Eric Kluthe  
CNA 432 Final Project

Thwarting Remote OS Fingerprinting/Detection

Introduction

With the advent of easy to use network scanners and scripting frameworks finding vulnerabilities in systems has been made relatively easy. System Administrators try their best to lock down ports and services but they often overlook the most commonly exploitable platform, the operating system. Several tools have been developed, including recent updates to Nmap, to scan not only for open ports but also to determine the operating system of the remote system.

The purpose of my research was to identify software packages that are effective in both fingerprinting operating systems and evading detection. After reviewing the available literature on the subject, I felt that it was a interesting area to write about since it is a subject area that is seldom written about and that has not been researched about in a while. While my project does not purpose any new techniques, it does evaluate the most current detection techniques against the most current fingerprinting evasion techniques. I feel that a current evaluation is necessary since no one has really officially evaluated this in years.

Review Of Literature

Techniques in OS-Fingerprinting by Nostromo, 2005
This paper defines OS fingerprinting and also describes OS fingerprinting techniques in depth using both manual and automated tools. The interesting section of this paper though is the last section, which describes tools and techniques available to thwart OS fingerprinting in Linux and BSD systems.
This document describes every technique and methodology used by the Nmap network scanner to fingerprint and detect remote OS’s.

A practical approach for defeating Nmap OS-fingerprinting by David Barroso Berrueta, http://voodoo.somoslopeor.com
Like the title says, this document does not present ways of fingerprinting, but is simply a practical guide to locking down systems so that your operating system is virtually impossible to determine remotely.

A Virtual Honeypot Framework by Niels Provos, Google Inc., niels@google.com
This paper describes how to setup and use honeypots and also has a section on how the honeypot framework thwarts OS Fingerprinting.

Frustrating OS Fingerprinting with Morph by Kathy Wang, synacklabs.net
This presentation overviews OS fingerprinting history, and then launches into a detailed description of Morph, a tool that is used to hide the operating system variables that allow for remote detection.

How Fingerprinting Works
On a general scale, fingerprinting works by sending various crafted packets with different flags and values set. The program then records the responses to those packets and forms a ‘fingerprint’ which it uses to match against a database of fingerprints and their matching operating systems.

A fine example of one of these tests is the ICMP echo test, which sends out two unique packets. The first one has the IP DF bit set, a type-of-service (TOS) byte value of zero, a code of nine (even though it should be zero), the sequence number 295, a random IP ID and ICMP request identifier, and 120 bytes of 0x00 for the data payload. The second ping query is similar, except a TOS of four (IP_TOS_RELIABILITY) is used, the code is zero, 150 bytes of data is sent, and the ICMP request ID and sequence numbers are incremented by one from the previous query values.

The resulting fingerprint from this test is as follows from our test machines:

Linux: IE(R=Y%DFI=N%T=40%CD=S)
Windows: IE(R=Y%DFI=N%TG=80%CD=Z)

R=Y means that both ICMP packets warranted a response. DFI is a test to see whether the Don’t Fragment bit was set in the response ICMP packets. T and TG are TTL and TTL guess. CD is the ICMP response code. As we can see the two fingerprints differ in the TTL and the ICMP responses. This is how we can tell the difference between two operating systems, and it is also how we can fool the detection system by changing the default TCP/IP variable values.

**Design and Methodology**
In order to complete the research I did the following:

1. Research the different software packages that are able to fingerprint operating systems.
2. Research the different software packages that are designed to protect against operating system fingerprinting.
3. Setup a virtual machine environment, which included one windows 2008 server, one Ubuntu 10.04 server, and one Backtrack 5 workstation.
4. Setup common services on the server machines. (See Appendix A for network/setup details).
5. Install the fingerprinting tools on the Backtrack 5 virtual machine.
6. Run the fingerprinting tools against the untouched server machines.
7. Implement firewall rules to limit footprint of the servers. (See Appendix B)
8. Run the fingerprinting tools against this variation of the servers.
9. Revert the snapshot back to the original untouched virtual machines.
10. Install fingerprinting evasion tools on the server machines.
11. Run the fingerprinting tools against this variation of the servers.
12. Revert the snapshot back to the original untouched virtual machines.
13. Use both firewall rules and fingerprinting evasion tools.
14. Run the fingerprinting tools against this variation of the servers.
15. Revert the snapshot back to the original untouched virtual machines.

The point of this methodology is to test my selected evasion techniques and tools alone and then in concert with one another. By doing this I can test the individual effectiveness of the techniques, and also see how well the fingerprinting tools stand up to a well defended endpoint.
The methodology of the analysis of the data was to compare the top results of the tool/technique to the results of the run against the clean unprotected machines.

**Results**

The first result of my research is that there are no up to date fingerprint obfuscation software packages available for any recent OS (any OS that was released in 3 years.) All of the windows software packages, which toy with certain registry values, no longer work on Windows 2008 server because they haven’t been upgraded in 5 years. The software packages available for Linux were not able to compile because the dependencies had matured so much that they no longer worked with the software. Since I had a pretty good idea of how the tools in question worked, I could easily mimic the effects of the software manually, see Appendix C for the scripts and registry values I used to do this.

Another result I quickly obtained was that many of the scanners no longer work since they were based on the linux 2.4 kernel, and every Linux distribution since 2008 has been using kernel version 2.6. This was not a problem as nmap and the SinFP scanners still worked and gave me adequate information regarding the fingerprint of the test machines.

When I ran Nmap and SinFP against the newly installed and configured test machines, they gave me the results that one would expect, one machine is running windows 2008 server and the other is running Linux kernel 2.6.19 - 2.6.35.

I then implemented firewalls (See appendix B) on the test machines. On both machine’s
nmap and SinFP were able to still detect the operating system perfectly. Nmap however reported that is was less sure of it’s guess and when looking at the fingerprint we can see that this is because nmap can no longer use ICMP detection methods.

Because the tools I expected to use were outdated or failed, I had to mimic their techniques by changing the values of certain tcp stack variables by hand, these include MTU, default TTL, and window size. (See Appendix C for more details.) When I changed these values on the Linux machine, Nmap totally failed at detecting the OS with it’s number one guess being the OS of a Linksys WRV54G WAP. SinFP because it relies heavily on TCP packet values, completely failed at detecting matching OS fingerprint. When I changed certain values in Windows, Nmap reported that their was no exact OS match, but Nmap’s first guess was Windows 2008 server. SinFP reacted much the same as it did with linux: it completely failed.

The most interesting results are when I used both firewall rules and modified TCP stack variables in concert with one another. In this test, I did not use SinFP as it had already failed in a previous test on both OS’s. On Linux, when I implemented both, Nmap reported no OS matches. On Windows, Nmap reported no exact OS matches and it also guessed Windows 2008 server, but it also guessed FreeBSD 6.3 with the same probability.

**Discussion**

It is clear from my research into the different free OS fingerprinting software packages, that interest in OS detection has dipped as their are really only two decent software packages that fingerprint and run on a recent version of the Linux operating system. Also, protecting against such detection has run into the same problem, which is a lack of updated tools to easily protect a machine from fingerprint detection.
The outcome of my research is that it is really easy to detect operating systems and pretty hard to defend against detection, with the current tool set. Nmap, which is used by almost everyone in the networking world, is really good at detecting operating systems and uses a variety of tests to do so. But what is the defense against Nmap’s fingerprinting utility? A whole host of tools that on average, were made in 2003. I would say that it is time for newer and better fingerprint evasion tools to be made, since the current tools are basically useless in 2011.

Works Cited


Appendix A

Network Setup/Configuration

I setup the test machines on a VMware Xen Server, so the tests were completely virtualized but that should have no effect on the test themselves. I first setup my attack machine with the following packages:
Backtrack 5 Linux
- Updated nmap to 5.61
- Downloaded and compiled SinFP

I then setup two test machines, since I wanted to get data on machines that you would see running in a typical corporate environment I chose Ubuntu 10.04 server, and Windows 2008 server. I setup the servers with the following services.

Ubuntu 10.04 Server
- Bind9 DNS Server (port 53)
- OpenSSH (port 22)
- Lighttp Web Server (port 80)
- NFS services (port 111 and 2049)

Windows 2008 Server
- Active Directory (Multiple ports)
- DNS (port 53)

Before OS fingerprinting, I tested all services for proper operation.

Appendix B

Firewall Rules

My methodology when setting up and creating these rules was that if these were production servers the only thing that would be exposed (the attack surface) would be public services such as DNS, and HTTP. Most production servers also don’t allow ICMP, so I firewalled that off as well.

On the Windows box I used Windows Firewall and reconfigured it so that the only service open
to the outside was DNS.

On the Ubuntu box I used iptables with the following script:

```bash
# Auto DROP everything incoming, accept everything out.
#
#iptables -P INPUT DROP
#iptables -P FORWARD DROP
#iptables -P OUTPUT ACCEPT

# Flush (-F) all specific rules
#iptables -F INPUT
#iptables -F FORWARD
#iptables -F OUTPUT
#iptables -F -t nat

# Permit packets in to the machine itself that are part of existing and related connections.
#iptables -A INPUT -i eth0 -m state --state ESTABLISHED,RELATED -j ACCEPT

# Accept all tcp SYN packets for protocols HTTP and SSH:
#iptables -A INPUT -i eth0 -p tcp -s 0/0 -d 0/0 --destination-port 22 --syn -j ACCEPT
iptables -A INPUT -i eth0 -p tcp -s 0/0 -d 0/0 --destination-port 80 --syn -j ACCEPT

# NFS requirements to the internal network
#iptables -A INPUT -i eth0 -p tcp -s 172.20.240.0/24 -d 0/0 --destination-port 111 --syn -j ACCEPT
iptables -A INPUT -i eth0 -p tcp -s 172.20.240.0/24 -d 0/0 --destination-port 2049 --syn -j ACCEPT

# When we run a DNS server, we have to accept UDP from anywhere to port 53
#iptables -A INPUT -i eth0 -p udp -s 0/0 -d 0/0 --destination-port 53 -j ACCEPT
```

**Appendix C**

**TCP Value Tweaking**

Tweaking the TCP/IP values on the Ubuntu box was admittedly easy I produced the following script to change the TCP window size and the MTU

```bash
sudo sysctl -w net.ipv4.tcp_rmem="4096 390000 1870000" net.ipv4.tcp_wmem="4096 390000 1870000"
net.ipv4.tcp_no_metrics_save=1 net.ipv4.tcp Moderate_rcvbuf=1

sudo ifconfig eth0 mtu 1000
```
I also changed the variable `/proc/sys/net/ipv4/ip_default_ttl` to 32.

On the windows box it was considerably more difficult to change the values and I was unable to change the TCP window size at all. I changed the following registry keys below:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters
```

Keys:

- `DefaultTTL` set to 32
- MTU set to 1000