Global Information Assurance Certification Paper

A practical approach for defeating Nmap OS-Fingerprinting

Author: David Barroso Berrueta

GSEC Practical Version 1.4b

Remote OS Fingerprinting is becoming more and more important, not only for security pen-testers,but for the black-hat. Just because Nmap is getting popularity as the tool for guessing which OS is running in a remote system, some security tools have been developed to fake Nmap in its OS Fingerprinting purpose. This paper describes different solutions to defeat Nmap and behave like another chosen operating system, as well as a demonstration on how can be accomplished.

Table of Contents

1. Introduction

The purpose of this paper is to try to enumerate and briefly describe all applications and technics deployed for defeating Nmap OS Fingerprint, but in any case, security by obscurity is not good approach; it can be a good security measure but please take into account that is more important to have a tight security environment (patches, firewalls, ids, ...) than hiding your OS.

Learning which Operating System is running in a remote system can be very valuable for both the pen-tester and the black-hat. Suppose that they find an open port in their (approved or not) penetration; knowing the OS makes easier to find and execute an exploit against that service, because often an exploit is OS version specific, and an exploit for Sendmail running on HP-UX won't work for Sendmail running on AIX, or being more accurate, an AIX 4.3.3 exploit could not work in a system running 4.3.3 with the latest maintenance code applied. Fyodor (Nmap's author) has written a detailed article about remote OS Fingerprint, describing some different methods to successfully detect the remote OS, from the basic ones, to the more powerful ones.

In the beginning, guessing the remote OS was done grabbing the banner that a specific service was serving. For example, a typical telnet or FTP banner was always shown to the entire world, telling which OS was running, or if the banner has been changed or removed, some service commands could be executed to know the OS (remember the SYST in the FTP). Other basic ways to know the OS could be searching for HINFO entries in the DNS server, or trying to get information using snmp (lot of devices have enabled by default snmp access using the 'public' community string). Even searching for the target company jobs posting in the Internet, dumpster diving looking for OS manuals, or social engineering are valid methods for trying to know the remote OS.

Then, some more advanced network solutions were deployed, taking advantage of each OS vendor TCP/IP stack implementation. The idea is to send some crafted packets to the remote OS and wait for its answer. Those packets are "nasty" packets, crafted with uncommon TCP options or with 'impossible' options. Each OS has its own TCP/IP stack implementation, there isn't a common stack implementation for every OS and this issue allows to create a classification of different OS and versions according to their answers. Playing around with those tricky packets is how remote OS Fingerprinting tools work; some of them using the TCP/IP protocol, and others using the ICMP protocol.

There is a paper about 'Defeating TCP/IP Stack Fingerprinting' that describes in high level the design and implementation of a TCP/IP Stack fingerprint scrubber. That paper outlines why and how you can defeat TCP/IP OS Fingerprinting, so I am not going to talk too much about that; therefore I will focus on the solutions available out there.

---

2. Reasons to hide your OS to the entire world

Perhaps you are wondering why do you want to spend your precious time changing your Linux kernel to hide your real OS version against Nmap 'bad purposes' users. Maybe the following reasons can convince you:

- Revealing your OS makes things easier to find and successfully run an exploit against any of your devices.
- Having and unpatched or antique OS version is not very convenient for your company prestige. Imagine that your company is a bank and some users notice

that you are running an unpatched box. They won't trust you any longer! In addition, these kind of 'bad' news are always sent to the public opinion.
- Knowing your OS can also become more dangerous, because people can guess which applications are you running in that OS (data inference). For example if your system is a MS Windows, and you are running a database, it's highly likely that you are running MS-SQL.
- It could be convenient for other software companies, to offer you a new OS environment (because they know which you are running).
- And finally, privacy; nobody needs to know the systems you've got running.

---

3. Nmap

Nmap is one of such tools. It sends seven TCP/IP crafted packets (called tests) and waits for the answer. Results are checked against a database of known results (OS signatures database). This database is a text file that contains the result answered (signature) by each OS known. Thus, if the answer matches any of the entries in the database, we can guess that the remote OS is the same that the one in the database. Some Nmap packets are sent to an open port and the others to a closed port; depending on that results, the remote OS is guessed. A sample entry could be:

```
        /* OS Comment. Yes, we want to be a Sega Dreamcast console */
        Fingerprint Sega Dreamcast

        /* ISN predictibilty; TD: time dependant */
        TSeq(Class=TD%gcd=<780%SI=<14)

        /* Test 1 result: SYN packet with some options to an open port.
We got
        a SYN+ACK, acknowledgment seq +1, window size 0x1d4c, don't
fragment
        bit not activated, and only the MSS returned */
        T1(DF=N%W=1D4C%ACK=S++%Flags=AS%Ops=M)

        /* Test 2 result: Null packet with options to an open port. We
got a
        ACK+RST, acknowledgment seq, window size 0x0, don't fragment bit
not
        activated */
        T2(Resp=Y%DF=N%W=0%ACK=S%Flags=AR%Ops=)

        /* Test 3 result: SYN, FIN, URG, PSH with options to an open
        port. We got a SYN+ACK, acknowledgment seq +1, window size
0x1d4c,
        don't fragment bit not activated, and only the MSS returned */
        T3(Resp=Y%DF=N%W=1D4C%ACK=S++%Flags=AS%Ops=M)

        /* Test 4 result: ACK packet to an open port. We got a RST,
        acknowledgment seq, window size 0x0, don't fragment bit not
activated */
```

```
        T4(DF=N%W=0%ACK=S%Flags=R%Ops=)

        /* Test 5 result: SYN with options to a closed port. We got a
        ACK+RST, acknowledgment seq, window size 0x0, don't fragment bit
not
        activated */
        T5(DF=N%W=0%ACK=S%Flags=AR%Ops=)

        /* Test 6 result: ACK with options to a closed port. We got a
RST,
        acknowledgment seq, window size 0x0, don't fragment bit not
activated */
        T6(DF=N%W=0%ACK=S%Flags=R%Ops=)

        /* Test 7 result: FIN, PSH, URG with options to a closed port.
We
        got a ACK+RST, acknowledgment seq+1, window size 0x0, don't
fragment
        bit not activated */
        T7(DF=N%W=0%ACK=S++%Flags=AR%Ops=)

        /* Port unreachable message result. No response */
        PU(Resp=N)
```

Then, if we want to defeat Nmap and tell the attacker that we are running a different operating system, we only need to fake the responses to the Nmap tests. The solution that is going to describe is only valid for defeating Nmap and not other remote OS Fingerprinting tools. In the Conclusion section, other tools will be mentioned, as well as some recommendations for the pen-tester and/or the attacker.

---

4. Linux solutions

Methods to defeat Nmap OS Fingerprinting in Linux are written as kernel modules, or at least, as patches to the Linux kernel. The reason is that if the aim is to change Linux TCP/IP stack behavior, and if we want to achieve it, we need to do it in the kernel layer.

Three kernel module solutions are going to be described, all of them independent from the Linux kernel tree; you have to download them and patch your kernel to add the feature. The first one requires netfilter enabled in your kernel (what I think it's a must if you want to start to have a secure system), but the other two don't.

4.1. IP Personality

The first and probably, best option is IP Personality. It's netfilter module (then, only available for 2.4 Linux kernels) that allows us to change the IP stack behavior and

'personality', having multiple network personalities depending on parameters that you can specify as an iptables rule. Actually, we can change the following options:

- TCP Initial Sequence Number (ISN)
- TCP initial window size
- TCP options (their types, values and order in the packet)
- IP ID numbers
- answers to some pathological TCP packets
- answers to some UDP packets

An IP Personality overall summary is that we can change the way we answer to some packets, and we can specify which packets we want to answer in such way (it could be depending on the source ip address, the destination port, or, and that's we are going to use, those crafted packets coming from Nmap)

Installation is fairly straight forward and well explained in the INSTALL file provided by the package; for our test purposes, our test box is a stable Debian box running a 2.4.19 kernel. By default, IP Personality netfilter module is not available in latest kernel, so we need to patch our kernel sources. Patch for adding IP Personality feature to our netfilter core is available in the IP Personality site. We also need to patch the iptables command so that it can recognize our new feature available. Once the kernel is patched and compiled, we need to reboot our box just because the patch also modifies other netfilter files (the connection tracking).

Next step is include our iptables rules related to IP Personality in our working kernel. Before doing it, we run Nmap to check our current OS:

```
          # nmap (V. 3.10ALPHA4) scan initiated Wed Feb 19 20:26:52
2003 as: nmap -sS -O -oN nmap1.log 192.168.0.19
          Interesting ports on 192.168.0.19:
          (The 1597 ports scanned but not shown below are in state:
closed)
          Port       State       Service
          22/tcp     open        ssh
          25/tcp     open        smtp
          80/tcp     open        http
          143/tcp    open        imap2
          Remote operating system guess: Linux Kernel 2.4.0 - 2.5.20
          Uptime 106.832 days (since Tue Nov  5 00:29:33 2002)
          # Nmap run completed at Wed Feb 19 20:26:58 2003 -- 1 IP
address (1 host up) scanned in 7.957 seconds
```

Now, we can reboot to run our new patched kernel and add the iptables rules needed to fake Nmap OS guess:

```
          voodoo:~/ippersonality-20020819-
2.4.19/samples#/usr/local/sbin/iptables -t mangle -A PREROUTING -s
192.168.0.50 -d192.168.0.19 -j PERS --tweak dst --local --conf
dreamcast.confi
```

```
          voodoo:~/ippersonality-20020819-
2.4.19/samples#/usr/local/sbin/iptables -t mangle -A OUTPUT -s
192.168.0.19 -d192.168.0.50 -j PERS --tweak src --local --conf
dreamcast.conf
```

What we are doing with those filter rules is:

- The first one means that all packets coming from 192.168.0.50 (me) against 192.168.0.19 (server) have to be mangled and rewritten simulating a Dreamcast behavior. The PREROUTING chain is the one that can do that.
- The second one means that all packets coming from 192.168.0.19 (server) against 192.168.0.50 (me) have to be mangled and rewritten simulating a Dreamcast behavior. As is a packet going out the server, the OUTPUT chain is the responsible for that.

Checking our set-up:

```
          voodoo:~/ippersonality-20020819-
2.4.19/samples#/usr/local/sbin/iptables -L -t mangle
          Chain PREROUTING (policy ACCEPT)
          target     prot opt source               destination
          PERS       all      192.168.0.50      192.168.0.19
tweak:dst local id:Dreamcast
          Chain INPUT (policy ACCEPT)
          target     prot opt source               destination
          Chain FORWARD (policy ACCEPT)
          target     prot opt source               destination
          Chain OUTPUT (policy ACCEPT)
          target     prot opt source               destination
          PERS       all      192.168.0.19      192.168.0.50
tweak:src local id:Dreamcast
          Chain POSTROUTING (policy ACCEPT)
          target     prot opt source               destination
```

It's time to see if Nmap can report that we are still running a Linux kernel 2.4.0-2.5.20 or perhaps we can find out that our OS has changed:

```
          # nmap (V. 3.10ALPHA4) scan initiated Wed Feb 19 21:49:18
2003 as: nmap -sS -O -oN nmap2.log 192.168.0.19
          Interesting ports on 192.168.0.19:
          (The 1597 ports scanned but not shown below are in state:
closed)
          Port       State       Service
          22/tcp     open        ssh
          25/tcp     open        smtp
          80/tcp     open        http
          143/tcp    open        imap2
          Remote operating system guess: Sega Dreamcast
          # Nmap run completed at Wed Feb 19 21:49:23 2003 -- 1 IP
address (1 host up) scanned in 5.886 seconds
```

As you can see, we've fooled Nmap with our response. It's easy to choose the OS we want to 'run' in the Nmap OS fingerprint and tell IP Personality to behave like that chosen OS. Let's take a look to the dreamcast.conf file that we've specified when adding our iptables rules:

```
          /* Our new OS identification */
          id "Dreamcast";

          /* only incoming packets will be mangled and TCP window sizes
will not be changed*/
          tcp {
            incoming yes;
            outgoing no;
            max-window 32768;
          }

          /* We need to emulate the Dreamcast ISN time dependant
generator; this can be done with the fixed-inc generator and a small
increment */
          tcp_isn {
            type fixed-inc 2;
            initial-value random;
          }

          tcp_options {
            keep-unknown yes;
            keep-unused no;
            isolated-packets yes;
            code { copy(mss); }
          }

          /* now we have to follow nmap Dreamcast signature and answer
like a Dreamcast */
          tcp_decoy {
            code {
              if (option(mss)) { /* nmap has mss on all of its pkts */
                set(df, 0);
                if (listen) {
                  if (flags(syn&ece)) { /* nmap test 1 */
                    set(win, 0x1D4C);
                    set(ack, this + 1);
                    set(flags, ack|syn);
                    insert(mss, this+1);
                    reply;
                  }
                  if (flags(null)) { /* nmap test 2 */
                    set(win, 0);
                    set(ack, this);
                    set(flags, ack|rst);
                    reply;
                  }
                  if (flags(syn&fin&urg&push)) { /* nmap test 3 */
                    set(win, 0x1D4C);
                    set(ack, this + 1);
                    set(flags, ack|syn);
                    insert(mss, this+1);
```

```
                    reply;
                  }
                  if (ack(0) && flags(ack) && !flags(syn|push|urg|rst))
{ /* nmap test 4 */
                    set(win, 0);
                    set(ack, this);
                    set(flags, rst);
                    reply;
                  }
                } else {
                  set(win, 0);
                  if (flags(syn) && !flags(ack)) { /* nmap test 5 */
                    set(ack, this);
                    set(flags, ack|rst);
                    reply;
                  }
                  if (ack(0) && flags(ack) && !flags(syn|push|urg|rst))
{ /* nmap test 6 */
                    set(ack, this);
                    set(flags, rst);
                    reply;
                  }
                  if (flags(fin&push&urg)) { /* nmap test 7 */
                    set(ack, this + 1);
                    set(flags, ack|rst);
                    reply;
                  }
                }
              }
            }
          }

          /* No ICMP response for connections to closed UDP ports */
          udp_unreach {
            reply no;
            df no;
            max-len 56;
            tos 0;

            mangle-original {
              ip-len 32;
              ip-id same;
              ip-csum zero;
              udp-len 308;
              udp-csum same;
              udp-data same;
            }
          }
```

IP Personality is even more powerful. You can set up a Linux firewall/router that will change the answer of the hosts behind it. All your hosts protected by that Linux router can appear to be Sega Dreamcast consoles to any attacker!

There is also a great Nmap patch in the same site, named osdet, that allows us to OS Fingerprint an ip address (using Nmap engine), but with the fancy add-on that we can see the packets that are sent and their answer in tcpdump output format. Sometimes it's very helpful and easier to understand the OS Fingerprint technique watching the packets flowing on our screen (all Nmap tests and its answers).

4.2. Stealth patch

The solution that we are going to describe is the stealth patch, available from Security Technologies. It's available as a kernel modules for Linux kernels 2.2.x (and in a near future for 2.4.x) ,and as a kernel patch (without module support for Linux kernel 2.4.x). We're going to test the patch for the 2.4.19 kernel. When patched, two new options appear in our config file:

- IP: TCP Stack Options: option that we have to choose if want to use the stealth patch. If you select this option, it's enable by default when you boot your system. To disable them, you need to execute:

```
echo 0 > /proc/sys/net/ipv4/tcp_ignore_ack
echo 0 > /proc/sys/net/ipv4/tcp_ignore_bogus
echo 0 > /proc/sys/net/ipv4/tcp_ignore_synfin
```

- Log all dropped packets: logs all packets with bad options.

This patch simply discards the TCP/IP packets received with the following matches:

1. Packets with both SYN and FIN activated (tcp_ignore_synfin) (QueSO probe).
2. Bogus Packets: if the TCP header has the res1 bit active (one of the reserved bits, then it's a bogus packet) or it does not have any of the following activated: ACK, SYN, RST, FIN (Nmap test 2).
3. Packets with FIN, PUSH and URG activated (Nmap test 7).

This is a simpler solution than the one described earlier. We cannot behave like any other Operating System, we just silently drop all 'strange' packets that are supposed to be destinated to guess our OS, and hope that it will be enough to fool our attacker, or at least, make the things harder. These kernel modifications are easy to understand, and it would be relatively easy to add our own homemade 'bad packets' detection.

4.3. Fingerprint F**ker

Fingerprint F**ker is a kernel module available for Linux kernel 2.2.x which also can hide your OS and behave like another. It's a kernel module which accepts parameters

from the command line to configure the answer. By default, it simulates a VAX. There is also another file, called fing_parses.c, which parses a Nmap signature file and loads the Fingerprint F**ker module with the right parameters (when executing fing_parses, you have to specify which OS you want to emulate). It also waits for receiving a Nmap bogus packet, and then answers as you have configured. As far I've seen, only some Nmap tests are treated (T1, T2 and T7).

The code is not very stable. I loaded the module and in a few moments my Linux box got frozen.

4.4. IPlog

IPlog is a TCP/IP logger that also detects some scans (XMAS, FIN, SYN, ...). For our purposes, it has an option (-z) that allows to fool Nmap queries, and, although we can't behave as other OS, we can completely fool Nmap when guessing remotely our OS.

Now it's time to run IPlog to check the results:

```
voodoo:~#iplog -o -L -z -i eth0
```

The options are the following: -o (don't fork and stay in foreground), -L (results to stdout), -z (fool Nmap), -i eth0 (listen to eth0).If I run a Nmap against the box, iplog starts to write a lot of information to stdout, about all connections made, and even which type of scanning is being performed; I've included only the relevant information about Nmap OS Fingerprinting in the iplog's output:

```
        Feb 20 13:20:54 TCP: SYN scan detected [ports
10082,1430,770,815,440,86,848,797,560,5998,...] from 192.168.0.50 [port
49047]
       Feb 20 13:20:56 TCP: Bogus TCP flags set by 192.168.0.50:49054
(dest port 22)
       Feb 20 13:20:56 UDP: dgram to port 1 from 192.168.0.50:49047
(300 data bytes)
       Feb 20 13:20:56 ICMP: 192.168.0.50: port is unreachable to (udp:
dest port 1, source port 49047)
       Feb 20 13:20:58 UDP: dgram to port 1 from 192.168.0.50:49047
(300 data bytes)
       Feb 20 13:20:58 ICMP: 192.168.0.50: port is unreachable to (udp:
dest port 1, source port 49047)
       Feb 20 13:21:01 UDP: dgram to port 1 from 192.168.0.50:49047
(300 data bytes)
       Feb 20 13:21:01 ICMP: 192.168.0.50: port is unreachable to (udp:
dest port 1, source port 49047)
       Feb 20 13:21:04 TCP: Xmas scan detected [ports
1,9,49055,49056,49054] from 192.168.0.50 [ports 49060,49056,49054,9]
       Feb 20 13:21:05 UDP: dgram to port 1 from 192.168.0.50:49047
(300 data bytes)
       Feb 20 13:21:05 ICMP: 192.168.0.50: port is unreachable to (udp:
```

```
dest port 1, source port 49047)
        Feb 20 13:21:12 TCP: null scan detected [ports
9,49056,49060,49054] from 192.168.0.50 [ports
49055,9,1,49056,49054,...]
        Feb 20 13:21:13 TCP: FIN scan detected [ports 49060,49054,9,1]
from 192.168.0.50 [ports 1,9,49055,49056,49054,...]
        Feb 20 13:21:56 TCP: SYN scan mode expired for 192.168.0.50 -
received a total of 1647 packets (33440 bytes).
        Feb 20 13:21:56 TCP: Xmas scan mode expired for 192.168.0.50 -
received a total       of 33812 packets (676300 bytes).
        Feb 20 13:22:03 TCP: null scan mode expired for 192.168.0.50 -
received a total of 16462 packets (329300 bytes).
        Feb 20 13:22:04 TCP: FIN scan mode expired for 192.168.0.50 -
received a total of 16343 packets (326860 bytes)
```

Iplog does recognize the bogus TCP flags, null packet, ... every Nmap OS Fingerprint attempt. That's why it can act accordingly and send a fake answer to fool Nmap. Nmap output is the following:

```
        # nmap (V. 3.10ALPHA4) scan initiated Thu Feb 20 13:20:54 2003
as: nmap -vv -sS -O -oN nmap3.log 192.168.0.19
        Insufficient responses for TCP sequencing (1), OS detection may
be less accurate
        Insufficient responses for TCP sequencing (1), OS detection may
be less accurate
        Insufficient responses for TCP sequencing (1), OS detection may
be less accurate
        Interesting ports on voodoo (127.0.0.1):
        (The 1599 ports scanned but not shown below are in state:
closed)
        Port       State       Service
        22/tcp     open        ssh
        25/tcp     open        smtp
        80/tcp     open        http
        143/tcp    open        imap2
        No exact OS matches for host (If you know what OS is running on
it, see http://www.insecure.org/cgi-bin/nmap-submit.cgi).
        TCP/IP fingerprint:
        SInfo(V=3.10ALPHA4%P=i586-pc-linux-
gnu%D=2/20%Time=3E54C833%O=9%C=1)
        T1(Resp=Y%DF=Y%W=7FFF%ACK=S++%Flags=AS%Ops=MNNTNW)
        T2(Resp=Y%DF=N%W=0%ACK=O%Flags=BA%Ops=)
        T2(Resp=Y%DF=Y%W=100%ACK=O%Flags=BARF%Ops=)
        T2(Resp=Y%DF=Y%W=100%ACK=O%Flags=BPF%Ops=)
        T3(Resp=Y%DF=N%W=0%ACK=O%Flags=BA%Ops=)
        T4(Resp=Y%DF=Y%W=0%ACK=O%Flags=R%Ops=)
        T5(Resp=Y%DF=Y%W=0%ACK=S++%Flags=AR%Ops=)
        T6(Resp=Y%DF=Y%W=0%ACK=O%Flags=R%Ops=)
        T7(Resp=Y%DF=N%W=0%ACK=O%Flags=BA%Ops=)
        T7(Resp=Y%DF=Y%W=0%ACK=S++%Flags=AR%Ops=)
        PU(Resp=Y%DF=N%TOS=C0%IPLEN=164%RIPTL=148%RID=E%RIPCK=E%UCK=E%UL
EN=134%DAT=E)

        # Nmap run completed at Thu Feb 20 13:21:07 2003 -- 1 IP address
```

```
(1 host up) scanned in 13.633 seconds
```

As you can see, iplog answers to all the packets with specific options; we can have a look to iplog source code:

```
         file iplog_tcp.c, line 99:

         if (opt_enabled(FOOL_NMAP) &&
         ((tcp_flags & TH_BOG) || (tcp_flags == TH_PUSH) || (tcp_flags
== 0) ||
         ((tcp_flags & (TH_SYN | TH_FIN | TH_RST)) && (tcp_flags &
TH_URG)) ||
         ((tcp_flags & TH_SYN) && (tcp_flags & (TH_FIN | TH_RST)))))
```

That 'if' statement means that if we have executed iplog with the '-z' switch (fool Nmap), and the TCP header options are:

- bogus (use of the reserved bits), or
- only PUSH , or
- NULL (no options), or
- SYN+URG, FIN+URG, RST+URG, or
- SYN+FIN, SYN+RST

then it will create a new packet for answering with the options we want (some options depend on the machine time, for example DF, that's why sometimes is 1 and other 0, or the window size which is defined as current_time & 1).

Of course we could change the file iplog_tcp.c so that iplog always behave as a Sega Dreamcast for those nasty packets, but we do not have the flexilibity to have multiple personalities or specify that we want to behave as a Dreamcast only for a specific traffic or ip address. It's a good idea to answer in this way to abnormal packets, but it's better to have the control and be more granular.

---

5. *BSD solutions

5.1. Blackhole

Blackhole is a special option present in the *BSD kernel to control system behavior when someone is connecting to closed TCP or UDP ports. There are two options that we can change:

```
sysctl -w net.inet.tcp.blackhole=[0 | 1 | 2]
sysctl -w net.inet.udp.blackhole=[0 | 1]
```

The TCP blackhole behaves as following: if the value is 0, whenever a packet connects a TCP closed port, it returns a RST. If the value is 1, if a SYN packet connects a TCP closed port, it's dropped; and if the value is 2, if any packet tries to connect to a TCP closed port, it's dropped.

The UDP blackhole is similar; if the value is 0, any connection to an UDP closed port, returns an ICMP port unreachable; if the value is 1, it does not return the ICMP port unreachable.

If we enable these settings in paranoid mode, tests 5, 6, 7 and the unreachable port test won't work when running Nmap to remotely guess the OS, so we'll not be able to know the OS.

5.2. Fingerprint F**ker

There is also another Fingerprint fucker for the FreeBSD systems, written by Darren Reed, that simply rewrites the TCP/IP stack and sends packets with other settings (different window size, ttl, ...) trying to hide its real OS.

5.3. OpenBSD packet filter

The OpenBSD packet filter can also be configured to try to defeat remote OS Fingerprint. There are some options in the ip.conf configuration file (Traffic Normalization) where you can change some IP fields (DF bit, TTL, MSS, ID), as you can see in ip.conf's man page:

```
           no-df
           Clears the don't-fragment bit from a matching ip packet.

           min-ttl _number_
           Enforces a minimum ttl for matching ip packets.

           max-mss _number_
           Enforces a maximum mss for matching tcp packets.

           random-id
           Replaces the IP identification field with random values to
compen-
           sate for predictable values generated by many hosts.  This
option
           only applies to outgoing packets that are not fragmented
after the
           optional fragment reassembly.
```

5.4. FreeBSD TCP_DROP_SYNFIN

FreeBSD kernel has got a special option, TCP_DROP_SYNFIN, which actually drops all packets with the SYN and FIN flags activated (Nmap test #3 sends a SYN+FIN+PSH+URG TCP packet); this special option could be also a valid method for defeating Nmap when performing its tests (be sure to activate it at startup in `/etc/rc.conf`).

---

6. General solutions

We saw when talking about IP Personality, that we could set up a linux router protecting our internal network, and that router could fool Nmap and other OS Fingerprinting tools when trying to remotely guess our internal network hosts' OS. If we haven't got a linux box, but we've got a Checkpoint FW-1, then we can do something similar because of the fw-1 INSPECT language. Using this language, it's easy to create your own 'packet inspector' for the packets that are going through your fw-1. There is a reference in the FW-1 mailing list describing a fw-1 service to manage those bogus packets:

---

7. More things to play with

Next solution won't allow us to hide or change our OS, but we'll be able to create as many virtual devices as we want with every valid Operating System you can imagine. This idea is being applied to the honeypots field, just because you can create a entire C class virtual network with lots of different OS flying around; the black-hat can be easily attracted by all those boxes running so many vulnerable services...It could be an attacker's heaven.

Honeypots in general, and this approach in particular, can be highly recommended not only for learning the black-hat tools and tactics, but for also divert attackers to your honeynet and not your production boxes. It can also make attackers think that you have an entire farm of a specific OS (the virtual one) and hide your real OS.

The package I'm going to briefly describe is honeyd, from Niels Provos. One of its greatest feature is that we can give each virtual device a specific OS personality. That personality is also fed by a standard nmap fingerprinting file, allowing us to become the OS we want. I'm not going to deeply describe this great tool, I'm only going to run the sample config file to demonstrate what it can do.

After installing it, there is a file which name is config.localhost with a lot of virtual devices configured in. For instance, if we get the device 10.0.0.1 definition:

```
route entry 10.0.0.1
route 10.0.0.1 link 10.0.0.0/24
```

```
[snip]
create routerone
set routerone personality "Cisco 7206 running IOS 11.1(24)"
set routerone default tcp action reset
add routerone tcp port 23 "router-telnet.pl"
[snip]
bind 10.0.0.1 routerone
[snip]
```

The high level explanation is that we have a device which ip address is 10.0.0.1, which will act as a Cisco 7206 running IOS 11.1(24), will reset all TCP connections except for connections to TCP port 23, because then the script router-telnet.pl (an emulation of the telnet daemon) will be executed. Well, let's run Nmap to check the OS running in the virtual device we've just created:

```
        # nmap (V. 3.10ALPHA4) scan initiated Thu Feb 20 16:17:44 2003
as: nmap -v -sS -oN nmap4.log -O 10.0.0.1
        Warning:  OS detection will be MUCH less reliable because we did
not find at least 1 open and 1 closed TCP port
        Interesting ports on 10.0.0.1:
        (The 1604 ports scanned but not shown below are in state:
filtered)
        Port       State       Service
        23/tcp     open        telnet
        Remote OS guesses: Cisco 7206 running IOS 11.1(24), Cisco 7206
(IOS 11.1(17)
        TCP Sequence Prediction: Class=random positive increments
        Difficulty=26314 (Worthy challenge)
        IPID Sequence Generation: Incremental

        # Nmap run completed at Thu Feb 20 16:20:42 2003 -- 1 IP address
(1 host up) scanned in 178.847 seconds
```

Again, when receiving Nmap bogus packets, honeyd answers with the device's personality we've chosen.

---

8. Conclusion

As stated in the IP Personality Limitations, changing your TCP/IP stack behavior when receiving Nmap bogus packets can create some troubles:

- some characteristics of OS are related to the host architecture (for instance page sizes on various CPU) which could lead to performance issues;
- some of these changes are more "political" choices of the IP stack (initial sequence numbers, window sizes, TCP options available...). Tweaking those allow to fool a scanner but might break regular connectivity by changing network

parameters. It could also make the system weaker if the emulated IP stack is not as strong as the initial one

In my opinion, it's pretty clear that we can't rely on only one security tool to remotely guess the Operating System. This paper has shown that it's very easy to fool Nmap (and other similar tools) when trying to profile a remote device, and that all those attempts can be properly logged by the remote administrator. To successfully remotely fingerprint an OS, all possible methods have to be gathered, starting with the simpler ones (banner grabbing, seeking for job posts, social engineering, ...) to the more complex ones (network fingerprinting). Every open service in a remote device has to be properly analyzed (banner, responses, behavior against attacks, DoS, known errors) and documented. It could be even possible (although not ethical) to run some tools that are known to crash specific OS versions (nuke, land, teardrop, ...) to clarify our guess.

Although all these solutions can be modified to detect and fool any other TCP/IP fingerprint tool (just knowing which packets are sent), it is highly recommended to use various tools when doing a remote OS Fingerprint. Nmap is perhaps the most widely used, but there is another tool that also works great: Xprobe. Xprobe also has got a signatures database (not updated very often), and the final guess it's a probabilistic guess (fuzzy matching) depending on various answers. One of xprobe's biggest problem is that it's rarely updated and it includes very few signatures. Nmap detects the remote OS if its tests' result is exactly equal to that OS signature in the database, but you can run Nmap with the switch (*--osscan_guess or --fuzzy*, and then it performs a more aggressive OS guess trying to find the best match available in its signatures database. There is a paper about Xprobe specification and usage where explains why its idea and implementation seems to be so good and so valid. I think it should be executed as a partner with Nmap, in case you can send both TCP and ICMP packets against the target host. Xprobe could be an effective tool in poorly secured networks, just because it sends ICMP timestamps and ICMP netmask requests, which can become suspicious for a network administrator. It does not sent bogus packets (uncommon TCP packets, since the reserved bits are rarely used) to detect the remote OS, it simply sends 'normal' traffic (ICMP) to the target host, making harder (if not impossible) to detect such packets (and therefore, act accordingly). This approach was first used in sing (Send Internet Nasty Garbage), which can be executed with the -O switch for doing OS Fingerprint (with the ICMP type you choose). It should be difficult to any IDS or network implementation to detect that those ICMP packets have other function, just because there are a huge number of those ICMP packets daily in our networks. On the other hand, ICMP now is getting blocked by default from almost every network environment, making impossible to do an ICMP OS remote fingerprint, but usually you can find some TCP services in those network environments and shoot your Nmap packets.

Just for being accurate, there is also another OS Fingerprint tool, named p0f; p0f listens to your network looking for the first SYN in a TCP connection and grabs that packet options. If it matches with its signature database, then we can guess the OS; again, changing any of the options that p0f is looking for, will completely fool it. If, for

instance, using IP Personality, we change every packet's window size, we can fake our responses and fool p0f.

Administrators should also carefully configure all their devices for not showing anything that can be used for identified them (banners, issue, common services open by default, ...) and run one of these tools that can log the OS Fingerprint attempts, because it's very likely that, those ip addresses wanting to know your OS, will be attacking your network in a short period of time. Besides, setting up a linux router using IP Personality and fooling everyone outside your network that you're using a different OS (with any of the options shown in this paper), could be a good security measure.

References

Matthew Smart, Robert Malan, and Farnan Jahanian, *Defeating TCP/IP Stack Fingerprinting*, Usenix Security Symposium 2000, URL: http://www.usenix.org/publications/library/proceedings/sec2000/smart.html .

Fyodor, *Remote OS Detection via TCP/IP Stack Fingerprinting*, June 11, 2002, URL: http://www.insecure.org/nmap/nmap-fingerprinting-article.html .

Gael Roualland and Jean-Marc Saffroy, *IP Personality*, URL: http://ippersonality.sourceforge.net/ .

Sean Trifero and Derek Callaway, *Stealth*, URL: http://www.innu.org/%7Esean/ .

Ryan McCabe, *IPlog*, URL: http://ojnk.sourceforge.net/stuff/iplog.readme .

Fusys and |CyRaX|,*Fingerprint F**ker*, URL: http://www.s0ftpj.org/tools/fingfuck.tgz .

FreeBSD, *Blackhole*, URL: http://www.gsp.com/cgi-bin/man.cgi?section=4&topic=blackhole .

Darren Reed, *Fingerprint F**ker*, URL: http://packetstormsecurity.org/UNIX/misc/bsdfpf.tar.gz .

OpenBSD, *ip.conf manual*, URL: http://www.openbsd.org/cgi-bin/man.cgi?query=pf.conf&sektion=5&arch=i386&apr .

FreeBSD, *Kernel Options*, URL: http://www.freebsd.org/doc/en_US.ISO8859-1/articles/dialup-firewall/kernel.html .

Alfredo Andrés Omella, *Trying to stop the security tool queSO*, URL: http://www.phoneboy.com/fom-serve/cache/82.html , October, 6th, 1998.

Niels Provos, *Honeyd - Network Rhapsody for You"*, URL: http://www.citi.umich.edu/u/provos/honeyd/ .

Gael Roualland and Jean-Marc Saffroy, *IP Personality Limitations*, URL: http://ippersonality.sourceforge.net/doc/ippersonality-en-2.html .

Fyodor Yarochkin and Ofir Arkin, *Xprobe*, URL: http://www.sys-security.com/html/projects/X.html .

Fyodor Yarochkin and Ofir Arkin, *Xprobe2 - A'Fuzzy' Approach to Remote Active Operating System Fingerprinting*, URL: http://www.sys-security.com/archive/papers/Xprobe2.pdf .

Alfredo Andrés Omella, *Sing*, URL: http://sing.sourceforge.net , October, 6th, 1998.

Michael Zalewski and William Stearns, *p0f*, URL: http://www.stearns.org/p0f/ .