is up, received udp-response (0.13s latency).
Nmap done: 1 IP address (1 host up) scanned in 0.16 seconds
Raw packets sent: 1 (40B) | Rcvd: 1 (40B)

**Figure 2.6: UDP ping of open port 53 (nmap-payloads probe)**

**UDP Ping of Open or Filtered Port 53 (ignored payload or dropped packets)**

A UDP ping of port 53 on the same Yahoo DNS server as in the previous example. This time an empty probe is sent instead of the service-specific probe (the relevant line was commented out of the nmap-payloads file). The empty UDP probe (iplen=28) is unsuccessful in getting the host to reply, so Nmap can not conclude that the host is active. The port is actually open as demonstrated by the previous example, but in this case it is indistinguishable from a filtered port. This situation represents the main issue with UDP pings sent to open ports.

```
# nmap -d --packet-trace -n -sn -PU53 ns1.yahoo.com
```

Starting Nmap 6.25 ( http://nmap.org )
Initiating Ping Scan
Scanning ns1.yahoo.com (68.180.131.16) [1 port]
SENT (0.0144s) UDP 192.168.1.20:57132 > 68.180.131.16:53 iplen=28
SENT (1.0156s) UDP 192.168.1.20:57133 > 68.180.131.16:53 iplen=28
Completed Ping Scan, 2.00s elapsed (1 total hosts)

Nmap scan report for ns1.yahoo.com (68.180.131.16)
[host down, received no-response]
Nmap done: 1 IP address (0 hosts up) scanned in 2.02 seconds
Raw packets sent: 2 (56B) | Rcvd: 0 (0B)
UDP Ping of Closed Port 53

A UDP ping of port 53 on an IRC server gets an ICMP port unreachable reply, which indicates that the port is closed and the host active. The IRC server was unlikely to also function as a DNS server. UDP probes work best when sent to ports unlikely to be in use, because they can cause hosts to reply with such ICMP port unreachable errors.

```
# nmap -d --packet-trace -n -sn -PU53 quakenet.underworld.no
```

Starting Nmap 6.25 ( http://nmap.org )
Initiating Ping Scan
Scanning quakenet.underworld.no (158.38.8.251) [1 port]
SENT (0.0947s) UDP 192.168.1.20:40033 > 158.38.8.251:53 iplen=40
RCVD (0.1952s) ICMP 158.38.8.251 > 192.168.1.20
  Port 53 unreachable (type=3/code=3)
Got port unreachable for 158.38.8.251
Completed Ping Scan, 0.10s elapsed (1 total hosts)

Nmap scan report for quakenet.underworld.no (158.38.8.251)
Host is up, received port-unreach (0.10s latency).
Nmap done: 1 IP address (1 host up) scanned in 0.20 seconds
  Raw packets sent: 1 (40B) | Rcvd: 1 (68B)

UDP Ping of Filtered Port 53 (Other ICMP Destination Unreachable Errors)

A UDP ping of port 53 discovers a Deutsche Telekom router. It replied with an ICMP destination unreachable: "Communication administratively prohibited" packet (type 3, code 13). Nmap classifies UDP ports as filtered not only if the host does not reply, but also if it replies with certain ICMP destination unreachable packets (type 3, codes 1, 2, 9, 10, or 13). The latter kind of filtered port reveals the host as active.

```
# nmap -d --packet-trace -n -sn -PU53 pD9E227CA.dip0.t-ipconnect.de
```

Starting Nmap 6.25 ( http://nmap.org )
Initiating Ping Scan
2.9. **SCTP INIT Ping (−PY)**

An SCTP packet containing a minimal INIT chunk is sent as a host discovery probe. It cannot be used by an unprivileged user on a Unix system. The default destination port is 80. Alternate ports can be specified as a parameter (e.g. −PY22 or −PY22,80,179,5060).

The INIT chunk suggests to the remote system that there is an attempt to establish an association. Normally the destination port will be closed, and an ABORT chunk will be sent back. If the port happens to be open, the target will take the second step of an SCTP four-way-handshake by responding with an INIT-ACK chunk. If the machine running Nmap has a functional SCTP stack, then it breaks the association by responding with an ABORT chunk rather than sending a COOKIE-ECHO chunk which would be the next step in the four-way-handshake. The ABORT packet is sent by the kernel of the machine running Nmap in response to the unexpected INIT-ACK, not by Nmap itself. The INIT-ACK and ABORT responses tell Nmap that the host is active. Hosts may more often reply with ICMP errors such as "protocol unreachable" or various "administratively prohibited" messages, indicating that SCTP is not supported or is being politely filtered. These responses also give away the host’s presence.

The following examples were produced with a version of Nmap 6.26 patched to make the -PY host discovery work properly. The patch has been submitted, and will be included in the next Nmap update. More details about it can be found in "A. SCTP INIT Ping (−PY) Patch".

### 2.9.1. **Examples**

**SCTP INIT Ping of Open Port 80**

An SCTP INIT ping discovers an active host when an INIT-ACK packet is received in response to the INIT probe sent to port 80.
# nmap -d --packet-trace -n -sn -PY scanme.roe.ch

Starting Nmap 6.26SVN ( http://nmap.org )
Initiating Ping Scan
Scanning scanme.roe.ch (213.144.141.28) [1 port]
SENT (0.0368s) SCTP 192.168.1.20:35160 > 213.144.141.28:80
RCVD (0.0980s) SCTP 213.144.141.28:80 > 192.168.1.20:35160
Completed Ping Scan, 0.06s elapsed (1 total hosts)

Nmap scan report for scanme.roe.ch (213.144.141.28)
Host is up, received init-ack (0.061s latency).
Nmap done: 1 IP address (1 host up) scanned in 0.10 seconds
Raw packets sent: 1 (52B) | Rcvd: 1 (236B)

Figure 2.10: SCTP INIT ping of open port 80

**SCTP INIT Ping of Closed Port 22**

An SCTP INIT ping probe sent to the closed port 22 also reveals the active host when an ABORT packet is received in response.

# nmap -d --packet-trace -n -sn -PY22 scanme.roe.ch

Starting Nmap 6.26SVN ( http://nmap.org )
Initiating Ping Scan
Scanning scanme.roe.ch (213.144.141.28) [1 port]
SENT (0.1889s) SCTP 192.168.1.20:55071 > 213.144.141.28:22
RCVD (0.2397s) SCTP 213.144.141.28:22 > 192.168.1.20:55071
Completed Ping Scan, 0.05s elapsed (1 total hosts)

Nmap scan report for scanme.roe.ch (213.144.141.28)
Host is up, received abort (0.051s latency).
Nmap done: 1 IP address (1 host up) scanned in 0.24 seconds
Raw packets sent: 1 (52B) | Rcvd: 1 (36B)

Figure 2.11: SCTP INIT ping of closed port 22
SCTP INIT Ping of Filtered Port 80 (Packet Ignored)

An SCTP INIT ping of one of Google’s web servers is ignored and the host is not found to be active.

```
# nmap -d --packet-trace -n -sn -PY www.google.com
```

```
Starting Nmap 6.26SVN ( http://nmap.org )
Initiating Ping Scan
Scanning www.google.com (173.194.35.51) [1 port]
SENT (0.0853s) SCTP 192.168.1.20:63640 > 173.194.35.51:80
SENT (1.0872s) SCTP 192.168.1.20:63641 > 173.194.35.51:80
Completed Ping Scan, 2.01s elapsed (1 total hosts)
```

Nmap scan report for www.google.com (173.194.35.51)
[host down, received no-response]
Nmap done: 1 IP address (0 hosts up) scanned in 2.09 seconds
Raw packets sent: 2 (104B) | Rcvd: 0 (0B)

Figure 2.12: SCTP INIT ping of filtered port 80

---

SCTP INIT Ping of Filtered Port 80 (ICMP Protocol Unreachable)

An SCTP INIT (IP protocol number 132) ping of one of Microsoft Bing web servers. The server shows its activity by replying with an ICMP protocol 132 unreachable (type 3, code 2) error packet.

```
# nmap -d --packet-trace -n -sn -PY www.bing.com
```

```
Starting Nmap 6.26SVN ( http://nmap.org )
Initiating Ping Scan
Scanning www.bing.com (213.191.147.214) [1 port]
SENT (0.0148s) SCTP 192.168.1.20:53817 > 213.191.147.214:80
RCVD (0.0306s) ICMP 213.191.147.214 > 192.168.1.20
Protocol 132 unreachable (type=3/code=2)
Got destination unreachable for 213.191.147.214
Completed Ping Scan, 0.02s elapsed (1 total hosts)
```

Nmap scan report for www.bing.com (213.191.147.214)
Host is up, received proto-unreach (0.016s latency).
Nmap done: 1 IP address (1 host up) scanned in 0.03 seconds
Raw packets sent: 1 (52B) | Rcvd: 1 (80B)
### 2.10. ICMP Ping Types (−PE, −PP, −PM)

An ICMP packet of one of the following types is sent as a host discovery probe:

−PE an ICMP type 8 (echo request) packet is sent, expecting a type 0 (echo reply)

−PP an ICMP type 13 (timestamp request) packet is sent, expecting a type 14 (timestamp reply)

−PM an ICMP type 17 (address mask request) packet is sent, expecting a type 18 (address mask reply)

If specified by an unprivileged user on a Unix system, the `connect` workaround will be used instead (effectively sending a SYN packet instead of an ICMP packet). The echo request is the standard ICMP query as sent by the ping program. The timestamp and address mask requests are also specified as ICMP standards and can be used for host discovery [3][4]. These two queries can be valuable when administrators specifically block echo request packets while forgetting that other ICMP queries can be used for the same purpose. If the target host responds to any sent ICMP packet, it is marked active.

Many hosts and firewalls now block these packets, rather than responding as required by RFC 1122 [5]. For this reason, ICMP-only scans are rarely reliable enough against unknown targets over the Internet. Nonetheless, they can be a practical and efficient approach for system administrators monitoring an internal network.

#### 2.10.1. Examples

**ICMP Ping Type 8 (Echo Request) With a Reply**

Google’s web servers respond to ICMP echo requests.

```
# nmap -d --packet-trace -n -sn -PE www.google.com
```

Starting Nmap 6.25 ( http://nmap.org )
Initiating Ping Scan
Scanning www.google.com (173.194.35.18) [1 port]
SENT (0.0194s) ICMP 192.168.1.20 > 173.194.35.18
  Echo request (type=8/code=0)
RCVD (0.0665s) ICMP 173.194.35.18 > 192.168.1.20
  Echo reply (type=0/code=0)
We got a ping packet back from 173.194.35.18
Completed Ping Scan, 0.05s elapsed (1 total hosts)

Nmap scan report for www.google.com (173.194.35.18)
Host is up, received echo-reply (0.047s latency).
Nmap done: 1 IP address (1 host up) scanned in 0.07 seconds
  Raw packets sent: 1 (28B) | Rcvd: 1 (28B)
ICMP Ping Type 8 (Echo Request) With a Reply

Microsoft Bing web servers respond to ICMP timestamp request pings.

```
# nmap -PE
ICMP echo request (type 8, code 0)
ICMP echo reply (type 0, code 0)
```

**Figure 2.14:** ICMP ping type 8 (echo request) with a reply

---

ICMP Ping Type 13 (Timestamp Request) With a Reply

Microsoft Bing web servers respond to ICMP timestamp request pings.

```
# nmap -d --packet-trace -n -sn -PP www.bing.com
Starting Nmap 6.25 (http://nmap.org)
Initiating Ping Scan
Scanning www.bing.com (213.191.147.216) [1 port]
SENT (0.0145s) ICMP 192.168.1.20 > 213.191.147.216
  Timestamp request (type=13/code=0)
RCVD (0.0252s) ICMP 213.191.147.216 > 192.168.1.20
  Timestamp reply (type=14/code=0)
We got a ping packet back from 213.191.147.216
Completed Ping Scan, 0.01s elapsed (1 total hosts)
```

Nmap scan report for www.bing.com (213.191.147.216)
Host is up, received timestamp-reply (0.011s latency).
Nmap done: 1 IP address (1 host up) scanned in 0.03 seconds
Raw packets sent: 1 (40B) | Rcvd: 1 (40B)

```
# nmap -PP
ICMP timestamp request (type 13, code 0)
ICMP timestamp reply (type 14, code 0)
```

**Figure 2.15:** ICMP ping type 13 (timestamp request) with a reply

---

ICMP Ping Type 17 (Address Mask Request) With No Reply

Google’s web server seems to be ignoring ICMP address mask requests.

```
# nmap -d --packet-trace -n -sn -PP www.google.com
Starting Nmap 6.25 (http://nmap.org)
Initiating Ping Scan
Scanning www.google.com (173.194.35.19) [1 port]
SENT (0.0105s) ICMP 192.168.1.20 > 173.194.35.19
  Address mask request (type=17/code=0)
SENT (1.0116s) ICMP 192.168.1.20 > 173.194.35.19
  Address mask request (type=17/code=0)
Completed Ping Scan, 2.00s elapsed (1 total hosts)
```

Nmap scan report for www.google.com (173.194.35.19)
IP Protocol Ping (–PO)

An IP packet with the specified protocol number set in its IP header is sent as a host discovery probe. It cannot be used by an unprivileged user on a Unix system. If no protocols are specified, the default is to send multiple IP packets for ICMP (protocol 1), IGMP (protocol 2), and IP-in-IP (protocol 4). Alternate protocols can be specified as a parameter (e.g. –PO136 or –PO6,17,132).

For the ICMP, IGMP, TCP (protocol 6), UDP (protocol 17) and SCTP (protocol 132), the packets are sent with the proper protocol headers while other protocols are sent with no additional data beyond the IP header (unless the --data-length option is specified).

If the protocol is supported on the target host, the host should respond using the same protocol as the probe. Otherwise, the host should respond with an ICMP protocol unreachable message, indicating that it does not support the specified protocol. Any response would mark the target host as active.

2.11.1. Examples

IP Protocol Ping of Supported Protocol 1 (ICMP Echo Request)

An IP protocol 1 ping is actually equivalent to an ICMP echo request ping, as illustrated in "2.10.1 ICMP Ping Types (–PE, –PP, –PM): Examples". Like before, the probe elicits an echo reply which reveals an active Google web server.

# nmap -d --packet-trace -n -sn -PO1 www.google.com

Starting Nmap 6.25 ( http://nmap.org )
Initiating Ping Scan
Scanning www.google.com (83.139.106.230) [1 port]
SENT (0.0183s) ICMP 192.168.1.20 > 83.139.106.230
 Echo request (type=8/code=0)
RCVD (0.0348s) ICMP 83.139.106.230 > 192.168.1.20
 Echo reply (type=0/code=0)
Completed Ping Scan, 0.02s elapsed (1 total hosts)

Nmap scan report for www.google.com (83.139.106.230)
Host is up, received proto-response (0.017s latency).
Nmap done: 1 IP address (1 host up) scanned in 0.04 seconds
IP Protocol Ping of Unsupported Protocol 222 (ICMP Protocol Unreachable)

An IP protocol 222 ping probe references an unassigned protocol number, to which a Microsoft Bing web server responds with an ICMP protocol unreachable error.

```
# nmap -d --packet-trace -n -sn -PO222 www.bing.com
```

Starting Nmap 6.25 ( http://nmap.org )
Initiating Ping Scan
Scanning www.bing.com (213.191.147.216) [1 port]
SENT (0.0141s) Unknown protocol (222) 192.168.1.20 > 213.191.147.216
RCVD (0.0217s) ICMP 213.191.147.216 > 192.168.1.20
   Protocol 222 unreachable (type=3/code=2)
Got destination unreachable for 213.191.147.216
Completed Ping Scan at 10:11, 0.01s elapsed (1 total hosts)

Nmap scan report for www.bing.com (213.191.147.216)
Host is up, received proto-unreach (0.0077s latency).
Nmap done: 1 IP address (1 host up) scanned in 0.02 seconds

```
# nmap -PO
```

IP protocol 222 packet
ICMP protocol 222 unreachable (type 3, code 2)

Figure 2.17: IP protocol ping of unsupported protocol 222 (ICMP protocol unreachable)

IP Protocol Ping of Unsupported Protocol 222 (Packet Ignored)

Google’s web servers ignore the IP protocol 222 ping, and are not discovered by the probe.

```
# nmap -d --packet-trace -n -sn -PO222 www.google.com
```

Starting Nmap 6.25 ( http://nmap.org )
Initiating Ping Scan
Scanning www.google.com (83.139.106.212) [1 port]
SENT (0.0194s) Unknown protocol (222) 192.168.1.20 > 83.139.106.212
SENT (1.0205s) Unknown protocol (222) 192.168.1.20 > 83.139.106.212
Completed Ping Scan at 10:18, 2.00s elapsed (1 total hosts)

Nmap scan report for www.google.com (83.139.106.212)
 [host down, received no-response]
Nmap done: 1 IP address (0 hosts up) scanned in 2.02 seconds

```
# nmap -PO
```

IP protocol 222 packet
IP protocol 222 packet (no response; try again)

Figure 2.18: IP protocol ping of unsupported protocol 222 (packet ignored)
2.12. ARP Ping (−PR)

An ARP request is sent as a host discovery probe. It cannot be used by an unprivileged user on a Unix system.

One of the most common Nmap usage scenarios is to scan a local area network. On most LANs, especially those using private address ranges (as specified by RFC 1918 [6]), the vast majority of IP addresses are unused at any given time. When Nmap tries to send a raw IP packet such as an ICMP echo request, the operating system must determine the destination hardware (ARP) address corresponding to the target IP so that it can properly address the Ethernet frame. This is often slow and problematic, since operating systems were not written with the expectation that they would need to do millions of ARP requests against unavailable hosts in a short time period.

ARP scan uses Nmap’s optimized algorithms for ARP requests. If it gets a response back, Nmap knows the host is up and has no further need for IP-based ping packets. This makes the ARP scan much faster and more reliable than IP-based scans.

The ARP scan is done by default when scanning hosts that Nmap detects are on a local area network. Even if different host discovery methods (such as −PE or −PS) are specified, Nmap uses ARP instead for any of the targets which are on the same LAN (unless --disable-arp-ping or --send-ip are specified).

When used with IPv6 (the −6 option), the −PR option uses ICMPv6 Neighbor Discovery (as specified by RFC 4861 [7]) instead of ARP. Neighbor Discovery can be seen as the IPv6 equivalent of ARP.

2.12.1. Examples

ARP Ping

A LAN gateway router responds to an ARP ping.

# nmap −d --packet-trace −n −sn −PR 192.168.1.1

Starting Nmap 6.25 ( http://nmap.org )
Initiating ARP Ping Scan
Scanning 192.168.1.1 [1 port]
SENT (0.0200s) ARP who-has 192.168.1.1 tell 192.168.1.20
RCVD (0.0210s) ARP reply 192.168.1.1 is-at 38:C8:5C:00:DE:AD
Completed ARP Ping Scan, 0.00s elapsed (1 total hosts)

Nmap scan report for 192.168.1.1
Host is up, received arp-response (0.0010s latency).
MAC Address: 38:C8:5C:00:DE:AD (Cisco Spvtg)
Nmap done: 1 IP address (1 host up) scanned in 0.05 seconds
Raw packets sent: 1 (28B) | Rcvd: 1 (28B)
3. Service Detection in Nmap

3.1. Service Detection

Service detection is the process of identifying the service running on a specific port. A simple port scan may reveal an open port at a certain port number. An educated guess based on the port number can then be made as to what service the open port offers. Such guesses are often based on common protocol/port to service name mappings (e.g. the IANA service registry [8]). This method does not require any additional network communication, which makes it useful for quick assessments. However, services can run on arbitrary port numbers, and multiple services can share a common port number. The information gained is minimal even when the port number is correctly resolved to a service name. These drawbacks mean that this method is not reliable enough to be used by itself in any serious network exploration endeavor. This is especially relevant in the case of vulnerability assessments, network inventory, and any other tasks related to maintaining security where knowing the exact version of a service is crucial.

A more reliable and informative method involves gathering and analyzing responses to service-specific probes sent to the target host. This approach can yield invaluable additional details, such as the service version number, host operating system, host device type, and other miscellaneous information. Developing service detection additions for Nmap by using this method will be the main focus of the thesis. The following chapter describes service detection in greater detail through Nmap’s implementation.

3.2. Introduction to Service Detection in Nmap

A more comprehensive look at these and other Nmap features can be found in Nmap Network Scanning [1] and the online Nmap Reference Guide [2]. These are the sources from which most of the information in this chapter was gathered (some taken verbatim or with minor alterations).

Nmap does the basic protocol/port to service name resolution by querying its database (the nmap-services file) of about 2200 well-known services. This approach will report the service most likely running on a scanned port, e.g. port 25/tcp would be identified as a probable SMTP service port. As discussed in the previous chapter, this provides little information and can generally not be relied upon.

Performing service detection means accurately identifying a service, including detecting its version and any other distinguishing details. This is referred to in the Nmap documentation as "Service and Application Version Detection", "Service and Version Detection", or just "Version Detection". In the context of describing Nmap’s features, service detection may further be referred to as in the official Nmap documentation (mostly as version detection).

Nmap’s version detection starts up after a port scan has been completed and open ports have been discovered. Using a database of service-specific probes (the nmap-service-probes file) is the first of two approaches to implementing service detection in Nmap. The second approach involves utilizing the Nmap Scripting Engine (NSE) which runs after the
previous approach has finished. The \texttt{-sv} option enables version detection of both kinds.

### 3.3. The nmap-service-probes Approach

The \texttt{nmap-service-probes} database contains probes for querying various services, and match expressions to recognize and parse responses. Nmap tries to determine the service protocol (e.g. FTP, SSH, Telnet, HTTP), the application name (e.g. ISC BIND, Apache httpd, Solaris telnetd), the version number, hostname, device type (e.g. printer, router), the OS family (e.g. Windows, Linux), miscellaneous details (e.g. whether an X server is open to connections, the SSH protocol version), and the Common Platform Enumeration (CPE) [9] representation of this information. If Nmap was compiled with OpenSSL support, it will connect to SSL servers to deduce the service listening behind that encryption layer. Some ports may be left in the open|filtered state after a port scan if the port scan was unable to determine whether the port is open or filtered. Version detection will try to elicit a response from these ports (just as it does with open ports), and change their state to open if it succeeds.

When RPC services are discovered, the Nmap "RPC grinder" is automatically used to determine the RPC program and version numbers. All the ports detected as RPC are flooded with SunRPC program NULL commands in an attempt to determine whether they are RPC ports, and if so, what their program and version numbers are.

When Nmap receives responses from a service but cannot match them to its database, it prints out a special service fingerprint and a URL. This URL can be used to submit information about the unknown service if the user knows what service version it is.

Short descriptions of options related to version detection using \texttt{nmap-service-probes}:

\begin{description}
\item[\texttt{--allports}] Disables excluding some problematic ports from version detection, e.g. printer ports that print anything sent to them. These ports are specified by the \texttt{Exclude} directive in \texttt{nmap-service-probes}.
\item[\texttt{--version-intensity <intensity>}] Sets the version scan intensity to an integer value between 0 and 9. Nmap’s version scan probes are assigned a rarity value between 1 and 9. Probes with a low rarity value are effective against many common services, while the high rarity probes are not often useful. The default intensity is 7, which means that all probes with a lower or equal rarity value than 7 will be sent. High intensity scans have a higher chance to identify the service, but take longer. A probe registered to the target port in the \texttt{nmap-service-probes} file will be tried regardless of intensity level.
\item[\texttt{--version-light}] A convenience alias for \texttt{--version-intensity 2}. Light version detection. Faster but less likely to identify services.
\item[\texttt{--version-all}] A convenience alias for \texttt{--version-intensity 9}. Full version detection. Slower but maximizes chances of identifying services.
\item[\texttt{--version-trace}] Causes Nmap to print out extensive debugging information about what version scanning is doing.
\end{description}
3.3.1. The nmap-service-probes File Format

Version detection probes and match strings are stored in a flat line-oriented file named nmap-service-probes. Understanding the file format allows users to add their own services to the detection engine. Lines starting with a "#" (comments) and blank lines are ignored by the parser. Other lines must contain one of the directives described below.

Exclude <port specification>
This directive excludes the specified ports from the version scan. It can only be used once and should be near the top of the file, above any Probe directives. The Exclude directive uses the same format as the Nmap -p switch, so ranges and comma separated lists of ports are supported. The only ports excluded in nmap-service-probes by default are TCP ports 9100 through 9107. These are common ports for printers to listen on and they often print any data sent to them, such as Nmap’s service probe payloads, which is often undesirable behavior during a scan. The --allports option overrides the Exclude directive, causing version detection to interrogate all open ports.

Example:
Exclude 53,T:9100,U:30000-40000

Probe <protocol> <probename> <probestring>
This directive tells Nmap what payload string to send to an open port in order to provoke a response which can then be matched to a service. All of the directives discussed later operate on the most recent Probe statement. The arguments to this directive are:

<protocol>
This must be either TCP or UDP. Nmap only uses probes that match the protocol of the service it is trying to scan.

<probename>
This is a plain English name for the probe. It is used in service fingerprints to describe which probes elicited responses.

<probestring>
A formatted payload string to send. It must start with a q, then an arbitrary delimiter character (that is not a part of the string) which begins and ends the string that represents the data that is actually sent. It is formatted similarly to a C or Perl string in that it allows the following standard escape characters: \, \0, \a, \b, \f, \n, \r, \t, \v, and \xHH (where H is any hexadecimal digit). Only the TCP NULL probe has an empty probe string in nmap-service-probes, as it just listens for initial banners that many services send.

Examples:
Probe TCP NULL q||
Probe TCP GetRequest q|GET / HTTP/1.0\r\n\r\n|
Probe UDP DNSStatusRequest q|\0\0\x10\0\0\0\0\0\0\0\0\0\0\0\0\0

match <service> <pattern> [ <versioninfo> ]
This directive tells Nmap how to recognize services based on responses to the string sent by the previous Probe directive. A single Probe line may be followed by numerous match statements. If the given pattern matches, optional specifiers can extract the application name, version number, and additional info for Nmap to report. It is sometimes referred to as a hard
match, as opposed to a soft match (by the softmatch directive described below). The arguments to this directive are:

<service>
This is the service name that the pattern matches (e.g. ssh, smtp, http, snmp). Services tunneled by SSL which can be fully recognized without the overhead of making an SSL connection can have their name prefixed with ssl/ (e.g. ssl/vmware-auth).

<pattern>
This pattern is used to determine if the received data matches what is known about the service’s response. If the received data matches the pattern, the service has been conclusively identified. The pattern format is similar to Perl’s, with the syntax being m/[regex]/[opts]. The "m" tells Nmap that a match string is beginning. The slashes ("/") are delimiters and can be substituted by almost any printable character (that is not in the pattern). The regex is a Perl-style regular expression [10]. The only options currently supported are "i" which makes a match case-insensitive, and "s" which includes newlines in the "." specifier. Subexpressions to be captured, such as version numbers and other information worthy of extraction, are surrounded by parentheses as shown in most of the examples.

<versioninfo>
The <versioninfo> section is comprised of several optional fields:

p/vendorproductname/
Includes the vendor and often service name and is of the form "Sun Solaris rexecd", "ISC BIND named", or "Apache httpd".

v/version/
The application version string, which may include non-numeric characters.

i/info/
Miscellaneous information which was immediately available and might be useful, e.g. whether an X server is open to unauthenticated connections, or the protocol number of SSH servers.

h/hostname/
The hostname (if any) offered up by a service. This is common for protocols such as SMTP and POP3 and is useful because these hostnames may be for internal networks or otherwise differ from the straightforward reverse DNS responses.

o/operatingsystem/
The operating system the service is running on. This may legitimately be different than the OS reported by Nmap IP stack based OS detection. For example, the target IP might be a Linux box which uses NAT to forward requests to an Microsoft IIS server in the DMZ. In this case, stack OS detection should report the OS as Linux, while service detection reports port 80 as being Windows.

d/devicetype/
The type of device the service is running on, a string like "print server" or "webcam". Some services disclose this information, and sometimes it can be inferred (e.g. the "HP-ChaiServer" web server runs only on printers).

cpe:/cpename/[a]
A CPE name [9] for some aspect of the service. This may be used multiple times to identify the service (cpe:/a names), the operating system (cpe:/o names) or hardware platform (cpe:/h names).
The preferred delimiter character is slash (“/”), but almost any printable character not part of the field value string can be arbitrarily chosen. Any, and even all of the fields can be omitted if no such information on the service is available. Any of the version fields can include numbered strings such as $1 or $2, which are replaced with the corresponding parenthesized substring in the <pattern>.

Examples:

```plaintext
match ftp m/^220.*Welcome to.*Pure-?FTPd \(\d\S+\s*\)/ p/Pure-FTPd/ v/$1/
cpe://a:pureftpd:pure-ftpd:$1/
match ssh m/^SSH-\[(\d.)+\]-OpenSSH\[\-\]{{\w.}}/r?\n/i p/OpenSSH/ v/$2/
i/protocol $1/ cpe://a:openssl:openssl:$2/
multimysql m/^\x10\0\0\x01\xff\x13\x04Bad handshake$| p/MySQL/
cpe://a:mysql:mysql/
multichargen m/@ABCDEFGHIJKLMNOPQRSTUVWXYZ/
multichargen m|^login: login: login: $| p/NetBSD uucpd/ o/NetBSD/
cpe://o:netbsd:netbsd/a
match afs m|\([\d\D\]{28}\s*(OpenAFS)([\d.]{3}[\d\D]*)&\)| p/$1/ v/$2/
```

**softmatch <service> <pattern>**

This directive is similar in format to the match directive. The main difference is that scanning continues after a softmatch, but it is limited to probes that are known to match the specified service. This allows for a hard match to be found later, which may provide useful version information. Arguments are the same as with match, except there is no <versioninfo> argument.

Examples:

```plaintext
softmatch ftp m/^220 \[-.\w ]+ftp.*\r
$/i
softmatch smtp m|^220 \[-.\w ]+SMTP.*\r
|
softmatch pop3 m|^\+OK \[-\[\]\(\)\!,/:<>@.\w ]+$|
```

**ports <portlist>**

This directive tells Nmap what ports the services identified by this probe are commonly found on. It should only be used once within each Probe section. The syntax is a simplified version of what is accepted by the Nmap -p option.

Examples:

```plaintext
ports 21,43,110,113,199,505,540,1248,5432,30444
ports 111,4045,32750-32810,38978
```

**sslports <portlist>**

This directive is similar to the ports directive described above, except that these are the ports often used to wrap the service in SSL. For example, the HTTP probe declares sslports 443 and SMTP probes have an sslports 465 line because those are the standard ports for HTTPS and SMTPS respectively.

Example:

```plaintext
sslports 443
```

**totalwaitms <milliseconds>**

This optional directive specifies the amount of time Nmap should wait before giving up on the most recently defined Probe against a particular service. It is rarely necessary as Nmap’s default value is usually suitable.
Example:

```
totalwaitms 5000
```

**rarity <value between 1 and 9>**
This directive roughly corresponds to how infrequently this probe can be expected to return useful results. The higher the number, the rarer the probe is considered and the less likely it is to be tried against a service. By default, probes with a rarity of 7 or less are used during version detection. The `--version-intensity` option (described in "3.3 The nmap-service-probes Approach") can be used to make Nmap consider less rare or even rarer probes.

Example:

```
rarity 9
```

**fallback <comma separated list of probes>**
This optional directive specifies which probes’ match patterns should be used as fallbacks if there are no matches in the current probe section. Nmap first tries match lines from the probe itself, then those from the probes specified in the fallback directive (from left to right), if the directive is present. Finally, if it is a TCP probe, Nmap will always try to fall back to the TCP NULL probe.

Example:

```
fallback GetRequest,GenericLines
```

**An nmap-service-probes Example**
A somewhat representative example of the nmap-service-probes file follows. Many lines have been cut for brevity.

```
# The Exclude directive takes a comma separated list of ports.
# The format is exactly the same as the -p switch.
Exclude T:9100-9107
# This is the NULL probe that just compares any banners given to us
# The match patterns are the same as the -p switch.
# Windows 2003
match ftp m/^220[ -]Microsoft FTP Service\r\n/ p/Microsoft ftpd/
match ftp m/^220 ProFTPD \(\d\d\d\) Server/ p/ProFTPD/ v/$1/
softmatch ftp m/^220 [.-\w ]+ftp.*\r\n$/i
match ident m|^flock\(\) on closed filehandle .*midentd| p/midentd/
match imap m|^\* OK Welcome to Binc IMAP \(\d[-.-\w]+\) 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/
```

```
Probe UDP Help q|help\r\n
rarity 3
ports 7,13,37
match chargen m|@ABCDEFGHIJKLMNOPQRSTUVWXYZ|
match echo m|^help\r

An nmap-service-probes Example
A somewhat representative example of the nmap-service-probes file follows. Many lines have been cut for brevity.

```
# The Exclude directive takes a comma separated list of ports.
# The format is exactly the same as the -p switch.
```

```
3.3.2. The Process

Nmap first does a port scan as instructed, and then passes all the open and open|filtered ports to the service scanning module. Those ports are then interrogated in parallel, although a single port is described here for simplicity. Service probes are defined in the nmap-service-probes file, and many of the terms used in this chapter have been described previously in "3.3.1 The nmap-service-probes File Format".

1. Nmap checks to see if the port is one of the ports to be excluded, as specified by the Exclude directive in nmap-service-probes. If it is, this port will not be scanned.

2. If the port protocol is TCP, Nmap starts by connecting to it (UDP probes continue at step 4). If the connection succeeds and the port had been in the open|filtered state, it is changed to open.

3. Once the TCP connection is made, Nmap listens for roughly six seconds. Many common services, including most FTP, SSH, SMTP, Telnet, POP3, and IMAP servers, identify themselves in an initial welcome banner. Nmap refers to this as the TCP NULL probe because Nmap just listens for responses without sending any probe data. If a response is received and a hard match is made, the service is fully identified and the process ends for this port. If a softmatch is made, Nmap continues but only considers probes that are known to recognize the soft-matched service type.

4. This point is where Nmap starts for UDP probes, and TCP connections continue here if the TCP NULL probe fails (no data received) or soft-matches. Every probe has a list of port numbers it is considered to be most effective against, i.e. most probable to provoke a reply from (the ports directive). Nmap sequentially (in the order they appear in the file) executes the probes that match the port number being scanned. Responses are handled in the same way as for the TCP NULL probe described above. The exact list of regular expressions that Nmap uses to test for a match depends on the probe fallback configuration. If any response during version detection is ever received from a UDP port which was in the open|filtered state, that state is changed to open. This makes version detection an excellent complement to the UDP scan, which is forced to label all scanned UDP ports as open|filtered when some common firewall rules are in effect. While combining UDP scanning with version detection can take many times as long as a plain UDP scan, it is an effective and useful technique.

5. The scan continues here if the TCP NULL probe and probable port probes have failed to produce and hard-match a response. Nmap goes through other existing probes sequentially. In the case of TCP, Nmap must make a new connection for each probe to avoid having previous probes corrupt the results. This worst-case scenario can take time, especially since Nmap must wait about five seconds for the results from each probe because of slow network connections and otherwise slowly responding services.

6. One of the probes tests whether the target port is running SSL. If so (and if OpenSSL is available), Nmap connects back via SSL and restarts the service scan to determine what is listening behind the encryption. This new service scan probing behind the SSL layer uses probable ports from the sslports directive.

7. Another generic probe identifies RPC-based services. When these are found, the Nmap "RPC grinder" is initiated to brute force the RPC program number/name and supported version numbers. Similarly, an SMB post-processor for fingerprinting Windows services is available as part of the Nmap Scripting Engine (NSE).
8. If at least one of the probes elicits some sort of response, yet Nmap is unable to recognize the service, the response content is printed to the user in the form of a fingerprint. If users know what services are actually listening, they are encouraged to submit the fingerprint to Nmap developers for integration into Nmap.

A summary of the algorithm Nmap uses when determining which probes to use follows:

1. For TCP, the TCP NULL probe is always tried first.
2. All probes that have the port being scanned listed as a probable port (the \texttt{ports} directive) are tried in the order they appear in \texttt{nmap-service-probes}.
3. All other probes that have a \texttt{rarity} value less than or equal to the current intensity value of the scan are tried, also in the order they appear in \texttt{nmap-service-probes}.

Nmap uses several automatic techniques to speed up scans:

- Most probes are generic enough to match many services, thus minimizing the amount of connections necessary. For example, the \texttt{GenericLines} probe sends two blank lines (\texttt{
\n}) to the service. This matches daemons of many diverse service types, including FTP, ident, POP3, UUCP, Postgres, and whois.
- If a service matches a \texttt{softmatch} directive, only probes that can potentially match that service are tried.
- The \texttt{rarity} metric is used to avoid trying probes that are extremely unlikely to match.

3.4. The Nmap Scripting Engine (NSE) Approach

The regular expression based approach is powerful, but it cannot recognize everything. Some services cannot be recognized by simply sending a standard probe and matching a pattern to the response. The service protocol might require a multi-step handshaking process, or communication with more than one port, protocol or host. The received response may need more advanced processing, beyond what a regular expression can provide. Responses can have valuable information not encoded in text form, which can be matched by hardcoding it into the pattern (e.g. for a single binary version number), but cannot be captured and presented. A less than elegant solution for that particular case could involve forging a separate pattern for every possible combination of binary field values. Such tasks are not well suited for traditional Nmap version detection, but are easily accomplished with the Nmap Scripting Language (NSE).

The NSE allows users to write scripts to automate a variety of networking tasks. Its possible applications include network discovery, backdoor detection, vulnerability detection and exploitation, and more sophisticated version detection. Script results are integrated into Nmap’s normal and XML output. Scripts are written in the embedded Lua programming language (version 5.2) \cite{lua}, for which there is a freely available reference manual online \cite{luaManual}. Lua was chosen as the base for Nmap scripting because it is a lightweight language, distributed under the liberal MIT open-source license, allows the scripts to be efficiently executed in parallel, was designed with embeddability in mind, has excellent documentation, and is actively developed by a large and committed community. General programming in Lua is outside the scope of this thesis and will be discussed mostly in the context of the NSE.

The core of the NSE is an embeddable Lua interpreter, and the NSE Library is what connects Lua to Nmap. The NSE Library layer handles issues such as initialization of the
Lua interpreter, scheduling of parallel script execution, and script retrieval. It is a key part of the NSE network I/O framework and the exception handling mechanism. It also includes many useful utility libraries.

Scripts are not run in a sandbox and thus could accidentally or maliciously damage the system or invade privacy. Running scripts from untrusted third parties without carefully auditing them is strongly discouraged. The online NSE Documentation Portal [13] documents the scripts included with Nmap.

This chapter is not meant as an exhaustive reference manual on NSE (such sources already exist [1][2][13]). Rather, it is meant as an overview of basic concepts helpful in reading and using NSE scripts, as well as understanding the context and approaches in developing them. A service detection script is very similar to any other script, so this chapter will mostly be introducing NSE scripting in general.

The NSE is controlled by the following options:

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

--script <file>|<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 argument all may be used to specify every script in Nmap’s database. Prefixing script names and expressions with "+" will force them to run even if they normally would not, e.g. when the relevant service was not detected on the target port.

File and directory names may be relative or absolute. Relative paths are looked for in the scripts subdirectory of each of the following places until found:

1. --datadir
2. $NMAPDIR
3. ./nmap (not searched on Windows)
4. <HOME>\AppData\Roaming\nmap (only on Windows)
5. the directory containing the nmap executable
6. the directory containing the nmap executable/../share/nmap
7. NMAPDATADIR
8. the current directory

When a directory name is given, Nmap loads every .nse file in the directory. All other files are ignored and directories are not searched recursively.

Nmap scripts are stored in the scripts subdirectory of the Nmap data directory by default. For efficiency, scripts are indexed in a database stored in scripts/script.db. When referring to scripts from script.db by name, a shell-style "*" wildcard can be used.

An example command loading scripts in the default, safe, and intrusive categories, except for those whose names start with "http-":

nmap --script ".(default or safe or intrusive) and not http-*"
--script-args <n1>=<v1>, <n2>=<v3>, <n4>=<v4>, <v5>

Used for providing arguments to NSE scripts. Arguments are a comma-separated list of name=value pairs. Values may also be tables enclosed in curly brackets ("{}"), just as in Lua. A table may contain simple string values or more name-value pairs, including nested tables. The online NSE Documentation Portal [13] lists the arguments that each script accepts.

--script-args-file <filename>

Used for providing arguments to NSE scripts from a file. Any arguments on the command line supersede ones in the file. Arguments can be comma-separated or newline-separated.

--script-help <file>|<category>|<directory>|<expression>|all[,...]

Shows help about specified scripts. The specifications are the same as those accepted by --script. For each script matching the given specification, Nmap prints the script name, its categories, and its description. This option can be used as a sanity check by previewing which scripts will be run for a specification.

--script-trace

If this option is specified, all communication performed by scripts is printed. The displayed information includes the communication protocol, the source, the target and the transmitted data. If more than 5% of all transmitted data is not printable, then the trace output is in a hex dump format.

--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 after adding or removing NSE scripts from the default scripts directory, and after changing the categories of any script. This option is generally used by itself.

3.4.1. Script Categories

NSE scripts define a list of categories they belong to. The following paragraphs describe each currently defined category.

auth

These scripts deal with authentication credentials on the target system. Scripts which use brute force attacks to determine credentials are placed in the brute category instead.

Examples include x11-access (checks for allowed access to the X server), ftp-anon (tests whether an FTP server allows anonymous access), and oracle-enum-users (attempts to enumerate valid Oracle user names against unpatched Oracle 11g servers).

broadcast

These scripts typically broadcast on the local network in order to discover hosts not listed on the command line. The newtargets script argument allows these scripts to automatically add the hosts they discover to the Nmap scanning queue.

Examples include broadcast-ms-sql-discover (discovers Microsoft SQL servers), broadcast-netbios-master-browser (discovers master browsers and the domains they manage), and broadcast-wake-on-lan (wakes remote systems up from sleep by sending a Wake-On-Lan packet).
brute
These scripts use brute force attacks to guess authentication credentials of a remote server.
Examples include http-brute, oracle-brute, and snmp-brute.

default
These scripts are the default set run when using the -sC or -A options instead of specifying scripts to run with --script. Scripts that run by default must finish quickly, produce valuable and actionable information, have readable and concise output, reliably get results, be unintrusive, and must not divulge much information to third parties. Many of these criteria do not have exact thresholds and are subjective.
Examples include identd-owners (determines the username running remote services by using identd), http-auth (obtains authentication scheme and realm of web sites requiring authentication), and address-info (shows extra information about IPv6 addresses, such as embedded MAC or IPv4 addresses when available).

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).

dos
These scripts may cause denial of service, usually because they crash a service as a side effect of testing it for a vulnerability.
Examples include http-slowloris (tests a web server for vulnerability to the Slowloris DoS attack by launching a Slowloris attack), smb-check-vulns (checks for various Windows vulnerabilities), and smb-flood (exhausts a remote SMB server’s connection limit by by opening as many connections as possible).

exploit
These scripts aim to actively exploit some vulnerability.
Examples include afp-path-vuln (detects the Mac OS X AFP directory traversal vulnerability, CVE-2010-0533 [14]) and http-vuln-cve2012-1823 (Detects PHP-CGI installations that are vulnerable to CVE-2012-1823 [15]).

external
These scripts may send data to a third-party database or other network resource (which may log anything sent to them). Any scripts that do not involve traffic strictly between the scanning machine and the target are placed in this category.
Examples include whois (connects to whois servers to learn about the address of the target), asn-query (maps IP addresses to autonomous system (AS) numbers), and dns-blacklist (checks IP addresses against multiple DNS anti-spam and open proxy blacklists).

fuzzer
These scripts are designed to send server software unexpected or randomized fields in each packet. This technique can be useful for finding undiscovered bugs and vulnerabilities in software, but is a slow and bandwidth intensive process.
Examples include dns-fuzz (launches a DNS fuzzing attack against DNS servers) and http-form-fuzzer (performs a simple form fuzzing against forms found on websites).
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. Unless a script is in the special version category, it should be categorized as either safe or intrusive.

Examples include http-open-proxy (attempts to use the target server as an HTTP proxy), dns-zone-transfer (requests a zone transfer (AXFR) from a DNS server), and mysql-empty-password (checks for MySQL servers with an empty password for root or anonymous).

malware
These scripts test whether the target platform is infected by malware or backdoors.

Examples include irc-unrealircd-backdoor (checks if an IRC server is backdoored), smtp-strangeport (watches for SMTP servers running on unusual port numbers), and auth-spoof (detects identd spoofing daemons which provide a fake answer before even receiving a query).

safe
Scripts which were not designed to crash services, use large amounts of network bandwidth or other resources, or exploit security holes are categorized as safe. Unless a script is in the special version category, it should be categorized as either safe or intrusive.

Examples include ssh-hostkey (retrieves an SSH host key), html-title (retrieves the title from a web page), and bitcoin-info (extracts version and node information from a Bitcoin server).

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 include 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 http-slowloris-check (tests a web server for vulnerability to the Slowloris DoS attack without actually launching a DoS attack), netbus-auth-bypass (checks a NetBus server for an authentication bypass vulnerability which allows full access without knowing the password), and irc-botnet-channels (checks an IRC server for channels that are commonly used by malicious botnets).

3.4.2. Script Format
NSE scripts consist of descriptive fields, rules defining when the script should be executed, and an action function containing the actual script instructions. Values can be assigned to the descriptive fields just as they would be to any other Lua variable.
**Descriptive Fields**

The examples shown in the following paragraphs are snippets taken from the `smb-server-stats.nse` script [16] distributed with Nmap.

**description**

This field describes what a script is testing for and any important notes the user should be aware of. 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.

Example:

```
description = [[ Attempts to grab the server’s statistics over SMB and MSRPC, which uses TCP ports 445 or 139.

An administrator account is required to pull these statistics on most versions of Windows, and Vista and above require UAC to be turned down.

Some of the numbers returned here don’t feel right to me, but they’re definitely the numbers that Windows returns. Take the values here with a grain of salt.

These statistics are found using a single call to a SRVSVC function, `<code>NetServerGetStatistics</code>`. This packet is parsed incorrectly by Wireshark, up to version 1.0.3 (and possibly higher).
]]
```

**categories**

This field defines the categories to which a script belongs (as described in "3.4.1 Script Categories"). The categories are case-insensitive and may be specified in any order. They are listed in an array-style Lua table.

Example:

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

**author**

This field contains the script authors’ names and can also contain contact information.

Example:

```
author = "Ron Bowes"
```

**license**

All of the scripts which come with Nmap currently use the standard Nmap license. This optional field helps ensure legal permission to distribute all those scripts.

Example:

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

**dependencies**

This field is an array containing the names of scripts that should run before this script, if they are also selected. Listing a script in `dependencies` does not by itself cause that script to be run, but merely forces an order of execution among the scripts that have been selected.
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.

Example:

```lua
dependencies = {"smb-brute")
```

**Rules**

Script rules are used to determine whether a script should be run against a target. A rule is a Lua function that returns either `true` or `false`. The script’s action function is only performed if the rule evaluates to `true`. A script may run in more than one phase if it has several rules.

A script must contain one or more of the following functions, i.e. rules, that determine when the script will be run:

- **prerule()** Receives no arguments. Runs only once, before any hosts are scanned. Used when doing host discovery or any other network operation.

- **hostrule(host)** Accepts a host table argument containing information such as the IP address and hostname. Runs once for each host target scanned.

- **portrule(host, port)** Accepts a host table and a port table argument containing information such as the port number, port state, and listening service name. Runs once for each port in the open, open|filtered, or unfiltered port state. Service detection scripts use a portrule.

- **postrule()** Receives no arguments. Runs only once, after all hosts have been scanned. Used when reporting data and statistics gathered during the scan.

Example:

```lua
-- Check whether or not this script should be run.
hostrule = function(host)
  -- Determines whether or not SMB checks are possible on this host.
  return smb.get_port(host) ~= nil
end
```

**Action**

The action is a Lua function executed when the script’s prerule, portrule, hostrule, or postrule trigger. It accepts the same arguments as the rule that triggered it. The script’s action function return values are interpreted as follows:

- If the return value is a table, it is automatically formatted in a structured fashion for inclusion in the normal (`-oN`) and XML (`-oX`) output formats.

- If the return value is a string, the text is displayed directly in normal output and a blob of text is written as an XML attribute in XML output. This is a deprecated feature.

- If the first return value is a table and the second a string, the table is used in XML and the string in normal output.

- If the return value is `nil`, no output is produced.

Example:
action = function(host)
    local result, stats
    local response = {}
    local subresponse = {}

    result, stats = msrpc.get_server_stats(host)

    if(result == false) then
        return stdnse.format_output(false, response)
    end

    table.insert(response, string.format("Server statistics collected since %s (%s):", stats['start_str'], stats['period_str']))
    table.insert(subresponse, string.format("%d bytes (%.2f b/s) sent, %d bytes (%.2f b/s) received", stats['bytessent'], stats['bytesrcvd'], stats['bytessentpersecond'], stats['bytesrcvdpersecond']))
    table.insert(subresponse, string.format("%d failed logins, %d permission errors, %d system errors, %d print jobs, %d files opened", stats['pwerrors'], stats['permerrors'], stats['syserrors'], stats['jobsqueued'], stats['fopens']))
    table.insert(response, subresponse)

    return response
end

Environment Variables

Each script has its own set of environment variables:

**SCRIPT_PATH** The script path.

**SCRIPT_NAME** The script name. This variable can be used in debug output.

**SCRIPT_TYPE** This environment variable will show which rule has activated the script, since a script can have multiple rule functions. This can be useful if the script wants to share code between different script scan phases. The variable value will be one of four strings: "prerule", "hostrule", "portrule", or "postrule". This variable is only available during and after the evaluation of the rule functions.

Example (from the dns-zone-transfer.nse script [17] distributed with Nmap):

```lua
stdnse.print_debug(3, "Skipping '%s' %s, 'dnszonetransfer.server' argument is missing.", SCRIPT_NAME, SCRIPT_TYPE)
```

**OUTPUT:**

NSE: Skipping 'dns-zone-transfer' prerule, 'dnszonetransfer.server' argument is missing.

### 3.4.3. Information Passed to a Script

The information Nmap has learned about the host targets is passed to the NSE script’s action method via its arguments. If a script matched a hostrule it gets only the host table, and if it matched a portrule it gets both the host and the port table. Service detection scripts can then update the port table and its fields with newfound information. 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 IP address, host name, and operating system of the target host (if OS detection was specified), as well as other potentially useful information also described below.

**host.os**
An array of OS match tables. An OS match consists of a human-readable name and an array of OS classes. Each OS class consists of a vendor, OS family, OS generation, device type, and an array of CPE entries for the class. Only entries corresponding to perfect OS matches are put in the `host.os` table. If Nmap was run without OS detection (the `-O` option), then `host.os` is **nil**.

**host.ip**
Contains a string representation of the IP address used for the scan of the target host.

**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 the host running Nmap, i.e. if they are on the same network segment.

**host.mac_addr**
MAC address of the destination host (six-byte-long binary string) if available, otherwise **nil**. The MAC address is generally only available for hosts directly connected on a LAN and only if Nmap is doing a raw packet scan such as SYN scan.

**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**
MAC address of the host running Nmap, which was used to connect to the host. It is either the network card’s MAC address, or the spoofed address (with `--spoof-mac`).

**host.interface**
A string containing the interface name through which packets to the host are sent.

**host.interface_mtu**
The MTU (maximum transmission unit) for `host.interface`, or 0 if unknown.

**host.bin_ip**
The target host’s IP address as a 4-byte (IPv4) or 16-byte (IPv6) string.

**host.bin_ip_src**
The Nmap running host’s source IP address as a 4-byte (IPv4) or 16-byte (IPv6) string.

**host.times**
This table contains Nmap’s timing data for the host. Its keys are `srtt` (smoothed round trip time), `rttvar` (round trip time variance), and `timeout` (the probe timeout), all given in floating-point seconds.
host.traceroute
This is an array of traceroute hops, present when the --traceroute option was used. Each entry is a host table with fields name, ip and times.srtt (round trip time). The TTL for an entry is implicit given its position in the table. An empty table represents a timed-out hop.

port
The port table is passed to NSE scripts with a portrule as a parameter to the rule and action functions in the same way the host table is. 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 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 guessed based on the port number or as detected by version detection. Its value is equal to the value of port.version.name.

port.version
This entry is a table which contains information retrieved by the version scanning engine. Values which were not determined default to nil. Once modified, port.version can be updated in Nmap by using the nmap.set_port_version() call. This is normally done when a service detection script acquires new information about the service running on the port. The meaning of each key of the port.version table follows:

name
Contains the service name for the port.

name_confidence
Evaluates how confident Nmap is about the accuracy of name, from 1 (least confident) to 10. Services determined by guessing the port have it set to 3, whereas those detected by version detection have it set to 10.

product, version, extrainfo, hostname, ostype, devicetype
These five variables are the same as those described under <versioninfo> in "3.3.1 The nmap-service-probes File Format".

service_tunnel
Contains the string "none" or "ssl" based on whether or not SSL tunneling was used to detect the service.

service_fp
Contains the service fingerprint used for reporting information about encountered unrecognized services.

cpe
List of CPE codes for the detected service. As described in the official CPE specification [9] these strings all start with the cpe:/ prefix.
port.state
Contains information on the state of the port. Service scripts are only run against ports in the open or open|filtered states. Other values might appear if the port table is a result of the get_port_state or get_ports functions. Once modified, port.state can be updated in Nmap by using the nmap.set_port_state() call. This is normally done when an open|filtered port is determined to be open.

3.4.4. The Registry
Scripts can share information by storing values in a registry, which is a special table that can be accessed by all scripts. There is a global registry with the name nmap.registry, shared by all scripts. The global registry persists throughout an entire scan session. Scripts can use it, for example, to store values that will later be displayed by a postrule script. Each host additionally has its own registry called host.registry, where host is the host table passed to a script (discussed in "3.4.3 Information Passed to a Script"). The per-host registries only exist while a host is being scanned. They can be used to send information from one script to another running against the same host. Information in the registries is not stored between Nmap executions.

Because every script can write to the global registry table, it is important to make the keys used unique, to avoid overwriting the keys of other scripts (or the same script running in parallel). It is recommended to use the per-host registry when possible because it allows the memory used by the registry to be reclaimed when it is no longer needed.

For example, the ssh-hostkey script collects SSH key fingerprints and stores them in the global nmap.registry so they can be printed later by the postrule. The ssl-cert script collects SSL certificates and stores them in the per-host registry so that the ssl-google-cert-catalog script can use them without having to make another connection to the server.

Scripts that use the results of another script must declare it using the dependencies variable (described in "3.4.2 Descriptive Fields") to make sure that the earlier script runs first.

3.4.5. Mutexes
Each script execution thread yields to other script threads whenever it makes a call on network objects (sending or receiving data), but some scripts require more concurrency control over thread execution. Semantics of the mutex are as expected, it allows for only one thread to be working on an object at a time. Competing threads waiting to work on this object are put in the waiting queue until they can get a "lock" on the mutex.

For example, this can be useful when a script will run for multiple ports or hosts but query the same resource and get the same response, thus duplicating the work. Each thread could block on the mutex prior to querying the remote resource. When the first thread that got the lock on the mutex has finished the query, it can store results in the NSE registry and unlock the mutex. Other scripts waiting to query the remote server can then (one by one) obtain a lock, check the NSE registry for a usable result from previous queries, make their own queries if they have to, store their results in the NSE registry, and unlock the mutex.

A mutex can be created by calling nmap.mutex(object). It returns a function which works as a mutex for the object passed in. This object can be any Lua data type except nil, Boolean, and number. The returned function takes acts based on a sole argument:
"lock" Makes 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, it immediately returns false. Otherwise, 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. 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 mutex function. However, if the reference to the mutex is discarded then it may be collected and subsequent calls to nmap.mutex with the object will return a different function. Therefore a mutex should be saved to a (local) variable that persists as long as it is needed.

3.4.6. Exception Handling

NSE provides an exception handling mechanism not present in the base Lua language. The nmap.new_try API method returns a function (the exception handler) which takes a variable number of arguments that are assumed to be the return values of a function compatible with this exception handling mechanism, such as the NSE network I/O functions. 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, a function can be passed to nmap.new_try which will be called if an exception is caught. That catch function generally performs any required cleanup operations, such as closing newly created sockets. If no such catch function is specified, execution of the script aborts while open sockets 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 (the -d option), a description of the uncaught error condition is printed on standard output.

Example:

```lua
local result, socket, try, catch

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

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

For a function to be treated properly by the try/catch exception handling mechanism, its first return value should be a Boolean which is true upon successful completion of the function and false (or nil) otherwise. If the function completed successfully, the try construct consumes this Boolean value and returns the remaining values. If the function failed, the second returned value must be a string describing the error condition.
4. Developing Extensions to Nmap’s Service Detection

4.1. Introduction

Nmap’s service detection relies heavily on community contributions to its database of known useful service probes and rules. This thesis entails the development of such additions, and they are described in the following chapters along with the service protocols involved. The relevant service protocols are generally described only in as much detail as is necessary to understand the service detection solution.

The choice of services to develop version detection for was arbitrary. The main deciding factor was whether or not Nmap already identified the service accurately. This can be determined by searching the \texttt{nmap-service-probes} file for lines containing the service name, and by looking through Nmap’s online NSEDoc Reference Portal [13] for version detection scripts that might already detect this service. Looking through locally installed scripts (\texttt{.nse} files) can also be helpful. They are usually located in \texttt{/usr/local/share/nmap/scripts/} or \texttt{/usr/share/nmap/scripts/} on UNIX systems, and in \texttt{C:\Program Files\Nmap\Scripts} on Windows systems. Launching Nmap’s service detection against a target known to be running the service in question is ultimately necessary to confirm that the service is, or isn’t, being properly identified. This should always be done while using the latest available Nmap version. The testing can be performed with a locally set up service host instance, e.g. on a virtual machine (VM), or with a public Internet host. Using a personal testing environment is preferable, especially during development, because public servers and their administrators might not respond well to the aggressive probing.

If Nmap does not identify a service accurately, but one of the existing probes got a response, a service fingerprint is printed out. Developers can use this fingerprint to help them in creating a regular expression match rule that extracts relevant version information from the response, i.e. to implement service detection for the service. If no existing probe elicited a response (there is no fingerprint), a custom probe payload and match rule must be made in order to detect the service. In the case of a more complex service that cannot be identified with a simple static payload and regular expression match rule, the Nmap Scripting Engine (NSE) can be utilized instead. Regardless of the chosen approach, the service protocol must be known in sufficient detail before crafting a detection solution.

Monitoring and analyzing service network traffic is done during all stages of service detection development in order to see how a service reacts to given inputs. It may be helpful, and sometimes even necessary, to look for clues in the service client and server executable assembly code (e.g. if the traffic is encrypted). Learning about the service protocol also includes looking into any relevant available documentation, version and development history, source code, as well as any other existing work and research done on the subject. Such analyses give insight into the inner workings of the service protocol, which makes possible devising a service detection mechanism. The mechanism should be well thought-out in terms of memory and time complexity, and the amount of network communication necessary. It should cover as broad a range of service versions and operating systems as possible, while extracting as much useful information as possible. The collected knowledge about the service is also useful when setting up the necessary development and testing environments.
to run as many supported platforms and service versions as the planned implementation can feasibly detect. The testing environments should include installations of legitimate client programs whose communication with the service can be analyzed, as parts of the traffic could be replicated to perform service detection. A perfect service detection solution would be able to consistently send one small packet and derive detailed and exact version information from a single response packet, while using little memory and CPU, and still being effective against all known and future service versions in all known and future environments. Such solutions do not exist in practice and compromises between priorities must be made. For example, a tradeoff can be made in favor of extracting more detailed version information for newer versions of a service, over fully supporting some of the older versions.

Finally, the resulting code must be well tested and documented in accordance with Nmap guidelines. Detailed guidelines for documenting NSE scripts can be found in the chapter “Writing Script Documentation (NSEDoc)” of the online Nmap book [2]. Examples of probe payloads and their regular expression match rules can be found in the nmap-service-probes file, although they usually require little or no documentation.

4.2. Murmur

Murmur is a server service for the Mumble voice communication client. Mumble is a voice chat application for groups. While it can be used for any kind of activity, it is primarily intended for gaming. It can be compared to programs like Ventrilo or TeamSpeak. It is free and open-source software licensed under the BSD license [18][19].

**Supported platforms:** Windows, Linux, OS X [18]

**Release versions:**

- 0.1 (2005-09-02) - 1.1.8 (2009-03-22): not detected
- 1.2.0 (2009-12-10) - 1.2.4 RC1 (2013-01-15; latest release): detected

**Ports:**

- TCP - the control port; port number 64738 by default
- UDP - the voice port; port number 64738 by default

**Protocol description:**

The Mumble/Murmur protocol is documented at the Mumble website protocol page [20], in the Mumble protocol 1.2.X reference [21], and in the source code [19].

Mumble is based on a standard server-client communication model. It utilizes two channels of communication. The first one is a TCP connection which is used to reliably transfer control data between the client and the server. The second one is a UDP connection which is used for unreliable, low latency transfer of voice data. The TCP and UDP ports share the same port number. Both are protected by strong cryptography. This encryption is mandatory and cannot be disabled [21]. Querying version data through unencrypted communication on the UDP port by sending a ping packet to the target server is also supported. This ping packet is used by the Mumble client to measure latency and gather other basic information for display on the user's public or favorite server lists. It is used for version detection of the Murmur server service. Both the request and the response packets are formatted in big endian. The request UDP payload has a fixed size of 12 bytes, and the response UDP payload has a fixed size of 24 bytes.
<table>
<thead>
<tr>
<th>Offset</th>
<th>Type</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>int32</td>
<td>0</td>
<td>Denotes the request type.</td>
</tr>
<tr>
<td>4-11</td>
<td>int64</td>
<td>ident</td>
<td>Used to identify the response.</td>
</tr>
</tbody>
</table>

**Table 4.3: The Mumble UDP ping response packet**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Type</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>int8</td>
<td>0</td>
<td>Always zero.</td>
</tr>
<tr>
<td>1</td>
<td>int8</td>
<td>major version number</td>
<td>E.g. the '1' in 1.2.4</td>
</tr>
<tr>
<td>2</td>
<td>int8</td>
<td>minor version number</td>
<td>E.g. the '2' in 1.2.4</td>
</tr>
<tr>
<td>3</td>
<td>int8</td>
<td>patch number</td>
<td>E.g. the '4' in 1.2.4</td>
</tr>
<tr>
<td>4-11</td>
<td>int64</td>
<td>ident</td>
<td>The ident value sent with the request.</td>
</tr>
<tr>
<td>12-15</td>
<td>int32</td>
<td>number of connected users</td>
<td></td>
</tr>
<tr>
<td>16-19</td>
<td>int32</td>
<td>maximum number of users</td>
<td></td>
</tr>
<tr>
<td>20-23</td>
<td>int32</td>
<td>allowed bandwidth per user</td>
<td>In bits per second.</td>
</tr>
</tbody>
</table>

**Approach:** Nmap Scripting Engine (NSE)

**Script file:** murmur-version.nse

**Script description:**
Detects the Murmur service (server for the Mumble voice communication client) versions 1.2.X on all supported platforms.

The Murmur server listens on a TCP (control) and a UDP (voice) port with the same port number. This script activates on both a TCP and UDP port version scan. In both cases probe data is sent only to the UDP port because it allows for a simple and informative ping command.

The single probe will report on the server version, current user count, maximum users allowed on the server, and bandwidth used for voice communication. It is used by the Mumble client to ping known Murmur servers.

The IP address from which service detection is being run will most likely be temporarily banned by the target Murmur server due to multiple incorrect handshakes from other preceding Nmap service probes. This ban makes identifying the service via TCP impossible in practice, but does not affect the UDP probe used by this script.

It is possible to get a corrupt user count (usually +1) when doing a TCP service scan due to previous service probe connections affecting the server.

See http://mumble.sourceforge.net/Protocol [20].

**Example output:**

```
PORT     STATE SERVICE VERSION
64740/tcp open  murmur Murmur 1.2.4  
   (control port; users: 35; max. users: 100; bandwidth: 72000 b/s)
```
64740/udp open murmur Murmur 1.2.4
  (voice port; users: 35; max. users: 100; bandwidth: 72000 b/s)

**Approach:** nmap-service-probes entry

**nmap-service-probes file modifications:**

Add a new probe entry:

```plaintext
Probe UDP Murmur q|\0\0\0\0abcdefgh|
rarity 9
ports 64738
match murmur m|^\0.{3}abcdefgh.{12}$|s p/Murmur/ v/1.2.X/
```

**Solution description:**

This edition of the probe detects no extra version information, but still hardmatches a Murmur 1.2.X service. It can be useful when not using NSE or when using a stricter portrule in the NSE script that doesn’t necessarily match on all open ports, but only ones that are probably running the service. In the latter case this solution can act as "softmatch" that enables the NSE solution and is controllable by the rarity metric.

This probe only detects the UDP port. The same probe payload is sent as in the NSE solution, but the received information is not gathered.

**Example output:**

```plaintext
PORT     STATE SERVICE VERSION
64740/tcp open murmur Murmur 1.2.X
```

**UDP payload (the nmap-payloads file entry):**

```plaintext
# Murmur 1.2.X (Mumble server)
# UDP ping. "abcdefgh" is an identifier. See
# http://mumble.sourceforge.net/Protocol.
udp 64738 "\x00\x00\x00\x00abcdefgh"
```

### 4.3. Ventrilo

Ventrilo is a voice chat application for groups. While it can be used for any kind of activity, it is primarily intended for gaming. It can be compared to programs like Mumble or TeamSpeak. It is proprietary software, but the Ventrilo client and server are both available as freeware for use with some limitations. This chapter addresses the Ventrilo server when not explicitly stated otherwise.

**Supported platforms:** Windows, Linux i386, FreeBSD i386, NetBSD i386, Solaris Sparc 64bit, Solaris i386, Mac OSX PowerPC 32bit [22]

**Release versions:**

1.0 (2002-08-03) - 2.1.1 (2003-07-17): not detected

2.1.2 (2003-09-23) - 3.0.3 (2008-08-24; latest release): detected

---

1Some older versions (pre 3.0.0) may not have the UDP service this probe relies on enabled by default.
Ports:

TCP - the voice and control port; port number 3784 by default

UDP - the ping and status port; port number 3784 by default

Protocol description:

Ventrilo is proprietary software and there is no source code available. However, Mangler [23] is a free and open-source client for Linux capable of connecting to Ventrilo 3.X servers, and its source code and documentation may be of use when learning about the Ventrilo protocol. The main and only other significant resource on the Ventrilo protocol seems to be Luigi Auriemma’s website [24] which hosts his implementations of various aspects of the protocol. It includes C source code for handling UDP status queries which can be used to detect the service. This code emulates the "ventrilo_status.exe" executable (included with Ventrilo versions 2.1.2 and above) in creating, encrypting, decrypting, sending and receiving of UDP status queries.

Ventrilo uses two channels of communication, a TCP connection which is used to transfer control and voice data, and a UDP connection which is used for pings and status queries. TCP and UDP ports share the same port number. Freeware Ventrilo servers are restricted to using port number 3784. The UDP status query is used for service detection in this script.

Both the status request and response packets have the same header field structure, followed by text data formatted in ASCII. The text data following a status query request contains the first 16 bytes of the Ventrilo server status password. This is not the server password clients use when connecting, and is most often empty as it is set to empty by default. The rest of the bytes, or all of them if no password is given, are zeroes. The text data following a detailed status query response contains a description of various server properties and its current state, as described in the script description and shown in the output example. The header has a fixed size of 20 bytes, and the data that follows it has a maximum size of 492 bytes. Thus the total UDP payload size of any sent or received packet cannot exceed 512 bytes. As the text data of a request packet has a fixed size of 16 bytes, the UDP payload size of a request always totals 36 bytes. If the text data of a response packet exceeds 492 bytes, it can be split and sent in multiple packets (all following the same header and data section structure). The text is then simply concatenated when all packets are received.

Table 4.5: The Ventrilo UDP status packet header

<table>
<thead>
<tr>
<th>Offset</th>
<th>Type</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–1</td>
<td>uint16</td>
<td>key for decrypting this header</td>
<td></td>
</tr>
<tr>
<td>2–3</td>
<td>uint16</td>
<td>0</td>
<td>Always zero.</td>
</tr>
<tr>
<td>4–5</td>
<td>uint16</td>
<td>command number</td>
<td>1 (general) or 2 (detailed status).</td>
</tr>
<tr>
<td>6–7</td>
<td>uint16</td>
<td>query ID</td>
<td>Used to identify the response.</td>
</tr>
<tr>
<td>8–9</td>
<td>uint16</td>
<td>total data size of the query</td>
<td>In bytes.</td>
</tr>
<tr>
<td>10–11</td>
<td>uint16</td>
<td>data size of this packet</td>
<td>In bytes. Maximum of 492.</td>
</tr>
<tr>
<td>12–13</td>
<td>uint16</td>
<td>total number of packets in the query</td>
<td>Maximum of 32.</td>
</tr>
<tr>
<td>14–15</td>
<td>uint16</td>
<td>the packet’s ordinal in the query</td>
<td>Starts from zero.</td>
</tr>
<tr>
<td>16–17</td>
<td>uint16</td>
<td>key for decrypting the data</td>
<td></td>
</tr>
<tr>
<td>18–19</td>
<td>uint16</td>
<td>checksum of the total plain text data</td>
<td></td>
</tr>
</tbody>
</table>
The UDP payload is encrypted in both query and response packets. The header and data are encrypted separately with different algorithms and keys. The cryptography algorithms used are both symmetric-key algorithms.

An example of text data received after sending a detailed status query request to an arbitrarily chosen public server follows.

NAME: Sigma Squawk Service
PHONETIC: Sigma Squawk Service
COMMENT:
AUTH: 0
MAXCLIENTS: 200
VOICECODEC: 3, Speex
VOICEFORMAT: 16,16 KHz%2C 16 bit%2C 5 Qty
UPTIME: 4112812
PLATFORM: FreeBSD-1386
VERSION: 3.0.6
CHANNELCOUNT: 1
CHANNELFIELDS: CID, PID, PROT, NAME, COMM
CHANNEL: CID=904, PID=0, PROT=0, NAME=Sigma Squawk, COMM=
CLIENTCOUNT: 53
CLIENTFIELDS: ADMIN, CID, PHAN, PING, SEC, NAME, COMM
CLIENT: ADMIN=0, CID=904, PHAN=0, PING=28, SEC=2038634, NAME=user068, COMM=
CLIENT: ADMIN=0, CID=904, PHAN=0, PING=224, SEC=1711430, NAME=user054, COMM=
CLIENT: ADMIN=0, CID=904, PHAN=0, PING=83, SEC=1305773, NAME=user061, COMM=
CLIENT: ADMIN=1, CID=904, PHAN=0, PING=4, SEC=1287818, NAME=SIGMA SQUAWK, COMM=
CLIENT: ADMIN=0, CID=904, PHAN=0, PING=3, SEC=1085812, NAME=user089, COMM=
CLIENT: ADMIN=0, CID=904, PHAN=0, PING=310, SEC=1039354, NAME=user233, COMM=
CLIENT: ADMIN=1, CID=904, PHAN=0, PING=2, SEC=1036810, NAME=user140, COMM=

An example of the process for the sender of a packet follows. In this case it is an example of a client sending a detailed status query request with an empty password. The process of a server responding to a request is largely analogous, only differing in data length and the number of packets sent.

1. A buffer large enough to host a header (20 bytes) and the text data (maximum 492 bytes) is established, e.g. a 512 byte buffer.

2. The first 20 bytes represent the header structure and are initialized with currently available information.
   - Bytes at offsets 2-3 are set to zero.
   - The command number field (offsets 4-5) is set to 2, indicating a a detailed status request.
   - The query ID (offsets 6-7) is set to a random 2 byte value.
   - The total data size field (offsets 8-9) is set to 16 in the case of a status request query because its data contains only the 16 bytes of the server status password.
   - The data size of this packet (offsets 10-11) is set to 16 because it is the only packet sent (the total data of a status request query fits inside it).
   - The total number of packets in the query (offsets 12-13) is set to 1 because only one packet is sent.
   - Current packet’s ordinal number in the query (offsets 14-15) is set to 0 because it is the first and only packet that will be sent as part of this status request query.

3. Bytes following the header are initialized with this packet’s share of text data. This text data contains only a 16 byte password in the case of a status query request.
4. A 2 byte checksum of the total plain text data (of all packets combined) is calculated, and the relevant field (offsets 18-19) is set to the checksum value.

5. A 2 byte pseudorandom key for encrypting and decrypting the text data of this packet is created, the packet’s data is encrypted in place with the key, and the relevant field (offsets 16-17) is set to the key value.

6. A 2 byte pseudorandom key for encrypting and decrypting the header of this packet is created, the packet’s header is encrypted in place with the key, and the relevant field (offsets 0-1) is set to the key value.

7. At this point the header and data are encrypted in the buffer each with their own key and algorithm, with the exception of the header’s first field (offsets 0-1) where the header key is set in plaintext. The 36 byte (20 byte header + 16 byte data for a status query request) packet is then sent to the server’s UDP port.

An example of the process for the receiver of a packet follows. It is essentially the sending process done in reverse.

1. The packet is received into a buffer large enough to host a header (20 bytes) and the text data (maximum 492 bytes), e.g. a 512 byte buffer.

2. The header is decrypted in place with the header key (offsets 0-1). The data key, query ID, and other header fields are revealed in plain text.

3. The query ID is checked to identify if this is a response to the sent request.

4. The other fields, such as packet numbers and sizes, are checked for inconsistencies.

5. The data is decrypted in place with the data key (offsets 16-17).

6. This process repeats from step 1. until all expected packets have been received in full, i.e. until all packet ordinal numbers up to the total number of packets have been encountered, and the total received size equals the total size indicated by the packet headers.

7. The text data sections of all received packets are concatenated in order of their respective packet ordinal numbers.

8. The checksum is calculated from the total plain text data, and is checked against the checksum field (offsets 18-19) in any of the received packets. All received packets should have the same checksum field value, as it is calculated for the full text data.

9. At this point the total plain text data has been verified and is available for any further processing. A server responding to a status query request would form a status query response and send it to the client, in the same way the client sent the initial request (as described in the previous example). The client would then receive and interpret the response in the same way the server received the initial request (as described in this example).

The probe sent with this solution has a static payload generated from a detailed (command number 2) status query request made with an empty password and a query ID of 0x33CF. The UDP payload made as an entry for the nmap-payloads file differs from it in that it is generated from a general (command number 1) status query request with a random and irrelevant query ID. This is done because a general request causes the server to respond with minimal information that fits in a single reply packet, which is enough for detecting an open UDP port when no version detection is being done and the response is not going to be interpreted.
Approach: Nmap Scripting Engine (NSE)

Script file: ventrilo-info.nse

Script description:
Detects the Ventrilo voice communication server service versions 2.1.2 and above. Some of the older versions (pre 3.0.0) may not have the UDP service this probe relies on enabled by default.

The Ventrilo server listens on a TCP (voice and control) and an UDP (ping and status) port with the same port number (fixed to 3784 in the free version, otherwise configurable). This script activates on both a TCP and UDP port version scan. In both cases probe data is sent only to the UDP port because it allows for a simple and informative status command as implemented by the "ventrilo_status.exe" executable which has shipped alongside the Windows server package since version 2.1.2 when the UDP status service was added.

When run as a version detection script (-sv), the script will report on the server version, name, uptime, authentication scheme, and OS. When run explicitly (--script ventrilo-info), the script will additionally report on the server name phonetic pronunciation string, the server comment, maximum number of clients, voice codec, voice format, channel and client counts, and details about channels and currently connected clients.

Original reversing of the UDP status query protocol was done by Luigi Auriemma (http://aluigi.altervista.org/papers.htm#ventrilo) [24].

Example output (full invocation with --script ventrilo-info):

```
PORT STATE SERVICE VERSION
9408/tcp open ventrilo Ventrilo 3.0.3.C
 | ventrilo-info:
 | name: TypeFrag.com
 | phonetic: Type Frag Dot Com
 | comment: http://www.typefrag.com/
 | auth: pw
 | max. clients: 100
 | voice codec: 3,Speex
 | voice format: 32,32 KHz%2C 16 bit%2C 10 Qlty
 | uptime: 152h:56m
 | platform: WIN32
 | version: 3.0.3.C
 | channel count: 14
 | channel fields: CID, PID, PROT, NAME, COMM
 | client count: 6
 | client fields: ADMIN, CID, PHAN, PING, SEC, NAME, COMM
 | channels:
 | <top level lobby> (CID: 0, PID: n/a, PROT: n/a, COMM: n/a): <empty>
 | Group 1 (CID: 719, PID: 0, PROT: 0, COMM: ):
 | stabya (ADMIN: 0, PHAN: 0, PING: 47, SEC: 206304, COMM: )
 | Group 2 (CID: 720, PID: 0, PROT: 0, COMM: ): <empty>
 | Group 3 (CID: 721, PID: 0, PROT: 0, COMM: ):
 | Group 4 (CID: 722, PID: 0, PROT: 0, COMM: ):
 | Group 5 (CID: 723, PID: 0, PROT: 0, COMM: ):
 | Sir Master Win (ADMIN: 0, PHAN: 0, PING: 32, SEC: 186890, COMM: )
 | waterbukk (ADMIN: 0, PHAN: 0, PING: 31, SEC: 111387, COMM: )
 | likez (ADMIN: 0, PHAN: 0, PING: 140, SEC: 22457, COMM: )
 | Tweet (ADMIN: 0, PHAN: 0, PING: 140, SEC: 21009, COMM: )
 | Group 6 (CID: 724, PID: 0, PROT: 0, COMM: ):
 | Raid (CID: 725, PID: 0, PROT: 0, COMM: ):
45
```
Approach: nmap-service-probes entry

nmap-service-probes file modifications:
Add a new probe entry:

```plaintext
Probe UDP Ventrilo q|\x23\xce\52\x96\01\xfc\18\x07\xc5\x45\xbd\x62
 \x24\x62\x01\x35\xf3\6f\x99\ec\xe4\01\xba\xa4\x72\x5d\xe3\xb0\df
 \xe2\x12\x96\50\5e\24|
rarity 9
ports 3784
```

match ventrilo m|^.{111}|s p/Ventrilo/ v/2.1.2+/

Solution description:
This edition of the probe detects no extra version information, but still hardmatches a Ventrilo 2.1.2+ service. It can be useful when not using NSE or when using a stricter portrule in the NSE script that doesn’t necessarily match on all open ports, but only ones that are probably running the service. In the latter case this solution can act as "softmatch" that enables the NSE solution and is controllable by the rarity metric.

This probe only detects the UDP port. A different probe payload is sent than in the NSE solution. The difference is in the level of detail requested. The NSE script sends a detailed status request, whereas a general status request is used here. A general status response was chosen because it comes as a single packet and the received data is discarded in any case. Because the payload is encrypted, a match can only be made on the minimum possible length of the reply payload, which has been estimated at 111 bytes. This is a conservative assumption based on a response that has all its information fields empty, i.e. a response with only overhead payload data.

Example output:
PORT STATE SERVICE VERSION
3784/udp open ventrilo Ventrilo 2.1.2+

UDP payload (the nmap-payloads file entry):

```plaintext
# Ventrilo 2.1.2+
# UDP general status request (encrypted).
# See http://aluigi.altervista.org/papers.htm#ventrilo
udp 3784 "\x01\xe7\xe5\x75\x31\xa3\x17\x0b\x21\xc5\oord\x99\x4e\xdd
 \x19\xac\xde\x08\x5f\x8b\x24\x0a\x11\x19\xb6\x73\x6f\xad\x28\x13\x0a\xb9\x12\x75"
```
4.4. TeamSpeak 2

TeamSpeak 2 is a voice chat application for groups. While it can be used for any kind of activity, it is primarily intended for gaming. It can be compared to programs like Ventrilo or Mumble. It is proprietary software, but the TeamSpeak 2 client and server are both available as freeware as long as they are not used within a commercial environment [25]. This chapter addresses the TeamSpeak 2 server when not explicitly stated otherwise.

**Supported platforms:** Windows 32-bit, Linux 32-bit [25]

**Release versions:**
- 2.0.17.17 (2002-08-26) - 2.0.24.1 (2007-09-02; latest version): detected

**Ports:**
- **TCP** - the TCPQuery interface (admin telnetd) port; port number 51234 by default
- **TCP** - the web admin interface (HTTP) port; port number 14534 by default
- **UDP** - the voice and control port (the TeamSpeak 2 protocol); port number 8767 by default

These three services are distinct, and a separate version detection solution has been developed for each of them. The TCP ports are administrative services, and are not used within normal client-server communication. Mentions of the TeamSpeak 2 (TS2) protocol refer to the UDP protocol if not explicitly stated otherwise.

### 4.4.1. TCP: The TCPQuery Interface (Admin Telnetd) Port

**Protocol description:**
This service is an administration telnet daemon. Upon establishing a TCP connection it will greet with a [TS]\r\n banner. This greeting will identify the service as a TS2 TCPQuery service, but more detailed information can be gathered by interrogating the service further. The user can issue various commands upon connecting, such as `login`, `playerlist`, `kick`, etc. The command of interest for version detection is `ver`, because it gets the server to respond with a string like **2.0.23.19 Win32 Freeware**. This command is used to extract the version number, the platform, and what seems to be license information.

**Approach:** nmap-service-probes entry

**nmap-service-probes file modifications:**
Remove the following line from Probe TCP NULL (if it exists):

```
match telnet m[^\[TS\]\r\n] p/Teamspeak VoIP Information telnetd/
```

Add the following line to Probe TCP NULL:

```
softmatch ts2-TCPQuery m[^\[TS\]\r\n]
```

Add a new probe entry:

```
Probe TCP verLine q|ver|\r\n| rarity 9
ports 51234

match ts2-TCPQuery m[^\[TS\]\r\n(S+) (S+) (S+)\r\nOK\r\n(S+)\r\nOK\r\n(S+)\r\nOK\r\n(S+)\r\nOK\r\n(S+)
  p/TeamSpeak 2 server TCPQuery interface (telnetd)/ v/$1/ i/$3/ o/$2/
match ts2-TCPQuery m[^\[TS\]\r\n(S+)\r\nOK\r\n(S+)\r\nOK\r\n(S+)
  p/TeamSpeak 2 server TCPQuery interface (telnetd)/ v/$1/ o/$2/
```

47
Solution description:
An existing hardmatch line for the TCP NULL probe is replaced by a softmatch, and a new probe with a hardmatch rule is added that triggers on that softmatch. The new probe will garner a more precise service name, version number, platform, and some extra information in newer versions (usually the license status).

Example output:
```
PORT STATE SERVICE VERSION
51234/tcp open  ts2-TCPQuery TeamSpeak 2 server TCPQuery interface
   (telnetd) 2.0.24.1 (Freeware)
Service Info: OS: Linux
```

### 4.4.2. TCP: The Web Admin Interface (HTTP) Port

**Protocol description:**
This is a HTTP web administration service. It allows administrators to log in and monitor, configure, and moderate the server. It is recognizable by its specific HTTP `<title>` element.

**Approach:** `nmap-service-probes` entry

**nmap-service-probes file modifications:**
Remove the following line from `Probe TCP GetRequest` (if it exists):
```
match http m|^HTTP/1.1 200 OK\r\n.*Server: Indy/(\[\w._-]+)\r\n|s p/Indy/ v/$1/
```
Add the following lines to `Probe TCP GetRequest` (if they don’t already exist):
```
match http m|^HTTP/1.1 \d+\d\r\nConnection: keep-alive\r\nContent-Type: text/HTML\r\nContent-Length: \d+\r\nServer: Indy/(\[\d.]+)\r\nSet-Cookie: .\r\n<-- header.html
.--.*TeamSpeak\s p/TeamSpeak admin httpd/ v/1.X / i/Indy httpd $1/
```
```
match http m|^HTTP/1.1 \d+\d\r\nConnection: keep-alive\r\nContent-Type: text/HTML\r\nContent-Length: \d+\r\nServer: Indy/(\[\d.]+)\r\nSet-Cookie: .\r\n<title>TeamSpeak 2 - Server-Administration</title>|s p/TeamSpeak admin httpd/ v/2.X / i/Indy httpd $1/
```

**Solution description:**
A match rule seems to already exist, but requires a modification in order to become operational. Another, more general, match line blocked the execution of relevant more specific match lines. That line is removed, and the TS2 specific match lines are added if they didn’t already exist.

Example output:
```
PORT STATE SERVICE VERSION
51234/tcp open  http TeamSpeak admin httpd 2.X (Indy httpd)
```

### 4.4.3. UDP: The Voice and Control Port (The TeamSpeak 2 Protocol)

**Protocol description:**
Some parts of the TeamSpeak 2 protocol (TS2) have been documented and implemented
as a Wireshark protocol dissector [26]. Project TeamBlibbityBlabbity [27] is an attempt to
document and provide an example implementation of the proprietary TS2 protocol, and the
source code may be useful when looking into the protocol.

This UDP port handles everything in a regular connection between a TS2 client and
server. This includes initiating, establishing, and maintaining a connection, transferring
channel and client data, transferring voice data, keepalive pings, etc. The developed ser-
vice detection solution sends a login request packet as if it wanted to establish a connection
to the server. The server then replies with an informative login reply packet from which
version details can be discerned. The packet fields are formatted in little endian. The login
request UDP payload seems to have a fixed size of 180 bytes, while the login reply UDP pay-
load seems to have a fixed size of 436 bytes. The reply size and packet structure may vary in
versions prior to 2.0.19.16, and in that case detection will not gather any extra information
other than hardmatching the TS2 service.

The probe packets described in the following tables show the traffic of a TS2 2.0.32.60
client connecting to a TS2 2.0.23.19 server, both running on Windows XP. The response
packet is described in a more general fashion since the values depend more on the replying
server, as opposed to the request packet which is sent with a static payload. The service
detection probe uses an empty login name and password. If the server requires a password,
the reply packet will lack version number, server name, and server platform data. However,
it will still detect such a TS2 server and note that it has a password.

Table 4.7: The TeamSpeak 2 UDP login request packet

<table>
<thead>
<tr>
<th>Offset</th>
<th>Type</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>uint16</td>
<td>0xBEF4</td>
<td>Class: connection.</td>
</tr>
<tr>
<td>2-3</td>
<td>uint16</td>
<td>0x0003</td>
<td>Type: login request.</td>
</tr>
<tr>
<td>4-7</td>
<td>uint32</td>
<td>0</td>
<td>Session key.</td>
</tr>
<tr>
<td>8-11</td>
<td>uint32</td>
<td>0</td>
<td>Client id. Always zero?</td>
</tr>
<tr>
<td>12-15</td>
<td>uint32</td>
<td>1</td>
<td>Sequence number.</td>
</tr>
<tr>
<td>16-19</td>
<td>uint32</td>
<td>0x85BA7832</td>
<td>A CRC32 checksum. Unknown what of.</td>
</tr>
<tr>
<td>20</td>
<td>uint8</td>
<td>9</td>
<td>Length of the protocol string.</td>
</tr>
<tr>
<td>21-49</td>
<td>string</td>
<td>&quot;TeamSpeak&quot;</td>
<td>Protocol string.</td>
</tr>
<tr>
<td>50</td>
<td>uint8</td>
<td>10</td>
<td>Length of the platform string.</td>
</tr>
<tr>
<td>51-79</td>
<td>string</td>
<td>&quot;Windows XP&quot;</td>
<td>Platform string.</td>
</tr>
<tr>
<td>80-81</td>
<td>uint16</td>
<td>2</td>
<td>1. client version (e.g. the ‘2’ in 2.0.32.60).</td>
</tr>
<tr>
<td>82-83</td>
<td>uint16</td>
<td>0</td>
<td>2. client version (e.g. the ‘0’ in 2.0.32.60).</td>
</tr>
<tr>
<td>84-85</td>
<td>uint16</td>
<td>32</td>
<td>3. client version (e.g. the ‘32’ in 2.0.32.60).</td>
</tr>
<tr>
<td>86-87</td>
<td>uint16</td>
<td>60</td>
<td>4. client version (e.g. the ‘60’ in 2.0.32.60).</td>
</tr>
<tr>
<td>88</td>
<td>uint8</td>
<td>0</td>
<td>Unknown.</td>
</tr>
<tr>
<td>89</td>
<td>uint8</td>
<td>1</td>
<td>Unknown.</td>
</tr>
<tr>
<td>90</td>
<td>uint8</td>
<td>0</td>
<td>Length of the login name.</td>
</tr>
<tr>
<td>91-119</td>
<td>string</td>
<td>&quot;&quot; (zeroes)</td>
<td>Login name.</td>
</tr>
<tr>
<td>120</td>
<td>uint8</td>
<td>0</td>
<td>Length of the login password.</td>
</tr>
<tr>
<td>121-149</td>
<td>string</td>
<td>&quot;&quot; (zeroes)</td>
<td>Login password.</td>
</tr>
<tr>
<td>150</td>
<td>uint8</td>
<td>8</td>
<td>Length of the nickname.</td>
</tr>
<tr>
<td>151-179</td>
<td>string</td>
<td>&quot;nickname&quot;</td>
<td>Nickname.</td>
</tr>
</tbody>
</table>
### Table 4.9: The TeamSpeak 2 UDP login reply packet

<table>
<thead>
<tr>
<th>Offset</th>
<th>Type</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td><code>uint16</code></td>
<td>0xBEF4</td>
<td>Class: connection.</td>
</tr>
<tr>
<td>2-3</td>
<td><code>uint16</code></td>
<td>0x0004</td>
<td>Type: login reply.</td>
</tr>
<tr>
<td>4-7</td>
<td><code>uint32</code></td>
<td>0</td>
<td>Session key; zero on first reply.</td>
</tr>
<tr>
<td>8-11</td>
<td><code>uint32</code></td>
<td>client id</td>
<td>Possibly # users + 1?</td>
</tr>
<tr>
<td>12-15</td>
<td><code>uint32</code></td>
<td>2</td>
<td>Sequence number; 2 on first reply.</td>
</tr>
<tr>
<td>16-19</td>
<td><code>uint32</code></td>
<td>a crc32 checksum</td>
<td>Unknown what of.</td>
</tr>
<tr>
<td>20</td>
<td><code>uint8</code></td>
<td>server name length</td>
<td></td>
</tr>
<tr>
<td>21-49</td>
<td><code>string</code></td>
<td>server name</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td><code>uint8</code></td>
<td>platform length</td>
<td></td>
</tr>
<tr>
<td>51-79</td>
<td><code>string</code></td>
<td>platform</td>
<td></td>
</tr>
<tr>
<td>80-81</td>
<td><code>uint16</code></td>
<td>1. server version</td>
<td>E.g. the &quot;2&quot; in &quot;2.0.23.19&quot;.</td>
</tr>
<tr>
<td>82-83</td>
<td><code>uint16</code></td>
<td>2. server version</td>
<td>E.g. the &quot;0&quot; in &quot;2.0.23.19&quot;.</td>
</tr>
<tr>
<td>84-85</td>
<td><code>uint16</code></td>
<td>3. server version</td>
<td>E.g. the &quot;23&quot; in &quot;2.0.23.19&quot;.</td>
</tr>
<tr>
<td>86-87</td>
<td><code>uint16</code></td>
<td>4. server version</td>
<td>E.g. the &quot;19&quot; in &quot;2.0.23.19&quot;.</td>
</tr>
<tr>
<td>88-179</td>
<td><code>bytes</code></td>
<td>unknown</td>
<td></td>
</tr>
<tr>
<td>180</td>
<td><code>uint8</code></td>
<td>welcome message length</td>
<td></td>
</tr>
<tr>
<td>181-435</td>
<td><code>string</code></td>
<td>welcome message</td>
<td></td>
</tr>
</tbody>
</table>

**Approach:** Nmap Scripting Engine (NSE)

**Script file:** teamspeak2-version.nse

**Script description:**

Detects the TeamSpeak 2 server UDP voice communication service.

A single UDP packet (a login request) is sent. If the server does not have a password set, the exact version, name, and OS type will also be reported on.

For more info on the protocol see:

- http://wiki.wireshark.org/TeamSpeak2 (and the Wireshark TS2 protocol dissector) [26]
- http://sourceforge.net/projects/teambb/ [27]

**Example output:**

```
PORT STATE SERVICE VERSION
8767/udp open ts2 TeamSpeak 2 server 2.0.23.19
   (name: COWCLANS; no password)
Service Info: OS: Win32
```

**Approach:** nmap-service-probes entry

**nmap-service-probes file modifications:**

Add a new probe entry:

```
Probe UDP TeamSpeak2 q|\xf4\xbe\x03\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
Solution description:
This edition of the probe doesn’t detect the exact service version. It can be useful when not using NSE or when using a stricter portrule in the NSE script that doesn’t necessarily match on all open ports, but only ones that are probably running the service. In the latter case this solution can act as "softmatch" that enables the NSE solution and is controllable by the rarity metric.

The same probe payload is sent as in the NSE solution. Precise version information can not be gathered with this approach as the version data is not formatted as text. The server name and OS can still be had if the server doesn’t have a password set. The first match line matches the response of a server with no password set, while the second matches the response of a server with a password set. If the server has a password set, the server name and OS fields will be empty (zeros).

Example output:
PORT STATE SERVICE VERSION
8767/udp open ts2 TeamSpeak 2 (name: COWCLANS; no password)
Service Info: OS: Win32

UDP payload (the nmap-payloads file entry):
# TeamSpeak 2
# UDP login request
# See http://wiki.wireshark.org/TeamSpeak2

4.5. TeamSpeak 3

TeamSpeak 3 is a voice chat application for groups. While it can be used for any kind of activity, it is primarily intended for gaming. It can be compared to programs like Ventrilo or
Mumble. It is proprietary software, but the TeamSpeak 3 client and server are both available as freeware as long as they are not used within a commercial environment [25]. This chapter addresses the TeamSpeak 3 server when not explicitly stated otherwise.

**Supported platforms:** Windows 32/64-bit, Linux 32/64-bit, Mac OS X [25]

**Release versions:**


**Ports:**

TCP - the ServerQuery interface (admin telnetd) port; port number 10011 by default

UDP - the voice and control port (the TeamSpeak 3 protocol); port number 9987 by default

These two services are distinct and a separate version detection solution has been developed for each of them. The TCP port is an administrative service, and is not used within normal client-server communication. Mentions of the TeamSpeak 3 (TS3) protocol refer to the UDP protocol if not explicitly stated otherwise.

### 4.5.1. TCP: The ServerQuery Interface (Admin Telnetd) Port

**Protocol description:**

This service is an administration telnet daemon. Upon establishing a TCP connection it will greet with a `TS3
\n` banner or the similar longer one mentioned above in a `softmatch` directive. This greeting will identify the service as a TS3 ServerQuery service, but more detailed information can be gathered by interrogating the service further. The user can issue various commands upon connecting, such as `login`, `clientlist`, `clientkick`, etc. The command of interest for version detection is `version`, because it gets the server to respond with a string like `version=3.0.7.2 build=1368605352 platform=Windows`. This command is used to extract the version number, build id, and platform.

**Approach:** `nmap-service-probes` entry

**nmap-service-probes file modifications:**

Remove the following line from `Probe TCP NULL` (if they exist):

```
match teamspeak m|^TS3\n\r$| p/TeamSpeak voice communication/ v/3/
```

Add the following lines to `Probe TCP NULL`:

```
softmatch ts3-ServerQuery m|^TS3\n\rWelcome to the TeamSpeak 3 ServerQuery
   interface, type \"help\" for a list of commands and \"help <command>\" for information on a specific command\n\n\r$| p/TeamSpeak voice communication/ v/3/
```

Add a new probe entry:

```
Probe TCP versionLine q|version\n\r
rarity 9
ports 10011
```
Solution description:
Two existing hardmatch lines for the TCP NULL probe are replaced by softmatch lines, and a new probe with a hardmatch rule is added that triggers on those softmatches. The new probe will garner a more precise service name, version number, build id, and platform.

Example output:

```
PORT STATE SERVICE VERSION
10011/tcp open ts3-ServerQuery TeamSpeak 3 server
   ServerQuery interface (telnetd) 3.0.6.1 (build: 1340956745)
Service Info: OS: Linux
```

4.5.2. UDP: The Voice and Control Port (The TeamSpeak 3 Protocol)

Protocol description:
The TeamSpeak 3 (TS3) protocol is a complete rewrite of the TeamSpeak 2 protocol, and the UDP payloads are encrypted. The approach to creating the service detection solution remains the same, but in the case of TS3 the payload to be sent is copied from a TS3 client’s network traffic with no understanding of the payload structure. The only thing known is that it is used as a connection initiator.

This UDP port handles everything in a regular connection between a TS3 client and server. This includes initiating, establishing, and maintaining a connection, transferring channel and client data, transferring voice data, keepalive pings, etc. The developed service detection solution sends an encrypted packet as if it wanted to establish a connection to the server. The server then replies with an encrypted packet which has some specific constant bytes that can be matched, as indicated by the `match ts3` directive described below.

The login request UDP payload seems to have a fixed size of 162 bytes, while the login reply UDP payload size seems to vary between 182 and 183 bytes. Until further work on decrypting the traffic is done, a TS3 server can merely be identified and no additional version information can be gathered.

Approach:
nmap-service-probes entry
nmap-service-probes file modifications:
Add a new probe entry:

```
Probes UDP TeamSpeak3 q|\x05\xca\x7f\x16\x9c\x11\xf9\x89\x00\x00\x00\x00
   \x02\x9d\x74\x8b\x45\xaax\x7b\xef\xb9\x9e\xfe\xad\x08\x19\xba\xcf\x41
   \xe0\x16\xa4\x32\x6c\xf3\xcf\xf4\x8e\x3c\x44\x83\xc8\x8d\x51\x45\x6f
   \x90\x95\x23\x3e\x00\x97\x76\x8b\x54\xad\x79\xe3\af\x87\eb
   \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
```

```
match ts3 m|^.{8}00\x00\x02\x97\x76\x8b\x54\xad\x79\xe3\xf87\xeb
   \xa5\x19\xba\xcf\x41\xe0\x16\xa4\x32\x6c\xf3\xcf\xf4\x8e\x3c\x44
```
Solution description:
A new probe is added that delivers the UDP payload that a TS3 client sent when connecting to a server. The probe elicits a response of a server that accepted the initial connection packet. This response has distinct properties and is matched with a regular expression match rule. No extra information is collected apart from establishing that it is a TS3 server.

Example output:

<table>
<thead>
<tr>
<th>PORT</th>
<th>STATE</th>
<th>SERVICE</th>
<th>VERSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>9997/udp</td>
<td>open</td>
<td>ts3</td>
<td>TeamSpeak 3 server</td>
</tr>
</tbody>
</table>

UDP payload (the nmap-payloads file entry):

```plaintext
# TeamSpeak 3
# UDP login request (encrypted)
udp 9987 "\x05\xca\xf7f\x16\x9c\x11\xf9\x89\x00\x00\x00\x02\x9d\x74
\x8b\x45\x7b\xef\xb9\x8b\x45\x7a\xe8\x83\xc8\x8d\x51\x45\x6f\x90\x95\x23
\x3e\x00\x97\x2b\x1c\x71\xb2\x4e\xc0\x61\xf1\xd7\x6f\x57\xe6\x48
\x52\x82\xa6\x9a\x23\x65\xaa\x18\x7a\x17\x38\xc3\x81\x27\xc3\x47
\xcf\xa7\x35\xba\xcf\x0f\xd9\x9d\x72\x24\x9d\xcf\x02\x17\x6d\x6b\xb1
\x2d\x72\xc6\xe3\x17\xc1\x95\x9d\x69\x99\x57\xce\x6d\x05\xdc\x03
\x94\x56\x04\x3a\x14\xe5\xad\x9a\x2b\x14\x30\x3a\x23\xa3\x25\xad\xe8
\xe6\x39\x8a\x85\x2a\xc6\xdf\xe5\x5d\x2d\x02f\x5d\x9c\xd7\x2b\x24
\xfb\xb0\x9c\xc2\x8a\x89\xb4\x1b\x17\xa2\xb6"
```

4.6. The All-Seeing Eye Protocol

The All-Seeing Eye (ASE) was a game server browser designed by UDP Soft. It was released on 2001-06-15 and was discontinued on 2008-05-15. Many game servers implement the ASE status querying protocol. Even though the ASE application has ceased to function, some games still respond to the ASE protocol status query. Some games servers that support the ASE protocol can be found at the GameStat [28][29] and GameQ [30] project web pages. Those listings are incomplete and may have inaccurate information. This chapter addresses the ASE protocol (as opposed to the application) when not explicitly stated otherwise.

Ports:

UDP - ASE; default port number is the game port number + 123

Protocol description:

The GameStat [28][29][31] and GameQ [30][32] project pages and their source code were used as primary references on the protocol. Another resource on the ASE protocol is Luigi Auriemma’s website [33] which hosts his implementations of various aspects of the protocol.

The ASE protocol runs either on the game server port (within the actual game server service), or on a separate port number. If ASE is on a separate port, its port number will most often be the game server port number + 123.

There are 4 different query types (UDP payloads) that an ASE game server might respond to [33]:

"p" - used only for pinging the server; the response should be "P"
"g" - the response should be the name of the server’s game; untested
"v" - used by ASE scanner servers to verify the remote game server; untested
"s" - returns server status information; used in version detection solutions

The ASE protocol uses a string type that starts with a 1-byte integer which specifies the string length (with the byte included). The byte 0x01 represents an empty string ("""). The only exception is the "EYE1" identifier string at the beginning of the response data which consists of only the 4 ASCII bytes (no initial length byte).

**Table 4.11:** The ASE status query ("s") response

<table>
<thead>
<tr>
<th>Offset</th>
<th>Type</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>bytes</td>
<td>&quot;EYE1&quot;</td>
<td>Not an ASE string; just the 4 ASCII bytes.</td>
</tr>
<tr>
<td>4- (A-1)</td>
<td>string</td>
<td>game name</td>
<td>E.g. &quot;chrome&quot;.</td>
</tr>
<tr>
<td>A- (B-1)</td>
<td>string</td>
<td>game port</td>
<td>The game server can run on a port separate from ASE.</td>
</tr>
<tr>
<td>C- (D-1)</td>
<td>string</td>
<td>server name</td>
<td>E.g. &quot;My Awesome Chrome Server!&quot;.</td>
</tr>
<tr>
<td>D- (E-1)</td>
<td>string</td>
<td>game type</td>
<td>E.g. &quot;Team Death Match&quot;.</td>
</tr>
<tr>
<td>E- (F-1)</td>
<td>string</td>
<td>map name</td>
<td>E.g. &quot;Data/LevelsNet/Narrow/Narrow.map&quot;.</td>
</tr>
<tr>
<td>F- (G-1)</td>
<td>string</td>
<td>game version</td>
<td>E.g. &quot;1.2.0.ww&quot;.</td>
</tr>
<tr>
<td>G- (H-1)</td>
<td>string</td>
<td>has password?</td>
<td>&quot;1&quot; if password set, &quot;0&quot; otherwise.</td>
</tr>
<tr>
<td>H- (I-1)</td>
<td>string</td>
<td>num. players</td>
<td>Number of players currently playing.</td>
</tr>
<tr>
<td>I- (J-1)</td>
<td>string</td>
<td>max. players</td>
<td>Maximum number of players.</td>
</tr>
<tr>
<td>J- (K-1)</td>
<td>strings</td>
<td>key-value pairs</td>
<td>Various server settings as key-value pairs of strings.</td>
</tr>
<tr>
<td>K</td>
<td>string</td>
<td>&quot;&quot; (0x01)</td>
<td>End of key-value pairs.</td>
</tr>
<tr>
<td>(K+1)-</td>
<td>players</td>
<td>player data</td>
<td>Repeats for every player in the server. Described below.</td>
</tr>
</tbody>
</table>

The player data structures at the end of the response consist of a byte (bitwise flags) that describes strings that follow it, and of the strings themselves. The bits are tested starting from the least significant bit, and if a bit is found to be set, it means that a corresponding string follows:

- **bit 0 (LSB)** (flags & 1 != 0) - player name
- **bit 1** (flags & 2 != 0) - team name
- **bit 2** (flags & 4 != 0) - skin name
- **bit 3** (flags & 8 != 0) - score
- **bit 4** (flags & 16 != 0) - ping (latency)
- **bit 5** (flags & 32 != 0) - time spent in server (current session)

For example, if the byte containing flags was 0x13, the strings following it would describe (in order): the player name, team name, and ping. There is such a player data structure for every player in the server.

**Approach:** Nmap Scripting Engine (NSE)

**Script file:** ase-info.nse
Script description:
Detects the All-Seeing Eye service. Provided by some game servers for querying the server’s status.

The All-Seeing Eye service can listen on a UDP port separate from the main game server port (usually game port + 123). On receiving a packet with the payload "s", it replies with various game server status info.

When run as a version detection script (-sV), the script will report on the game name, version, actual port, and whether it has a password. When run explicitly (-script ase-info), the script will additionally report on the server name, game type, map name, current number of players, maximum number of players, player information, and various other server settings.

For more info on the protocol see:
http://int64.org/docs/gamestat-protocols/ase.html [29]
http://aluigi.altervista.org/papers.htm#ase [33]

Example output (full invocation with --script ase-info):
PORT  STATE  SERVICE    REASON      VERSION
27138/udp open  ase   udp-response All-Seeing Eye
(game: chrome 1.2.0.0ww; port: 27015; no password)
| ase-info:
| game: chrome
| port: 27015
| server name: ChromeNet Server
| game type: Team Death Match
| map: Data/LevelsNet/Narrow/Narrow.map
| version: 1.2.0.0ww
| passworded: 0
| num players: 2
| max players: 16
| settings:
| Dedicated: No
| Password Required: No
| Time Limit: 30
| Points Limit: 200 min.
| Respawns Limit: unlimited
| Respawn Delay: 10 sec.
| Enemies Visible On Map: No
| Available Inventory Room: Yes
| Identify Enemy Players: No
| Available Vehicles: Yes
| Vehicle Respawns Limit: unlimited
| Vehicle Respawn Delay: 30 sec.
| Vehicle Auto Return Time: 90 sec.
| Vehicles Visible On Map: Yes
| Team Balance: Off
| Friendly Fire: On
| Friends Visible On Map: Yes
| players:
| player 0:
| name: NoVoDondo
| team: BLUE
| skin:
| score: 71
| ping: 0
The script also produces XML output when Nmap is given the -oX (XML output) switch. An example can be found in the script file (ase-info.nse).

**Approach:** nmap-service-probes entry

**nmap-service-probes file modifications:**

Add a new probe entry:

```
Probe UDP ASE q|s|
rarity 9
ports 1258,2126,3123,12444,13200,23196,26000,27138,27244,27777,28138
```

```
match ase m/^EYE1.(.*?)(?i)|\x02(\d)|\x03(\d{2})|\x04(\d{3})|\x05(\d{4})|\x06(\d{5}))/s p/All-Seeing Eye/ i/game: $1; port: $2/
```

**Solution description:**

This edition of the probe detects only the game name and the actual game server port number. It can be useful when not using NSE or when using a stricter portrule in the NSE script that doesn't necessarily match on all open ports, but only ones that are probably running the service. In the latter case this solution can act as "softmatch" that enables the NSE solution and is controllable by the rarity metric.

The same probe payload is sent as in the NSE solution, but most of the received information is ignored as it cannot be parsed with regular expressions.

**Example output:**

```
PORT STATE SERVICE VERSION
27138/udp  open  ase  All-Seeing Eye  (game: chrome 1.2.0.0ww; port: 27015)
```

4.7. Freelancer

Freelancer is a space trading and combat simulation video game developed by Digital Anvil and published by Microsoft Game Studios. This chapter addresses the Freelancer multiplayer game server (FLServer.exe) service when not explicitly stated otherwise.

**Supported platforms:** Windows

**Release versions:**

```
1.0 (2003-03-04) - 1.1 (2003-06-06; latest release): detected
```

**Ports:**

```
UDP - the game/status port; port number 2302 by default
```
Protocol description:
There is little documentation on the Freelancer protocol apart from the UDP status query payload and some response parsing code found in the GameQ project [32].

The status query UDP payload has a fixed size of 21 bytes, and the response is a single UDP packet whose payload size depends on the server name and description string lengths. The packet fields are apparently formatted in little endian.

Table 4.13: The Freelancer UDP status query response payload

<table>
<thead>
<tr>
<th>Offset</th>
<th>Type</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–3</td>
<td>uint32</td>
<td>0x26F10300</td>
<td>Presumably indicates a status query reply.</td>
</tr>
<tr>
<td>4–7</td>
<td>uint32</td>
<td>payload offset of server name’s last character</td>
<td></td>
</tr>
<tr>
<td>8–11</td>
<td>uint32</td>
<td>unknown</td>
<td></td>
</tr>
<tr>
<td>12–15</td>
<td>uint32</td>
<td>0x50</td>
<td></td>
</tr>
<tr>
<td>16–19</td>
<td>uint32</td>
<td>bit 7 (LSB is 0) indicates a password</td>
<td>has_password = x &amp; 128</td>
</tr>
<tr>
<td>20–23</td>
<td>uint32</td>
<td>max. players + 1</td>
<td></td>
</tr>
<tr>
<td>24–27</td>
<td>uint32</td>
<td>num. players + 1</td>
<td></td>
</tr>
<tr>
<td>28–31</td>
<td>uint32</td>
<td>0x58</td>
<td></td>
</tr>
<tr>
<td>32–35</td>
<td>uint32</td>
<td>unknown</td>
<td></td>
</tr>
<tr>
<td>36–59</td>
<td>bytes</td>
<td>zeroes</td>
<td></td>
</tr>
<tr>
<td>60–75</td>
<td>bytes</td>
<td>random?</td>
<td>Changes with each server restart.</td>
</tr>
<tr>
<td>76–91</td>
<td>bytes</td>
<td>26 F0 90 A6 F0 26 57 4E AC A0 EC F8 68 E4 8D 21</td>
<td>UTF-16 string.</td>
</tr>
<tr>
<td>92–(A−1)</td>
<td>string</td>
<td>server name</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>char</td>
<td>allow players to harm other players</td>
<td>True if '1', false if '0'.</td>
</tr>
<tr>
<td>A+1</td>
<td>char</td>
<td>':'</td>
<td></td>
</tr>
<tr>
<td>A+2</td>
<td>char</td>
<td>allow new players</td>
<td>True if '1', false if '0'.</td>
</tr>
<tr>
<td>(A+3)−B</td>
<td>string</td>
<td>&quot;.<em>:.</em>:<em>:</em>:<em>:</em>:*:&quot;</td>
<td>Doesn’t change on server restart. Server signature?</td>
</tr>
<tr>
<td>(B+1)−</td>
<td>string</td>
<td>server description</td>
<td>UTF-16 string.</td>
</tr>
</tbody>
</table>

Approach:  Nmap Scripting Engine (NSE)

Script file:  freelancer-info.nse

Script description:
Detects the Freelancer game server (FLServer.exe) service by sending a status query UDP probe.

When run as a version detection script (−sV), the script will report on the server name, current number of players, maximum number of players, and whether it has a password set. When run explicitly (−script freelancer-info), the script will additionally report on the server description, whether players can harm other players, and whether new players are allowed.
Example output (full invocation with --script freelancer-info):

```
PORT    STATE     SERVICE REASON     VERSION
2302/udp open  freelancer udp-response Freelancer
   (name: Discovery Freelancer RP 24/7; players: 152/225; password: no)
   freelancer-info:
   |   server name: Discovery Freelancer RP 24/7
   |   server description: This is the official discovery freelancer RP server. To know more about the server, please visit www.discoverygc.com
   |   players: 152
   |   max. players: 225
   |   password: no
   |   allow players to harm other players: yes
   |_  allow new players: yes
```

The script also produces XML output when Nmap is given the --oX (XML output) switch. An example can be found in the script file (freelancer-info.nse).

**Approach:** nmap-service-probes entry

**nmap-service-probes file modifications:**

Add a new probe entry:
```
Probe UDP FreelancerStatus q|\x00\x02\xf1\x26\x01\x26\xf0\x90\xa6\xf0 \x26\x57\xe4\xac\xa0\xec\xf8\xe4\xe4\x21|
rarity 9
ports 2302

match freelancer m|^\x00\x03\xf1\x26.{88}(.*)\0\0(?:.*?:){5}(.*)\0\0$|s
   p/Freelancer/ i/name: $P(1); description: $P(2)/
```

**Solution description:**

This edition of the probe detects only the server name and description (which may get truncated if Nmap deems it too long for display). It can be useful when not using NSE or when using a stricter portrule in the NSE script that doesn’t necessarily match on all open ports, but only ones that are probably running the service. In the latter case this solution can act as "softmatch" that enables the NSE solution and is controllable by the rarity metric.

The same probe payload is sent as in the NSE solution, but most of the received information is ignored as it cannot be parsed with regular expressions.

**Example output:**

```
PORT    STATE     SERVICE VERSION
2302/udp open  freelancer Freelancer
   (name: MyServer; description: It rocks!)
```

**UDP payload (the nmap-payloads file entry):**
```
# Freelancer game server status query
# http://sourceforge.net/projects/gameq/
# (relevant files: games.ini, packets.ini, freelancer.php)
udp 2302 "\x00\x02\xf1\x26\x01\x26\xf0\x90\xa6\xf0\x26\x57\xe4\xac\xa0\xec\xf8\xe4\xe4\x21"
```
## 5. Summary

Several extensions to Nmap’s service detection have been developed and described along with the related network service protocols. A summary listing of developed solutions follows.

<table>
<thead>
<tr>
<th>Service</th>
<th>Detected versions</th>
<th>Version information</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Murmur UDP (voice)</td>
<td>1.2.0 - 1.2.4 RC1</td>
<td>version, user count, max. users, bandwidth/user version, user count, max. users, bandwidth/user version, name, uptime, auth. scheme, OS</td>
<td>Will likely detect versions 1.2.X.</td>
</tr>
<tr>
<td>Murmur TCP (control)</td>
<td>1.2.0 - 1.2.4 RC1</td>
<td>version, name, uptime, auth. scheme, OS</td>
<td>Indirect, detects the UDP service (same port number). Detailed server info if script is invoked explicitly.</td>
</tr>
<tr>
<td>Ventrilo UDP (status)</td>
<td>2.1.2 - 3.0.3</td>
<td>version, name, uptime, auth. scheme, OS</td>
<td>Indirect, detects the UDP service (same port number).</td>
</tr>
<tr>
<td>Ventrilo TCP (voice)</td>
<td>2.1.2 - 3.0.3</td>
<td>version, name, uptime, auth. scheme, OS</td>
<td></td>
</tr>
<tr>
<td>TeamSpeak 2 TCP (telnetd)</td>
<td>2.0.17.16 - 2.0.24.1</td>
<td>version, OS, license</td>
<td>Admin telnet daemon.</td>
</tr>
<tr>
<td>TeamSpeak 2 TCP (HTTP)</td>
<td>2.0.17.16 - 2.0.24.1</td>
<td>n/a</td>
<td>Admin web interface.</td>
</tr>
<tr>
<td>TeamSpeak 2 UDP (control/voice)</td>
<td>2.0.17.16 - 2.0.24.1</td>
<td>password(?), version, name, OS version, build id, OS</td>
<td>If it has a password, no other info can be had. Admin telnet daemon.</td>
</tr>
<tr>
<td>TeamSpeak 3 TCP (telnetd)</td>
<td>3.0.0 beta1 - 3.0.7.2</td>
<td>n/a</td>
<td>Sending and matching encrypted traffic. Game server status query protocol. Game server.</td>
</tr>
<tr>
<td>TeamSpeak 3 UDP (control/voice)</td>
<td>3.0.0 beta1 - 3.0.7.2</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>The All-Seeing Eye Protocol UDP (status)</td>
<td>n/a</td>
<td>game name, version, port, password(?) server name, description, num. players, max. players, password(?), PvP(?), new players(?)</td>
<td></td>
</tr>
<tr>
<td>Freelancer UDP (game/status)</td>
<td>1.0 - 1.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As new network services and protocols are constantly being introduced, there is always more work to be done on detecting them. Future work may also include developing NSE libraries that implement some commonly used network traffic encryption and decryption schemes not included in the currently available OpenSSL library wrapper. The NSE bin library used for packing and unpacking binary data could also do with some added functionality. Such improvements would allow for simpler script development with less focus on creating ad hoc cryptography and binary data handling solutions.
BIBLIOGRAPHY


Appendix A
SCTP INIT Ping (−PY) Patch

This patch fixes a simple coding error that prevented the SCTP INIT ping from working properly in Nmap versions 6.26 (r30897) and below. The error seems to have been present since r23778. The patch has been submitted to the Nmap developers mailing list, and was subsequently committed in r30898. It will be included in following Nmap releases.

Index: scan_engine.cc
===================================================================
--- scan_engine.cc (revision 30896)
+++ scan_engine.cc (working copy)
@@ -5317,7 +5317,7 @@
 /* Ensure the connection info matches. */
 if (probe->dport() != ntohs(sctp->sh_sport) ||
     probe->sport() != ntohs(sctp->sh_dport) ||
-    sockaddr_storage_cmp(&target_src, &hdr.dst) == 0)
+    sockaddr_storage_cmp(&target_src, &hdr.dst) != 0)
     continue;

     /* Sometimes we get false results when scanning localhost with
Network Host Discovery and Service Detection Tools

Abstract

Network host discovery and service detection are introduced alongside relevant terminology, issues, and motivation in pursuing their development. Nmap is chosen as the token tool through which these topics are further examined. Host discovery is discussed through Nmap’s host discovery techniques and examples depicting their use in some common scenarios. Service detection is also addressed in greater detail through Nmap’s implementation, including both the simple pattern matching and scripting methods. Basic approaches to developing service detection solutions for Nmap are introduced. Several extensions to Nmap’s service detection have been developed. These solutions and related service protocols are described.

Keywords: security, network, network service, host discovery, ping, service detection, version detection, Nmap

Alati za otkrivanje i prepoznavanje mrežnih usluga

Sažetak

Uvodi se u tematiku otkrivanja i prepoznavanja mrežnih usluga te se navode bitna terminologija, motivacija i problemi u razvoju rješenja. Nmap je odabran kao reprezentativan alat kroz čiju se implementaciju tema dalje proučava. Otkrivanje mrežnih usluga se razmatra kroz mogućnosti alata Nmap i primjere njihovog korištenja u nekim uobičajenim situacijama. Prepoznavanje mrežnih usluga se također detaljnije razmatra kroz implementaciju alata Nmap, uključujući jednostavnu metodu prepoznavanja uzorka i metodu korištenjem skriptnog jezika. Uvode se osnovni pristupi razvoju rješenja prepoznavanja mrežnih usluga za alat Nmap. Razvijeno je nekoliko dodataka prepoznavanju mrežnih usluga alata Nmap. Ta rješenja i protokoli povezanih usluga su opisani.

Ključne riječi: računalna sigurnost, računalna mreža, mrežna usluga, otkrivanje mrežnih usluga, prepoznavanje mrežnih usluga, Nmap