SANS

www.sans.org

SECURITY 560
NETWORK PENETRATION TESTING AND ETHICAL HACKING

560.2

Scanning

The right security training for your staff, at the right time, in the right location.

Copyright © 2011, The SANS Institute. All rights reserved. The entire contents of this publication are the property of the SANS Institute.

IMPORTANT-READ CAREFULLY:

This Courseware License Agreement ("CLA") is a legal agreement between you (either an individual or a single entity; henceforth User) and the SANS Institute for the personal, non-transferable use of this courseware. User agrees that the CLA is the complete and exclusive statement of agreement between The SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA. If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this courseware. BY ACCEPTING THIS COURSEWARE YOU AGREE TO BE BOUND BY THE TERMS OF THIS CLA. IF YOU DO NOT AGREE YOU MAY RETURN IT TO THE SANS INSTITUTE FOR A FULL REFUND, IF APPLICABLE. The SANS Institute hereby grants User a non-exclusive license to use the material contained in this courseware subject to the terms of this agreement. User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of this publication in any medium whether printed, electronic or otherwise, for any purpose without the express written consent of the SANS Institute. Additionally, user may not sell, rent, lease, trade, or otherwise transfer the courseware in any way, shape, or form without the express written consent of the SANS Institute.

The SANS Institute reserves the right to terminate the above lease at any time. Upon termination of the lease, user is obligated to return all materials covered by the lease within a reasonable amount of time.

V2011_0418

Network Penetration Testing and Ethical Hacking

Scanning

SANS Security 560.2

Copyright 2011, All Rights Reserved
Version 3Q11

Hello, and welcome back. Today's section is called 560.2, Scanning.

This component of the course focuses on the vital task of scanning a target environment, creating a comprehensive inventory of machines, and then evaluating those systems to find potential vulnerabilities. We'll look at some of the most useful scanning tools freely available today, experimenting with them in our hands-on lab. Because vulnerability-scanning tools inevitably give us false positives, we'll conduct an exercise on false-positive reduction, analyzing several methods for getting inside of what our tools are telling us to ensure the veracity of our findings. Our hands-on exercises include the creative use of packet crafting to measure the fine-grained behavior of target machines, all while watching the action from a custom-configured sniffer. We also look at some of the late-breaking features of popular tools, including the latest Nmap Scripting Engine capabilities. And, we'll perform vulnerability scans, looking at the fine-grained configuration options of Nessus.

Without further ado, let's begin.

560.2 Table of Contents

Slide #

- Scanning Goals and Types 3
- Overall Scanning Tips 7
- Sniffing with tcpdump 14
- Scapy Packet Manipulation 20
  - **Scapy/tcpdump Exercise 42**
- Network Tracing, Port Scanning and Nmap 52
  - **Nmap Exercise 96**
- OS Fingerprinting 105
- Version Scanning 110
  - **Nmap –O –sV and Amap Exercise 114**
- Vulnerability Scanning 126
  - Nmap Scripting Engine 130
  - **NSE Exercise 135**
  - Nessus 146
  - **Nessus Exercise 154**
  - Other Vulnerability Scanners 166
- Enumerating Users 168
  - **Enumerating Users Exercise 176**
- Netcat for the Pen Tester 181
  - **Netcat Exercise 190**
- Optional Appendix: Hping Packet Crafting 198

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 2

This slide is a table of contents. Note that exercises are in bold face, so you can more easily find and refer to them.

Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

- ***Scanning Goals and Types***
- Overall Scanning Tips
  - Sniffing with tcpdump
- Network Sweeping with Scapy
  - Scapy/tcpdump Exercise
- Network Tracing
- Port Scanning
  - Nmap
  - Nmap Exercise
- OS Fingerprinting
- Version Scanning
  - Nmap -O -sV and Amap Exercise
- Vulnerability Scanning
  - Nmap Scripting Engine
  - NSE Exercise
  - Nessus
  - Nessus Exercise
  - Other Vuln Scanners
- Enumerating Users
  - Enumerating Exercise
- Netcat for the Pen Tester
  - Netcat Exercise

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 3

We'll start this section by discussing the goal of scanning and the different kinds of scans. We'll then proceed to go over some tips to help improve the effectiveness of your scans and analysis of the results. We then proceed through various scan types, including network sweeps, port scanning, and version scanning, culminating with an analysis of vulnerability scanning.

Goals of Scanning Phase

- Overall: Learn more about targets and find openings by interacting with the target environment
  - Determine network addresses of live hosts, firewalls, routers, etc. in the network
  - Determine network topology of target environment
  - Determine operating system types of discovered hosts
  - Determine open ports and network services in target environment
  - Determine lists of potential vulnerabilities
  - Do these in a manner that minimizes risk of impairing host or service

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 4

The overarching goal of the scanning phase is to learn more about the target environment and find openings by directly interacting with the target systems. Particular objectives under this goal include determining the addresses used by systems on the target environment, including hosts (servers and clients), network equipment (firewalls, routers, switches), and other devices. We also want to learn the topology of the target environment, creating a diagram that shows how various systems interconnect: in effect drawing a network map. From this map, we can plan further attacks with more confidence.

We also want to determine the operating system types of our target machines, so that we can tailor follow-up activity (including exploitation) based on vulnerabilities associated with those kinds of machines.

Next, we want a list of listening TCP and UDP ports on the target systems because each open port offers a potential avenue for compromise. In addition to determining which ports are open, we also want to verify which service is listening on each port and the version of the given application or application-level protocol (e.g., HTTP version, SMTP version, SSH protocol version) that it speaks.

We then want a list of potential vulnerabilities, which may be determined from the version numbers determined earlier or based on the behavior of the target system in light of certain kinds of network interactions.

We want to do all of the above in a manner that minimizes the chance of damaging the target machine(s), although there is always a possibility that our interactions could cause a target system or service to slow down or crash.

Scan Types

- Network sweeping:
  - Send a series of probe packets to identify live hosts at IP addresses in the target network
- Network tracing:
  - Determine network topology and draw a map
- Port scanning:
  - Determine listening TCP and UDP ports on target systems
- OS fingerprinting:
  - Determine target operating system type based on network behavior
- Version scanning:
  - Determine the version of services and protocols spoken by open TCP and UDP ports
- Vulnerability scanning:
  - Determine a list of potential vulnerabilities (misconfigurations, unpatched services, etc.) in the target environment

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 5

To achieve our goals, we'll perform several types of scans during the test, including:

Network sweeping: This kind of scan identifies which addresses are in use by sending probe packets to all network addresses in a target range. Wherever we receive a response during our network sweep, there is likely a system using that address.

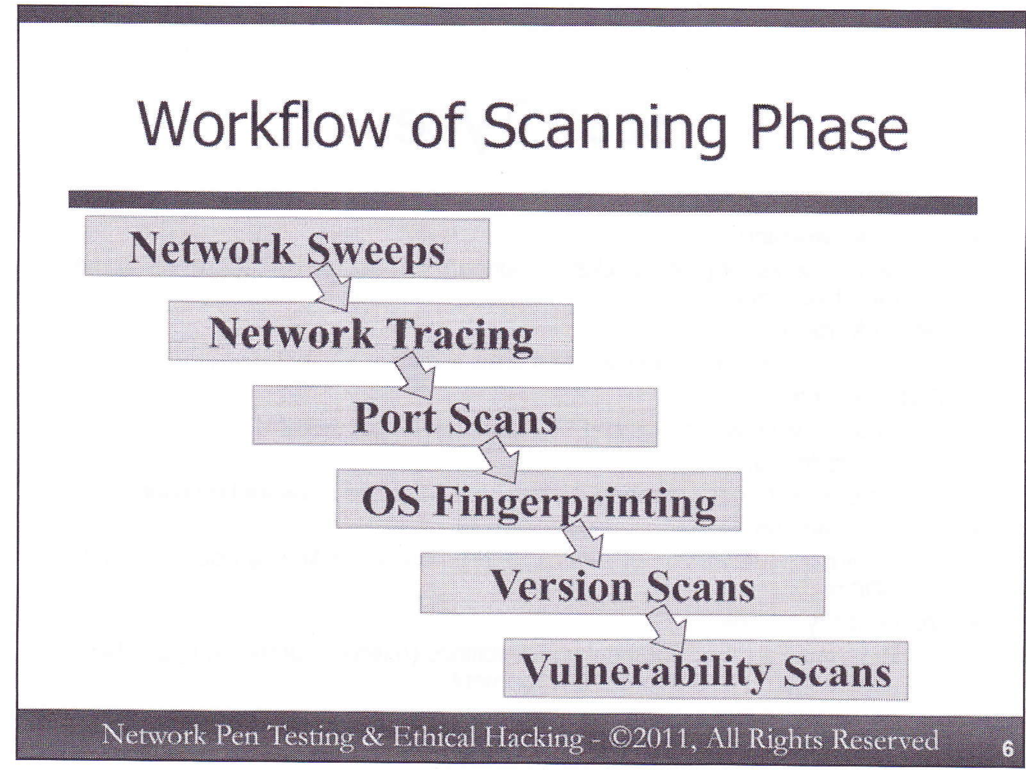
Network tracing: This is a closely related activity to network sweeping, in which we attempt to discern the topology of the target network by drawing a network map.

Port scanning: This kind of scan discerns potential openings in target machines by looking for listening TCP and UDP ports. Open ports indicate that a service is listening. If that service is vulnerable, we may have found an avenue to compromise the target.

OS fingerprinting: Different operating systems have different network behaviors that can be measured. By crafting specific test packets designed to measure the different behaviors, we can remotely determine the target's operating system type using a technique called "Active OS Fingerprinting". Alternatively, some sniffing tools include functionality to discern what kind of operating system formulated given packets in an entirely passive sense. Without sending any packets, but merely by receiving them, these "Passive OS Fingerprinting" tools can be helpful to a tester.

Version scanning: The tester needs to know which services are listening on which ports. Although many major services listen on well-known ports (e.g., sshd on TCP 22 and web servers on TCP 80), an administrator may put these services on alternative ports. By interacting with ports during a version scan, we can check which protocols they speak and possibly the version of the service listening on the given port.

Vulnerability scanning: In these scans, we measure whether the target machine has any one of thousands of potential vulnerabilities, which could include misconfigurations or unpatched services.



The workflow of a tester during the scanning phase generally progresses through the different kinds of scans indicated on this slide. We start with network sweeps to identify potential targets and the addresses they use. We then try to discern the network architecture to see how these targets are connected together. Next, we move on to port scans, identifying openings in the targets. We also perform OS fingerprinting to see what kinds of target machines we are testing. We then move on to version scanning to discern the services and protocols we face, ultimately culminating in a vulnerability scan. Each of these phases provides vital information we'll use in future phases of testing.

The order of these scans presented on the slide is very common among most testers, but it is not universal. Some testers may perform these scans out of order, or, given the scope of a test, may skip some steps altogether. For example, sometimes the scope of a test is merely to find unexpected machines in a target network range. Thus, the scope of the test may merely include network sweeps. Or, some testers may invert the port scan and OS fingerprinting phases of the workflow because they find that they can do more targeted port scans if they know the operating system type in advance.

While conducting scans, a tester should observe some particular tips to help ensure a successful scan, with results that provide insights into what is really happening on target systems. Let's go over some of those tips now.

Scanning Tip: Usually Scan Target IP Address... Not Name

- When scanning (and exploiting) systems, configure scanning tools to use target IP addresses or address ranges, not system names
  - For example, target 10.10.10.10, instead of www.target.tgt
  - If you attack based on name, round robin DNS may alter a target system while the test is occurring
  - That will corrupt results
    - Port scans with results from two targets merged into one
    - Exploiting service, try to connect to it, but it's now a different machine

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 8

When conducting scanning or exploitation of a target system, we recommend that you indicate the target network or machine in your tools based on its destination IP address and not its domain name. For example, if you want to conduct a port scan or launch an exploit against a machine called www.target.tgt with an IP address of 10.10.10.10, you should configure a target of 10.10.10.10, not www.target.tgt. You may think, "Well, DNS will just convert www.target.tgt to 10.10.10.10 for me, so what's the big problem?"

The concern is that many networks use DNS to perform load balancing and other traffic distribution schemes across multiple targets. So, if you attack a single domain name, www.target.tgt, you may actually be going after multiple hosts simultaneously without knowing it. This could lead to highly erroneous results. For example, in a port scan, you will see the merged results from multiple machines as though they were one box, likely missing some open ports. Or, if you exploit a target and create a listening port to connect to, when you connect to that port, there may be nothing there waiting for you because you exploited it on a different machine.

For these reasons, identify target systems for your tools based on their IP addresses.

There is still a possibility that the target environment will be load balancing the same IP address across multiple physical machines, which makes our jobs as testers harder.

Tip: Dealing with Very Large Scans

- Occasionally, testers are asked to scan a very large set of targets
  - Consider a request to scan 1,000 machines, all ports
    - 65,536 TCP ports and 65,536 UDP ports
  - If it took 1 second for each port (which is a low estimate), the scan alone would take:
    - 131 Million Seconds = 4.15 Years
    - Even if you scanned 100 ports at a time, it would still take 15 days of round the clock scanning
  - There must be better ways

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 9

Occasionally, penetration testers and ethical hackers are asked to conduct comprehensive scans of very large environments. A very expansive scope could mean a huge, almost impossibly large, amount of work, and the numbers grow more quickly than many people assume.

Consider this example: suppose an organization wants a full port scan of 1,000 machines. It may sound simple enough. The organization wants to know if there are any unexpected ports, such as those associated with backdoors or unauthorized software in their environment. And, 1,000 machines represents a mid-sized organization, not tiny by any means, but also not a giant enterprise either. But, let's look at the math.

If we take port 0 into account, there are 65,536 TCP and 65,536 UDP ports. For 1,000 target machines, that would be about 131 million ports to measure. Measuring one port per second would take 4.15 years. Now, depending on network performance and the behavior of target machines (whether they silently drop packets to closed ports or send TCP RESETs or ICMP port unreachable messages back), this 1 second may be way too short a timeframe for our estimate. Acting very optimistically and going with that 1 second estimate, if we could scan 100 ports at a time (perhaps using one system, or dividing the work among five or ten machines), we'd still chew up 15 days with round-the-clock scanning.

Clearly, there must be a better way.

Tip: Handling Large Scans by Limiting Scope (1)

- Numerous approaches to dealing with very large scans, some of which involve cutting down the number of ports measured

1) Sample a subset of target machines
   - Look for representative targets
   - Downside: How representative is the sample, really?

2) Sample target ports
   - Look for most interesting ports, such as TCP 21, 22, 23, 25, 80, 135, 137, 139, 443, 445, etc.
   - Downside: What about other ports?

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 10

We actually have many different approaches to dealing with requests for very large scans. The specific approach chosen for a given test will ultimately be a management decision, informed by the recommendations of the target organization's technical personnel and possibly the testers themselves.

One set of methods deals with cutting down the number of ports that need to be measured. We want to still have useful and meaningful results, but need to bring the amount of work down to a more manageable project and lower the budget. Some common and effective ways to do this include:

1) *Sample a subset of target machines:* Instead of scanning the entire target environment, some organizations are comfortable narrowing scope by selecting a representative sample of machines in the target environment. For example, instead of scanning all desktop machines, the testers could choose a dozen that have typical configurations representing the remainder of those systems. Likewise, instead of scanning every web server, three or four representative servers with common configurations representing other servers in the environment could be scanned. The downside, of course, is that these servers may not accurately represent the other systems.

2) *Sample a set of ports:* Instead of scanning every port, target organization personnel and the testers can agree upon a subset of the most interesting ports to measure. For example, a TCP port scan might focus on a dozen, a hundred, or a thousand ports, but not all 65,536, thereby reducing the scope of the work. For TCP, some of the most interesting ports include 21 (FTP), 22 (SSH), 23 (telnet), 25 (SMTP), 80 (HTTP), 135 (NetBIOS over TCP), 137 (again NetBIOS over TCP), 139 (yes, NetBIOS over TCP), 443 (HTTPS), 445 (SMB over TCP), and so on. The downside of this approach is that it only measures a set of ports, leaving the organization unaware of the status of other ports.

Tip: Handling Large Scans by Limiting Scope (2)

3) Review network firewall ruleset and measure only those ports that could reasonably make it through the firewall
   - In effect, this is part configuration review and part port scan
   - Overcomes the downsides of only sampling targets or sampling specific ports
     - By sampling ports on a more intelligent basis
   - Often a very effective approach
   - Downside: Doesn't measure potential firewall bugs
     - And requires more work from target organization personnel
     - Also, doesn't lend itself to a black-box approach
   - Combining method 3 for large-scale scan with method 1 (sampling a subset of targets for comprehensive scans) is a solid approach

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 11

A third approach to focusing the scope of scans in a large target environment is quite promising, and overcomes some of the downsides of the two other approaches we've discussed:

3) *Review network firewall rule set:* Target organization personnel could provide the testing team with a set of network firewall configuration rules. The testing team could then perform the scan on only those ports that would be allowed through the firewall rule set. In effect, this approach bundles a focused configuration review with a scan to help make the scan more efficient. While this approach is often quite effective, its limitation is that it requires target personnel to provide the testers with the configurations, making it more invasive, and it doesn't measure potential failures in the firewall technology itself. Furthermore, this method doesn't really work with a black-box penetration test, in which the testers are given as little information about the target organization as possible. Still, it is a good approach, and one that professional penetration testers and ethical hackers often rely upon.

Penetration testers can achieve a nice balance where you can conduct large-scale scanning while still verifying that a firewall faithfully implements its filter configuration by combining method number 3 and method number 1. For the large-scale scan, consult the firewall ruleset, and scan only those ports that the firewall is configured to allow through. But, for some sample of target machines, conduct entire scans of all ports. That way, you can verify that the firewall is actually filtering appropriately, while still touching a large number of target machines.

Tip: Handling Large Scans by Speeding Up (1)

- Other approaches deal with scanning all ports, but as quickly as possible:

4) Tweak firewall rules to send RESETs and ICMP Port Unreachable messages from closed ports
   - Several downsides:
     - Often undesirable because of changes to production environment
     - You've changed firewall rules so that you can measure their effectiveness?
     - Also, a large scan will still take a lot of time even with this approach

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 12

Instead of narrowing the scope of a project to deal with a very large scan, another set of options involves trying to speed up the scan itself, including:

4) *Alter firewall rules for closed ports:* Target organization personnel could alter firewall rules to send TCP RESET messages for closed TCP ports and ICMP Port Unreachable messages for closed UDP ports, which will prevent most scanning tools from waiting for a time-out to expire before moving to the next port. In fact, it's quite possible that the target organization's network firewalls already function in this way, helping to speed up a scan. While this technique can be helpful, it has some pretty big limitations. First, it may involve making changes to the firewall configuration of a target environment, something most organizations will not want to do for a penetration test. Secondly, even though such a configuration will speed up scans, it will still take time to measure each port. That time will quickly add up, and the scan will still likely have a very long duration.

Tip: Handling Large Scans by Speeding Up (2)

- A final approach for speeding up scans of large numbers of ports:

5) Use hyper-fast port scanning methods
   - Large number of scanning machines, and/or
   - Much faster packet send-rate from existing machine, lowering time outs (but may lose packets), and/or
   - Moving closer to targets, near high-bandwidth backbone, and/or
   - Very fast scanning tools, like those featured in Dan Kaminsky's ScanRand
   - Downside: You could create a denial of service attack
     - Be careful of network bottlenecks in attacking and target infrastructure!

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 13

There are more options we have for large port scans that involves speeding up the scan, which can be accomplished via several mechanisms:

5) *Send packets much more quickly:* The attacker could use hyper-fast scanning methods for measuring large numbers of ports quickly on the target environment.

   First off, the attacker could use a large number of scanning machines. Instead of one or two, the tester could rely on ten, twenty, or more machines distributed at various locations to conduct the scan.

Secondly, the tester could configure machines to send packets more quickly by lowering the timeout values for unresponsive ports, with some specialized configuration options that we'll cover for the Nmap port scanning tool later in this class. The tester has to be careful here, however, or he or she will miss important packets indicating the status of a port if the timeout is lowered too much.

Thirdly, we could move our testing machines closer to the target, near a point in the network with higher bandwidth.

Fourth, the attacker could use tools that conduct port scanning in untraditional ways to make them even faster, such as those embodied in Dan Kaminsky's ScanRand tool. We'll briefly discuss ScanRand later in the class, but, in essence, it allows for hyper-fast TCP port scanning by separating the sending and receiving mechanisms. The sending component sends TCP SYN packets as fast as possible, and the receiver component of the tool sniffs for SYN-ACK responses indicating that a port is open. Using this approach, the sender doesn't have to wait for a timeout to expire on the receiver before sending more packets.

Any of these mechanisms for option 5, however, consume a lot of bandwidth. Thus, testers have to be very careful of inadvertently causing a denial of service on the testing network and the target infrastructure. When using these approaches, the testers should carefully measure target systems to ensure that their legitimate services are still available to third-parties while the scan ensues.

Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

- Scanning Goals and Types
- Overall Scanning Tips
  - ***Sniffing with tcpdump***
- Network Sweeping with Scapy
  - Scapy/tcpdump Exercise
- Network Tracing
- Port Scanning
  - Nmap
  - Nmap Exercise
- OS Fingerprinting
- Version Scanning
  - Nmap -O -sV and Amap Exercise
- Vulnerability Scanning
  - Nmap Scripting Engine
  - NSE Exercise
  - Nessus
  - Nessus Exercise
  - Other Vuln Scanners
- Enumerating Users
  - Enumerating Exercise
- Netcat for the Pen Tester
  - Netcat Exercise

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 14

Our next set of tips will focus on sniffers, specifically the very useful tcpdump tool. Professional penetration testers and ethical hackers need to be familiar with sniffers for several reasons, including:

- To watch the packets generated by their scanning tools and other tools while they run so that they can make sure their tools appear to be operating properly.
- To gain insight into the behavior of target machines at a fine-grained level, perhaps getting more information from their sniffer than their particular scanning tool is capable of revealing.
- If the Rules of Engagement allow for it, to sniff useful and interesting information from the target environment, possibly including userIDs and passwords or other sensitive information passing by the machines that the tester has compromised during a project.

While sniffers can be useful for all of the above items, we need to understand how to configure a sniffer so that it focuses on specific packets that interest us during a test. This section provides tcpdump configuration advice specifically targeted at penetration testers and ethical hackers.

Scanning Tip: While Scanning, Run a Sniffer

- Whenever you run a scan, run a sniffer so that you can monitor network activity
  - You don't have to *capture* all packets in the file system
    - That would likely require huge storage space
  - Instead, display them on the screen so you can visualize what is happening in the scan
- Which sniffer to use?
  - Any sniffer that shows packet headers will do, but you want something small, flexible, and fast
  - tcpdump is ideal for this purpose

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 15

When running any kind of scan (ping sweep, port scan, vulnerability scan, and others), we recommend that you also run a sniffer on the testing machine that is running the scanning tool. The sniffer should be configured to display packets on the screen in real time, so you can have an at-a-glance view of activity from your system. That way, while the test is running, you can verify that the scanning tool is functioning properly. If the packet display stops, either the tool has finished or encountered some sort of problem.

You don't have to capture the packets into a packet capture file, because that file would grow immense over time given the sheer number of packets that are typically generated by a scan. Still, displaying the packets on standard output is quite useful.

Any sniffer will suffice, but a simple, flexible, low-cost, and fast tool is best. Tcpdump works really well as a sniffer to use while scanning. Let's explore it in more depth.

Scanning Tip: Use tcpdump

- Free, open source sniffer
  - www.tcpdump.org
  - Ported to Windows as WinDump at www.winpcap.org/windump/default.htm
- Supports various filtering rules
- While testing, you will likely have it display all packets leaving from and coming to your scanning machine
- But, for specific issues, you may need to focus on specific packets
  - We'll address some configuration options to do that

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 16

Tcpdump is a free, open source sniffer that is quite flexible and fast. It runs on most Linux and Unix variants (in fact, it is installed by default on many Linux distributions), and it has been ported to Windows as WinDump.

Tcpdump supports a variety of filters, with a powerful language for specifying individual filter types. We won't go over all of the filtering options in this class (they are covered in detail in SANS Intrusion Analysis course, SANS Security 503, which focuses on packet analysis). Instead, we'll go over the most common options of tcpdump used by penetration testers to view packets generated by their scanning and attack tools while a test is underway.

Tip: Helpful tcpdump Options to Use While Scanning

- Often, just running tcpdump with no special options while scanning provides the information you need
  $ **sudo tcpdump**
- But, you may want to rely on various options:
  -n: Use numbers instead of names for machines
  -nn: Use numbers instead of names for machines and ports
  -i [int]: Sniff on a particular interface (-D lists interfaces)
  -v: Be verbose (print TTL, IP ID, Total Length, IP options, etc.)
  -w: Dump packets to a file (use –r to read file later)
  -x: Print hex
  -X: Print hex and ASCII
  -A: Print ASCII (doesn't work in all versions... consider –X instead)
  -s [snaplen]: Snarf this many bytes from each packet, instead of the default
    - For older versions of tcpdump, default was to capture only first 68 bytes for most OSs...
    - For those versions of tcpdump, you had to specify -s 0 to get whole packets
    - On modern versions of tcpdump, default snaplength of zero grabs entire packets automatically

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 17

Most commonly, penetration testers simply run tcpdump without any special options, which by default will show all packets sent to and from the testing machine. The tool should be invoked with root-level privileges to make sure it can put the interface into promiscuous mode, grabbing all packets that pass by the network interface.

One relatively safe way to invoke a tool with root privileges is to use the sudo command, as follows:

```
$ sudo tcpdump
```

Then, provide the appropriate user password, and you are now sniffing.

Some useful command-line options for configuring tcpdump include:

-n: Use numbers for machines instead of the names available via /etc/hosts and DNS.

-nn: Use numbers for machines instead of the names available via /etc/hosts and DNS, and numbers for ports instead of names in /etc/services.

-i [interface]: Sniff on a specific network interface, such as the local loopback interface (usually lo) or the local ethernet (often eth0). For a list of interfaces, you can run "tcpdump –D".

-v: Print verbose output (shows TTL, IP ID, Total Length, and IP options). -vv shows more. -vvv shows even more.

-w: Write packets to a file (which can be read later with the –r option)

-x: Print out packet settings in hexadecimal form

-X: Print out packet settings in both hex and ASCII

-A: Print out packet settings in ASCII (This option doesn't work in all versions of tcpdump. If it doesn't work in a given instance, consider using the –X option to get ASCII and Hex).

-s [snaplength]: Grab this many bytes from each packet instead of the default. On modern versions of tcpdump, the default is to grab entire packets. With older versions of tcpdump, the default would only grab the first 68 bytes of each packet, unless you specified a snaplength of zero (-s 0) to indicate you wanted full packets, regardless of their length.

Tip: Helpful tcpdump Expressions to Use While Scanning

- Protocol:
  `ether, ip, ip6, arp, rarp, tcp, udp` – protocol type
- Type:
  `host [host]` – Only give me packets to or from that host
  `net [network]` – Only packets for a given network
  `port [portnum]` – Only packets for that port
  `portrange [start-end]` - Only packets in that range of ports
- Direction:
  `src` – Only give me packets from that host or port
  `dst` – Only give me packets to that host
- Use "and" or "or" to combine these together

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 18

Sometimes, however, you'll want to run tcpdump to focus on specific packets, such as those associated with certain protocols, ports, or addresses. You can use several primitives to formulate an expression, which will let you focus only on some specific packets.

Protocol primitives include `ether, ip, ip6, arp, rarp, tcp, and udp.`

Type primitives include `host, net, port, and portrange`.

Direction primitives allow you to specify whether you want packets from a given source (`src`) or destination (`dst`), which can be associated with a host, network, or port. Note that `src and dst and src or dest` are supported as well.

Note that these primitives can be combined to create more complex expressions, using the logical "and" and "or" terms. Also, there are additional primitives beyond the ones in this list. However, this list contains some of the most frequently used items by penetration testers.

Tip: Some Quick tcpdump Usage Examples

- Show TCP packets against target 10.10.10.10 in ASCII and Hex

```
# tcpdump -nnX tcp and dst 10.10.10.10
```

- Show all UDP packets from 10.10.10.10

```
# tcpdump -nn udp and src 10.10.10.10
```

- Show all TCP port 80 packets going to or from host 10.10.10.10

```
# tcpdump -nn tcp and port 80 and host
10.10.10.10
```

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved

Let's look at some examples of combinations of these primitives to form expressions.

If you want to view all TCP packets being sent to a target with IP address 10.10.10.10, with output that includes ASCII and Hex contents of packets, you could run:

```
# tcpdump -nnX tcp and dst 10.10.10.10
```

To see UDP packets with a source address of 10.10.10.10, you could run:

```
# tcpdump -nn udp and src 10.10.10.10
```

To see all packets associated with TCP port 80 going to or from host 10.10.10.10, you could run:

```
# tcpdump -nn tcp and port 80 and host 10.10.10.10
```

As we perform exercises over the next several days, feel free to formulate tcpdump expressions to focus on the most interesting packets associated with the scan.

Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

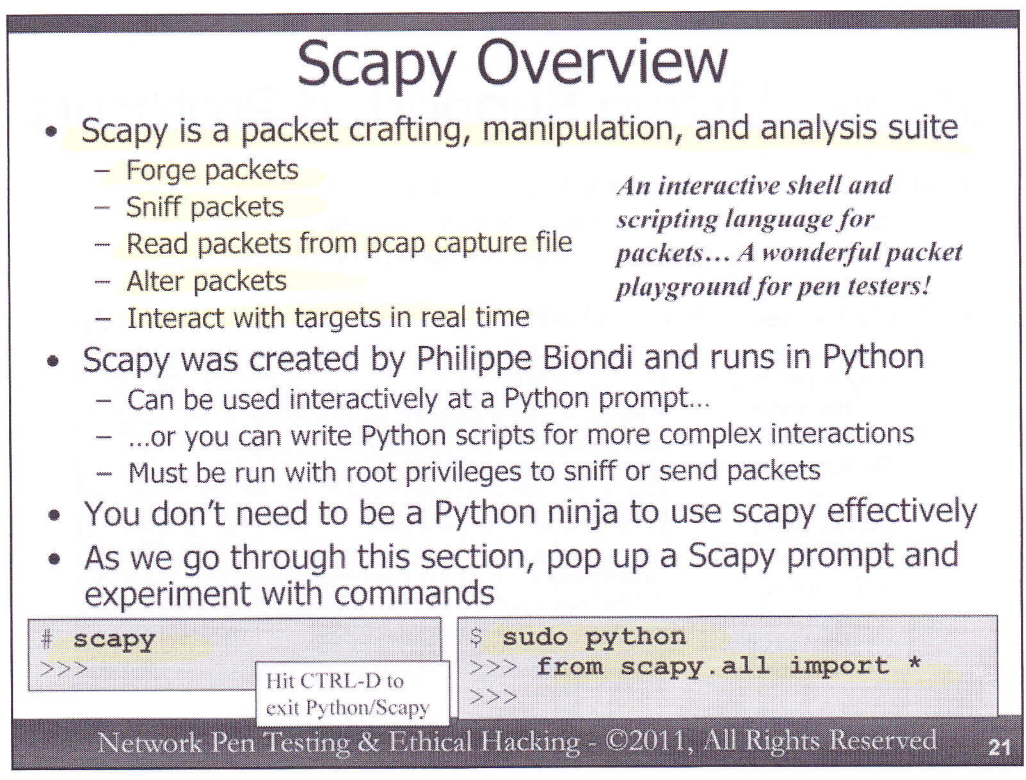
- Scanning Goals and Types
- Overall Scanning Tips
  - Sniffing with tcpdump
- ***Network Sweeping with Scapy***
  - Scapy/tcpdump Exercise
- Network Tracing
- Port Scanning
  - Nmap
  - Nmap Exercise
- OS Fingerprinting
- Version Scanning
  - Nmap -O -sV and Amap Exercise
- Vulnerability Scanning
  - Nmap Scripting Engine
  - NSE Exercise
  - Nessus
  - Nessus Exercise
  - Other Vuln Scanners
- Enumerating Users
  - Enumerating Exercise
- Netcat for the Pen Tester
  - Netcat Exercise

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 20

Now, we'll begin analyzing the different kinds of scans that testers perform, systematically stepping through the scanning workflow. We'll analyze each scan type, stopping periodically to do hands-on exercises for most of them.

We start with Network Sweeps. As we mentioned earlier, the purpose of this type of scan is to identify live hosts on the target network, determining their IP addresses so that we can later perform more detailed analysis. If we don't know it's there, we can't test it. Thus, network sweeps are a crucial part of our analysis.

One of the best tools available for formulating packets used in packet sweeps or many other forms of interaction with target systems is Scapy. Let's delve into this really remarkable tool, and see how we can use it to scan a target environment.



Scapy is a fantastic and flexible environment for creating and interacting with packets. It's incredibly full featured, allowing users to forge packets, sniff them, read them from a pcap-style packet capture file, edit packets, and interact with networked targets in real-time or via scripts. With all these capabilities, Scapy is an amazingly useful tool for penetration testers to use during scanning, exploitation, and researching target machines.

Created by Philippe Biondi, Scapy is an environment based on the Python programming language. Users invoke Scapy to get an interactive Python prompt (>>>). Or, you can invoke a Python script (typically with a filename suffix of .py) which can call Scapy features from the script itself. To craft packets with Scapy, you'll have to invoke it with UID 0 privileges on Linux or Unix. You can do this with "sudo scapy" followed by the user's password, as long as the user is allowed sudo privileges. Alternatively, you can invoke Python itself with UID 0 privileges, and then import all of the Scapy functionality as shown on the slide above.

It is important to note that you don't have to be a Python wizard to use Scapy, though. Even with just some fundamental knowledge, you can use Scapy to achieve great things.

In this section of the course, we'll be talking about many Scapy features, with numerous examples. As we go through this section, please pop up a command shell on your Linux virtual machine, and invoke Scapy. The easiest way to do this if you are logged in with root privileges (# prompt) is to simply run "scapy", which is included in the default PATH of the course Linux image:

```
# scapy
>>>
```

That >>> is the Python prompt, ready to run commands for us. To exit Scapy, hit CTRL-D.

Scapy - Listing Supported Protocols

- The ls() command by itself lists all protocols supported by Scapy
  - ARP, IP, IPv6, TCP, UDP, ICMP, and numerous app-layer
  - Protocol names tend to be all cap (but not always... for creating Ethernet frames, use Ether)
- To see the fields you can set within a given protocol, run ls([PROTO])
  - Field name, data type, and default value are shown
  - Default in parens
  - Defaults are usually quite reasonable
  - TCP defaults:
    - sport 20 (ftp-data)
    - dport 80 (http)
    - flags SYN

```
>>> ls(TCP)
sport      : ShortEnumField      = (20)
dport      : ShortEnumField      = (80)
seq        : IntField            = (0)
ack        : IntField            = (0)
dataofs    : BitField            = (None)
reserved   : BitField            = (0)
flags      : FlagsField          = (2)
window     : ShortField          = (8192)
chksum     : XShortField         = (None)
urgptr     : ShortField          = (0)
options    : TCPOptionsField     = ({})
```

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved

Let's jump right into Scapy by looking at the different protocols it supports. We can do this by running the ls() function:

```
>>> ls()
```

Here, we can see the over 300 different protocols Scapy supports, including application-layer protocols like HTTP, transport protocols such as TCP and UDP, Layer-3 protocols including IP (that's version 4) and IPv6, and data-link layer protocols such as Ether (which generates Ethernet frames). Most of he protocols are in all caps, with a few exceptions (such as IPv6 and Ether).

To see the fields available for us to interact with in a given protocol, we can run the ls() command on a specific protocol. For example, we could run:

```
>>> ls(TCP)
```

Here, we can see the various fields of the protocol (sport for source port, flags for the TCP Control Bits, and more), the data type of each field, and the default value Scapy will assign included in parentheses.

The default values assigned to most protocols are pretty reasonable, and of course we can change the packets away from their default field values to anything we want. For TCP, the default source port is 20 (which is typically associated with ftp-data), the destination port is 80 (HTTP, of course), and the TCP SYN Control Bit is set.

Scapy - Listing Commands

- The lsc() command shows all functions supported by scapy

```
>>> lsc()
arpcachepoison       : Poison target's cache
arping               : Send ARP who-has requests
corrupt_bits         : Flip a given percentage or number of bits
fragment             : Fragment a big IP datagram
fuzz                 : Transform a layer into a fuzzy layer by
replacing some default values by random objects
getmacbyip           : Return MAC address corresponding to a given
IP address
is_promisc           : Try to guess if target is in Promisc mode.
send                 : Send packets at layer 3
sendp                : Send packets at layer 2
```

- To get help with any function, run:

  `>>> help([function])`

  Hit Q to leave help

```
>>> help(arpcachepoison)
Help on function arpcachepoison in module scapy.layers.ls:
arpcachepoison(target, victim, interval=60)
    Poison target's cache with (your MAC,victim's IP) couple
```

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 23

In addition to its great protocol support revealed by the ls() function, Scapy also includes numerous functions, which can be inspected by running lsc(). Go ahead and run it on your system:

```
>>> lsc()
```

Here, we can see the numerous features of Scapy. Notice that there are numerous attack techniques embedded in Scapy itself, such as arpcachepoison, a topic we'll touch on in 560.4. There are also techniques for sending packets (send, sendp, sr, and sr1, among other things, fall into this category). There are functions for fuzzing, fragmenting, and much more.

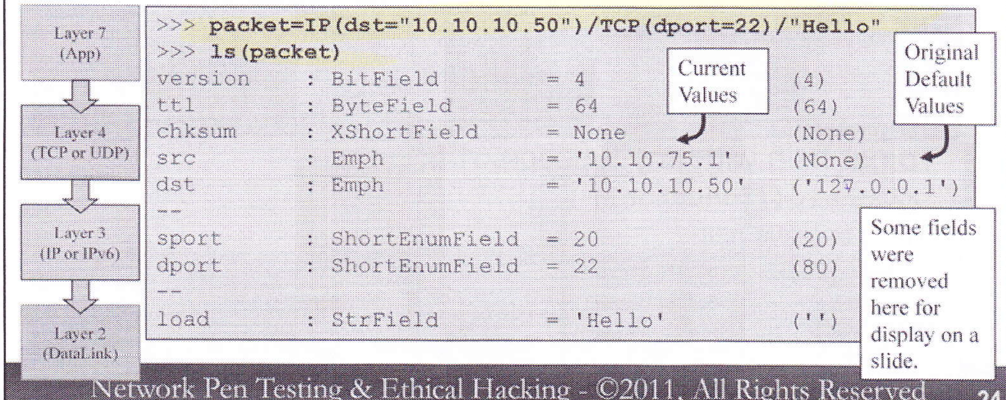
To learn more about a given function, as well as the arguments it supports, you can run "help ([function])", as in:

```
>>> help(arpcachepoison)
```

Here, we can see that the arpcachepoison function supports a target, a victim, and an interval (which defaults to 60 seconds) as arguments. These arguments are set by calling the function with variable=value pairs, as in arpcachepoison(target=10.1.1.1, victim=10.1.1.3, interval=2).

To get out of the help screen, simply hit the Q key.

- Packets are constructed by layers, simply calling the appropriate protocol
  - IP(), IPv6(), TCP(), UDP(), etc.
  - Build from lower layers up to higher layers moving left to right
  - Separate layers with a /
  - Override default value for field with <field>=<value>



Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 24

We've seen protocols and functions. It's now time to make some packets. We can easily construct them by calling the particular protocol we want with settings that we desire inside of parentheses, as in IP(dst="10.10.10.50"). We can build multi-layer packets by specifying from lower layers up to higher layers, separating each layer with a /. Remember that Scapy moves from lower layers on the left to higher layers on the right. For example, we can specify a TCP/IP packet with all default values by running packet=IP()/TCP(). If you reverse the order of these protocols and don't use lower-to-higher, you may not get what you are expecting.

Scapy lets us specify all the way down to Layer 2 if we want, via functions like Ether(). We don't have to specify Layer 2, though, as Scapy is happy to provide an Ethernet frame using default values (based on moving traffic around the LAN on which our system resides) around whatever we create at Layer 3 (typically IP or IPv6) when we go to send the packet. Most of the time, people use Scapy to specify Layers 3 and up, just relying on Scapy and the underlying operating system itself to construct Layer 2.

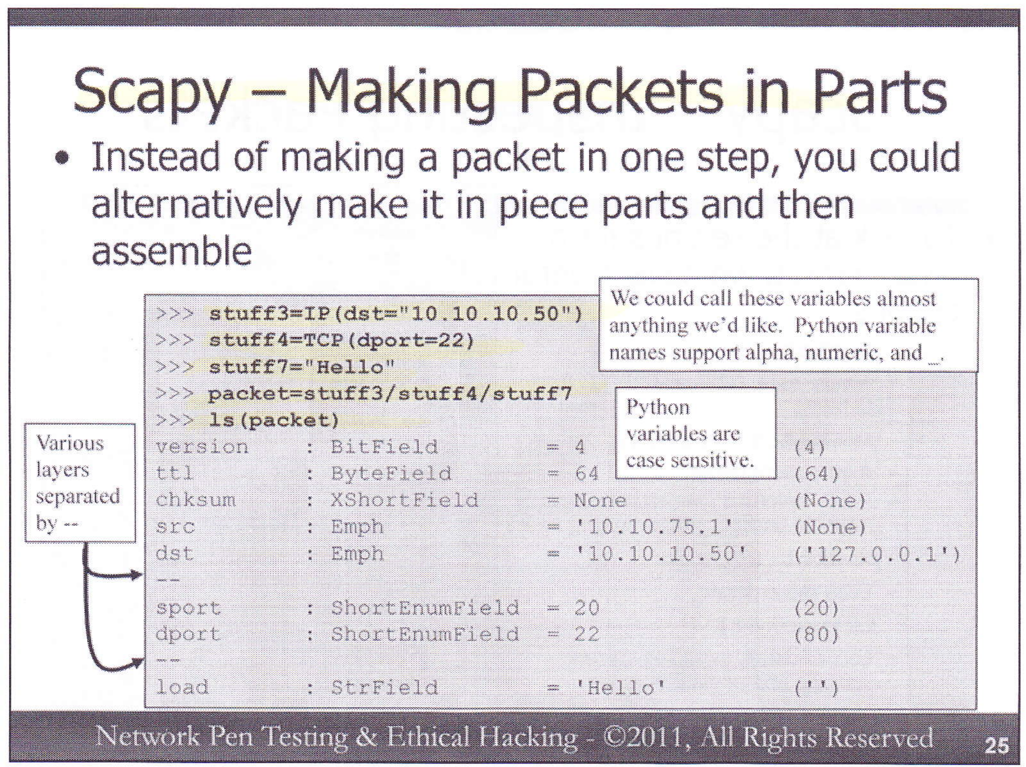
For example, we can create a packet by specifying Ether()/IPv6()/TCP()/"Application Data". If you don't need anything special for your Ethernet frame (such as spoofed MAC addresses), leave off the Ether() up front, and it will be taken care of for you.

We can override the default settings for fields in a packet by simply specifying variable=value pairs. For example, to create a packet called "packet" with a destination IP address of 10.10.10.50, going to TCP port 22, with a payload of "Hello", we could simply run:

```
>>> packet=IP(dst="10.10.10.50")/TCP(dport=22)/"Hello"
```

With our packet created, we can see all of its details using the ls command on the packet itself. The output of ls will show us each field, its data type, its current value, and its default value in parens.

```
>>> ls(packet)
```



In the previous slide, we made a packet in one whole shot, just separating the layers by a / while we were making the packet. Alternatively, we can make a packet in steps and then assemble it all into a single package. Consider the following:

```
>>> stuff3=IP(dst="10.10.10.50")
>>> stuff4=TCP(dport=22)
>>> stuff7="Hello"
```

Here, we've built each layer of the packet, storing each in a different variable. The stuff3 variable stores Layer 3 (the IP layer), stuff4 holds Layer 4, and so on. Note that Python supports variable names of almost anything we'd like, including alpha, numeric, and _ in variable names. It is also crucial to note that Python variables are case sensitive, as they should be for any reasonable system.

Now, let's take each of our layers and stuff them all together into a single packet:

```
>>> packet=stuff3/stuff4/stuff7
```

As before, we can see the detailed settings of our resulting packet (which could have a different name... we called it "packet" because that is easy to remember) using the ls() function:

```
>>> ls(packet)
```

Let's look a little more carefully at the output of ls(). Note how the different layers of the packet are separated by -- in the output. We first see our IP header fields, then our TCP fields, and finally our application layer payload.

Scapy – Inspecting Packets

- To look at the settings for a given packet, we have several options:
  - >>> **packet**
    - A very short summary (deltas from default)
  - >>> **packet.summary()**
    - A little more detail
    - Really helpful if [packet] contains multiple packets (more on that later)
  - >>> **packet.show()**
    - Even more detail
  - >>> **ls(packet)**
    - Lots of detail, including current settings and original defaults

```
>>> packet=IP(dst="10.10.10.50")
>>> packet
<IP  dst=10.10.10.50 |>
>>> packet.show()
###[ IP ]###
  version= 4
  ihl= None
  tos= 0x0
  len= None
  id= 1
  flags=
  frag= 0
  ttl= 64
  proto= ip
  chksum= None
  src= 10.10.75.1
  dst= 10.10.10.50
  \options\
>>> ls(packet)
```

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 26

We can inspect a great deal of packet details using ls(packet). But sometimes you don't want that much detail. Scapy includes numerous different methods for inspecting the fields of packets, with various levels of verbosity.

To see a brief summary of a packet, you can simply enter the variable name at the Python prompt:

```
>>> packet
```

This formulation essentially shows us the deltas from the defaults that you've set for this packet. For more details, we can call the .summary() method of the packet, as follows:

```
>>> packet.summary()
```

This summary is really helpful, as it displays some of the most interesting aspects of the packet information to us. As we'll see later, a packet data structure may hold multiple packets, and calling the .summary() method is a great way to see a synopsis of the packets contained in the structure.

For even more detail, we can call the .show() method:

```
>>> packet.show()
```

Now, we can see a bunch of the headers and the value assigned to them, either by default or by the user.

And, as we've seen, to get a huge amount of detail for the packet (including all values plus the original defaults), we could use:

```
>>> ls(packet)
```

Scapy – Interacting with Individual Fields & Altering Packets

- You can see the value of an individual field in a packet using [packet].[field] if the field name is unique across the protocol layers of the packet

```
>>> packet=IP(dst="10.10.10.50")/TCP(sport=80)
>>> packet.sport
80
```

- If it is not unique (such as IP flags and TCP flags), you can list the value using [packet][[PROTO]].[field]

```
>>> packet[TCP].flags
2
```

Decimal value with Control Bits in order: CEUAPRSF

- After creating a packet, you can change any field by simply using [packet].[field] = [value]

```
>>> packet.sport=443
```

- If field isn't unique (e.g., IP flags and TCP flags), use: [packet][[PROTO]].[field] = [value]

```
>>> packet[TCP].flags="SA"
>>> packet[TCP].flags
18
```

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 27

To see the value assigned to an individual field within a packet, you can simply enter [packet].[field]. Here, we're looking at the source port of a TCP packet:

```
>>> packet.sport
```

This formulation works well if the field name is unique across the different header layers of the packet (which is the case for TCP source port with a name of "sport"). This is not the case for "flags", which is the name of a field in both the IP and TCP headers. In TCP, this is the field that holds the TCP Control Bits (SYN, ACK, etc.) We have a field name collision between IP and TCP. We can look at packet.flags, but Scapy will give us the value of the first flag field it finds, which is the IP flags. What if we really want the TCP flags? We have a couple of different ways of seeing this (one on this slide, the other on the next).

First, we could specify the particular protocol layer we want to pluck the value from by using [packet][PROTO].[field]. The following example shows how we can pull TCP flags from packet:

```
>>> packet[TCP].flags
```

By default, the TCP flags value is 2. That is a decimal representation of the Control Bits, in the order in which they appear in the packet, starting with CWR, then ECE, followed by URG, ACK, PSH, RST, SYN, and FIN. If all of the Control Bits are set to 1, we'd have a value of 255. A value of 2 indicates that the SYN bit is set to 1. We have a SYN packet.

To change the value assigned to a field, we simply assign a fieldname=value, as in:

```
>>> packet.sport=443
```

Or, if the field doesn't have a unique name, we can specify the [PROTO] header where the field resides:

```
>>> packet[TCP].flags="SA"
```

Scapy - Using ".payload" to Reference Packet Parts

- Appending ".payload" to a packet variable name will show you info beyond the initial layer (that is, the lowest layer you've defined for the given packet)
- This technique can also be used to resolve the field-name-collision issue if you don't want to use packet[TCP].flags

```
>>> packet=IP(dst="10.10.10.50")/TCP(dport=22)/"Hello"
>>> packet.flags
0
>>> packet.payload
<TCP  dport=ssh |<Raw  load='Hello' |>>
>>> packet.payload.flags
2
>>> packet.payload.flags="SA"
>>> packet.payload.flags
18
>>> packet.payload.payload
<Raw  load='Hello' |>
```

This is IP flags, the first flags field Scapy encounters.

By using "packet.payload", we are telling it to skip the lowest layer in this packet, the IP layer.

We can set variables in higher layers using this .payload technique, even when there is a name collision with lower layers.

We can even double-up on the .payload to jump in 2 layers.

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 28

Instead of using [packet][PROTO] to access a field when there is a fieldname collision, we could alternatively use the [packet].payload construction. The .payload tells Scapy to jump in beyond the lowest layer of the defined packet. So, for example, suppose we create a packet with the IP, TCP, and Application Layer of the following:

```
>>> packet=IP(dst="10.10.10.50")/TCP(dport=22)/"Hello"
```

Let's look at the flags:

```
>>> packet.flags
0
```

But, we know that Scapy assigns a TCP Control Bit of SYN by default for TCP packets, resulting in a flags value of 2. We're not looking at the Control Bits here, because these flags are in the first header Scapy encounters, the IP header. We can skip past this first layer by using ".payload":

```
>>> packet.payload
<TCP  dport=ssh |<Raw  load='Hello' |>>
```

So, now we can look at the flags in the TCP layer by running:

```
>>> packet.payload.flags
2
```

With a value of 2, we see that the SYN bit is set. We can also change values using .payload:

```
>>> packet.payload.flags="SA"
```

And, we can even use multiple iterations of the .payload concept to jump past multiple layers in the packet:

```
>>> packet.payload.payload
<Raw  load='Hello' |>
```

Scapy – Specifying Dest Addresses

- We can specify destination IP addresses in numerous ways:
  - Via dotted-quad notation:

```
>>> packet=IP(dst="10.10.10.50")
```

  - Via domain name:

```
>>> packet=IP(dst="neo.target.tgt")
```

  - CIDR notation:

```
>>> packet=IP(dst="10.10.10/24")
```

  - Mixed notation:

```
>>> packet=IP(dst="neo.target.tgt/24")
```

  - Multiple targets:

```
>>> packet=IP(dst=["10.10.10.1","10.10.10.7","10.10.10.9"])
```

Remember the [ ] around this list of IP addresses.

Scapy provides great flexibility for specifying destination IP addresses, referred to by Scapy as the dst field in the IP header. We can use the familiar dotted-quad notation:

```
>>> packet=IP(dst="10.10.10.50")
```

Remember to put the address in quotation marks.

Alternatively, we can use the domain name, which will cause Scapy to do name resolution when we try to send the packet:

```
>>> packet=IP(dst="neo.target.tgt")
```

Scapy supports CIDR notation to choose subnets (/32 means match an IPv4 address precisely, the equivalent of using dotted-quad notation by itself).

```
>>> packet=IP(dst="10.10.10/24")
```

Here, we're starting to see how Scapy can take one packet structure we define, and send it to multiple targets. This formulation would send the packet to every IP address on the 10.10.10 subnet.

Scapy also includes a nifty mixed notation, which uses domain names and CIDR formulations. The following will cause Scapy to look up the IP address of neo.target.tgt, and then send the packet to various targets on the same /24 subnet.

```
>>> packet=IP(dst="neo.target.tgt/24")
```

And, finally, we can provide Scapy with a list of multiple targets, simply by putting [ ] around a comma-separated list, as follows:

```
>>> packet=IP(dst=["10.10.10.1","10.10.10.7","10.10.10.9"])
```

Scapy – Setting Port Ranges and TCP Control Bits

- For TCP() and UDP(), we can set dport port *ranges* by simply specifying the start and end ports in parens(), separated by a comma
  - To create packets destined for ports 1-1024, we could run:

```
>>> packet=IP(dst="10.10.10.50")/TCP(dport=(1,1024))
```

- For a *list* of ports, use [ ] and commas:

```
>>> packet=IP(dst="10.10.10.50")/TCP(dport=[22,80,445])
```

- For TCP(), we can set Control Bits using any combinations of the letters "CEUAPRSF", *in any order*
  - To create a RESET-ACK packet for port 80, we could do either:

```
>>> packet=IP(dst="10.10.10.50")/TCP(dport=80,flags="RA")
>>> packet=IP(dst="10.10.10.50")/TCP(dport=80,flags="AR")
```

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 30

We've seen that a packet data structure can have multiple destination IP addresses, but can it have multiple destination ports for TCP or UDP? Why yes, it can. We can specify a range of ports by using parentheses around the start port comma end port, as in:

```
>>> packet=IP(dst="10.10.10.50")/TCP(dport=(1,1024))
```

If you prefer a list of ports instead of range, you could simply create a comma-separated list, included between brackets, as follows:

```
>>> packet=IP(dst="10.10.10.50")/TCP(dport=[22,80,445])
```

As we saw earlier, we can specify TCP Control Bits using the appropriate letters from CEUAPRSF depending on the Control Bit combinations we want to set. It is important to note that you can specify these Control Bits in any order that you choose, so that we can create a RST-ACK packet by using:

```
>>> packet=IP(dst="10.10.10.50")/TCP(dport=80,flags="RA")
```

Or:

```
>>> packet=IP(dst="10.10.10.50")/TCP(dport=80,flags="AR")
```

Scapy - Expanding Multi-Targets into Individual Packets

- When we have a variable with multiple targets (ports and/or addresses), we can display all of the resulting packets using a Python "for" loop in a list structure [ ]

```
>>> packets=IP(dst="10.10.10.50/30")/TCP(dport=(21,23))
>>> [a for a in packets]
[<IP  frag=0 proto tcp dst=10.10.10.48 | <TCP dport=ftp|>>, <IP
frag=0 proto=tcp dst=10.10.10.48 | <TCP dport=ssh |>>, <IP…
<snip>
```

You've gotta include the [ ] around the for loop.

- Of course, you can still do packets.summary() for a shorter form of output

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 31

If you have a packet structure defined that includes multiple different target addresses and/or ports, you can expand it into its individual packets using a Python "for" loop, with the following syntax:

```
>>> [a for a in packets]
```

This syntax tells Python that I want to create a list (that's the purpose of the [ ] around the syntax) that contains a, where the variable a is set to each component of the structure of packets. That is, we use a for loop to iterate through packets, plucking out each value into a variable called a, and then we use those a's to create a list with the [ ]. We are essentially using the for loop to unpack the packets' structure.

Try it on your own system, by creating a packet structure (which we'll call "packets") that is going to destination IP address 10.10.10.50/30 (that is multiple target machines), going to a port range of 21 to 23.

```
>>> packets=IP(dst="10.10.10.50/30")/TCP(dport=(21,23))
```

Now, use the for loop notation to display each of the packets your structure will create:

```
>>> [a for a in packets]
```

Note also that we can get a nice summary of the packets in our structure by running:

```
>>> packets.summary()
```

Scapy – Sending Packets

- We have numerous options for sending packets
  - send()
    - Send packets at Layer 3 (and higher), doesn't receive anything
    - Uses OS defaults for Layer 2
  - sendp()
    - Send packets at Layer 2 -- Custom Layer 2 header included, often created using Ether() for Ethernet
  - sr()
    - Send and receive packets at Layer 3
  - srp()
    - Send and receive packets at Layer 2
  - sr1()
    - Send packets at Layer 3 and return only the first answer
  - srp1()
    - Send packets at Layer 2 and return only the first answer

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 32

So, we've spent all of this time creating packet data structures, but they are really only useful if we can do something with them. Scapy has numerous functions we can call to send packets. Remember, you can get help on any of these functions by running help([function]).

The send() function sends packets using Layer 3 and higher, and doesn't receive any response back. It is a "fire-and-forget" sender. It uses default settings of the operating system itself for all the Layer 2 frame elements.

The sendp() function is used if you have crafted a Layer 2 header (as well as higher layer headers and payloads) for the packet you want to send. This function will send your packet without waiting for a response. The Layer 2 header is often constructed using Ether() for Ethernet.

The sr() function will send your packet and wait to receive responses from the target. Like send(), this function sends packets without custom Layer 2 frames, instead just relying on the operating system defaults for data link functionality.

The srp() function sends and receives packets, using Layer 2 components you specify.

The sr1() function sends packets at Layer 3, grabs the first response, and returns. It will not wait for multiple responses.

And, as you might suspect by now, the srp1() function sends packets at Layer 2, grabs just the first response, and returns.

Scapy - Fine-Grained Options for Sending Packets

- Many of the sending functions have fine-grained options we can see via the help() feature
- Most of the send/receive functions have the following options:
  - filter=[bpf packet filter]
    - The same filters we used for tcpdump
  - retry=[number of times to resend unanswered packets]
  - timeout=[number of seconds to wait before giving up... decimals supported]
  - iface=[interface to send and receive]

```
>>> sr(packet,timeout=0.1,
filter="host 10.10.10.50
and port 22")
Begin emission:
Finished to send 1 packets.
*
Received 1 packets, got 1
answers, remaining 0
packets
(<Results: TCP:1 UDP:0
ICMP:0 Other:0>,
<Unanswered: TCP:0 UDP:0
ICMP:0 Other:0>)
```

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 33

Each of the send functions supported by Scapy can be called by itself, just providing it a packet to send, as in send(packet). However, these send functions also support finer-grained options to control more details of sending. The options are specified as variable=value pairs in the function call itself. To see these options and how they apply to each send function, remember to call help([function]).

Some of the most useful of these options in sending packets include:

filter=[bpf packet filter]: With this option, we can define a packet filter that tells Scapy to accept only responses that match certain characteristics we define according to Berkeley Packet Filter (bpf) notation. This is the same filter syntax we covered earlier for tcpdump.

retry=[N]: This tells Scapy to resend the packet up to N times if it doesn't get a response.

timeout=[X]: This option tells Scapy to wait only N seconds for a response. Most timing options in Scapy are based on a number of seconds, making it much more human-friendly than some other packet tools which are based on milliseconds or microseconds. We can specify decimal seconds, such as 0.1 for a tenth of a second or .000001 for a microsecond.

iface=[Interface Name]: This lets us specify the particular interface to send the packet on, such as eth0 or lo. By default, Scapy determines the interface to use based on the way the operating system would route the packet.

An example call that uses some of these options is:

```
>>> sr(packet,timeout=0.1,filter="host 10.10.10.50 and port 22")
```

Here, we're sending a packet, waiting to receive a response (sr). We'll only wait 0.1 seconds, and we only want to receive answers that match the filter of host 10.10.10.50 AND port 22 (i.e., packets must involve 10.10.10.50 to or from port 22, or else we'll ignore them).

Scapy - Dealing with Responses

- Store all results by using [var] = sr([packet])

```
>>> packet=IP(dst="10.10.10.50")/TCP(dport=22)
>>> response=sr(packet)
Begin emission: <snip>
>>> response
(<Results: TCP:1 UDP:0 ICMP:0 Other:0>, <Unanswered: TCP:0
UDP:0 ICMP:0 Other:0>)
```

Two sets: Results and Unanswered

- But, send/receive functions actually have *two* sets of responses: answered and unanswered, so we can use [var1],[var2]=sr([packet])

```
>>> packet=IP(dst="10.10.10.50")/TCP(dport=22)
>>> ans,unans=sr(packet)
Begin emission: <snip>
>>> ans
<Results: TCP:1 UDP:0 ICMP:0 Other:0>
>>> unans
<Unanswered: TCP:0 UDP:0 ICMP:0 Other:0>
```

- If you don't provide a variable, your last results are stored in _

```
>>> sr(packet)
Begin emission: <snip>
>>> ans,unans=_
```

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 34

When we use a send/receive function call, like sr, srp, sr1, or srp1, we can catch our responses in a variable using [var]=sr(packet), as in:

```
>>> response=sr(packet)
```

Then, we can review our responses with:

```
>>> response
```

When we use sr() or srp(), our responses are often broken into two sets, surrounded in parentheses, separated by a comma, each delineated with < and >. The first set is called "Results". The other is called "Unanswered". Each includes an inventory of the number of TCP, UDP, ICMP, and Other packets we either got back, or that were sent and received no answer.

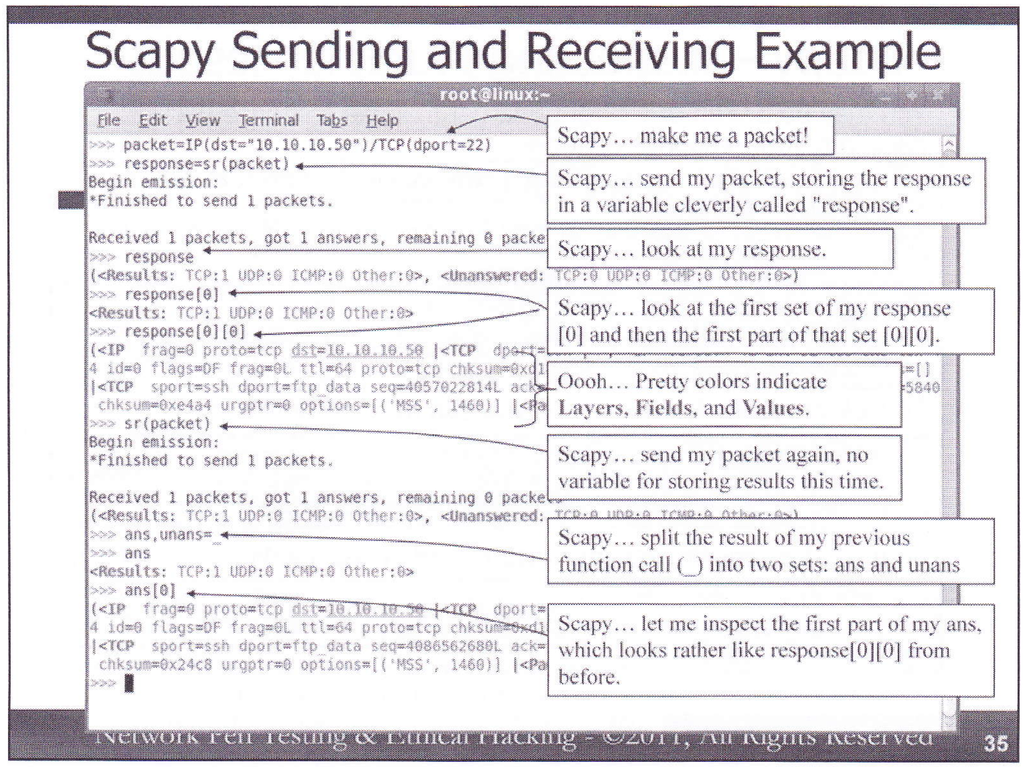
Often, it is useful to separate out the answered responses from the unanswered responses. We can do that using:

```
>>> ans,unans=sr(packet)
```

Then, we can view the answered packets, and also look at the unanswered packets we sent by simply referring to the ans and unans variables.

It's really important to note that if you call a function to send a packet, but don't provide a variable name to store the results (i.e., you don't use "something=sr(packet)" and instead just use "sr(packet)"), your results are automatically placed into a variable called _. When using Scapy interactively, this _ variable is very helpful, because sometimes you get ahead of yourself, building packets and sending them quickly, without remembering to store your results in a variable. Once you've sent some packets (via something like "sr(packet)"), you can split the results into answered and unanswered sets which you can then use later by running:

```
>>> ans,unans=_
```



Let's look at a quick example of building a packet with Scapy, and then sending it to a target machine and analyzing the responses. We'll start with making a packet, built from an IP component and a TCP component, separated by a /:

```
>>> packet=IP(dst="10.10.10.50")/TCP(dport=22)
```

We send the packet and get responses, storing our result in a cleverly named variable "response":

```
>>> response=sr(packet)
```

We can see that we received 1 packet back. We can look at our response data structure using:

```
>>> response
```

Here, we see that our response has two components: Results, which includes 1 TCP packet, and Unanswered, which doesn't have any packets in it. To look at the first component of our response data structure (that is, the "Results" piece), we can look at the first component (offset of 0) in that structure via the [0] notation:

```
>>> response[0]
```

Here, we see just the Results part. If we had looked at response[1], that would have shown us the unanswered component of our response structure. We can now look at the first packet (that is, the one with a zero offset) in our response Results with:

```
>>> response[0][0]
```

Here, we see the details of the packet we sent plus the response we got back.

Alternatively, we could have called sr(packet) without immediately storing the results in a variable:

```
>>> sr(packet)
```

Now, we immediately split the result (_) into a set of answered and unanswered results:

```
>>> ans,unans=_
```

We can then look at ans, unans, or the first part [0] of ans, which is what we sent and got back at a detailed packet level. It is worth noting that ans[0] will contain what we sent, and ans[1] contains the response we got back, similar to response[0][0] and response [0][1] from above.

Scapy - Inspecting Multiple Results

- You may do a scan of a target and get multiple response packets back into your response variable

```
>>> packet=IP(dst="10.10.10.50")/TCP(dport=(1,1024),flags="S")
>>> ans,unans=sr(packet)
Begin emission:
<snip>
Received 1361 packets, got 1024 answers, remaining 0 packets
>>> ans
(<Results: TCP:1024 UDP:0 ICMP:0 Other:0>, <Unanswered: TCP:0
UDP:0 ICMP:0 Other:0>)
```

Scapy got some packets that weren't responses to what we sent… it discarded those.

- To look at results for each port, you can use:

Sent portion comes first… received response comes second.

```
>>> ans.summary()
IP / TCP 10.10.75.2:ftp_data > 10.10.10.50:tcpmux S ==> IP / TCP
10.10.10.50:tcpmux > 10.10.75.2ftp_data RA / Padding <snip>
```

- To look at a specific response, use record offset number inside of [ ]:

```
>>> ans[2]
(<IP frag=0 proto tcp dst=10.10.10.50 | <TCP dport=3 |>>, <IP
version=4L ihl=5L tc
```

Remember these [ ]'s are offsets into an array, so [0] is port 1 if you specify a port range of 1,1024. You may just want to do ports 0,1024. Also, beware of ports that don't respond.

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 36

We've seen how we can pick off individual response components with [N] notation, but sometimes getting information in that way is just too fine grained. Consider the following port scan, in which we send a TCP SYN packet to ports 1 through 1024 on target 10.10.10.50:

```
>>> packet=IP(dst="10.10.10.50")/TCP(dport=(1,1024),flags="S")
>>> ans,unans=sr(packet)
```

First note on the slide that Scapy says that it received 1361 packets, and got 1024 answers. That is, while sr() was running, Scapy noticed that there were 1361 packets coming back to the machine on which it was running, but only 1024 of them were responses to packets Scapy sent. The other packets beyond the 1024 were discarded. We'll see shortly how we can use Scapy to sniff, grabbing all packets.

Anyway, with our Results stored in the ans variable (and unanswered responses, of which there are none in our unans variable), we can view a short survey of the results with:

```
>>> ans
```

Here, we see that we received 1024 TCP packets in response. Great! But, what are they? All open ports? That is very unlikely. Let's get a summary of them:

```
>>> ans.summary()
```

Here, we see that most of our results have "RA" in them, so we have RST-ACKs. Most of those ports are closed. Only where we see SA will we have an open port, because we got a SYN-ACK back.

As before, we can look at an individual response by the offset notation. So, to check the result for port 3, we could look at [2] (remember that 0 is the first item).

```
>>> ans[2]
```

Scapy Loops

- We often want to loop through a series of packets (to do an address sweep or port scan, for instance)
- We can do this with Layer-3 sending using the Scapy srloop() function, which sends the same packet and prints results continuously

```
>>> srloop(packet)
```

  - Similar to hping()

```
>>> packet=IP(dst="10.10.10.50")/ICMP()
>>> srloop(packet)
RECV 1: IP / ICMP 10.10.10.50 > 10.10.72.2 echo-reply 0 / Padding
RECV 1: IP / ICMP 10.10.10.50 > 10.10.72.2 echo-reply 0 / Padding
^C
Sent 4 packets, received 4 packets.  100.0% hits.
(<Results: TCP:0 UDP:0 ICMP:2 Other:0>, <PacketList: TCP:0 UDP:0
ICMP:0 Other:0>)
```

- Or, we can accomplish this with a Python for loop (which can let us change packet settings as the loop runs):

```
>>> for <var> in <list>:
...     statement
```

- Note mandatory indenting in the statement portion! Four spaces

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 37

We can loop through a series of packets using a variety of different constructs with Scapy. If you want to send the same packet again and again, printing out the response for each sent packet, you can call the srloop() function, as follows:

```
>>> srloop(packet)
```

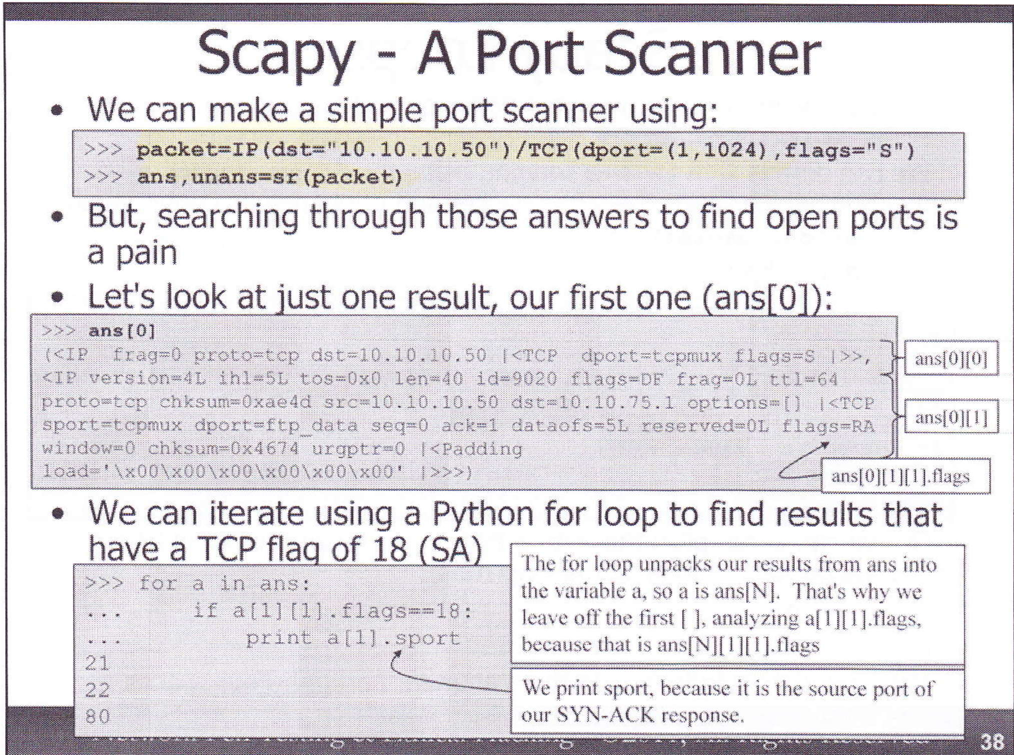
We can see the results here with RECV displayed directly on the screen, showing the result of our ICMP echo request packet going to target 10.10.10.50. This srloop feature is very similar to the behavior of the hping command, a packet crafting tool for Linux, Windows, and Mac OS X that isn't nearly as flexible as Scapy.

Alternatively, if we want more flexible looping, we can use a Python "for" loop. With this kind of structure, we can unpack results and even change packet settings in the middle of a loop. The syntax of a Python for loop is as follows:

```
>>> for <var> in <list>:
...     statement
```

On the next slide, we'll see an example of this for loop in action.

Note that the statement portion of the loop must be indented. Python requires mandatory indentation to make code more readable. Four spaces of indents is the recommended.



Let's see how we can use a Python loop to extract useful information about a target. We've already seen how we can do a port scan using the following syntax:

```
>>> packet=IP(dst="10.10.10.50")/TCP(dport=(1,1024),flags="S")
>>> ans,unans=sr(packet)
```

We can get a summary of our responses by simply looking at ans or ans.summary(). But, if the target machine is sending us RST-ACKs from closed ports, even ans.summary() will have over a thousand entries in it, making it tedious to see which ports are really opened. There are numerous ways for us to extract information about which ports are open, but one of the most flexible is to use a Python for loop. We'll be looping over the results in ans, selecting just those that have the SYN-ACK bits set. To figure out how to select the right responses, let's look at one of those responses, ans[0], the response from the target on port 1 (which is the tcpmux service).

In the output, we can see that ans[0] is made of two parts each included in < >: the packet we sent (which is ans[0][0]) and the response we got back (which is ans[0][1]). Within the response we got back, we see that there are three parts, again embedded in < >: the IP header (ans[0][1][0]), the TCP header (ans[0][1][1]), and the Padding (ans[0][1][2]).

We can use our for loop to unpack each ans[N] into a variable called a, as follows:

```
>>> for a in ans:
```

Now, variable "a" will have the result for an individual port. In other words, "a" represents ans[N]. So, a[0] is the packet we sent, and a[1] is the response we got back. Then, a[1][0] is the IP header of the response, and a[1][1] is the TCP header. So, we need to check a[1][1], which corresponds to ans[N][1][1], the TCP header. We'll look at the flags field, to see if it has a value of 18, which represents SYN-ACK:

```
...     if a[1][1].flags==18:
```

If this is a match, we'll print out the source port from the TCP header, which is where the SYN-ACK response came from, which should be an open port:

```
...         print a[1].sport
```

The result is a nice little report of open ports.

Scapy – Sniffing & Reading Packets

- To sniff, use the sniff() function

  `>>> packets=sniff(filter="[filter]")`
    - Gathers only certain packets

  `>>> sniff(count=[N])`
    - Sniffs only N packets
  - Warning! Can be slow; you may miss packets
  - Packets placed into _, or you can specify a variable, as in packets=sniff()
  - Look at them en masse with _.summary()
- To get packets from a pcap file, use:

  `>>> rdpcap("[filename]")`
- To write packets to a file, use:

  `>>> wrpcap("[filename]",[packets])`
- You can also invoke Wireshark directly from Scapy

  `>>> wireshark([packets])`

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 39

Scapy also includes a sniffer, which can be invoked using the sniff() function call. We can optionally specify filters (using bpf notation like we used for tcpdump) through the use of the filter="[filter]" notation. We can also put a limit on the number of packets we want to gather, by specifying "count=[N]". To get more detail about the various function call arguments besides filter and count of sniff(), please run help(sniff).

It is important to note that Scapy's sniffer isn't super fast. Its performance sometimes lags, causing you to miss packets. It is not as fast as tcpdump, a far simpler sniffer.

When you hit CTRL-C, sniff() stops grabbing packets, returning the results it captured so far.

As you might expect, by default, packets grabbed by sniff() are placed into _, or you can put them into a given variable using [var]=sniff().

To see a summary of all the packets you've sniffed, you can run _.summary(), or [var].summary().

Instead of pulling packets from a network interface with sniff(), Scapy can read them from a packet capture file using rdpcap(), where we specify a file name to pull the packets from. Again, packets are sent to _ or a variable name we provide in [var]=rdpcap().

We can likewise write our packets into a pcap file using the wrpcap() call, where we provide a filename and the packets we want to write.

Finally, Scapy can invoke the Wireshark sniffer to analyze a set of packets, right from the Scapy Python prompt, by simply calling wireshark([packets]). This provides a handy way to see the various fields in packets using the wonderful GUI of Wireshark.

Scapy Fuzzing

- Scapy includes the fuzz(packet) function
  - Applied across a whole layer, as in:
    ```
    >>> packet=fuzz(IP())
    ```
    - Or `>>> packet=IP()/fuzz(TCP())`
- Varies all of the fields in the given layer that you haven't tied down
  - Checksum fields are still calculated appropriately
  - Values aren't assigned until we use packet
  - If you run packet.show(), you'll see RandNum and RandByte in the to-be-fuzzed slots

```
>>> packet=IP(dst="10.10.10.50")/fuzz(TCP(sport=1025,dport=80))
>>> packet.show()
>>> sr1(packet,timeout=1)
```

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 40

Scapy also supports fuzzing, placing random data into fields of a given protocol to see how a target machine may respond to the garbage. To use the fuzzing option, we simply call the fuzz() function, with an argument of a protocol that we want fuzzed, as in fuzz(IP()). Any fields we haven't hard coded will be substituted with random data.

So, for example, to create a completely fuzzed IP packet, we could use:

```
>>> packet=fuzz(IP())
```

It should be noted that the random numbers for the various fields are not assigned until we actually send or otherwise use the packet. We can see this by looking at our fuzzed packet's settings:

```
>>> packet.show()
```

We'll see RandNum and RandBytes in various fields.

To create an IP packet with the default values, plus a fuzzed TCP layer, we could run:

```
>>> packet=IP()/fuzz(TCP())
```

For any field in the given fuzzed layer that we don't specify, Scapy will choose a pseudo-random value. Checksum fields are still calculated appropriately, however, to ensure that the packet is valid.

So, for example, to create a packet that is destined for IP address 10.10.10.50, with a fuzzed TCP layer that alters all fields except for the source port of 1025, the destination port of 80, and the TCP checksum field, sending the packet with a timeout of 1 second and receiving only the first response, we could run:

```
>>> packet=IP(dst="10.10.10.50")/fuzz(TCP(sport=1025,dport=80))
>>> sr1(packet,timeout=1)
```

Using Scapy in a Python Script

- We've focused on using Scapy at an interactive Python prompt
- We could use all of these techniques in a Python script, with a name suffix of .py
- At the start of your .py file, make sure to include:

```
#!/usr/bin/python
#
from scapy.all import *
## Your code goes here ##
```

- Remember mandatory indenting!
  - Four spaces is the recommended indentation

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 41

We've focused on using Scapy as an interactive shell, but everything we've discussed can be used to write Python scripts that call Scapy functions. Simply preface all of the commands you create with the lines shown on the slide, which include the following elements:

```
#!/usr/bin/python
```

This item tells the system to use /usr/bin/python to process this script.

```
#
```

This line is a comment. You can add any comments you'd like, prefaced with a # prompt.

```
from scapy.all import *
```

This vitally important line imports all of the Scapy functionality we'd like to use.

Remember, when you create Scapy scripts in Python, you must observe mandatory indenting. With for loops and if statements (among other Python commands), you must indent. If you do not, you'll get a Python error message. It is recommended to use four spaces for each level of indentation.

Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

- Scanning Goals and Types
- Overall Scanning Tips
  - Sniffing with tcpdump
- Network Sweeping with Scapy
  - ***Scapy/tcpdump Exercise***
- Network Tracing
- Port Scanning
  - Nmap
  - Nmap Exercise
- OS Fingerprinting
- Version Scanning
  - Nmap -O -sV and Amap Exercise
- Vulnerability Scanning
  - Nmap Scripting Engine
  - NSE Exercise
  - Nessus
  - Nessus Exercise
  - Other Vuln Scanners
- Enumerating Users
  - Enumerating Exercise
- Netcat for the Pen Tester
  - Netcat Exercise

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 42

Next, let's do some hands-on exercises with Scapy and tcpdump. These exercises will take the form of some challenges. We'll pose to you a scenario with packets you need to formulate with Scapy and tcpdump configurations you'll need to write to see these packets. All of the answers to the challenges are included on the page right after each challenge. Try to formulate the answers on your own system before peeking ahead to our suggested answers. If you cannot get one to work, though, feel free to look at the next page for some hints.

Exercise: Scapy and tcpdump

- We are going to experiment with tcpdump and Scapy…
- …Honing our skills to formulate packets and get responses
  - Sniffing default behavior of Scapy
  - Looking at ICMP payload behavior
  - Crafting Land packets
  - Conducting sweeps of a target environment

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 43

We are now going to perform an exercise to hone our skills in using both tcpdump and Scapy. We'll specify certain tcpdump configurations that will look for packets with specific settings. Then, we'll generate such packets using Scapy to verify that we can craft packets we want using Scapy and that we can detect them using tcpdump. Scapy and tcpdump do great duets!

For each of the exercise components we'll analyze, try to formulate the commands for tcpdump and Scapy yourself before flipping to the next slide, where solutions are included. If you need a hint, though, you can peek ahead.

1) Default Scapy Behavior

- The challenge:
  - Configure tcpdump to display all packets with your machine's IP address and the IP address of target machine 10.10.10.20, in either direction
  - In a separate window, run Scapy to craft a packet for 10.10.10.20 with no options
    - For the IP layer, set only the dst address of 10.10.10.20
    - For the TCP layer, use only the defaults
    - Use sr() to send your packet
  - In your sniffer output:
    - What is the default source port? What are the default Control Bits (flags) settings?
    - What is the default destination port?
    - What kind of response do you see?

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 44

We start out by measuring the default behavior of Scapy using tcpdump. Your challenge is to configure tcpdump to display all packets that include both your Linux machine's IP address (which is likely 10.10.75.X, with a specific X assigned to you) and the IP address of a host we are going to send packets to (in this case, 10.10.10.20). That way, we can capture only those packets that you are generating for 10.10.10.20. Configure tcpdump so that it does not resolve domain names or look up port numbers.

First, formulate that tcpdump command.

Then, run Scapy against the target, setting the dst to 10.10.10.20 for the IP layer, and using all of the defaults for the TCP layer.

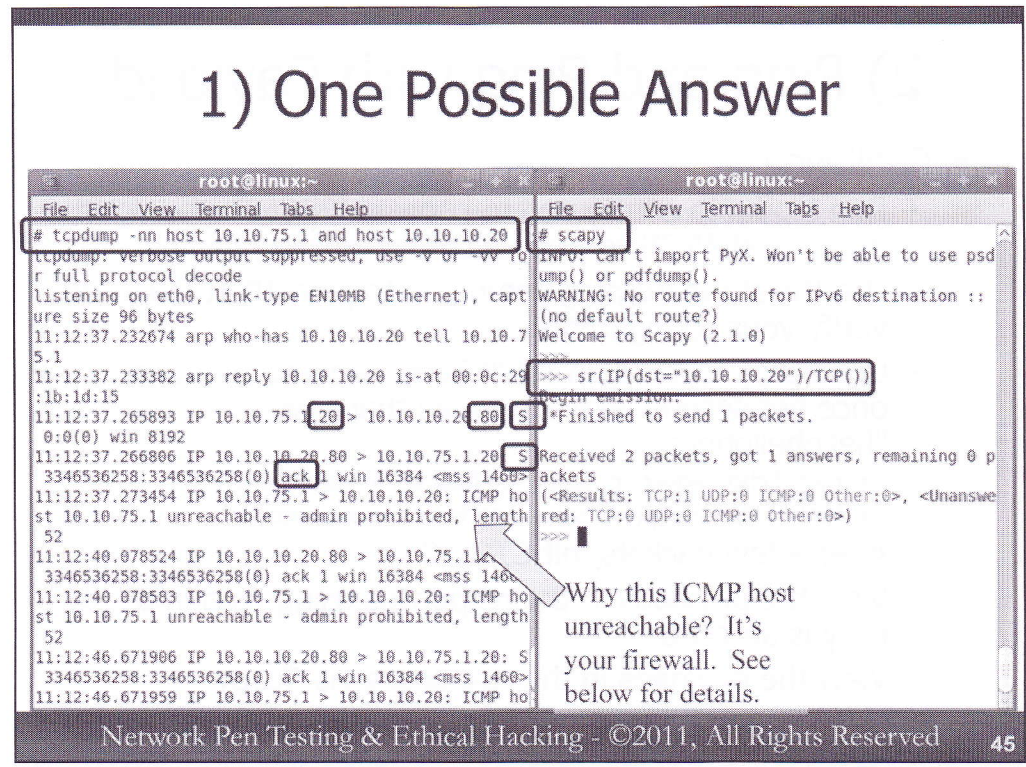
If your tcpdump has been configured appropriately, you should start seeing packets on its output. Based on those packets, discern the answer to the following questions:

What is the default source port? What is the default Control Bits (i.e., TCP flags) setting?

What is the default destination port?

What kind of response do you see?

The next slide includes answers… but try to complete this yourself before looking ahead. If you get stuck, feel free to turn to the next page.



One possible tcpdump command line that will focus on the packets we seek is:

```
# tcpdump -nn host [YourLinuxIPaddr] and host 10.10.10.20
```

Note that you should replace [YourLinuxIPaddr] with the IP address you assigned to your Linux machine. This syntax will make tcpdump look for packets that are associated with both hosts [YourLinuxIPaddr] and 10.10.10.20, while not looking up their names or services (-nn).

In your other window, you can run Scapy as follows to craft the requested packet and send it:

```
# scapy
>>> sr(IP(dst="10.10.10.20")/TCP())
```

As Scapy runs, we can see the packet it generates in the tcpdump output. Specifically, we see the packet with a source port of 20 (commonly associated with ftp-data), a destination port of 80 (associated with http). We also see that it is a SYN packet, given the S in tcpdump's output.

In tcpdump, we can also see the response coming back from the target, which has an S on the line, as well as an "ack" a bit later in the line. This is a SYN-ACK response. That port is open.

Note that you may see an ICMP host unreachable message, going from your machine [YourLinuxIPaddr] back to 10.10.10.20. Why is that packet being sent? This is an artifact of the built-in Linux firewall controlled via iptables. Scapy is crafting a TCP packet, directed at 10.10.10.20. That machine responds with a SYN ACK, sent back to [YourLinuxIPaddr]. Your Linux kernel had no idea that the earlier packet went out (because it was crafted by Scapy). Thus, it doesn't know what to do with the SYN ACK response, so it simply tells 10.10.10.20 that the host is unreachable. That's rather counter intuitive, telling 10.10.10.20 that the host is unreachable. But, it is the default behavior of this version of the Linux firewall.

To disable the Linux firewall (a reasonable thing to do when you are scanning and/or crafting packets), you should run:

```
# service iptables stop
```

2) Ping and Ping with Payload

- Challenge:
  - Run tcpdump configured to show only ICMP messages, in hex and ASCII format, without resolving names
  - Use the standard ping command to ping 10.10.10.20 to verify your configuration
  - Use Scapy to send an ICMP Echo Request Message once per second with a payload that says "hellohellohello"
    - Hint: Echo Request is Scapy's default ICMP message type
    - Hint 2: Use srloop() to send a packet once per second
  - After a few packets, hit CTRL-C
  - View the payloads in the responses using tcpdump... it truly is an echo
  - View the payloads in the responses via Scapy

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 46

For our next exercise, we are going to use Scapy to send packets with a payload, and look at the contents of that payload in tcpdump and Scapy.

To start, invoke tcpdump so that it will capture only ICMP messages, displaying both hex and ASCII formats of packets, without resolving names. Then, verify your tcpdump invocation by pinging 10.10.10.20 using a standard ping:

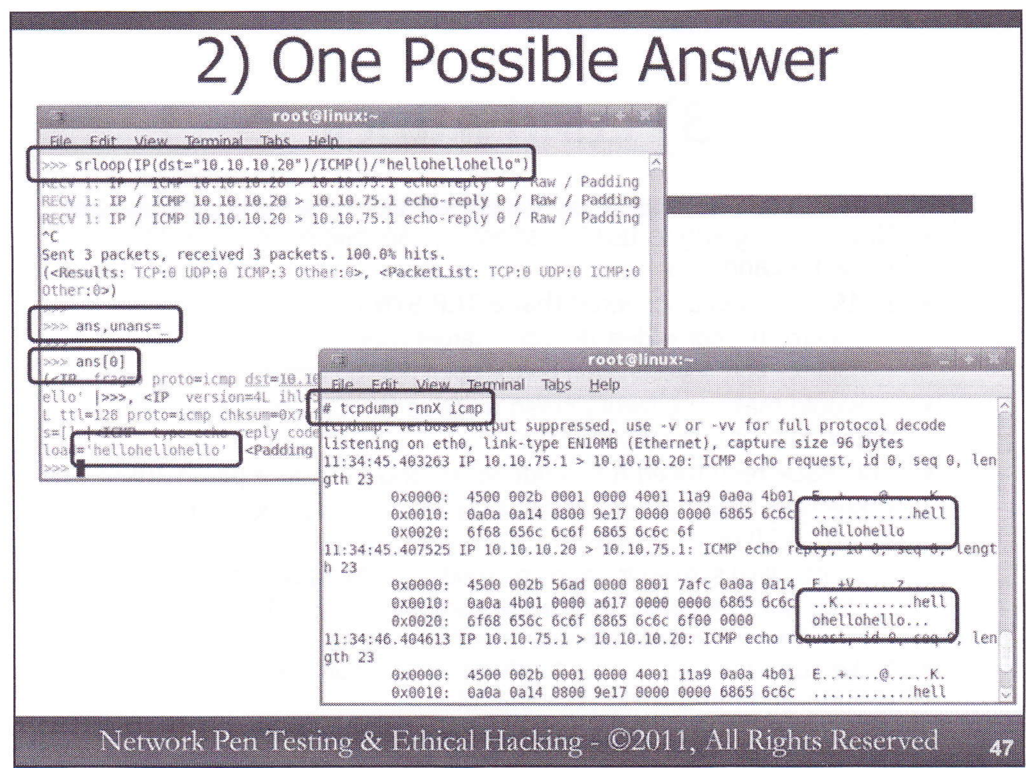
```
# ping 10.10.10.20
```

If you see the ping packets on your tcpdump output, your tcpdump syntax is good.

Then, use Scapy to send a payload of "hellohellohello" in ICMP Echo Request packets to the target machine. As a hint, remember that Scapy's ICMP uses Echo Request type messages as its default.

Now, tcpdump should show the ping and ping response messages. Look at the payload of each. Do you see that the ping response truly is an echo?

Try to formulate the commands for tcpdump and Scapy yourself before flipping to the next slide. If you need a hint, though, you can peek ahead.



First, we've run tcpdump, invoked with the –nn flag to make it show us only numbers and ports, not names, and the –X flag to display hex and ASCII output. We've specified that we only want packets associated with ICMP, as follows:

```
# tcpdump -nnX icmp
```

Then, we've run Scapy as follows:

```
>>> srloop(IP(dst="10.10.10.20")/ICMP()/"hellohellohello")
```

This invocation will make Scapy send ICMP packets with a payload of "hellohellohello" to 10.10.10.20, repeatedly, once per second.

And, note that in our tcpdump output, we see that hellohellohello was sent from our machine to the target (10.10.10.20). We also see that the ping response (from 10.10.10.20 to our IP address) includes the hellohellohello string coming back. Truly, ICMP Echo Request is an echo.

Hit CTRL-C in both windows to get your command prompt back. Now, in the Scapy window, let's analyze our result. First, we'll store it into ans and unans:

```
>>> ans,unans=_
```

Now, look at your first response:

```
>>> ans[0]
```

You should be able to see the payload ("hellohellohello") of both the request and the response in this data structure.

3) Land Attack

- Now, we're going to use the spoofing capabilities of Scapy to launch a Land attack
- In 1997, it was discovered that a TCP SYN packet with:
  - Source IP addr = dest IP addr = target addr
  - Source port = destination port = open port on target
- ... would make the target crash or drive the CPU to 100%, depending on the system type
- This issue resurfaced for Windows XP and 2003 in a patch 2005!
- *Challenge:* Using Scapy, create four Land-style packet for 10.10.10.20 on TCP port 80
  - Hint: Make sure your command sends only four packets
  - You won't be getting a response back, so use send()
    - Check out help(send) to see how to control the count
- Make sure you first invoke tcpdump with a suitable configuration to display your packet

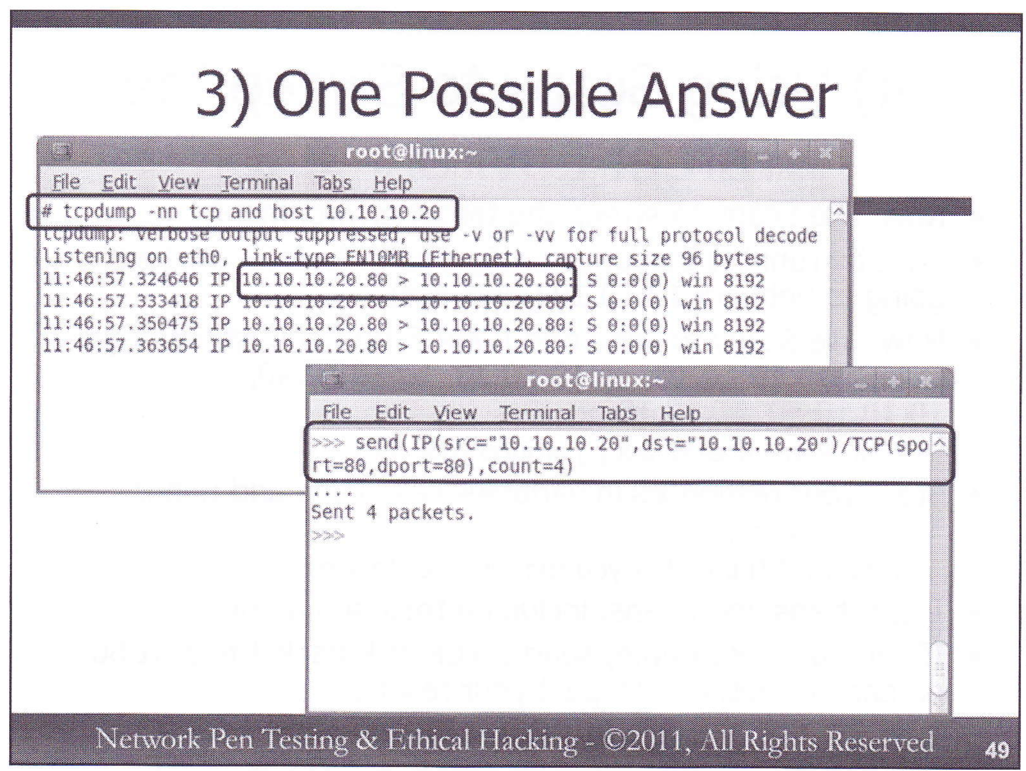
Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 48

For our next exercise component, we are going to use Scapy to generate a Land attack. Way back in 1997, a security researcher discovered that if you send a machine a spoofed TCP SYN packet to an open port with the source IP address set to the same value as the destination IP address, and the source port the same value as the destination port, the target system's CPU would spin up to 100% and in some cases even crash. The target machine would, in effect, experience a condition where it would look like it received a packet from itself, going in the same port that it is leaving, causing significant problems. Back in 1997, every major vendor fixed the problem.

In 2005, the issue resurfaced with a Microsoft patch for Windows XP and 2003 that re-introduced the flaw. Microsoft then released yet another patch to fix it, again.

We are going to verify our Scapy skills by recreating the Land attack.

Your challenge is to use Scapy to send a four Land-style packets for target 10.10.10.20 on TCP port 80. Before you run Scapy to do this, however, make sure that you first configured tcpdump appropriately so that you can see your packet as it is emitted. For this challenge, make sure you send only four identical Land packets to the target machine. Use help(send) for information about how to set the count.



Here is one possible solution to creating a Land attack and configuring tcpdump to sniff that packet.

We can capture our packets with a tcpdump command line that doesn't lookup names (-nn) but does show all TCP packets (tcp) that are also (and) going to or from the target IP address (host 10.10.10.20). You could narrow this down further by specifying particular ports, but it is often useful to invoke tcpdump with a broader configuration so we can see more activity than just what a given tool is sending.

We've run Scapy as follows:

```
>>> send(IP(src="10.10.10.20",dst="10.10.10.20")/TCP
(sport=80,dport=80),count=4)
```

Here, we've told Scapy to craft a packet with an IP header where the source and destination IP adddress are both 10.10.10.20. For the TCP layer, both our source and destination ports are 80. We'll use a count of 4 to send only four packets.

We can see in our tcpdump output the Land-style attack packets.

4) Using Scapy to Sweep the Target Environment

- Now, use Scapy to sweep the target environment
- Start by running tcpdump configured to look for packets going to network 10.10.10 (hint: net)
- Now, use Scapy to send one ICMP Echo Request message to host 10.10.10.10, 10.10.10.20, 10.10.10.40, 10.10.10.50, 10.10.10.60
  - Hint: Remember to put [ ] around your dst list
- Store your responses in variables called ans and unans
  - Hint: ans,unans=sr()
  - Hint: Hit CTRL-C after you see "Finished to send..."
- Inspect ans and unans, including their summaries
- If you have extra time, send a TCP ACK packet to port 80 on each target, and inspect your results

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 50

For our final challenge, we are going to send ICMP Echo Request messages to a list of target machines. Start off by configuring tcpdump to sniff for packets associated with the target network 10.10.10. As a hint, consider using the "net" keyword in your filter.

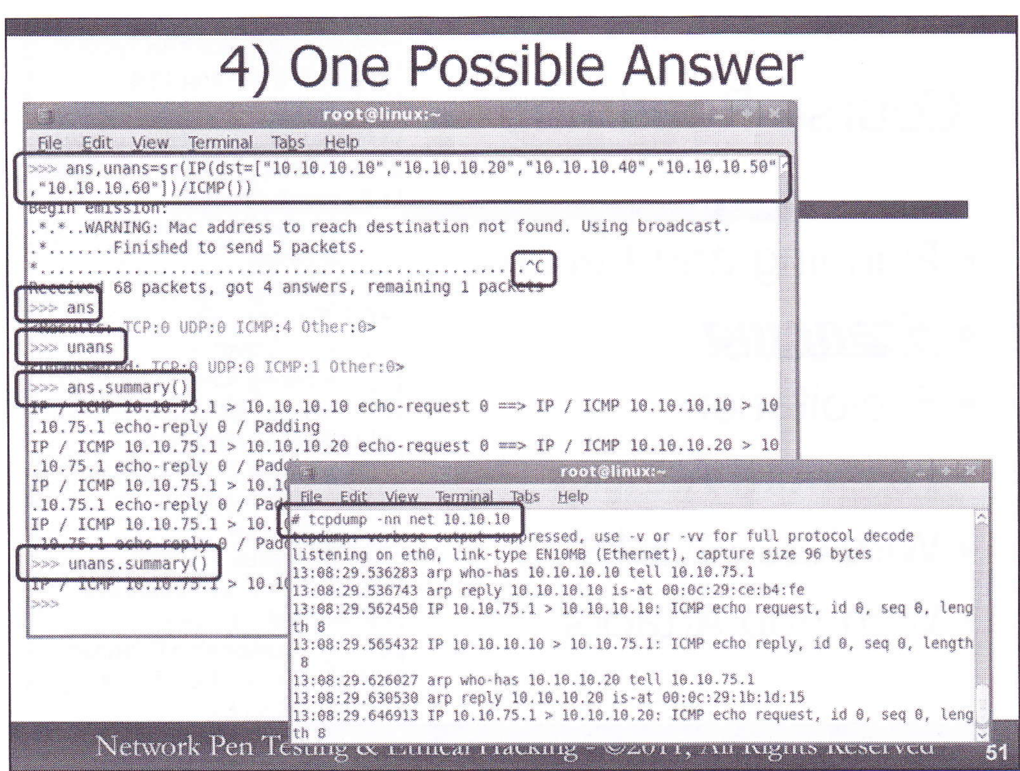
Now, use Scapy to send one ICMP Echo Request message to host 10.10.10.10, 10.10.10.20, 10.10.10.40, 10.10.10.50, and 10.10.10.60. Note that 10.10.10.40 isn't in use, so we shouldn't get a response from it. As another hint, don't forget to put [ ] around your list of dst addresses. Store your results in variables called ans and unans. As a hint for this, remember that you can use ans,unans=sr() to separate out the answered and unanswered results.

As Scapy runs, look at the output from your sniffer. Note that the sr() will keep running long after it has finished sending packets, waiting for possibly very late responses. Hit CTRL-C in Scapy after you see it indicate "Finished to send 5 packets."

Now, inspect ans and unans, looking at their summaries.

After you've reviewed their summaries, feel free to use offsets such as [0] and [1] to look at the components of ans and unans in more detail.

Finally, if you have extra time, repeat this challenge, but this time send TCP ACK packets to each of the target machines and inspect your results.



We start by running tcpdump looking for traffic associated with network 10.10.10:

tcpdump -nn net 10.10.10

Then, we tell Scapy to send packets to the targets, storing our results in ans,unans (or we could use ans,unans=_ after we run sr()).

>>>
ans,unans=sr(IP(dst=["10.10.10.10","10.10.10.20","10.10.10.40","10.10.10.50","10.10.10.60"])/ICMP())

We'll now see the packets going out. Note that when it gets to 10.10.10.40 (the unused address), Scapy will not get an ARP response, so it prints a WARNING on the screen.

After we see "Finished to send 5 packets", we hit CTRL-C.

Now, look at your ans:

>>> **ans**

We see that we got 4 ICMP responses.

Next, look at unans:

>>> **unans**

Here we see one ICMP message was unanswered.

We can review the summary of our results with:

>>> **ans.summary()**

Then, we can access individual responses with ans[0], ans[1], etc. We can also access the components of these responses with ans[0][1], and so on.

Try it again with TCP ACKs, and you'll see RESET messages coming back from the valid targets. You just conducted an ACK scan.

Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

- Scanning Goals and Types
- Overall Scanning Tips
  - Sniffing with tcpdump
- Network Sweeping with Scapy
  - Scapy/tcpdump Exercise
- ***Network Tracing***
- Port Scanning
  - Nmap
  - Nmap Exercise
- OS Fingerprinting
- Version Scanning
  - Nmap -O -sV and Amap Exercise
- Vulnerability Scanning
  - Nmap Scripting Engine
  - NSE Exercise
  - Nessus
  - Nessus Exercise
  - Other Vuln Scanners
- Enumerating Users
  - Enumerating Exercise
- Netcat for the Pen Tester
  - Netcat Exercise

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 52

Our next topic will be network tracing, figuring out the paths that packets take as they traverse the network. These methodologies and tools will be instrumental in our composing a network diagram of the target environment.

The IPv4 Header and TTL Field

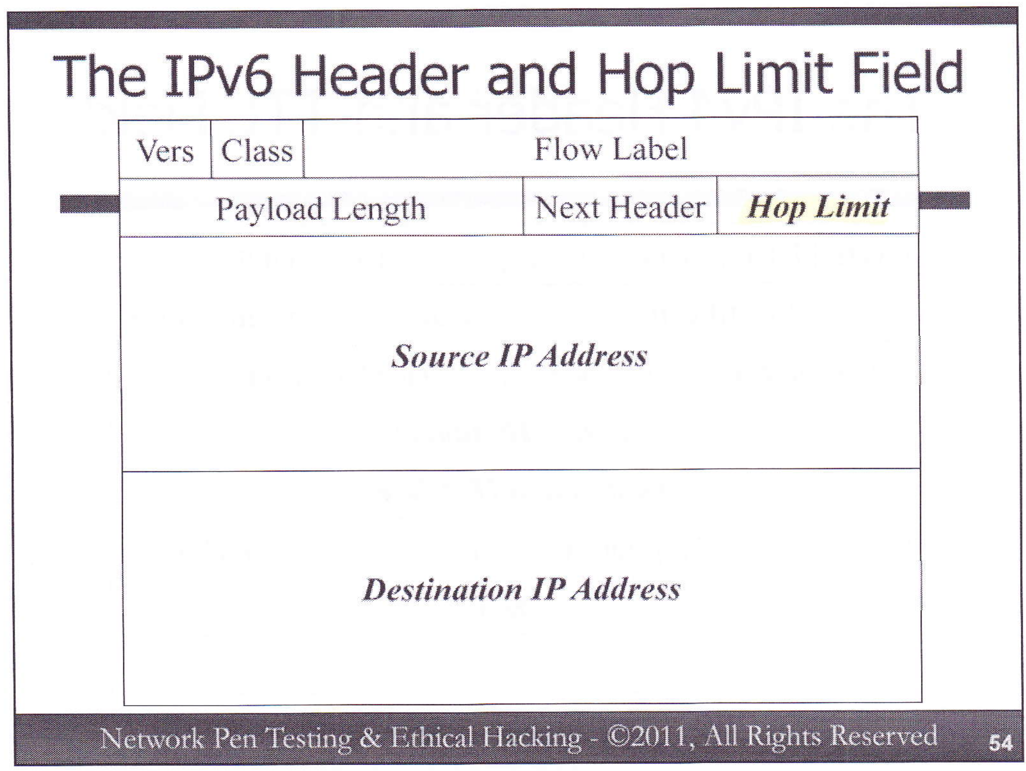
Vers	Hlen	Service Type	Total Length	
Identification			Flags	Fragment Offset
Time to Live	Protocol		Header Checksum	
Source IP Address				
Destination IP Address				
IP Options (if any)				Padding
Data				
.....				

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved

To understand how network tracing works, we need to analyze some of the fields of the IP packet header. This slide shows the IP version 4 (IPv4) header. Of particular interest to us now are the Time to Live (TTL), Source IP Address, and Destination IP Address fields, which we will use in determining the overall network topology. The source IP address is a 32-bit field indicating where the packet originated. This field will usually be set to the address of the machine running the scanning tools, unless we are using a technique that involves spoofing. The destination address is another 32-bits that identify where the network should carry this packet. During network sweeps, we often send large numbers of packets that vary this destination address.

The TTL field is 8-bits long and indicates how many hops this packet can travel before it must be discarded. When a router receives a packet, it is supposed to decrement the TTL field by one. When a given router decrements the TTL to zero, the router is supposed to drop the packet, and send a "TTL Exceeded in Transit" message (ICMP Type 11, Code 0) back to the source IP address of the discarded packet. The source address of this ICMP TTL Exceeded in Transit message is the router itself. This interesting TTL behavior allows us to perform network tracing, discerning the hops between the scanning machine and target systems.

Later in the class, we will look at some of the other fields of the IPv4 header.

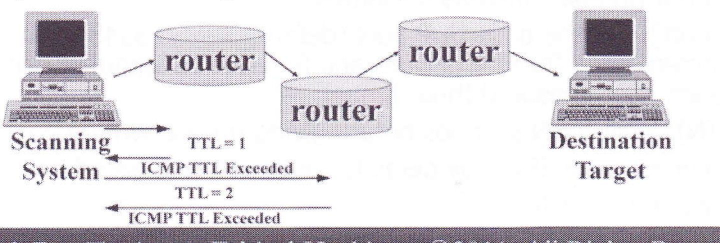


Here is the IPv6 header. First, note the massive size of the source and destination IP addresses, with each 128 bits in length. Further, notice that this packet structure is actually in many ways simpler than IPv4. For example, the fields associated with fragmentation (the IP Identification field, the fragment-associated flags Don't Fragment and More Fragment, and the Fragment Offset) are not present.

But, most important to us right now, there is a Hop Limit field, which behaves in a very similar way to the IPv4 TTL field. It's now named "Hop Limit" to remove any connotation of time from it, but it is still decremented by each router hop as the packet moves from its source to destination. Therefore, we can use it to determine the series of router hops between a source and destination.

Traceroute

- Discovers the route that packets take between two systems
- Helps a tester construct network architecture diagrams
- Included in most operating systems
  - Linux/Unix traceroute and traceroute -6
  - Windows tracert and tracert -6
- Sends packets to target with varying TTLs in the IP Header



Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 55

The traceroute technique uses this TTL behavior of routers to determine the addresses of routers between the scanning machine and a given target. On Linux and Unix machines, this technique is implemented in the traceroute command. On Windows, the tracert command provides similar functionality. Both traceroute and tracert support IPv4, and have an option on modern operating systems for using IPv6 if we invoke either command with a -6 flag. We'll discuss the differences between Linux/Unix traceroute and Windows tracert shortly.

But first, let's look at how both traceroute and tracert determine the hops between the scanning machine and the target. The scanning tool starts out by emitting a packet with the target machine's IP address in its destination field. The TTL of this first packet is very small: a value of one is inserted. When the first router receives this packet, it decrements the TTL to zero. Because the TTL is now zero, the router drops the packet, and sends an ICMP TTL Exceeded in Transit message back to the scanning tool. The source address of this ICMP packet is the first router. We now know the first router's IP address.

Then, the scanning tool sends another packet to the destination target's IP address, this time with a TTL of 2. The first router decrements the TTL to 1, and then routes the packet to the second router. The second router decrements the TTL to zero. Because the TTL has reached zero, the second router drops the packet, and sends an ICMP TTL Exceeded in Transit message back to the sender. We now know the second hop's IP address.

The traceroute tool proceeds in this fashion, measuring hop after hop, until it reaches the target itself. The target's response depends on the type of packet used by the traceroute tool. If a given hop doesn't return an ICMP TTL Exceeded in Transit message back (because it is configured to filter the inbound probe or omit the ICMP response), many traceroute tools will simply label that hop with a "*", meaning that no address information is known for it. In fact, if a given network device filters all ICMP messages going back, its hop and everything thereafter will be filled with a *.

Linux/Unix Traceroute

- Sends UDP packets with incrementing ports starting at base port of 33434, going up by one port for each probe packet sent (each hop measured three times)
- Some useful options:
  - -f [N]: Set the initial TTL for the first packet
  - -g [hostlist]: Specify a loose source route (8 maximum hops)
  - -I: Use ICMP Echo Request instead of UDP
  - -m [N]: Set the maximum number of hops
  - -n: Print numbers instead of names
  - -p [port]: Set the base UDP port (default base is 33434, which is incremented for first packet, and for each subsequent packet, with each hop measured three times)
  - -w [N]: Wait for N seconds before giving up and writing * (default is 5)
  - -4: Force use of IPv4 (by default, chooses 4 or 6 based on dest addr)
  - -6: Force use of IPv6

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 56

The Linux and Unix traceroute command utilized UDP messages with varying destination ports as its probe messages to elicit ICMP TTL Exceeded in Transit messages. As its starting point, traceroute begins with a UDP port of 33434, to which it adds one for each probe packet it sends. By default, each hop is measured three times. Thus, the first packet to measure the first hop has a TTL of 1 and a UDP port of 33434. The second packet measures the same hop, again with a TTL of 1, but this time a destination UDP port of 33435. The third packet again has a TTL of 1, but a UDP port of 33436. We then move on to the second hop, with a TTL of 2, and a UDP port of 33437.

The traceroute command supports some useful options, including:

-f [N]: This option sets the initial TTL of the traceroute to an integer N, thereby skipping over the first N-1 hops. If a tester wants to ignore their nearby network in tracerouting, they can set this value to skip some hops.

-g [hostlist]: Instead of having the network determine the routes that packets will take, the sender of a system can employ loose source routing, embedding the desired path of routers to take in the header of the IP packet itself. That way, the tester can control the flow of packets between some of the routers, measuring hops in between those routers specified. The traceroute command supports specifying up to 8 router hops.

-I: Use ICMP Echo Request messages as probes instead of UDP packets.

-m[N]: Set the maximum number of hops to measure (the default is 30).

-n: Don't resolve domain names, but print IP address numbers instead.

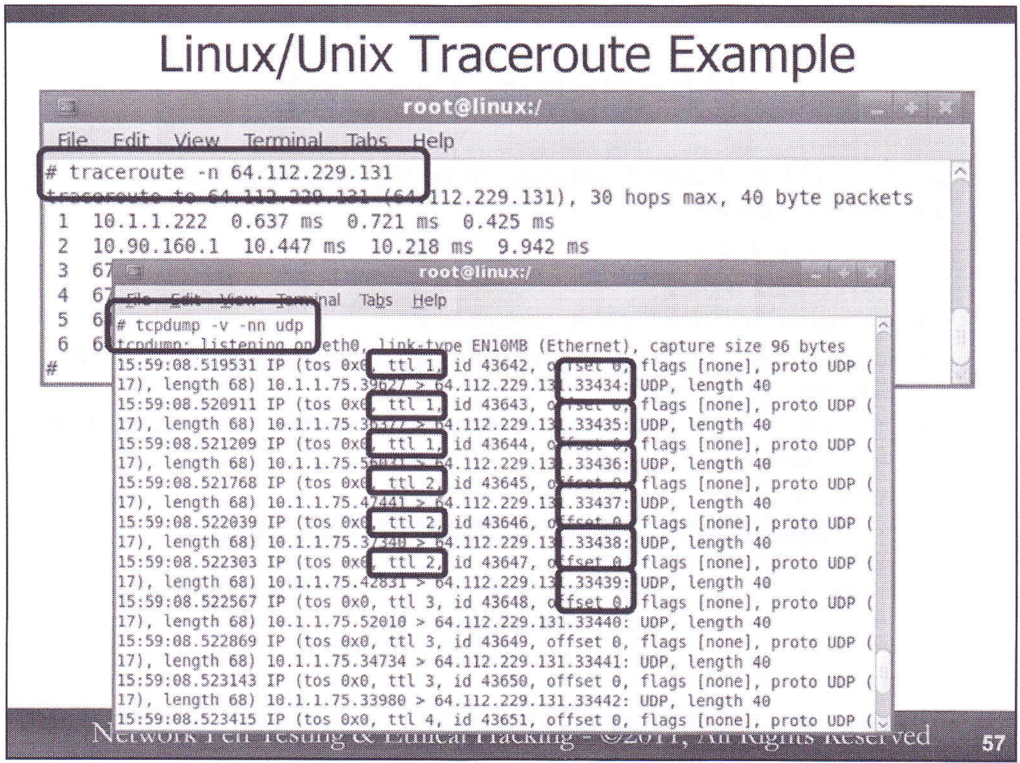
-p [port]: Set the base UDP port, which subsequent packets will increment, instead of the default of 33434.

-w [N]: Wait for an ICMP response for up to N seconds (default is 5 sec).

-4: Force the use of IPv4. By default, traceroute chooses IPv4 or IPv6 based on the destination address type provided. But, you can force it to use IPv4 with this option.

-6: Force use of IPv6.

For more options, please feel free to read the traceroute man page.



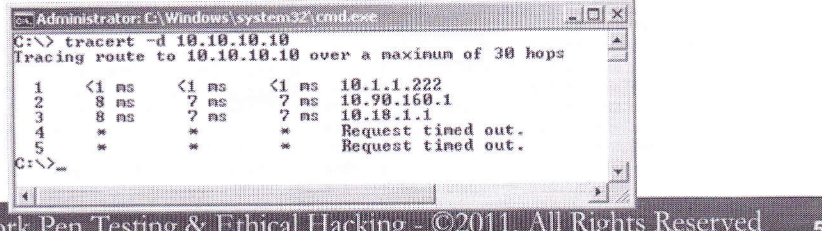
In this example of a Linux traceroute, we first started the tcpdump sniffer to verbosely (-v) use numbers instead of names (-nn) while printing out UDP packets (udp). We ran it verbosely so that tcpdump will show us the TTLs of packets.

We then ran the traceroute command, also configured to use IP address numbers instead of names (-n), to measure the router hops between the scanning machine and the target address of 64.112.229.131 (at the time, this was the address assigned to www.sans.org).

In the tcpdump output, we can see that the first three probe packets all have a TTL of 1, and destination UDP ports of 33434, 33435, and 33436. Next, we move onto the next hop, with a TTL of 2, and UDP ports of 33437, 33438, and 33439. Each of these TTLs and UDP ports are circled in the above packets.

It is vital to note that for the traceroute command to function using UDP, the network must transmit packets with these UDP ports toward the destination. If it does not, we can't measure those hops using this default invocation. We could use the –I option to send our probes via ICMP Echo Request messages.

- Sends ICMP Echo Request messages to target, starting with small TTLs and working upward
- Some useful options:
  - -d: Don't resolve names
  - -h [N]: Maximum number of hops (default is 30)
  - -j [hostlist]: Use loose source routing, with a space-separated list of router IP addresses (up to 9 max)
  - -w [N]: Wait for N milliseconds before giving up and writing a * (default is 4000)
  - -4: Force use of IPv4
  - -6: Force use of IPv6



Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 58

Next, let's look at the Windows tracert command, which has fewer options than the Linux/Unix traceroute command. By default, Windows tracert sends ICMP Echo Request messages probes, again varying the TTLs as before. Each hop is measured three times.

The following options can prove useful:

-d: Print IP addresses of discovered hops; don't resolve their names.

-h [N]: Measure only this number of hops. Give up if there are more than this number of hops between the scanning tool and the target. The default is a maximum of 30 hops.

-j [hostlist]: Use loose source routing, embedding a series of router hops in the IP header that should be used to carry the packet. The hostlist is a space-separated list of router IP addresses. Windows supports up to 9 router hops in its list.

-w [N]: Wait for N milliseconds for an ICMP TTL Exceeded in Transit message before giving up, printing a "*", and going to the next host. The default is 4000 milliseconds (4 seconds). Note that Windows tracert sets its timeout in milliseconds, while Linux/Unix traceroute uses seconds.

-4: Force use of IPv4.

-6: Force the use of IPv6.

In the example on this slide, we've done a tracert to 10.10.10.10. We've gotten results from the first three hops. For the next two hops, we did not receive a TTL Exceeded in Transit back within the 4 second timeout. In fact, we never received responses back from those hops, which are filtering either the inbound ICMP Echo Request or are blocking the response ICMP TTL Exceeded in Transit messages.

Other Traceroute Tools

- To conduct a traceroute, you have to be able to get a packet into the destination network...
  - ...and get an ICMP Time Exceeded message back
- If ICMP Echo Request is blocked, Windows tracert has problems
- If high UDP packets are blocked, Linux/Unix traceroute has problems
- We may need more flexible tracerouting options

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 59

To perform a traceroute, we need to be able to get our probe packets into the target network, so that we can elicit an ICMP TTL Exceeded in Transit message to receive back. If a router with ACLs, a firewall, or network-based IPS device blocks incoming UDP and ICMP Echo Request messages, the Linux/Unix traceroute and Windows tracert commands will not be able to do their work against the target environment. Are we out of luck?

No! We can use more flexible tracerouting tools beyond those built into the operating systems.

Layer Four Traceroute (LFT)

- Layer Four Traceroute (LFT) addresses this issue
  - Free at http://pwhois.org/lft/
  - Runs on Linux and Unix
- Supports a variety of Layer Four options for tracerouting
  - Use TCP (default), UDP (-u), or ICMP Echo Request (-p)
  - Choose destination port (-d [port]), default for TCP is 80
  - Choose source port (-s [port])
  - Set chosen length (-L [N]) including layer 3 and 4 header lengths
  - Looks up AS number (-A) using various whois servers
- Also support RFC 1393 Traceroute via IP options (-P)
  - Interesting, but support is not widely implemented in routers

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 60

Layer Four Traceroute (LFT) is a more flexible traceroute tool, which is not shackled to ICMP Echo Request and UDP for probe packets (although those are supported). This tool, which runs on Linux and Unix, supports TCP, UDP, and ICMP Echo Request messages for probes. TCP is the default, while UDP is invoked with the –u flag and ICMP Echo Request is activated with –p (for "ping").

Additionally, LFT lets a tester choose a destination port, setting it to something that the target network allows. By default, LFT uses TCP port 80 for probes, because any system running a web server on the default HTTP port can receive the packet. Alternatively, the tester could use –p 443 to use the HTTPS port, or any other port which might be permitted in.

Some networks only allow packets with certain source ports inbound, such as UDP 53 (so they can get responses from DNS servers). The LFT –s flag followed by a port number supports arbitrary source ports chosen by the attacker. If a source port isn't identified, a high numbered port (above 1024) is assigned.

The –L flag lets the tester specify the total length of the packet to be transmitted, which includes the IP header (Layer 3) and the TCP or UDP header (Layer 4). The payload of the Layer 4 packet is populated with padding to make the total exactly N bytes long. When invoked with the –A flag, LFT performs whois lookups to identify the Autonomous System (AS) number associated with each discovered router, so we can see which network cloud it belongs to and get a feel for when packets traverse between different ISP networks.

Finally, LFT supports an entirely different form of tracerouting. Instead of relying on sending a series of packets with incremental TTLs, RFC 1393 describes using a specialized IP Traceroute message, sent from the originator all the way to the destination. Each router hop along the way that supports this special kind of traceroute would respond with a special ICMP Traceroute message back to the sender, indicating its presence in the routing path. Also, the ICMP Traceroute response includes the hop count of the received IP traceroute message, so the sender can discern how many hops away each router is. This technique is more efficient than traditional tracerouting. If there are n router hops, this technique requires n+1 packets (the original probe, plus one response from each router hop). Traditional tracerouting requires 2n packets (one packet to each router, plus one response). Unfortunately, many routers do not respond to the IP Traceroute messages, making this technique of limited utility.

Web-Based Traceroute Services

- Instead of tracerouting from your address to the target, various websites allow you to traceroute from them to the target
  - In effect, you can traceroute from around the world...
  - ...By domain name or IP address
- Very useful in seeing if you are being shunned during a test!!
- Be careful with domain name, as that location may have a different IP address for that name
  - www.traceroute.org
  - www.kloth.net/services/traceroute.php
  - www.net.cmu.edu/cgi-bin/netops.cgi
  - www.tracert.com
- Realize that you are leaking some information to a third party... telling them that someone at your IP address has an interest in these target machines

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 61

Instead of running a local traceroute tool (such as traceroute, tracert, LFT, or 3D Traceroute), a tester could perform a traceroute using a web-based traceroute service. Several organizations provide a web server on the Internet that includes a form asking for a target IP address or domain name, as well as a geography (choosing from dozens of different countries) that the user would like to traceroute from. Upon receiving this information, the web server sends a request to an affiliated traceroute server in that given geographic location. The traceroute server performs the traceroute between it and the destination, returning the results to the web server, which forwards them back to the user's browser.

In this way, a tester can see what a traceroute against a target will look like from other parts of the world. These services are also helpful in determining if there is a localized outage or blockage on the network, or if the target system itself has gone down. For penetration testers and ethical hackers, these services are immensely valuable in differentiating whether a tester has been shunned by the target network administrators or automated detection technology, or if the target network or systems has gone down. If, at the start of a test, you can traceroute all the way to your destination, but during the test, you suddenly lose connectivity and traceroute ability, you can try tracerouting to the target using one of these services. If they can still reach the target, but you cannot, you either have a local network problem or have been shunned by the target. Please do note that any addresses used in these services can be recorded in their logs, so be careful in using them; you are revealing the IP addresses that you are testing to the organizations running these services.

Also, when using services like these, you may want to enter IP addresses of targets instead of domain names. If you enter a domain name for a traceroute server to test halfway around the world, that name may resolve to the IP address of a totally different system, one that you are not authorized to test. Thus, IP addresses are usually the best way to refer to targets with these web-based traceroute tools, unless you are specifically looking to see how a given domain name resolves in another part of the world.

It's important to note that, when you use such third-party external information sources, you are revealing to the people who run them that you have an interest in these target machines. You should always carefully consider the information you might be leaking to the third party while conducting a penetration test. Tracerouting usually doesn't contain very sensitive information, but other kinds of external lookups should be considered carefully so you can avoid violating your non-disclosure agreements.

Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

- Scanning Goals and Types
- Overall Scanning Tips
  - Sniffing with tcpdump
- Network Sweeping with Scapy
  - Scapy/tcpdump Exercise
- Network Tracing
- ***Port Scanning***
  - Nmap
  - Nmap Exercise
- OS Fingerprinting
- Version Scanning
  - Nmap -O -sV and Amap Exercise
- Vulnerability Scanning
  - Nmap Scripting Engine
  - NSE Exercise
  - Nessus
  - Nessus Exercise
  - Other Vuln Scanners
- Enumerating Users
  - Enumerating Exercise
- Netcat for the Pen Tester
  - Netcat Exercise

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 62

Our next topic is port scanning. We will use a variety of techniques, mostly centered around the mighty Nmap tool, to find open ports on the target machines. Each of these ports offers a potential vehicle for infiltrating the target environment. We want to use various tools to determine, with a high degree of certainty, which ports are open and which are closed. We'll send probe packets to the target machine, and, based on its responses, try to determine which ports are currently accessible to the tester on the target.

TCP vs. UDP

- Most services on the Internet are TCP or UDP
- Very different properties between these protocols, which impact our scanning
- TCP: Connection oriented, tries to preserve sequence, retransmits lost packets
- UDP: Connectionless, no attempt made for reliable delivery

Application	
TCP	UDP
Network Layer (IPv4 or IPv6)	
Data Link Layer	
Physical Layer	

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 63

To understand port scanning, we first need to discuss some protocol issues. Most services on the Internet use either TCP or UDP, which are carried end-to-end across the network using IP (either IPv4 or IPv6).

The Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are very different.

TCP is a connection-oriented protocol, which tries to ensure reliable, in-order delivery of packets. If packets are lost, TCP automatically retransmits them. If they arrive out of order, TCP will resequence them before handing them up to an application.

UDP is connectionless. The UDP software makes no attempt to associate streams of packets together. As far as UDP is concerned, each packet is completely independent, unrelated to other packets. No attempt is made by UDP for retransmission or resequencing. If a packet it lost via UDP, it's up to the higher layer application to resend it.

TCP Header

Source Port			Destination Port	
Sequence Number				
Acknowledgment Number				
Hlen	Rsvd	Control Bits	Window	
Checksum			Urgent Pointer	
TCP Options (if any)				Padding
Data				
…..				

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 64

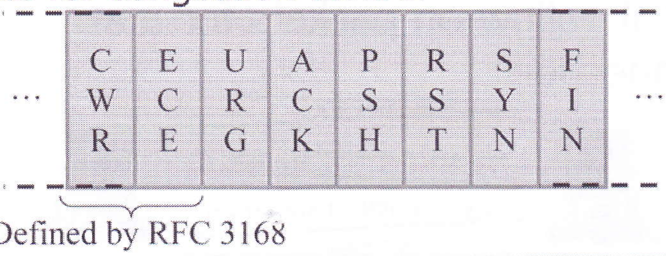
Here is the TCP header. Note that it includes a source port and destination port, each 16-bits in length. The source port is the port on the originating machine that emitted the packet. The destination port indicates the port on the target machine the packet should be delivered to.

We have a sequence number and an acknowledgment number, which allow TCP to track a series of packets to make sure they arrive reliably and in-order. If a packet is lost, TCP will retransmit it. If packets arrive out of order, TCP will adjust them to make sure they are delivered to the destination services in the proper order.

We also have the TCP Control Bits, which are incredibly important for tracking the state of a given TCP connection.

TCP Control Bits

- Control Bits are also known as "Control Flags" or "Communication Flags"
  - The RFC calls them Control Bits, though
- 6 traditional ones, with 2 newer extended ones for congestion control



Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 65

The TCP Control bits are sometimes called the "Control Flags" or "Communication Flags", but the RFC refers to them as the Control Bits. These bits in the TCP header help identify the state of the TCP connection and which components of the TCP connection the given packet is associated with. There are six traditional TCP Control Bits, with 2 newer extended ones defined by RFC 3168. These Control Bits provide numerous options for us to scan the target system and determine the status of its TCP ports. Each control bit can have a value of 0 or 1 (after all, each one is just one bit long). The six traditional control bits include:

- SYN: The system should synchronize sequence numbers. This Control Bit is used during session establishment.
- ACK: The Acknowledgment field is significant. Packets with this bit set to 1 are acknowledging earlier packets.
- RST:The connection should be reset, due to error or other interruptions.
- FIN: There is no more data from the sender. Therefore, the session should be gracefully torn down.
- PSH: This bit indicates that data should be flushed through the TCP layer immediately rather than holding it and waiting for more data.
- URG: The Urgent Pointer in the TCP header is significant. There is important data there that should be handled quickly.

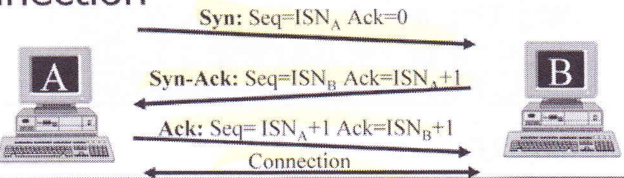
Note that this list doesn't show the Control Bits in the order in which they appear in the packet. Instead, we have sorted them in a more memorable fashion. The two additional control bits are CWR and ECE, which are:

- CWR: Congestion Window Reduced, which indicates that, due to network congestion, the queue of outstanding packets to send has been lowered.
- ECE: Explicit Congestion Notification Echo, which indicates that the connection is experiencing congestion.

Each of these control bits can be set independently of the others. Thus, we can have a single packet that is simultaneously a SYN and an ACK.

TCP Three-Way Handshake

- Every legit TCP connection starts with three-way handshake
- Used to exchange sequence numbers that will be applied in increasing fashion for all follow-on packets for that connection



Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 66

Every legitimate TCP connection begins with the TCP three-way handshake, which is used to exchange sequence numbers so that lost packets can be retransmitted and packets can be placed in the proper order.

If machine A wants to initiate a connection to machine B, it will start by sending a TCP packet with the SYN Control Bit set. This packet will include an initial sequence number (which we'll call ISNA because it comes from machine A), which is 32-bits long and typically generated in a pseudo-random fashion by the TCP software on machine A. The ACK number (another 32 bits in the TCP header) is typically set to zero, because it is ignored in this initial SYN. Some operating system variants may make this ACK number non-zero. Either way, it is ignored by the destination machine.

If the destination port is open (that is, there is something listening on that port), it must respond with a SYN-ACK packet back (a packet that has both the SYN and ACK Control Bits set at the same time). This packet will have a sequence number of ISNB, a pseudo-random number assigned by machine B for this connection. The SYN-ACK packet will have an acknowledgment number of ISNA+1, indicating that machine B has acknowledged the SYN packet from machine A.

To complete the three-way handshake, machine A responds with an ACK packet which has a sequence number of ISNA+1 (it's the next packet, so the sequence number has to change from the value in the original SYN packet). The acknowledgment number field is set to ISNB+1, thereby acknowledging the SYN-ACK packet.

We have now exchanged sequence numbers. All packets going from A to B will have increasing sequence numbers starting at ISNA+1, going up by a value of 1 for each byte of data transmitted in the payloads of A to B packets. Likewise, all responses back from B will have sequence numbers starting at ISNB+1 and going up for each byte of data from B to A. In essence, we have two streams of sequence numbers in this series of packets: one from A to B (originally based on ISNA) and the other from B to A (originally based on ISNB).

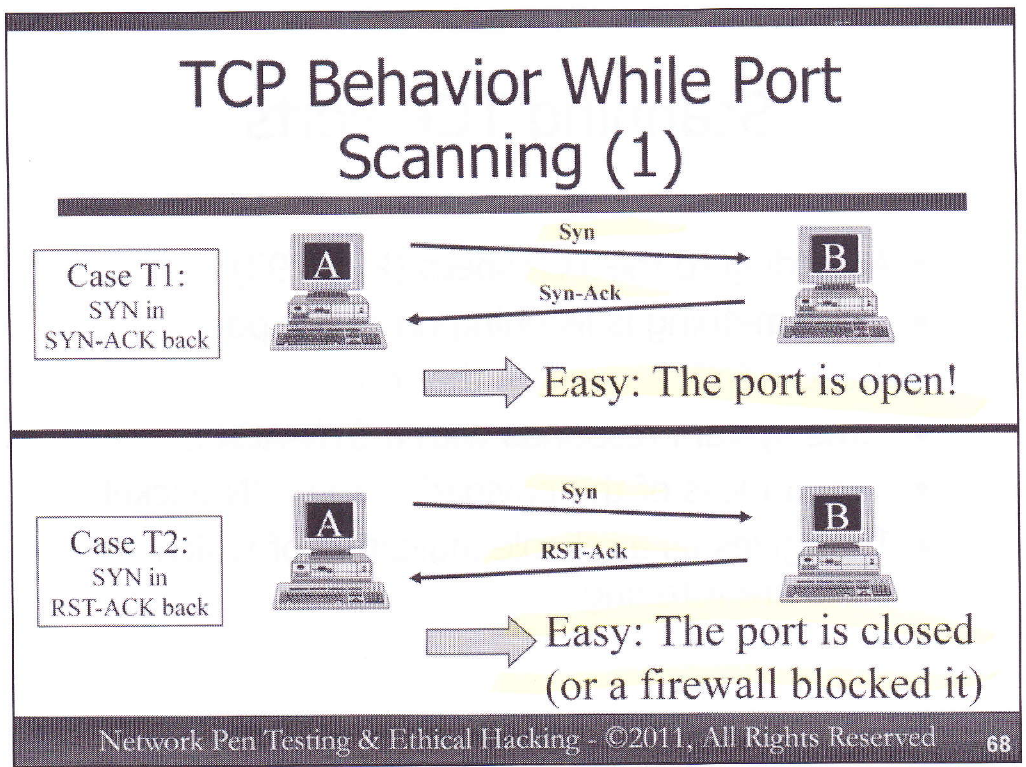
Scanning TCP Ports

- According to the TCP specs (RFC 793)...
- ...if something is listening on a TCP port...
- ...and a SYN arrives on that port...
- ...the system responds with a SYN-ACK...
- ...regardless of the payload of the SYN packet
- That gives us a reliable indication of which ports are listening

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 67

According to the original TCP specification (RFC 793), if a service is listening on a TCP port and a packet with the SYN Control Bit set arrives at that port, the TCP software must respond with a SYN-ACK packet. This response must be sent, regardless of the payload of the SYN packet itself.

Thus, even if we don't know what service is listening on the target port, we can still measure whether it is open by simply sending it a SYN packet. That gives us a reliable method for determining whether a TCP port is open or closed.

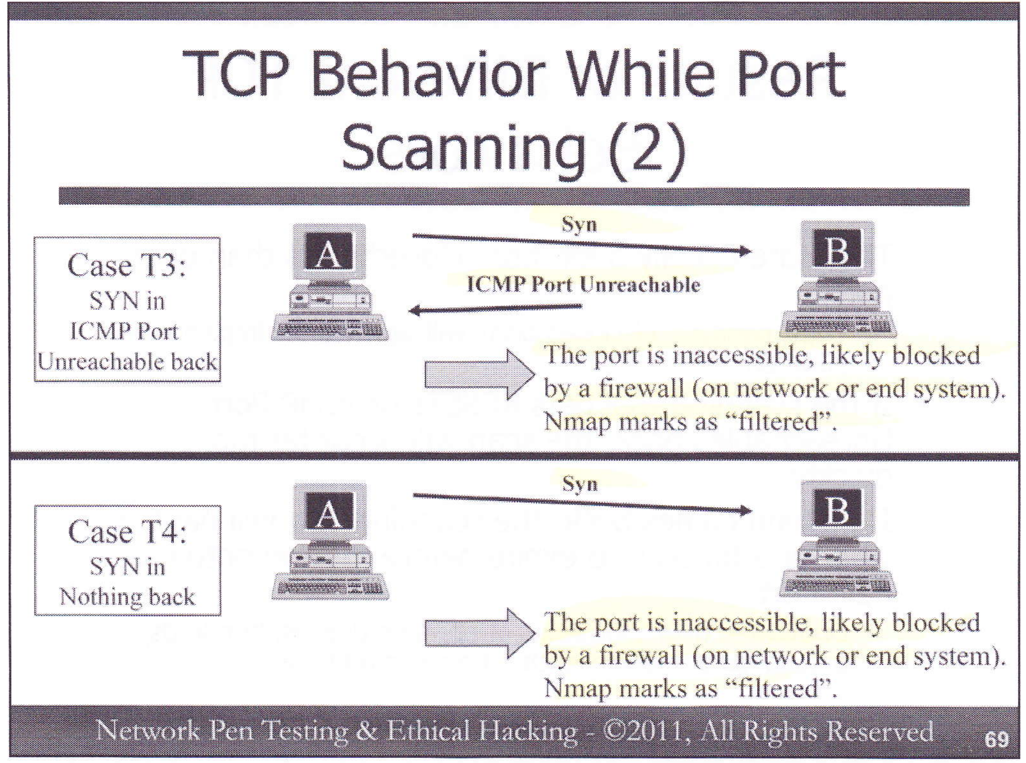


To understand the different options we have with TCP port scanning, let's explore TCP behavior under different conditions in more detail. Suppose machine A is being used to scan machine B to determine if a given port is open or closed. We start out by sending in a SYN packet. There are numerous possible responses:

Case T1: We receive a SYN-ACK response. This is an easy case, because we now know that the port is very likely open. There is a very small chance that there is some software on the target machine that is trying to trick us by responding with SYN-ACK packets from every possible TCP port on the box, but that is very unlikely.

Case T2: We receive a packet back with both the RST and ACK Control Bits set to 1. This RST-ACK packet represents another easy case: the port is likely closed, rejecting our connection request. There is also a chance that the RST-ACK came from a firewall instead of the target system. Either way, we cannot reach that port from where we sit, because it is effectively closed to us.

As a penetration tester or ethical hacker, we like to see packets with the RST Control Bit set to 1 coming back from closed ports during our scan because they make the scanning process significantly faster. Rather than having to wait for a timeout before we can move on to another port, we know very quickly that this port is closed and move on immediately upon receiving the packet with the RST Control Bit set to 1.



Case T3: We send in a SYN packet, and get an ICMP message back, such as an ICMP Port Unreachable message. The port is inaccessible to us, likely because it is blocked by a firewall which is creating the ICMP message. If the message is coming from the target machine itself, a local firewall on the machine (such as IPtables) is likely formulating the ICMP packet. Nmap marks this status as "filtered".

Case T4: We send in a SYN packet, and get nothing back. Nmap will try to retransmit the packet, but if nothing is received back within a certain timeout, the port will be marked as "filtered" as well. In all likelihood, either there is nothing listening on the end system (which has been configured via a personal firewall to silently drop all packets to closed ports) or a firewall is blocking our inbound SYN packet (again, silently rejecting it).

Each of these four cases is summarized well in the Nmap man page, which states:

"This technique is often referred to as half-open scanning, because you don't open a full TCP connection. You send a SYN packet, as if you are going to open a real connection and then wait for a response. A SYN/ACK indicates the port is listening (open), while a RST (reset) is indicative of a non-listener. If no response is received after several retransmissions, the port is marked as filtered. The port is also marked filtered if an ICMP unreachable error (type 3, code 1,2, 3, 9, 10, or 13) is received."

Results of Different TCP Behaviors

- There are usually a lot more closed ports than open ports
  - Thus, behavior of closed ports will significantly impact scan duration
- If the scanning tool gets RESETs or ICMP Port Unreachables back, the scan will occur far more quickly
- If nothing comes back, the scanning tool will have to wait for a timeout to expire before moving onto the next port
  - Duplicated over thousands of ports on dozens, hundreds, or thousands of machines, that time can add up!

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 70

When doing a port scan, you usually find far more closed ports than you do open ports. There are 65,536 possible TCP ports, and most systems have only a handful of ports open. Therefore, from a timing perspective, the behavior of the tens of thousands of closed ports could seriously slow down a scan. If the target sends back RESETs or ICMP Port Unreachables, our scan can occur more quickly, since we don't have to wait for a timeout to expire.

But, if nothing comes back, such as in case T4 that we discussed earlier, we have a problem for large-scale scans because it chews up a significant amount of time as the tool has to wait for a timeout to expire before it determines the state of this port. It may take 12 to 24 hours or more to conduct a port scan of all TCP ports when nothing comes back, and that is to scan just a single host.

UDP Header

Source Port	Destination Port
UDP Message Length	UDP Checksum
Data	
…..	

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved

Here is the UDP header. Note its relative simplicity when compared to the TCP header. We have a source port and destination port (each 16 bits in length, giving us potential values of between 0 and 65,535). We also have a message length and a checksum.

Specifically, note that there are no Control Bits in UDP, nor is there a sense of the "status" of a "connection". Because of these characteristics, our options for scanning UDP ports are far more limited than they are for TCP port scanning.

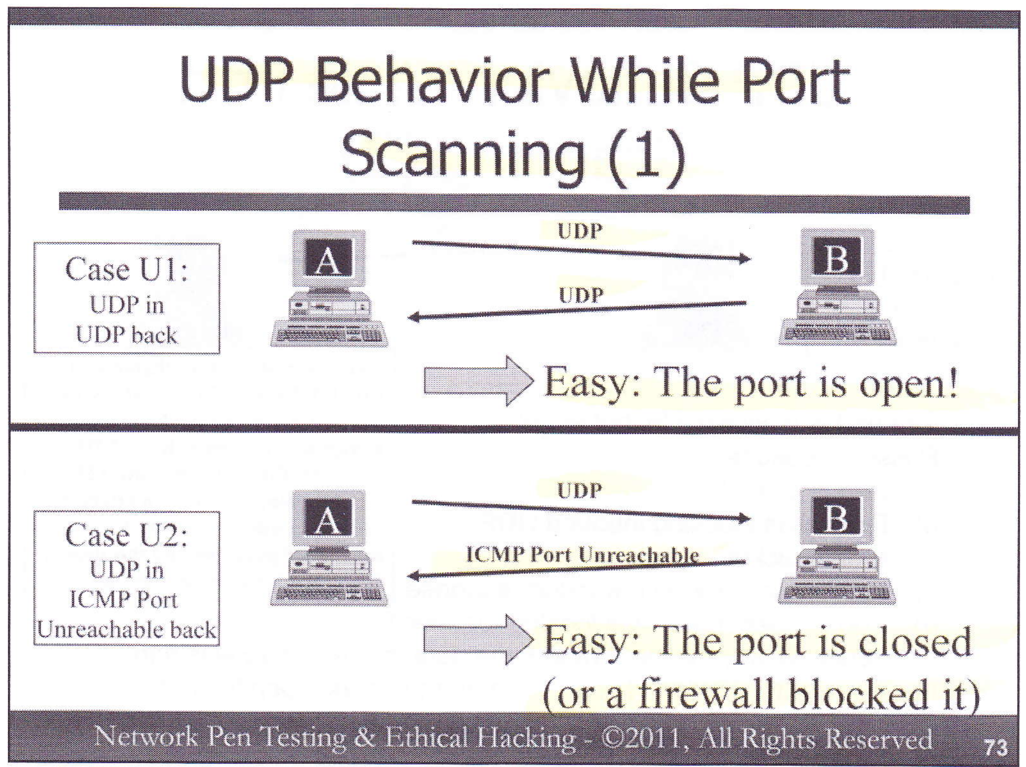
Scanning UDP Ports

- UDP is a far simpler protocol, without tracking of state of a "connection"
  - There is no connection with UDP
- Less options for scanning
- Often, slower scanning
- And, less reliable scanning

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved

UDP is a connectionless protocol. There is no concept within UDP of the state of a connection, as there is with TCP and its sequence numbers and window sizes. From a protocol perspective, UDP moves independent datagrams between systems.

Because there are no Control Bits in UDP, we have far fewer options for scan types. We can't vary the Control Bits to play with different target behavior to discern whether ports are open or closed. Because of this, UDP port scans are less reliable and often slower than TCP scans, for reasons that we'll cover shortly.



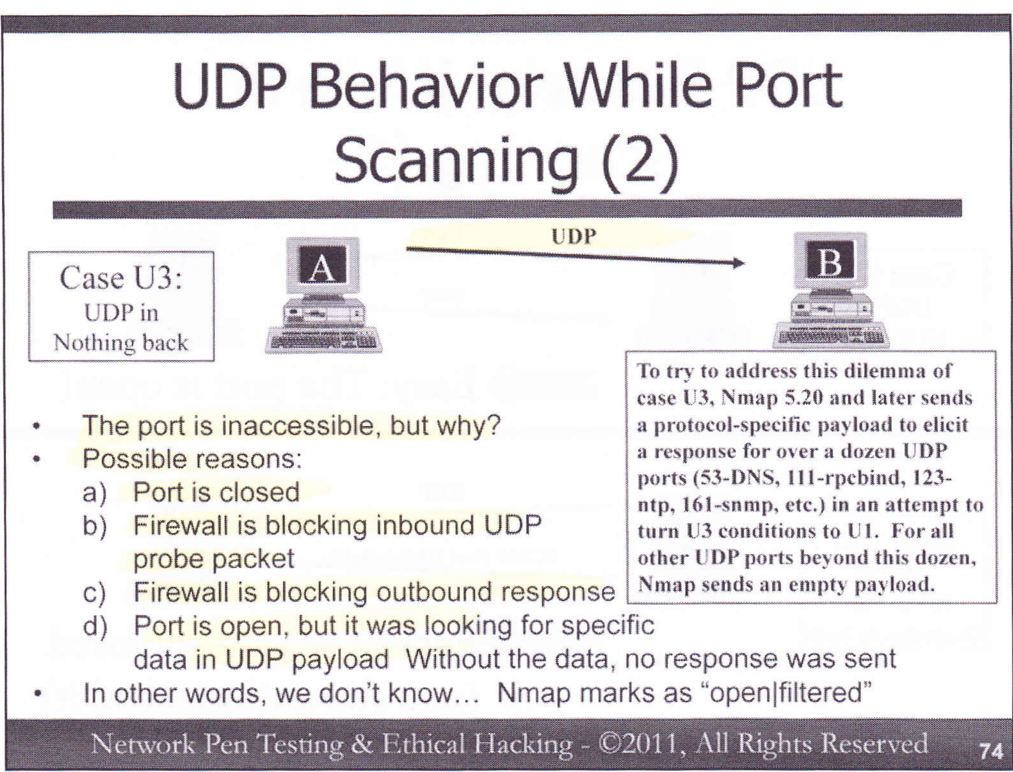
To see why UDP scanning is less reliable and often slower than TCP scanning, consider the cases that could occur when we perform a UDP port scan:

For each of these cases, the scanning system (System A in the figure) sends a UDP packet to the target machine (System B). With most port scanning tools (including Nmap), an empty UDP datagram is sent (with no payload).

Case U1: The target machine responds with a UDP packet. This is an easy case – something on the target machine received our UDP packet and responded to us. Thus, we can be fairly confident that there is something listening on that port on the target, so the port is open. Nmap lists the port as open.

Case U2: The UDP packet we send to the target may result in an ICMP Port Unreachable message coming back. This is an easy case for determining the status of the port as well, because we can be fairly certain that the port is closed. Nmap lists the port as closed. Unfortunately, some target systems rate-limit the number of ICMP Port Unreachable messages they send, specifically Linux and Solaris. The Linux 2.4 kernel, for example, will only send one ICMP Port Unreachable message per second. Thus, we have to go relatively slower in our UDP scans to make sure we allow adequate time for the ICMP Port Unreachable to come back.

By the way, there are variants of U2 in which the target system sends other ICMP message types back instead of "Port Unreachable" (Type 3, Code 3). According to the Nmap man page, "If an ICMP port unreachable error (type 3, code 3) is returned, the port is closed. Other ICMP unreachable errors (type 3, codes 1, 2, 9, 10, or 13) mark the port as filtered."



Now we get to the hard case.

Case U3: We send in a UDP packet, and we get nothing back. There are numerous possible reasons for this, including:

- The port is closed
- A firewall is blocking the probe packet inbound on its way to the target
- A firewall is blocking the response on its way back to us
- The port is open, but the service listening on the port was looking for a specific payload in the inbound UDP packet. We didn't include any payload, so it silently ignored us.

That last case is incredibly common. Nmap labels the result on its output as "open|filtered", which for UDP means that Nmap doesn't know whether the port is open or closed.

And, for that reason, UDP port scanning is less reliable than TCP port scanning. With TCP, according to the protocol spec itself, if we send a SYN packet to an open port, the target system must respond with a SYN-ACK, regardless of the payload of our SYN. That behavior gives us the assurance that the TCP port is open. We don't have that behavior and the resulting assurance with UDP, making it less reliable. Also, because we have to wait longer for the ICMP Port Unreachable messages, we have to go slower than we might with TCP.

To try to address this dilemma of case U3 and make UDP port scanning more reliable, Nmap 5.20 and later sends a protocol-specific payload to elicit a response for over a dozen UDP ports (53-DNS, 111-rpcbind, 123-ntp, 161-snmp, etc.) in an attempt to turn U3 conditions to U1. By sending a proper payload for a given layer-7 application that is designed to elicit a response, the target machine is more likely to send back a UDP packet, giving us a more reliable indication of whether the port is open or not (case U1). For all other UDP ports beyond this dozen or so port numbers, Nmap sends an empty payload, still resulting in a lot of case U3 conditions. Still, for the most common UDP ports in a production environment, this is a very good feature for identifying UDP-based services using their standard port.

Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

- Scanning Goals and Types
- Overall Scanning Tips
  - Sniffing with tcpdump
- Network Sweeping with Scapy
  - Scapy/tcpdump Exercise
- Network Tracing
- Port Scanning
  - ***Nmap***
  - Nmap Exercise
- OS Fingerprinting
- Version Scanning
  - Nmap -O -sV and Amap Exercise
- Vulnerability Scanning
  - Nmap Scripting Engine
  - NSE Exercise
  - Nessus
  - Nessus Exercise
  - Other Vuln Scanners
- Enumerating Users
  - Enumerating Exercise
- Netcat for the Pen Tester
  - Netcat Exercise

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 75

The most popular port scanner in the world is Nmap. Professional penetration testers and ethical hackers around the globe rely on this incredibly flexible and high-quality tool. In this section, we'll discuss some of the most useful features of Nmap for penetration testers and ethical hackers.

Even if you've run Nmap before, pay special attention to some of the new and more subtle features of Nmap that we'll address. In the past year, Nmap has been going through rapid change, with useful new features released on a regular basis. Understanding these features is important so that we can benefit from them in improving the accuracy and efficiency of our penetration testing and ethical hacking regimens.

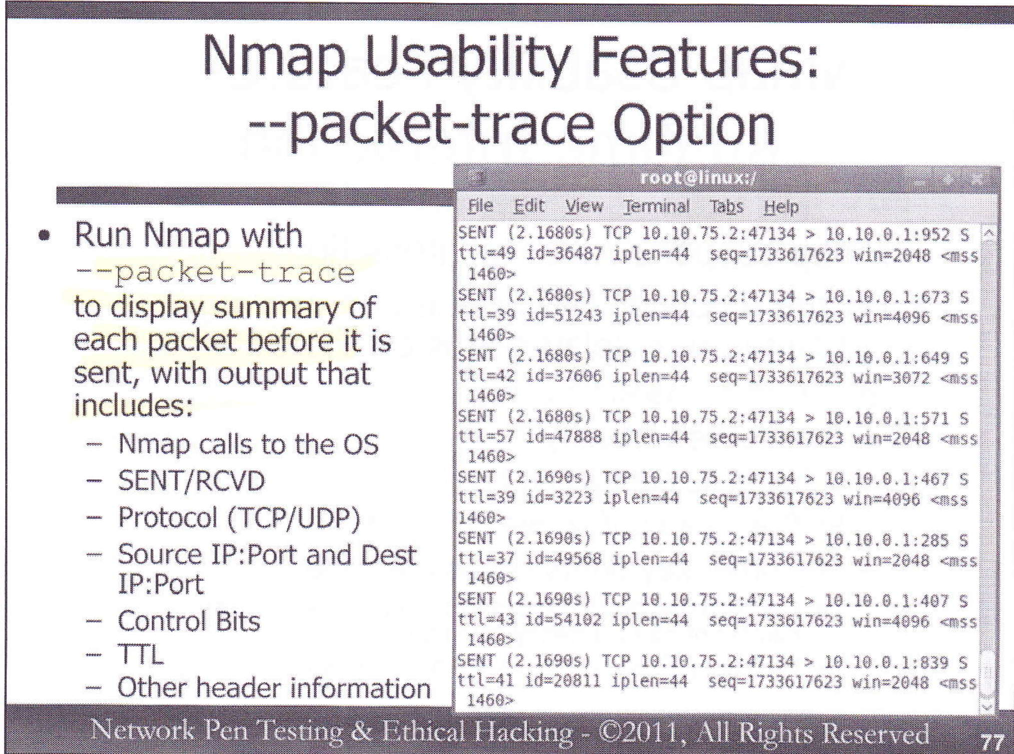
Nmap Port Scanner

- Written and maintained by Fyodor
  - Very popular, located at www.insecure.org and www.nmap.org
- Not *just* a port scanner
  - Port scanning is its focus
  - But, has been extended into a general-purpose vulnerability scanner via Nmap Scripting Engine (NSE)
    - More on that later

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 76

The Nmap tool is a critical element in the toolbox of most penetration testers and ethical hackers. Written and maintained by Fyodor, with a constant supply of updates and tweaks from an active group of contributors to this open source project, Nmap is primarily a port scanner, showing which TCP and UDP ports are open on a target system.

But, Nmap is not just a port scanner. It also provides numerous other features, including ping sweeps, operating system fingerprinting, tracerouting, and much more. With the Nmap Scripting Engine (NSE), Nmap can be extended to become a general purpose vulnerability scanner as well. We'll look at each of these features, building up to an exercise that analyzes the capabilities and results of NSE.



When using Nmap, it can be helpful to have the tool itself display a summary of the packets that it sends in real-time. When invoked with the --packet-trace feature, Nmap does just that. It displays various status messages on its output, including some of the calls it makes into the operating system, such as the connect() call that is made during TCP Connect scans (which we'll discuss later). It shows whether a given packet is sent or received, the protocol it used (TCP or UDP), and the source and destination IP addresses and ports. It also shows the control bits (the S in the screenshot above indicates a SYN packet). It also displays other header information, such as the IP Time-to-Live (TTL), the TCP Sequence number, etc.

The Nmap command line sequence that resulted in the screenshot on this page was:

```
# nmap -PN -sS 10.10.0.1 -p 1-1024 --packet-trace
```

The -PN indicates that we don't want to ping the target system, we just want to scan it.

The –sS indicates that we want a SYN scan (also known as a Stealth Scan or a Half-Open Scan).

The –p 1-1024 tells Nmap to scan ports 1 through 1024 only.

And, the --packet-trace makes Nmap display the status and packet summary information.

Nmap Usability Features - Runtime Interaction

- Nmap supports runtime interaction
- Hit the following keys while it is running to get Nmap to display status on the screen:
  - p = Turn on packet tracing
  - v = Increase verbosity
  - d = Increase debugging level
  - Shift with any of above inverts it
  - Any other key prints status message:
    - Elapsed time, hosts completed so far, number of hosts up, number of hosts currently being scanned
    - Percentage done, estimate of amount of time remaining

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 78

If a user forgets to invoke Nmap with the --packet-trace option, they can turn it on after invoking Nmap. While Nmap is running, the user can hit the p key to turn on packet tracing. Nmap will print on the screen a summary of each packet sent.

Furthermore, the user can hit any key to print a status message showing the elapsed time of the run so far, the number of hosts it has completed scanning, the number of hosts up, and the number of hosts currently being scanned in parallel. But, the best part of this output is an estimate of the time remaining for the given scan.

Additionally, the v and d keys increase the verbosity and debug modes, respectively.

Holding down the shift key with p, v, or d inverts the function (i.e., SHIFT-p turns off packet tracking, while SHIFT-v lowers verbosity). In other words, if you invoke Nmap with the --packet-trace option, hitting the Shift and p keys while it is running will turn off packet tracing.

Controlling Scan Speeds with Nmap's Timing Options

- By default, Nmap has a dynamic timing model
  - Adapts scan timeouts based on performance of initial packets
- Furthermore, Nmap has various options for scan speed built-in, invoked with the –T syntax

```
# nmap -T [timing_option] [other options]
```

  - 0: Paranoid – Waits 5 minutes between packets, scans serially
  - 1: Sneaky – 15 seconds between packets, scans serially
  - 2: Polite – 0.4 seconds between packets, scans serially
  - 3: Normal – Default, designed to not overwhelm network or miss targets/ports, scans in parallel (using -T3 changes nothing, because it is the default)
  - 4: Aggressive – Waits only 1.25 seconds for probe response, scans in parallel
  - 5: Insane – Spends up to 15 minutes per host (gives up on that host and moves on if scan taking longer for it), waits only 0.3 seconds for probe response, scans in parallel

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 79

Nmap supports a variety of scanning speeds built-in. They can be invoked at the command-line by adding a –T <Paranoid|Sneaky|Polite|Normal|Aggressive|Insane> to the Nmap invocation. Alternatively, they can be referred to by numbers, with 0 meaning Paranoid (-T0) and 5 meaning Insane (-T5).

Paranoid mode is designed to scan so slowly that it will avoid detection by IDS systems, falling outside of their time-sampling window. It sends packets approximately every 5 minutes. No packets are sent in parallel with a Paranoid scan; they are sent one at a time.

Sneaky mode sends packets every 15 seconds. As with Paranoid, no parallel sending is used with the Sneaky option.

Polite mode sends a packet every 0.4 seconds, again one-by-one (no parallel sending). This mode is designed to lower the load on a network and prevent targets and network equipment from crashing.

Normal mode is designed to run quickly, but without overwhelming the sending machine or the network. This mode, which is the default behavior of Nmap, is also designed to maximize the chance of successfully identifying target machines and open ports. It will scan in parallel, sending multiple packets to multiple target ports simultaneously. Invoking Nmap with the -T3 option actually doesn't change in any way the fashion that Nmap runs, because it simply selects the default timing model, which is used even if you don't specify -T3.

Aggressive mode will never wait more than 1.25 seconds for a response, and it scans in parallel. The Nmap documentation recommends using -T4 for "reasonably modern and reliable networks". However, some penetration testers use the default normal mode (-T3) to lower the chance of impairing the target network.

Insane mode spends only up to 15 minutes per target host, and waits only 0.3 seconds for a response to each probe. If Nmap cannot complete a given host within 15 minutes, it gives up on that host (with the scan only partially completed) and moves on to the next host. For protocols such as UDP or large-scale scans of ports for TCP services, that's not a lot of time to get results back, so it should only be used on a very fast network. Furthermore, sending packets at that clip could impact the target system or network equipment between the scanning machine and the target.

Finer-Grained Nmap Timing Options

- To get even more control over timing, Nmap supports these options (timeouts are in milliseconds):
  - --host_timeout: Max time spent on single host before moving on; default is no host timeout
  - --max_rtt_timeout: Max time to wait for probe response before retransmitting or timing out; default is 9 seconds
  - --min_rtt_timeout: To speed up a scan, Nmap measures timing of target and lowers timeouts to match its network behavior, speeding up a scan but possibly missing responses; this option can be set so that timeouts don't go below a given value
  - --initial_rtt_timeout: Sets the initial timeout for probes, which will be lowered automatically as Nmap measures the network performance of a target; default is 6 seconds
  - --max_parallelism: Sets the number of probes Nmap will send in parallel (1=serial)
  - --scan_delay: Sets minimum time Nmap waits between sending probe packets

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 80

Beyond its six levels of pre-canned timing options, Nmap also supports various finer-grained timing options. Most penetration testers use the default options, but some people fine tune their scans based on careful observations and measurements of the timing associated with the target network. The finer-grained Nmap timing options include:

--host_timeout: The maximum time in milliseconds spent on single host before moving on. The default is no host timeout.

--max_rtt_timeout: The maximum time to wait for probe response before retransmitting or timing out. The default is 9000 milliseconds.

--min_rtt_timeout: To speed up a scan, Nmap measures the timing of responses from a target and lowers its timeouts to match that target's network behavior, speeding up a scan but possibly missing responses on networks with high variability in their performance characteristics. This option can be set so that timeouts don't go below a given value, helping to ensure reliability of results.

--initial_rtt_timeout: This value sets the initial timeout for probes, which will be lowered automatically as Nmap measures the network performance of a target. The default is 6000 milliseconds.

--max_parallelism: This option sets the number of probes Nmap will send in parallel (with 1 indicating a serial scan with only one outstanding probe at a time).

--scan_delay: This value sets the minimum time Nmap waits between sending probe packets.

Nmap Output Options

- Nmap output is displayed on the screen in a handy format for humans
- We can also store output in a file by specifying an output type followed by a file name:
  - -oN [filename]: Store output in normal format, recording data typically displayed on screen
  - -oG [filename]: Store output in greppable format, with one line per host indicating all open ports, their status, their associated service, etc.
    - Very useful as input to other tools
  - -oX [filename]: Store output in XML format
  - -oS [filename]: Store output in script kiddie format (make "Elite speak" substitutions of O->0, mixed case, etc... not very useful, but sometimes comical)
  - -oA [basename]: Store in all three major formats (Normal, Greppable, and XML) at once, using basename.nmap, basename.gnmap, and basename.xml as file names

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 81

Nmap displays results to the screen in an easy-to-read, human-friendly format, indicating which hosts were scanned and the open ports and services associated with each host. Additionally, we can have Nmap store its results in a file by specifying various options at the command line prefaced with a dash and lower-case o (for "output").

The -oN [filename] option will store the normal human-readable output typically displayed on the screen in a file called "filename".

The -oG [filename] specification is highly useful, as it causes Nmap to store its results in a greppable format, with one target machine per line with each open port and its associated service all on that line in a comma and slash (/) separated list. Several other tools which rely on Nmap output (such as the Amap tool we'll discuss a little later) as well as Metasploit rely on this greppable format.

The -oX [filename] option causes Nmap to place its results in an XML format, which may be used as an import option for other tools.

The -oS [filename] option creates script-kiddie style output, which can be fun for laughs, but isn't terribly useful. O's become zeros, S's become dollar-signs, and mixed case prevails in this rather unreadable tongue-in-cheek format.

And, finally, to cover all bases, the -oA [basename] syntax tells Nmap to create normal, XML, and greppable output in three files, named basename.nmap, basename.gnmap, and basename.xml.

To make sure your results are as usable as possible, it often makes sense to specify -oA and a basename that includes the target IP address range and scan type so that the three files of output are immediately recognizable in the file system.

Nmap and Address Probing

- By default, Nmap probes a target address before scanning it
  - For UID 0 users, Nmap sends:
    - If on same subnet as Nmap box, just send ARP request
    - If on different subnet, send ICMP Echo Request, and...
    - TCP SYN to port 443, and...
    - TCP ACK to port 80, and...
    - ICMP Timestamp Request (Type 13)

    All sent immediately, not waiting for response between each packet
  - For non-UID 0 users, Nmap initiates 3-way handshake by sending:
    - TCP SYN to port 80, and...
    - TCP SYN to port 443
    - Note that no ICMP is used
  - These packet combinations are based on statistical analysis of actual systems that respond on large networks and the Internet
- Nmap with the –PN option (same as –P0) will not ping a target before scanning it

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved

Nmap is not just a port scanner, although that is one of its primary purposes. The tool does offer numerous other features, such as identifying which addresses are in use on a target network. In other words, Nmap can be used for network sweep scans.

By default, Nmap automatically probes a target address before it port scans it. The particular method of probing to determine whether the address is in use depends on whether Nmap has been invoked with UID 0 (root-level) privileges. If Nmap was invoked as root, it first checks if the target IP address to be scanned is on the same subnet as the machine running Nmap. If it is, Nmap sends an ARP request, waiting for an ARP response. If it gets an ARP response from the address on the same subnet, Nmap knows that the given address is in use. If the target address is on a different subnet (and Nmap was invoked with UID 0 privileges), Nmap then sends an ICMP Echo Request message (a standard ping), a TCP SYN packet to port 443, a TCP ACK packet to port 80, and an ICMP Timestamp Request message (Type 13) to the target address. Nmap sends all of these packets one right after another, not waiting for responses between them. After this small burst of packets, Nmap waits to determine whether any of them elicit a response from the target.

If Nmap is invoked without UID 0 privileges, it simply asks the OS to initiate connections, resulting in the sending of TCP SYN packets to port 80 and 443 and waits for a response. Without root privileges, Nmap cannot craft the specialized packets needed for the more complex and accurate probing done with root privileges. It's worthwhile noting that without UID 0, Nmap doesn't even send an ICMP Echo Request message to identify a host. It just uses two TCP SYNs.

The specific probe packets were chosen by the Nmap development team based on statistical analyses of scans of large networks, focusing probes on those packets most likely to get a valid response.

By default, Nmap will only scan the target if it gets a response to the messages described above. If it doesn't get a response, Nmap gives up on that address. To make Nmap skip this ping phase, the –PN option can be used. This –PN option does the same thing as –P0 option. More recent reference works on Nmap refer to the –PN option to minimize confusion between –P0 (zero, used for not pinging) and –PO (the capital letter O, used for IP Protocol Pings, which send a specified IP packet with a given number in the protocol field of the IP header).

Nmap and Network Sweeping

- Beyond pinging an individual host before port scanning, Nmap can also just probe for target hosts, launching a network sweep scan

```
# nmap -sP [options]
```

  - By itself, -sP uses default probing behavior listed on previous slide
  - Besides the default probes, there are numerous other options for network sweeping to determine which addresses are in use

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 83

Beyond this probing of an individual host before port scanning it, Nmap also offers network sweep capabilities to identify where hosts are located in a target network address range. The simplest version of an Nmap network sweep is initiated with the –sP option. With no further options, this simple syntax, as you might expect, performs the default probing behavior described on the previous slide.

Beyond this default behavior, Nmap supports numerous other probe types for network sweeping, which we'll explore in detail next.

Nmap Network Sweeping Options

- Attackers will choose a network sweep option based on what is allowed into the target network, measured by sending test probes using different protocols
- Nmap offers the following network sweep types:

-PN: Don't ping (also –P0)

-PB: Same as default, use ICMP Echo Request, SYN to TCP 443, ACK to TCP 80, and ICMP Timestamp Request (if UID 0)

-PE (formerly –PI): Send ICMP Echo Request (ICMP type 8)

-PS[portlist]: Use TCP SYN to specified ports in the port list (e.g., -PS80)

-PP: Send ICMP timestamp request (ICMP type 13) to find targets

-PM: Send ICMP address mask request (ICMP type 17) to find targets

-PR: Use ARP to identify hosts (only works with hosts on same subnet)

- Used by default for targets in the same subnet as scanning host

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 84

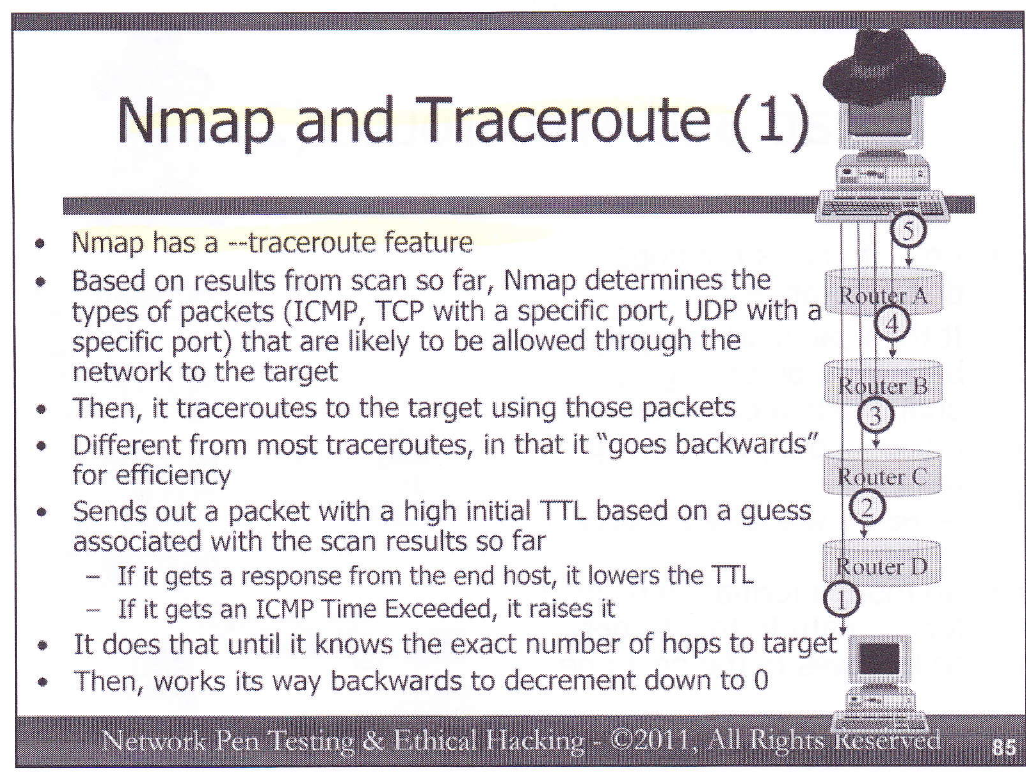
Here are the other ping sweep options for Nmap. The attacker will choose an appropriate option based on what is allowed into the target network. If a network firewall only blocks some ICMP types but not others, we might still be able to identify hosts on the other side.

As we've discussed, the –PN option tells Nmap not to ping at all. Some of the other useful options for a ping sweep include:

> -PB, which is the same as the default Nmap behavior for probing a target (if running with UID 0, send an ICMP Echo Request, a SYN to TCP port 443, an ACK to TCP port 80, and an ICMP Timestamp Request)
>
> -PE, which sends only an ICMP Echo Request message (formerly –PI)
>
> -PS[portlist], which sends a TCP SYN packet to each port in the port list. There is no space between the –PS and the port list. A useful port list is -PS22,25,80,135,139,443,445 which would identify systems using standard ports for Secure Shell, Simple Mail Transfer Protocol, HTTP, DCE Endpoint, NetBIOS Session, HTTPS, and Microsoft's SMB protocols, respectively. Note that Nmap identifies a host whether SYN/ACK or RESET packets come back. Either indicates that a target host responded.
>
> -PP, which sends ICMP Timestamp query messages.
>
> -PM, which sends ICMP Address Mask queries.
>
> -PR, which sends only ARP messages to identify hosts on the same subnet as the machine running Nmap.

As we have seen, that last one (-PR for ARP scanning) is used by default when Nmap determines that a host is on the same subnet as the machine on which Nmap is running. There's no sense doing a standard ping -- sending an ARP, and waiting for an ARP response, followed by an ICMP Echo Request, and waiting for its response – when the target is on the same subnet as the scanning machine. The ARP and ARP response suffice to tell us that there is a target host at the given address.

Nmap includes options beyond this list as well, but these are some of the most useful.



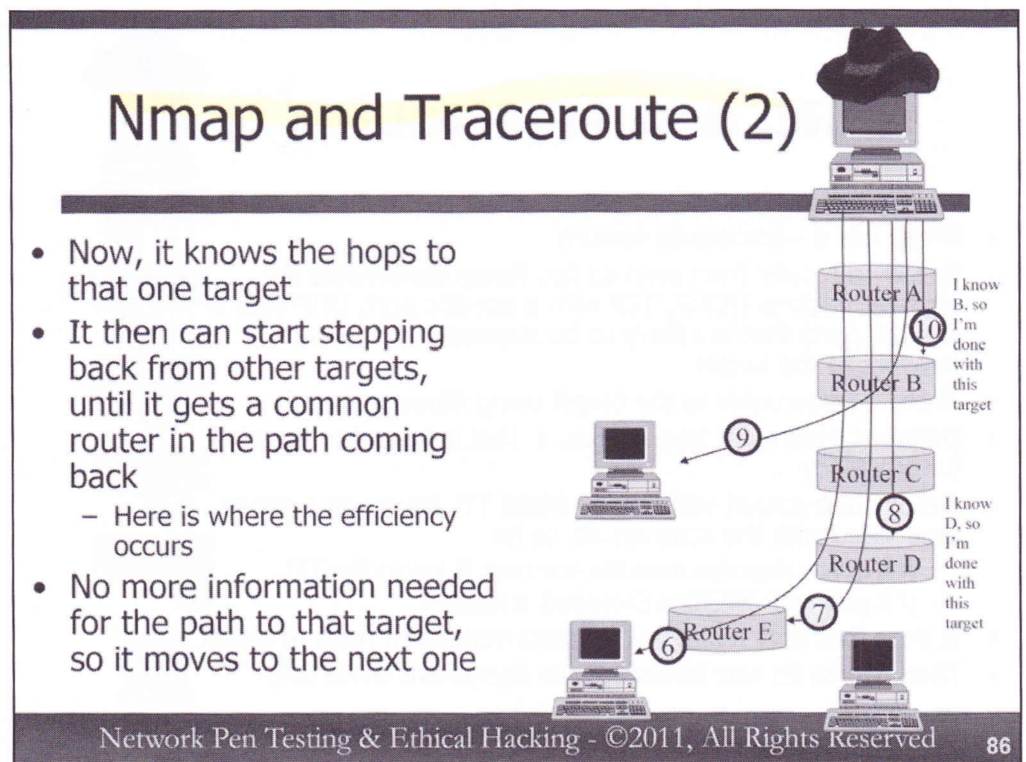
Nmap also included traceroute capabilities, invoked with the --traceroute syntax at its command line. The traceroute capabilities of Nmap have some interesting and useful differences from other traceroute tools, designed for effectiveness and efficiency.

From an effectiveness perspective, the Nmap traceroute functionality first consults the scan results obtained so far from the given target IP address. It then selects a protocol that the network allows to access that target to use for its traceroute, such as ICMP, TCP to a given port, or UDP to a given port.

Then, to determine the router between the attack machine and the target, Nmap works backward. Unlike most traceroute tools, which start with packets with small TTLs and work their way up to higher TTLs to measure the routers from the attacker to the target, Nmap measures from the target back. It's counter intuitive, but can lead to more efficiencies when scanning larger numbers of targets on more complex networks.

In Step 1, Nmap starts by sending a packet with the appropriate protocol to the target machine, using a TTL that it guesses for the target machine based on its earlier scan. If an ICMP Time Exceeded Message comes back from a router, Nmap didn't get the right TTL for the hops to the target, so it increments the TTL and sends another packet. If it gets a response from the target, but the TTL in the response is not 1, it doesn't have the exact number of hops to the target, so it lowers the TTL. All of this is happening in Step 1 in the figure above. Nmap steps up or down its TTL until it gets exactly the number of hops to that target, based on a response coming back with a TTL of 1.

Then, in Step 2, Nmap sends a packet with a TTL one lower than the number of hops to the target machine, finding the next earlier hop. It then tries a TTL of 2 lower, determining the hop before that in Step 3, and so on. Nmap has now figured out the routers between the target and the scanning machine. Essentially, we have mapped out one branch of a network tree. It's convoluted and backwards, but it works.



But how is this technique more efficient? To see the efficiencies, let's consider another target that is also part of the same scan, invoked with the –traceroute option.

In Step 6, Nmap does another guess of the TTL going to the second target. If it gets back a response with a TTL of one, its guess was correct. If not, it increments or decrements until it gets that TTL of one. Then, Nmap can send a packet with a TTL of one lower (Step 7), figuring out which router hop comes before that target. It then sends a packet with a TTL of two lower (Step 8), finding the hop before that.

Now, here's where the efficiency comes in. When Nmap, walking backwards like this, discovers a router hop already in the list of routers discovered for an earlier host, it doesn't have to walk backwards any more for that target. It has then attached a new branch to the network tree it had started constructing before, so it can move on to the next target.

In Step 9, we do the little back and forth shuffle to determine the TTL to that target. In Step 10, we start walking back, immediately seeing a router that we already have in the tree. Thus, we can attach that branch right away, saving us from having to retrace the same routers again and again.

Nmap Port Scanning

- Nmap doesn't check all ports by default
  - This is very important to note... it's not a comprehensive scan by default
- By default, Nmap checks only the top 1,000 most used ports for TCP and/or UDP
  - The nmap-services file indicates the ranking of the most common ports, based on widespread scanning research by Fyodor
  - Nmap *does not* check all ports less than 1024 by default anymore
- The –F option (which stands for "Fast") says to scan the top 100 ports
- The --top-ports [N] option tells Nmap to scan for the N most popular ports
- For a comprehensive or targeted scan, use the –p option
  - `-p 0-65535` will scan all ports
  - `-p 22,23,25,80,445` will check only those ports
  - The flag T: and U: can be included in the list to specify TCP or UDP
- Ports are scanned in random order, but –r makes them not randomized

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 87

A common error in running Nmap port scans is to simply run Nmap, specifying a scan type and a target IP address, thinking that Nmap will check all ports on the target system. For example, someone might run:

```
$ nmap -sT 10.10.10.10
```

This invocation will indeed run a TCP port scan against target 10.10.10.10. Unfortunately, it will not scan all TCP ports. In fact, Nmap won't even check all ports less than 1024 by default. Instead, by default, Nmap only checks the top 1,000 most widely accessible ports on the Internet, as specified in the nmap-services file. Fyodor conducted in-depth research with large scale scans to determine the most popularly used ports on the Internet. This ranking of port popularity is included in the latest versions of Nmap, within the nmap-services file. Nmap will scan the top 1,000 most popular TCP or UDP ports from that file when no port range is indicated.

If Nmap is invoked with the –F option (which stands for "Fast"), it will scan the top 100 most popular ports of TCP or UDP, depending on whether it is configured to conduct a TCP or UDP scan.

Instead of the top 1,000 or top 100 ports, the Nmap user can also specify "--top-ports [N]" to scan the N most popular ports from the nmap-services file.

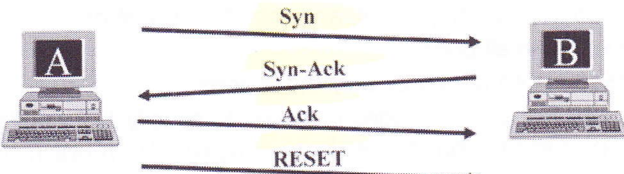
However, a given target environment may have a very specialized application listening on a port that is not in the top port listing in the nmap-services file. Thus, if testing time permits, you should consider doing a comprehensive port scan, checking all possible ports. The –p flag can indicate a port, port list, or port range for Nmap to scan. To scan all TCP ports on a target, you could specify:

```
$ nmap -sT 10.10.10.10 -p 0-65535
```

Alternatively, to check only a specific list of ports, you could invoke Nmap with –p 22,23,25,80,445 to measure only those ports. If you want to mix TCP and UDP ports, you can preface TCP ports with T: and UDP ports with U: in this list. By default, Nmap scans ports in a range or list in a random order. The –r flag makes Nmap scan linearly (in increasing port order).

Nmap TCP Port Scan Types: Connect Scan

- Nmap offers numerous TCP scanning options
- Most of these are based on varying the TCP control bits
- The most straightforward is the TCP Connect Scan, Nmap -sT
  - Completes three-way handshake
  - Connection then torn down using RESET
  - Slower, more likely to be logged
  - Less control for Nmap, because it uses OS connect() call
  - Can run with or without root or admin privileges



Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 88

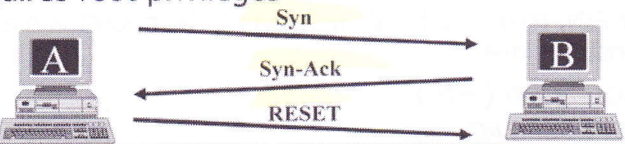
Nmap offers numerous types of TCP port scan options, most of which are based on triggering the behavior of target machines with various TCP Control Bits set.

The most straightforward Nmap TCP port scan is the Connect Scan. This option, invoked with the –sT flag, attempts to complete the TCP three-way handshake with each target port. If a connection is made, the port is labeled as open, and the connection is torn down with a RESET packet from the testing machine.

Connect scans are slower, in that they have to wait for the TCP three-way handshake to complete for all open ports. Furthermore, they are more likely to be logged. If the end system is logging completed connections, a connection will be recorded for each open port, unlike a SYN scan (discussed next), which never completes a connection.

Nmap TCP Port Scan Types: SYN Scan

- SYN scan, sometimes called "half-open" or "SYN Stealth" scan, invoked with -sS
  - SYN-ACK response = open
  - RST response = closed
  - No response = filtered
  - Often, not logged on the end system, because there is no connection
  - Firewalls, IDS sensors, and IPS tools may still detect it
  - Requires root privileges



Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 89

A SYN scan, also known as a "half-open" or "SYN Stealth" scan, doesn't complete the three-way handshake. Instead, it starts out by sending a SYN. Open ports are determined based on a SYN-ACK response. Then, Nmap sends a RESET to abort the connection initiation. Nmap interprets RESET responses as a closed port. If nothing comes back, Nmap labels the given port as filtered. These scans are invoked with the –sS option.

Because a connection never occurs, the target system is less likely to log this kind of activity. Any applications on the target that log connections will not see the activity. However, firewalls, Intrusion Detection System (IDS) sensors, and Intrusion Prevention System (IPS) tools may log, alert, or even block packets associated with a SYN scan.

Because it doesn't fully follow normal TCP behavior with a three-way handshake, this kind of scan requires root privileges so that Nmap can formulate the packets associated with the scan.

Additional Nmap TCP Scan Options

- ACK Scan (-sA)
  - Useful in scanning through an "established" filter on a router
  - But, doesn't reliably tell us if a port is open or closed... instead, it is useful for identifying hosts (network mapping)
- FIN Scan (-sF)
  - Set FIN bit of all scan packets
- Nmap Null Scan (-sN)
  - Set all control bits to 0 (Null)
- Nmap Xmas Tree Scan (-sX)
  - Set FIN, PSH, and URG, "lighting up the packet like a Christmas tree"
- Maimon Scan (-sM)
  - Set FIN and ACK bits

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 90

Nmap also supports ACK scanning to help get through certain kinds of packet filters. A router may have access control lists configured to allow outgoing SYNs from a protected network and their incoming responses (i.e., established connections with the ACK Control Bit set). These filters also block incoming SYNs. That way, users on the protected network can initiate sessions outbound and can receive responses. But, if someone on the outside tries to send in a SYN packet (without the ACK bit set), the router will block the connection. Nmap's ACK scan feature (invoked with –sA) generates packets with only the ACK bit set as it scans the target environment. It is important to note that ACK scans cannot reliably determine which ports are open or closed. Different systems respond in different ways to an unsolicited ACK. However, a response DOES indicate that there is a system at the address. So, the ACK scan result can be used to do network mapping through an established filter. But, it is not a useful port scan technique.

Nmap also offers other TCP scan options that involve unusual Control Bit combinations, which different end systems will respond to in different ways, and may be helpful in scanning through certain kinds of filters. A FIN scan, invoked with –sF, sends packets with the FIN Control Bit set. Null scans (-sN) set none of the Control Bits. Xmas tree scans (-sX) set the FIN, PSH, and URG Control Bits, making the packet resemble a Christmas tree (according to some people).

The point of all these variations is that, according to RFC 793, if the port "state is CLOSED... an incoming segment not containing a RST causes a RST to be sent in response." Later, the RFC further explains that systems should "drop the segment and return" for open ports that receive a packet without the SYN, RST, or ACK bits. Thus, if the target machine follows RFC 793 carefully, we can send packets without the SYN, RST, or ACK bits. A RST response means that the port is closed. No response means that the port may be open. Sadly, though, many systems do not follow this RFC-directed behavior, making these scans less reliable.

The Maimon Scan (-sM), named after its creator Uriel Maimon, sets the FIN and ACK bits. That's because some BSD-derived TCP stacks will respond to such a probe with a RESET if the port is closed, and nothing if the port is open.

Custom Control Bits in Scans

- To generate flags with your own desired TCP Control Bits, use:

```
--scanflags
  [URG|ACK|PSH|RST|SYN|FIN|ECE|CWR|ALL|NONE]
```

  - Include the three-letter reference for your desired Control Bits, in any order (or ALL or NONE)
  - For example, to send a SYN, PSH, ACK packet to port 139 on 10.10.10.10, you could run:

```
# nmap --scanflags SYNPSHACK -p 139
10.10.10.10
```

- Nmap is growing into a packet crafting tool

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 91

Beyond the pre-baked Control Bit scans (Xmas, Maimon, etc.), Nmap users can also specify arbitrary Control Bit settings, using the --scanflags option, followed by a list of the desired Control Bits. Control Bits are indicated based on three-letter abbreviations of URG, ACK, PSH, RST, SYN, and FIN, and can be specified in any order. Note that even the extended Control Bits (ECE and CWR) are supported now. Specifying ALL sets all of the control bits to 1 in TCP packets, while specifying NONE sets them all to zero.

For multiple flags, the three-letter abbreviations are just smashed together. The result looks like this:

```
# nmap --scanflags SYNPSHACK -p 139 10.10.10.10
```

That syntax will invoke Nmap to scan target IP address 10.10.10.10, sending a packet with the SYN, PUSH, and ACK control bits set to destination port 139.

With this kind of feature, Nmap is taking on characteristics of a packet crafting tool, being used to generate packets with settings determined by the user. We'll get more into packet crafting later in this course.

Nmap UDP Scans

- Far less options than with TCP
- Invoked with the –sU option
- Sends UDP packet with no payload to target for most ports
  - For a little more than a dozen of the most common UDP services, Nmap 5.20 and later send a protocol-specific payload to the standard port for the service, designed to elicit a response
  - Services include ports 7 (echo), 53 (domain), 111 (rpcbind), 123 (ntp), 137 (netbios-ns), 161 (snmp), 500 (isakmp), 1645/1812 (radius), 2049 (nfs), and others
  - Only sends the appropriate payload to those port numbers... all other UDP ports have blank payload
    - So, it won't detect a common UDP service listening on an unusual port... but how often do you see that in a production environment? Almost never.
- Attempts to detect response ICMP rate limiting in target, and slows down
  - Can really stretch out scan time
  - Remember, closed ports may respond with ICMP Port Unreachable
  - Linux will send only 1 per second...
  - For 65,536 ports, that's over 18 hours for a single target machine!

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 92

Nmap also supports UDP scanning, but note that we don't have as many options as we did with TCP scanning. There's no such thing as a UDP Connect, SYN Stealth, Xmas Tree, or Null scan. Instead, for UDP, we have one option, invoked with a –sU syntax.

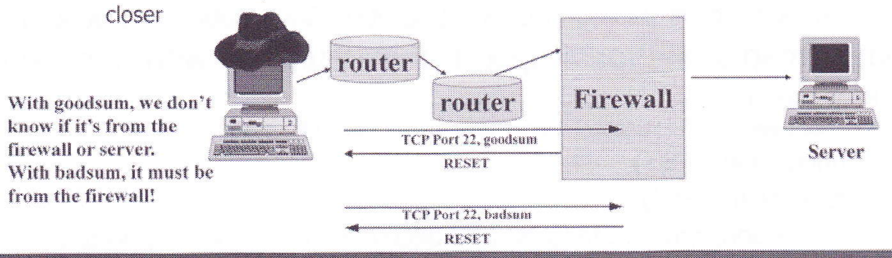
Nmap will send UDP packets with no payload to the target machine for most ports.

Starting with Nmap 5.20, for a little more than a dozen specific UDP ports associated with the most common UDP services, Nmap sends a protocol-specific payload in its UDP packet for each service, designed to get the target service to send a UDP response back. That way, we can get more reliable UDP port scanning for those services. The services Nmap measures this way include ports 7 (echo), 53 (domain), 111 (rpcbind), 123 (ntp), 137 (netbios-ns), 161 (snmp), 500 (isakmp), 1645/1812 (radius), 2049 (nfs), and others. Note that these payloads are only sent to those ports associated with the common UDP services. If someone has altered a standard UDP service to listen on an unusual port, this technique will not find it, because only a UDP packet without a payload will be sent to the unusual port. However, it is exceedingly rare to find a production network with a standard UDP service listening on an usual port. Therefore, for identifying these common over-a-dozen UDP services, Nmap's UDP payload feature is really useful.

Nmap also includes functionality that tries to detect whether the target machine is throttling the rate at which it sends ICMP Port Unreachable messages back. As you may recall, Linux will send only one ICMP Port Unreachable message to a given machine per second. Nmap interacts with the target to measure how quickly it gets ICMP Port Unreachable messages back, and then automatically slows down the rate at which it sends follow-up UDP packets to other ports to match the rate at which the target can send responses. With Linux throttling ICMP Port Unreachables down to 1 second per response, a UDP scan of 65,536 ports on a single target machine will take over 18.2 hours, a very long time.

Nmap Feature: --badsum scans

- Using Nmap with --badsum at the command line will generate packets with an invalid TCP or UDP checksum
- End system will reject these packets, silently dropping them
- But, some firewalls and IPSs do not calculate layer 4 checksums
  - They may send a RESET or ICMP Port Unreachable
  - Therefore, if any responses come back, it came from a firewall or IPS
- This technique is sometimes called "Firewall Spotting"
  - Another trick involves looking for a different TTL from SYN-ACK responses for allowed services versus the TTL on RESET responses for blocked services
  - They may be different because of a different initial TTL, or, even if they have the same initial TTL, the RESETs from the firewall are decremented less, because it is closer



Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 93

Another interesting Nmap feature involves sending packets with bad TCP or UDP checksums, calculated incorrectly on purpose. The resulting packets are bogus and should be ignored by any target operating system.

What value do they have, then? Well, as pointed out by Ed3f in Phrack magazine, although end systems silently drop bad checksum packets, most firewalls and IPS tools do not. They often send back a RESET or ICMP port unreachable message. These tools do not calculate the layer 4 checksum, but instead interact with even these bad packets. Again, you might think, so what?

A bad checksum scan can be used to determine if a firewall sits between the attacker and the target. Suppose an attacker does a scan with good checksums, and gets back a bunch of RESETs. The attacker is not sure if those packets came from a firewall on the network or from the end host. Is the traffic filtered or is the port legitimately closed? With a bad checksum scan, any RESET that comes back must be from a firewall or network-based IPS. Thus, the attacker knows that a firewall or IPS is in place between the attacker and the target, letting the attacker attempt to compromise that system. Additionally, by looking at differences in the TTL of the RESET, and any legit traffic that comes from the end system (such as a SYN-ACK response from an open port), the attacker can infer the number of hops to the firewall or IPS.

This technique is known as "Firewall Spotting", as it allows the penetration tester to spot a network firewall or similar device protecting the target systems.

Another trick for performing firewall spotting is to look at the TTL values in the responses coming back from the target environment. If the TTL values from allowed services (say, SYN-ACKs from the web server on port 80) are different from the TTL values of blocked services (indicated by RESETs coming back), that could be a sign that a firewall or similar network devices is sending the RESETs. For example, if the TTLs of the RESET packets coming back are higher than the TTLs of the SYN-ACKs, it implies the system generating the RESETs is closer (because the TTLs are decremented less). However, such a case depends on the target machine and the firewall device having the same initial TTL. Even if they don't have the same initial value, however, we still can spot a discrepancy in the TTLs for allowed versus blocked services.

Nmap Support for IPv6

- IPv6 access to systems is often not secured
  - Many firewalls and IPSs do not block IPv6 traffic
  - IPv6 is auto-configured on most Win, Linux, OS X, and other devices
  - Even if the target organization's ISP doesn't carry IPv6 traffic, it is often allowed within a DMZ or on an intranet
    - Exploit systems across the Internet via IPv4, and then locally pivot attacking IPv6
- IPv6 addrs are 128 bits (16 bytes): Groups of 4 hex digits separated by colons
  - Double colons (::) means to fill in with appropriate number of zeros
    - Can only use :: once in address, or else it is ambiguous
  - Local loopback is ::1 (0000:0000:0000:0000:0000:0000:0000:00001)
- Some Nmap scan types support IPv6 (launched with -6 option)
  - Ping sweeps (-sP)
    - Not overly useful because of enormous target network ranges
  - Connect scans (-sT)
  - Version scans (-sV)
  - OS detection, random targets, and decoys currently not supported

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 94

Nmap does support IPv6 for some of its scanning options. Scanning targets using the IPv6 protocol can be helpful for penetration testers, because many firewalls and IPSs do not filter, block, or detect attacks transmitted via IPv6. Even if the target organization's ISP does not transmit IPv6 traffic to the target over the Internet, chances are that the target systems themselves speak IPv6, and can be accessed locally within the DMZ and intranet using the protocol, especially from systems on the same subnet. This leads to some interesting pivot options for penetration testers. We can exploit a system across the Internet using IPv4 to gain access to a DMZ or internal network. Then, we can scan for and exploit other targets using IPv6. Most operating systems and even network appliances have IPv6 auto configured. Modern Windows systems, most Linux variants, Mac OS X, and several wireless access points all have IPv6 capabilities turned on by default, making them potentially juicy targets.

IPv6 addresses are 128 bits long, and are represented by groups of 4 hexadecimal digits separated by colons, as in 0102:0304:0506:0708:090A:0B0C:0D0E:0F00. To save space in printing addresses, double colons (::) mean that the given bits should be populated with all zeros. You can only use :: in an address one time. Otherwise, it would be ambiguous how many zeros to fill in with multiple :: indications. The local loopback address is ::1, which represents 0000:0000:0000:0000:0000:0000:0000:00001.

A few Nmap scanning capabilities support IPv6, which are invoked by adding "-6" to the Nmap command. In particular, ping sweeps (-sP) do. However, this feature isn't very useful because target IPv6 address spaces tend to be enormous. Subnets are often 64-bits long or even larger, making a single subnet have substantially more potential IP addresses than the *entire* current Internet, with its 32-bit address space. For that reason, penetration testers usually don't sweep subnets looking for IPv6 targets. Instead, as we'll see on the next slide, we usually send broadcast pings to find targets.

Nmap connect scans (-sT) and version scans (-sV) do support IPv6. These are highly useful, as they can allow us to find TCP services and the version of the protocols and software we may be able to exploit over the IPv6 protocol. Currently, Nmap's OS detection, random targets, decoys, and other scan types do not support IPv6.

Finding IPv6 Targets and Using Nmap to Scan Them

- To locate targets, you could use Neighbor Discovery feature based on broadcast addresses via ping6 command:
  - `$ ping6 -I eth0 ff02::1` (this is broadcast address for all link-local IPv6 nodes)
  - `$ ping6 -I eth0 ff02::2` (this is broadcast address for all link-local IPv6 routers)
  - Then, look at neighbors with:
    - `$ ip neigh`
- With Nmap, specify target IPv6 address as xxxx:xxx::xxxx%[int], as in fe80::20c0%eth0
- Finally, we can then port scan them with:
  - `$ nmap -PN -sV -6 fe80::20c0%eth0 --packet-trace`

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved

When using Nmap to perform connect and version scanning of targets, we first need the target's IPv6 address. We can use the ping6 command built into many Linux variations and Mac OS X to send a message to the broadcast address of a local subnet looking for neighbors, a feature of IPv6 known as "neighbor discovery". The broadcast address for a local subnet is ff02::1 to identify IPv6 hosts, and ff02::2 to identify IPv6 routers. Thus, we can use ping6 to find targets by running:

```
$ ping6 -I eth0 ff02::1
$ ping6 -I eth0 ff02::2
```

Now, you could look at the output of your ping6 command to identify targets. Alternatively, on Linux, you could run "ip neigh" to see which neighbors are currently cached based on the neighbor discovery done by ping6. On Mac OS X, you could also run the "ndp" command to discover neighbors.

Then, to run Nmap to launch a TCP connect scan and/or version scan against discovered targets, you need to specify the IPv6 address followed by a %[int] to indicate the interface the packets should be sent on. For example, you may scan an address such as fe80::20c0%eth0 to send packets on your eth0 interface to the target fe80::20c0.

For an example that puts this altogether, we could launch an Nmap version scan of a target (-sV), avoiding an initial ping (-PN), scanning using IPv6 (-6) of the target IP address fe80::20c0 sent through our eth0 interface (%eth0) invoking packet tracing to see all the action using the following command:

```
$ nmap -PN -sV -6 fe80::20c0%eth0 --packet-trace
```

Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

- Scanning Goals and Types
- Overall Scanning Tips
  - Sniffing with tcpdump
- Network Sweeping with Scapy
  - Scapy/tcpdump Exercise
- Network Tracing
- Port Scanning
  - Nmap
  - ***Nmap Exercise***
- OS Fingerprinting
- Version Scanning
  - Nmap -O -sV and Amap Exercise
- Vulnerability Scanning
  - Nmap Scripting Engine
  - NSE Exercise
  - Nessus
  - Nessus Exercise
  - Other Vuln Scanners
- Enumerating Users
  - Enumerating Exercise
- Netcat for the Pen Tester
  - Netcat Exercise

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 96

Now, we'll do some exercises with Nmap, exploring its run-time interaction abilities, ARP scanning, and Idle scanning. Get your Linux machine ready to go, logging in with root-level privileges, which Nmap will need to formulate most of the unusual packets we'll generate.

Exercise: Nmap ARP Scan and Run-Time Interaction

- Run a ping sweep of our local network

  `# nmap -n -sP 10.10.10.1-255 --packet-trace`

- While it is running, hit the following keys:
  - Shift-p = Turn off packet tracing
  - p = Turn it back on
  - v = Increase verbosity
  - Shift-v = Turn if off
  - d = Increase debugging level
  - Shift-d = Turn it off
- Note that you are just sending ARPs; no ICMP or HTTP
  - Nmap is smart enough to do that because you are on the same LAN

root@linux:/

File Edit View Terminal Tabs Help

```
# nmap -n -sP 10.10.10.1-255 --packet-trace

Starting Nmap 4.68 ( http://nmap.org ) at 2008-11-14 14:50 EST
SENT (0.0760s) ARP who-has 10.10.10.2 tell 10.10.75.2
SENT (0.0770s) ARP who-has 10.10.10.3 tell 10.10.75.2
SENT (0.0770s) ARP who-has 10.10.10.4 tell 10.10.75.2
SENT (0.0780s) ARP who-has 10.10.10.5 tell 10.10.75.2
SENT (0.0780s) ARP who-has 10.10.10.6 tell 10.10.75.2
SENT (0.0780s) ARP who-has 10.10.10.7
```

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 97

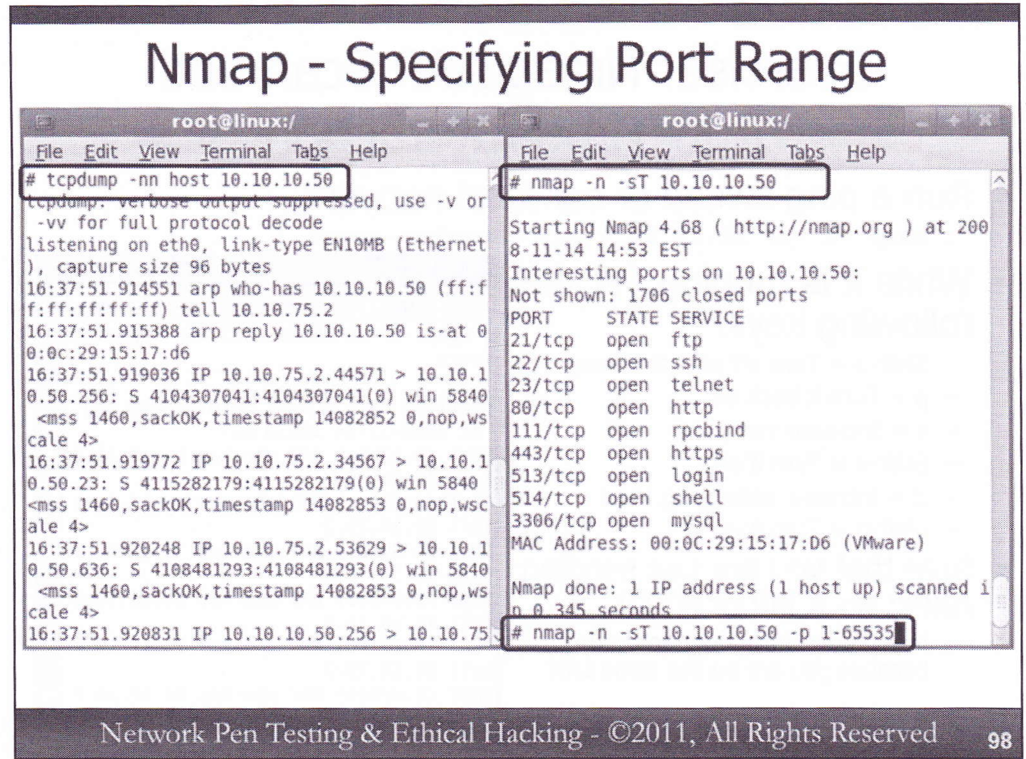
Let's run a scan of the target subnet. We will run:

```
# nmap -n -sP 10.10.10.1-255 --packet-trace
```

The –n means that Nmap should not resolve domain names. The –sP means do a ping sweep, but watch what happens… no ICMP (or TCP packets for that matter) will be sent for the ping sweep. Also, the --packet-trace option tells Nmap to display a summary of each packet before it sends it. While it runs, hitting Shift-p will turn this off, while hitting the p key will toggle it back on.

Also, try v and d multiple times each for verbosity and debug information. If you can't type that fast enough, try relaunching the scan and then hitting them.

Note that you are sending only ARPs, no ICMP or HTTP, despite the fact that you kicked off Nmap with a –sP for a "ping" sweep. Nmap did this because you are on the same subnets as the targets, so an ARP reply implies that the address is in use; no follow-up ICMP or TCP packets are required.



Next, let's conduct a TCP port scan of target machine 10.10.10.50. Stop tcpdump, and then restart it, configured to show traffic associated with host 10.10.10.50 (not resolving names):

```
# tcpdump -nn host 10.10.10.50
```

Next, invoke Nmap to scan that host, doing a TCP connect scan (full three-way handshake):

```
# nmap -n -sT 10.10.10.50
```

Nmap will display the total time it takes to complete the scan. Record how long it took for the scan here:

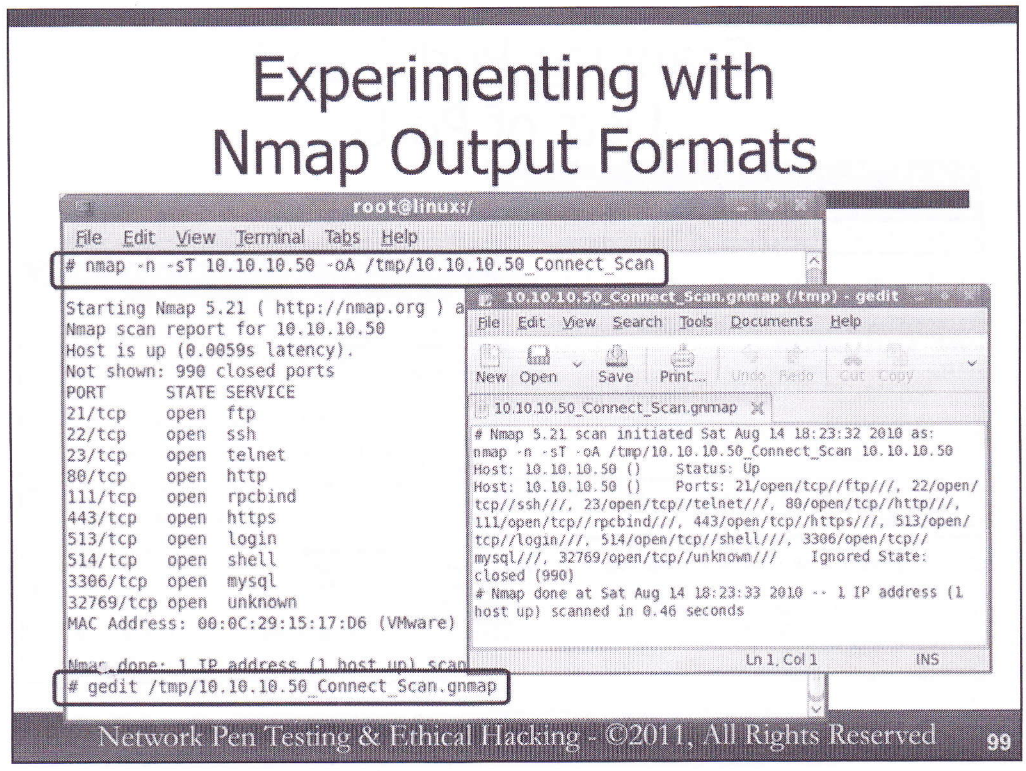
____

Nmap did not scan all TCP ports with that invocation, however. It only scanned the top 1,000 most frequently used ports as indicated in the nmap-services file. Let's see how much longer it takes to scan all TCP ports:

```
# nmap -n -sT 10.10.10.50 -p 1-65535
```

It should take a lot longer, given the higher number of ports it is scanning. But, do you notice any differences in the output of the narrower port scan and the complete port scan?

Also, look at the output of your sniffer. You should see lots of SYN packets (S) going from your machine to the target, as well as lots of RESETS (R) coming back. There will be a relatively smaller number of SYN-ACKs coming back, as well as ACKs going from your machine to complete the three-way handshake.



Next, let's look at the output format files that Nmap can create via the -oA option. Re-run your -sT scan with the default port, storing your results in all of the major format styles (-oA to indicate Normal, Greppable, and XML output). We'll store our results in files in the /tmp directory, with a base name of 10.10.10.50_Connect_Scan, which indicates the scan type and the IP address of the target:

```
# nmap -n -sT 10.10.10.50 -oA /tmp/10.10.10.50_Connect_Scan
```

Then, get a list of the files associated with 10.10.10.50 inside of /tmp:

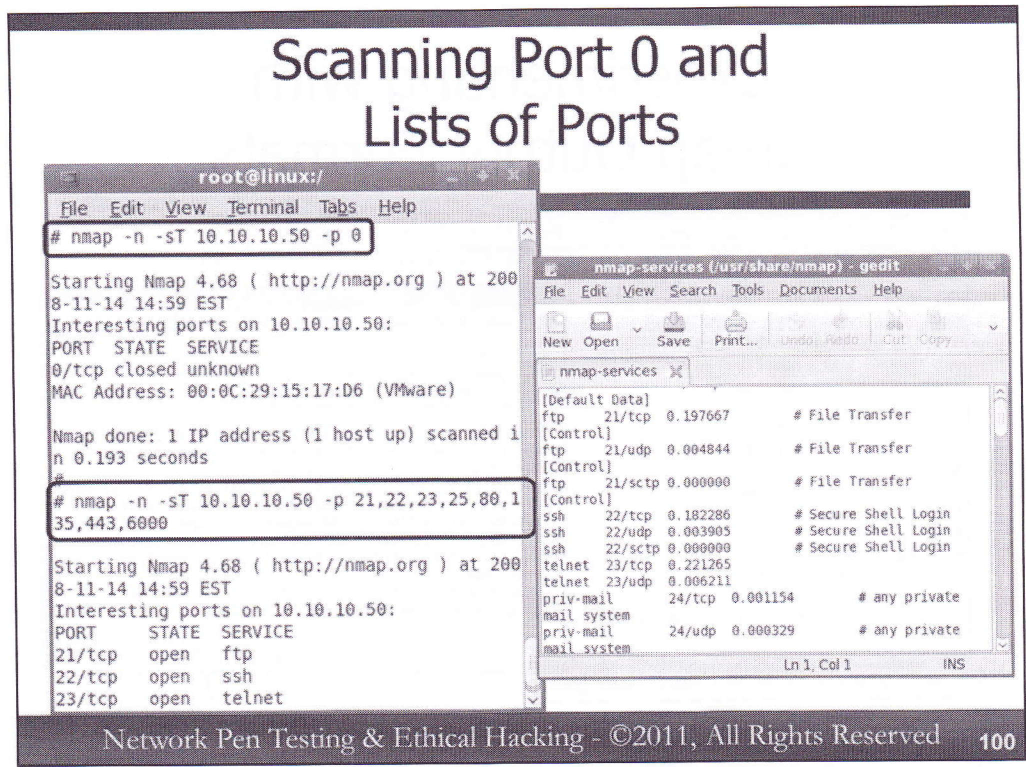
```
# ls /tmp/10.10.10.50*
```

You should see three files: the greppable form (with a .gnmap suffix), the normal form (with a .nmap suffix), and the XML form (with a .xml suffix).

Use the gedit tool to review these files, especially the greppable format:

```
# gedit /tmp/10.10.10.50_Connect_Scan.gnmap
```

Note that all of the results for a given host are stored on one line, with each open port and associated service identified.



By the way, in the TCP scans we just conducted, we omitted TCP port 0. Let's test that one port with:

```
# nmap -n -sT 10.10.10.50 -p 0
```

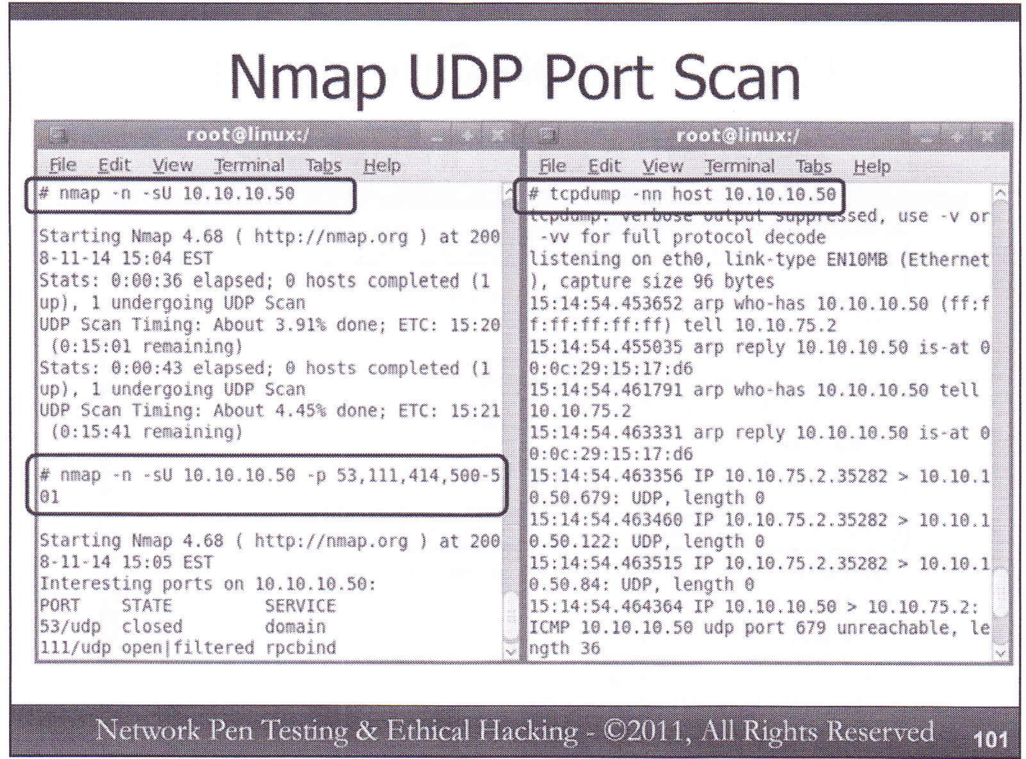
As we've seen, we can scan individual ports by just specifying –p [X] (where [X] is the port number we want to scan). We can do ranges of ports by specifying –p [X-Y]. And, we can do individual sets of ports by using a comma-separated list. Try the last one by scanning:

```
# nmap -n -sT 10.10.10.50 -p 21,22,23,25,80,135,443,6000
```

Next, review the ports in the nmap-services file (the file from which Nmap gets its list of most frequent ports to scan) by running:

```
# gedit /usr/share/nmap/nmap-services
```

The format of this file includes the service name (e.g., ftp), the associated port and protocol (e.g., 21/tcp), the relative frequency that the given port was discovered during Fyodor's widespread Internet scanning research, and an optional comment. Note that the ports themselves are typically TCP or UDP, although some are associated with the Stream Control Transmission Protocol (SCTP), an alternative layer-4 protocol defined by RFC 4960.



Now that we've looked at TCP port scanning with Nmap, let's try UDP port scanning. Remember we discussed earlier that Linux kernels throttle ICMP port unreachable responses so that they send only 1 every second? We'll see that behavior now, because 10.10.10.50 is a Linux machine. Keep your tcpdump sniffer running, showing packets going to and from host 10.10.10.50.

Now, invoke Nmap to perform a UDP port scan of 10.10.10.50, as follows:

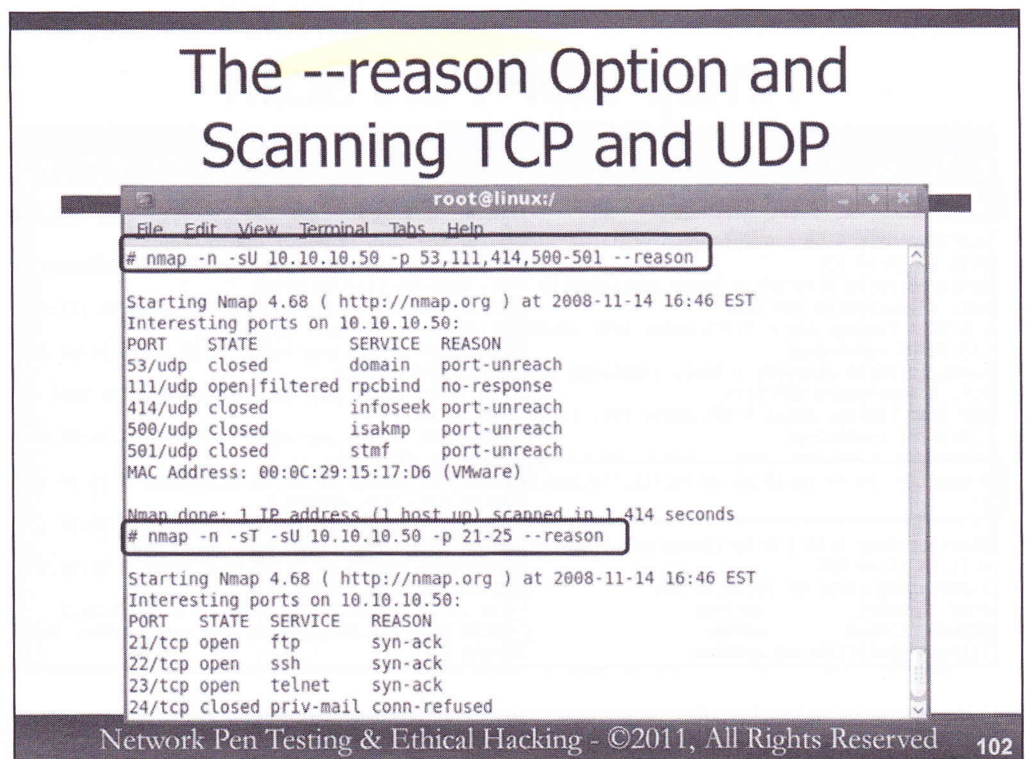
```
# nmap -n -sU 10.10.10.50
```

In your sniffer output, you will likely see several UDP packets, and some ICMP port unreachables sent periodically. But, these ICMP messages are coming very slowly.

In your Nmap window, hit the space key to get a status report. You will likely see that the scan is only a small percentage done, and will take far longer to complete than we have time for here, perhaps over an hour or more, depending on your system speed and the network speed. We can't wait, so hit CTRL-C to stop Nmap before the scan completes.

Now, re-run an Nmap UDP scan of the target, this time focusing on a narrower list of ports, as follows:

```
# nmap -n -sU 10.10.10.50 -p 53,111,414,500-501
```



Modern versions of Nmap provide the --reason option, which tells us why Nmap classifies a given port's open/closed/filtered state as it does. Let's re-run our previous scan, but with the --reason option:

```
# nmap -n -sU 10.10.10.50 -p 53,111,414,500-501 --reason
```

There are no spaces between those double dashes before the word reason. Note the REASON column in the output, telling us the behavior that caused Nmap to come to the conclusion it did about the port's state.

Next, let's see how we can scan for open TCP and UDP ports in the same command, while looking at the reasons that Nmap has labeled a port with a given state. Run Nmap as follows:

```
# nmap -n -sT -sU 10.10.10.50 -p 21-25 --reason
```

While it is running, note the output of your sniffer. It's always a good idea to keep an eye on what your sniffer is telling you about a scan.

Exercise: Nmap with Good Checksum and Bad Checksum

- Run a "normal" SYN scan of 10.10.10.10

```
# nmap -n -sS 10.10.10.10
```

- Note the results
- Now, run the same scan, but with a bad checksum

```
# nmap -n -sS 10.10.10.10 --badsum
```

- Hit the space bar to see the current estimate of % done and time remaining
- Nothing should come back, because the end host ignores the packets
- Why is it much slower with bad checksums?

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 103

Now, let's look at the bad checksum behavior of Nmap. First, let's try a normal SYN scan of the target machine at 10.10.10.10:

```
# nmap -n -sS 10.10.10.10
```

You should see some open ports on the target.

Next, try running the same scan with bad TCP checksums:

```
# nmap -n -sS 10.10.10.10 --badsum
```

This will take a lot longer. To see what your current status is, hit any key (such as the space bar) to see time remaining. In the end, you'll see that no ports appear to be open; they are all filtered. That's because the end system is ignoring these packets and sending nothing back.

But, why is it so much slower? Let's investigate.

Exercise: Nmap Checksums and Timing

- To shed some light on the difference in speed, run tcpdump:

```
# tcpdump -nn host 10.10.10.10
```

- Compare the tcpdump results of:

```
# nmap -n -sS 10.10.10.10
```

- Versus:

```
# nmap -n -sS 10.10.10.10 --badsum
```

- Bottom line:
  - RESETs really help to speed up a SYN scan
  - But the end system sends no RESETs during a badsum scan
  - If we do get a RESET, Nmap is smart enough to know it came from a firewall, and prints out "closed" instead of "filtered"

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 104

To determine why it is slower with bad checksums, try running tcpdump:

```
# tcpdump -nn host 10.10.10.10
```

Compare the tcpdump results when running:

```
# nmap -n -sS 10.10.10.10
```

Versus:

```
# nmap -n -sS 10.10.10.10 --badsum
```

Do you see why it is different? If the badsum result triggered a RESET for a given port, Nmap would label the port as "closed," not "filtered". What would that indicate?

Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

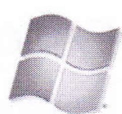
- Scanning Goals and Types
- Overall Scanning Tips
  - Sniffing with tcpdump
- Network Sweeping with Scapy
  - Scapy/tcpdump Exercise
- Network Tracing
- Port Scanning
  - Nmap
  - Nmap Exercise
- ***OS Fingerprinting***
- Version Scanning
  - Nmap -O -sV and Amap Exercise
- Vulnerability Scanning
  - Nmap Scripting Engine
  - NSE Exercise
  - Nessus
  - Nessus Exercise
  - Other Vuln Scanners
- Enumerating Users
  - Enumerating Exercise
- Netcat for the Pen Tester
  - Netcat Exercise

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 105

After the tester has determined open ports on systems in the target environment, we need to discern the operating system types of the target. Armed with OS types and open ports, we not only have a better idea of the kinds of targets we face, we can also begin researching known flaws (such as common misconfigurations and unpatched services) on those types of devices.

Nmap Active OS Fingerprinting

- Nmap attempts to determine the operating system of target by sending various packet types and measuring the response
- Different systems have different protocol behaviors that we can trigger and measure remotely
- Besides Nmap, another tool focused just on active fingerprinting is Xprobe2 by Ofir Arkin
  - http://sys-security.com/xprobe



In addition to finding out which ports are open on a system, an attacker also wants to determine which platform the system is based on. By determining the platform, the attacker can further research the system to determine the particular vulnerabilities it is subject to. For example, if the system is a Windows Server, the attacker can utilize various vulnerability disclosure sites to hone the attack.

The specifications for network protocols leave a lot of room for interpretation, and the software that implements this communication is quite complex. Thus, different vendor implementations of TCP, ICMP, IP and other protocol behavior differ. Nmap supports sending probes to a target machine to look for differences in these behaviors to identify the operating system type.

This technique is called Active OS Fingerprinting because it is sending packets out to measure the response of the machine in an effort to identify the OS type. It is active because it sends packets.

Another tool besides Nmap that focuses just on active fingerprinting is Xprobe2 by Ofir Arkin. This tool sends several test packets to a target machine, and then applies fuzzy logic to calculate the probabilities of its operating system type.

Nmap OS Fingerprinting Capability

- Recent versions of Nmap have dropped the first generation OS fingerprinting capability built into Nmap for years
  - Ran nine tests of a target, mostly associated with unusual Control Bit settings for different operating systems
  - Modern Nmap installs include only second generation OS fingerprinting functionality
- An avalanche of additional tests included in the second generation capability
- The –O option (and –O2) uses the second generation method
  - The -O1 option has been removed in modern Nmap versions

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 107

Nmap has included active OS fingerprinting functionality for many years. However, modern versions of Nmap have significantly changed this functionality from earlier versions. The original Nmap OS fingerprinting capabilities performed nine tests, most of which centered around how different operating systems respond to unusual TCP Control Bit settings. This older capability is often referred to as the "first generation" OS fingerprinting capabilities of Nmap.

Recent versions of Nmap have a "second generation" active OS fingerprinting ability. A huge number of new active fingerprinting techniques have been added in this suite. Currently, the second generation tests are more accurate than the first. The most recent Nmap releases have dropped support altogether for the first generation capability, and now rely exclusively on the second generation fingerprinting, which is invoked with either –O or –O2 at the Nmap command line. Older versions of Nmap supported –O1 for the first generation capability, but that support has been removed in recent versions.

It is important to note that Nmap focuses on *active* fingerprinting. That is, Nmap sends packets at a target machine to measure its behavior in responding to the packets Nmap generates. Nmap does not currently support passive fingerprinting, which involves sending no packets but merely listening for packets from a target. Other tools (such as the free P0f2) do support passive OS fingerprinting.

Tests Included in Nmap Second Gen OS Fingerprinting

- Over 30 different methods are included in the second generation fingerprinting, including:
  - TCP ISN greatest common denominator (GCD)
  - TCP ISN counter rate (ISR)
  - TCP IP ID sequence generation algorithm (TI)
  - ICMP IP ID sequence generation algorithm (II)
  - Shared IP ID sequence boolean (SS)
  - TCP timestamp option algorithm (TS)
  - TCP initial window size (W, W1 - W6)
  - IP don't fragment bit (DF)
  - IP initial time-to-live guess (TG)
  - Explicit congestion notification (CC)

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 108

The second generation active OS fingerprinting of Nmap includes over thirty different tests to determine the operating system type of a target. Included in these tests are measures of the TCP sequence numbers of responses, such as their greatest common denominator and how quickly they change over time. Also, Nmap measures the changes in IP ID values for responses to TCP and ICMP packets. Some operating system types have different sets of IP ID numbers for TCP versus ICMP, while others do not (Windows uses the same incremental number for both sets of protocols).

It also looks at TCP timestamp behavior and TCP window sizes the target system negotiates. Also, Nmap evaluates the behavior of the system to a message with the Don't Fragment bit set in its IP header. It attempts to guess the initial Time To Live for the packet by rounding it up to the next nearest power of 2, because many system types have a TTL of 2**n or (2**n)-1. Finally, Nmap analyzes the explicit congestion notification behavior of the target machine to see how it handles the extended control bits associated with congestion control.

Tests Included in Nmap First Gen OS Fingerprinting

- Older versions of Nmap (before 4.51) include first generation fingerprinting tests, invoked with –O1
  - TCP Sequence Prediction
  - SYN packet to open port
  - NULL packet to open port
  - SYN|FIN|URG|PSH packet to open port
  - ACK packet to open port
  - SYN packet to closed port
  - ACK packet to closed port
  - FIN|PSH|URG packet to closed port
  - UDP packet to closed port

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 109

The first generation active fingerprinting of Nmap is supported only in older versions of Nmap, those released before Nmap 4.51. Some testers still use these capabilities, because they are faster than the second generation feature, although they are less accurate.

The first test in this list looks at the predictability of the TCP sequence numbers of the SYN-ACK responses (which we called ISN_B when we covered the TCP three-way handshake) of the target. Some operating system types have more predictable sequence numbers than others. Thus, by looking at how these numbers change for subsequent connection, we may be able to narrow down the operating system type.

Many of the other tests look for variances in the behavior of the target's TCP stack with unusual combinations of TCP control bits.

Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

- Scanning Goals and Types
- Overall Scanning Tips
  - Sniffing with tcpdump
- Network Sweeping with Scapy
  - Scapy/tcpdump Exercise
- Network Tracing
- Port Scanning
  - Nmap
  - Nmap Exercise
- OS Fingerprinting
- ***Version Scanning***
  - Nmap -O -sV and Amap Exercise
- Vulnerability Scanning
  - Nmap Scripting Engine
  - NSE Exercise
  - Nessus
  - Nessus Exercise
  - Other Vuln Scanners
- Enumerating Users
  - Enumerating Exercise
- Netcat for the Pen Tester
  - Netcat Exercise

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 110

Now that we know the open ports and the operating systems behind them in the target environment, we need to discern the protocols spoken by each port and the versions of the services listening on those ports. We use version scans to gain this information.

Version Scanning

- When Nmap identifies an open port, it displays the default service commonly associated with that port
  - Based on nmap-services file, which lists about 2,200 services
  - Additional services are searchable at the Internet Assigned Numbers Authority (IANA) port assignments at http://www.iana.org/assignments/port-numbers
- But, what services are on ports not in that list?
- And, what about an admin who configures a service to listen on an unexpected port?
  - Example: Web server on TCP 90 or sshd on TCP 3322
- And, what service and protocol version is the target listening service using?
- Nmap version scanning has the answer

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 111

Once we've got a list of open ports, we have to determine which services are actually using those ports. One easy (and automatic) way to do this is to merely look up the common service associated with the port. These mappings of port numbers to services are available in several locations. Most Unix and Linux systems include a /etc/services file that includes rudimentary information of this form. More ports and detailed information are available in the nmap-services file (located by default in the /usr/share/nmap directory). This file contains approximately 2,200 common services and the well-known ports that they use. Nmap automatically checks this one by default as it displays its output. The official port assignment list maintained by the Internet Assigned Numbers Authority (IANA) can also be consulted.

However, while searching such common lists may be valuable, it is limited. The well-known service may note be on that well-known port. For example, what service is listening on a strange port, not included on any of these lists? Furthermore, what if an admin configures a common service on an unusual port, configuring a web server to listen on TCP port 90 or an sshd on TCP 3322? Even if a common service is using a common port, it could be helpful for us to know what version of the service is running and the protocol version number that it speaks.

Each of these questions is addressed by another very useful Nmap feature known as Version Scanning.

Nmap Version Scanning Functionality

- Version scan invoked with –sV
  - Or, use –A for OS fingerprinting, version scan, script scan, and traceroute (i.e., -A = -O + -sV + -sC + --traceroute)
- For each listening port discovered during the port scan, Nmap:
  - Makes a connection to TCP and listens for 5 seconds... if response with match: Done!
  - Sends probes to TCP and UDP ports, sending data designed to elicit a response to determine the service type
    - Over 1,000 service fingerprints in the nmap-service-probes file
  - Attempts SSL handshake over TCP ports, and, if successful, probes over SSL connection
  - Issues Null RPC commands to determine if RPC service is in use

  --version-trace option shows the details of the probes in real time

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 112

To invoke Nmap with its Version Scanning functionality, use the –sV option. Alternatively, Nmap executed with the –A option will conduct OS fingerprinting, Version Scanning, script scanning, and Nmap's tracerouting. In Nmap's algebra, it appears that –A = -O + -sV + -sC + --traceroute. In other words, running Nmap with the –A option is the same as running it with the –O (OS fingerprinting), –sV (version scan), -sC (run default Nmap Scripting Engine scripts), and --traceroute (use Nmap's traceroute feature) options.

The Nmap version scan functionality is executed after Nmap finishes conducting a port scan of the target. For each discovered open port, Nmap will probe the port to try to determine what is listening there. For TCP ports, Nmap connects to the port with a 3-way handshake and waits for a response. If a response comes across the connection, Nmap looks up that response in its nmap-service-probe file (also found by default in /usr/share/nmap). If it finds a match, Nmap prints out information about the service. If no strong match is found, Nmap starts probing the port further.

For open TCP and UDP ports, Nmap probes the target by sending a variety of packets defined in the nmap-service-probes file. There are over 1,000 signatures in this file, which are highly useful in determining various kinds of target services based on their network behavior. Nmap also attempts to conduct an SSL handshake over open TCP ports, and if SSL is supported, it then probes the target port across the SSL connection to get version information. Nmap also sends null Remote Procedure Call commands to listening ports, to determine if it has found a port mapper application that will provide more information for dynamic ports used by RPC services on the box, or whether it has found a particular RPC-based service.

When invoked with the --version-trace option, Nmap displays each step of its version probe on its output, to give its user a feel for how it is attempting to determine the target service in real time.

Other Version Scanning and Information Gathering Tools

- THC Amap
  - Free from http://freeworld.thc.org/thc-amap
  - Amap can do a port scan itself, or...
  - ...provide Amap with the output file from Nmap (generated using the Nmap "–oG filename" option)
  - It sends triggers to each open port (defined in the appdefs.trig file)
  - It looks for defined responses (from the appdefs.resp file)
  - A useful second opinion to the Nmap version scan

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 113

While Nmap's version probe functionality works quite well, a second opinion on what it discovers can be quite helpful. The Amap tool created by The Hacker's Choice (THC) group also performs quite accurate version scanning. Amap can perform a port scan by itself, or use the output of an Nmap scan saved to a file. Nmap, when invoked with the "–oG filename" option, will store its results in a format that Amap can read. The –oG stands for "greppable" format of output for Nmap.

Amap then sends triggers to each open port. These triggers include connecting to TCP ports and listening to what comes back, as well as making SSL connections. Amap's triggers are defined in the appdefs.trig file. When a response comes back from a target port, Amap consults its appdefs.resp file to see if it has a match for the given service type.

Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

- Scanning Goals and Types
- Overall Scanning Tips
  - Sniffing with tcpdump
- Network Sweeping with Scapy
  - Scapy/tcpdump Exercise
- Network Tracing
- Port Scanning
  - Nmap
  - Nmap Exercise
- OS Fingerprinting
- Version Scanning
  - ***Nmap -O -sV and Amap Exercise***
- Vulnerability Scanning
  - Nmap Scripting Engine
  - NSE Exercise
  - Nessus
  - Nessus Exercise
  - Other Vuln Scanners
- Enumerating Users
  - Enumerating Exercise
- Netcat for the Pen Tester
  - Netcat Exercise

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 114

To get a feel for version scanning, we'll now perform some exercises, both with Nmap version scanning and Amap. In these exercises, we'll also identify a false positive with Amap and determine why it is happening. We'll also see an unusual port in our Amap analysis, and attempt to discern its network behavior using Scapy and tcpdump.

In these exercises, we are attempting to model the troubleshooting and analysis process that penetration testers and ethical hackers must go through to refine their results when anomalies are discovered.

Exercise: Nmap OS Fingerprinting

- We will perform Nmap OS fingerprinting of all systems on our target network
- First, run tcpdump so that it sniffs all packets going between your machine and the target network of 10.10.10
- Then, invoke Nmap in one command configured as follows:
  - Don't resolve names
  - Use OS fingerprinting
  - Do a TCP connect scan (3-way handshake)
  - Scan target ports 1-1024
  - Scan the target network 10.10.10.1-255
- Use run-time interaction by hitting the space key to see Nmap's current activity and progress

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved

For this exercise, we are going to start by running Nmap's OS fingerprinting features. Start up tcpdump so that it will sniff all packets from your machine and the 10.10.10 network without resolving names, as follows:

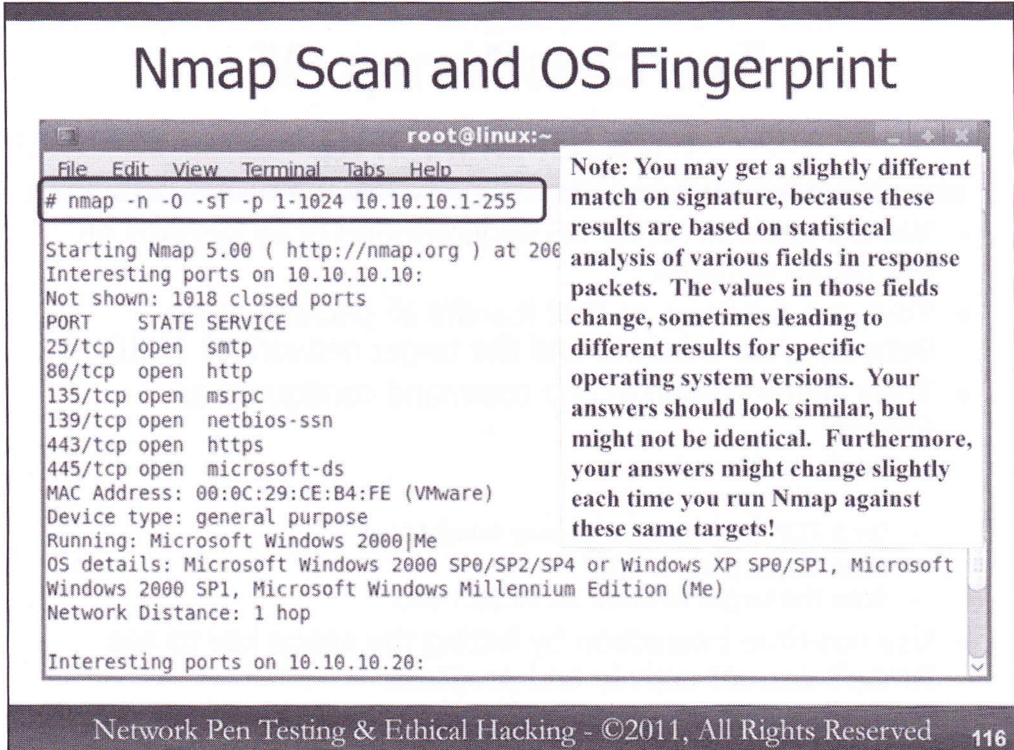
```
# tcpdump -nn host [YourLinuxIPaddr] and net 10.10.10
```

Now, invoke Nmap to do the following:

- Not resolve names (have it display IP addresses instead)
- Use OS fingerprinting
- Perform a TCP connect scan (the three-way handshake for each open port)
- Scan target ports 1 through 1024
- Scan the target network 10.10.10.1-255

Try to compose this Nmap command line yourself, without peeking at the next slide. If you must, flip the page to see the command.

While it is running, periodically check on Nmap's progress by looking at your sniffer output. Also, hit the space key every once and a while to see what Nmap is up to.



To make Nmap perform the scan described on the previous slide, we invoke it as follows:

```
# nmap -n -O -sT -p 1-1024 10.10.10.1-255
```

The –n option makes Nmap use IP address numbers instead of names. The –O (that's a letter O, not a zero) tells Nmap to perform OS fingerprinting, which uses the second generation capability. The –sT configures Nmap to do a TCP scan completing the three-way handshake (a connect scan). We've directed it to scan ports 1 to 1024 with the –p 1-1024 syntax. And, of course, our targets all fall on 10.10.10.1-255.

Note the results in Nmap's output. Was it able to identify the operating system types of 10.10.10.10, 10.10.10.20, 10.10.10.50, and 10.10.10.60?

Please note that you may get a slightly different match on signature from what you see on the slide above, because these results are based on statistical analysis of various fields in response packets, which vary from time to time even on the same target machine. The values in those fields change, sometimes leading to different results for specific operating system versions. Your answers should look similar, but might not be identical. In fact, your answers might change slightly each time you run Nmap against these same targets due to this field sampling and analysis performed by Nmap!

Nmap Version Scan

- Next, let's do a version scan of some of the hosts
  - Start with 10.10.10.10
  - Configure Nmap not to resolve domain names
  - Perform a version scan
  - Use target ports 1-150
- Nmap bases its version scan on the contents of the file nmap-service-probes
  - "Probe" lines indicate what to send
  - "match" lines indicate what to search for in responses

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 117

Next, we'll do an Nmap version scan, but only of ports between 1 and 150, and with one target host at a time. Start with 10.10.10.10.

Your Nmap command should look like:

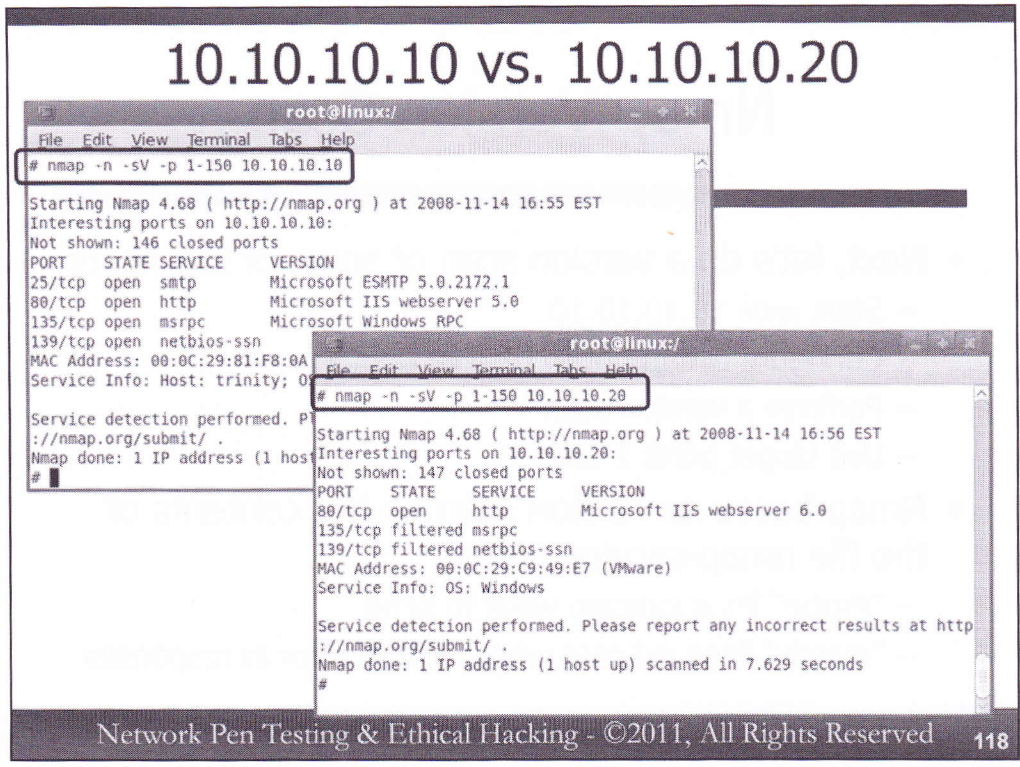
```
# nmap -n -sV -p 1-150 10.10.10.10
```

Compare your results to those discovered when you performed the –sT port scan on the previous slide for 10.10.10.10. Are they different? How?

Then, in another window right next to your first, proceed to do the same kind of scan against 10.10.10.20. Do you notice any differences in the –sV output of 10.10.10.10 and 10.10.10.20?

Next, try 10.10.10.50. It is a Linux machine, as is 10.10.10.60.

Nmap bases its analysis of services on the contents of a file called nmap-service-probes, located in the Nmap data directory (typically /usr/share/nmap). In that file, lines that start with "Probe" indicate the messages to send to target services, while lines that start with "match" indicate the response text to look for when identifying the given service.



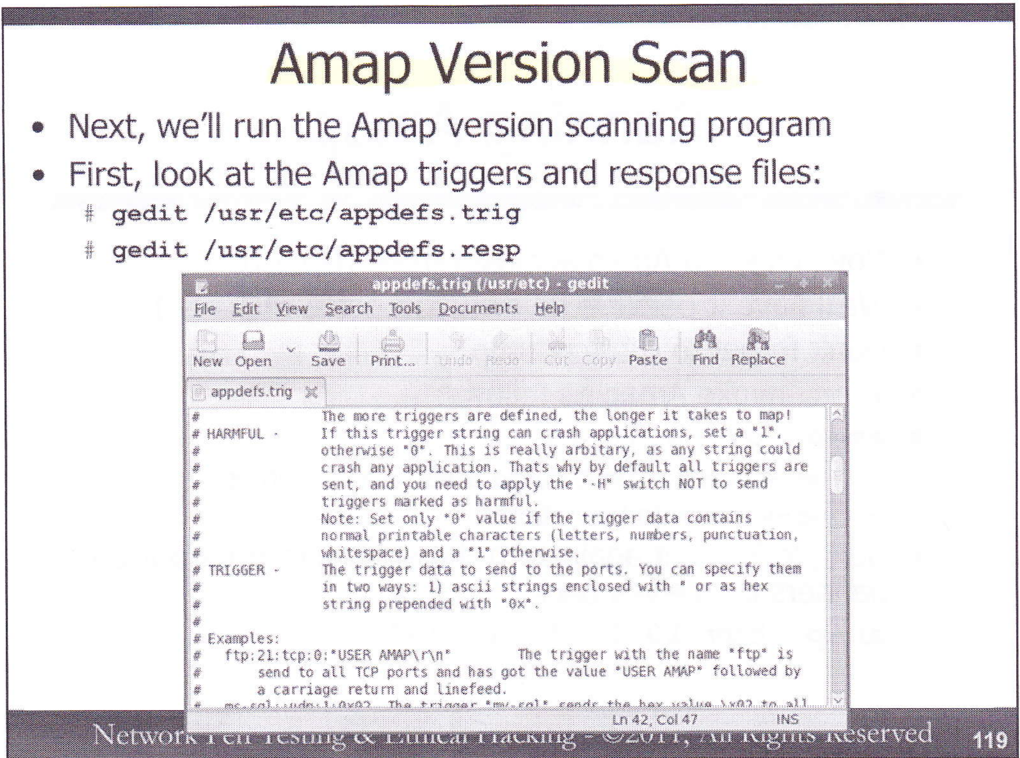
The screen shot on this page shows the results of the version scans of ports 1-150 on 10.10.10.10 and 10.10.10.20. Ideally, you should have similar (but possibly not exactly the same) results on your own system.

In particular, note that 10.10.10.10 is listening on TCP port 25, while 10.10.10.20 is not. This port speaks ESMTP, the Extended Simple Mail Transfer Protocol, with a very precise version number and an indication that it's Microsoft's version. That's a mail server all right.

Both machines are running web servers, identifying themselves as IIS. But, the IIS version on 10.10.10.20 is newer than the one on 10.10.10.10.

Additionally, note that TCP 135 and 139 are open on 10.10.10.10, while they are labeled as "filtered" on 10.10.10.20.

Both systems are Windows, but they are quite different versions with significant differences in configuration.



For the next component of the exercise, we'll run Amap against the same targets, comparing its findings with Nmap's results. First, let's look at the contents of the Amap triggers file (appdefs.trig) and its responses file (appdefs.resp). You can open these files on your Linux machine using any editor you are comfortable with, such as vi, emacs, or others. If you don't know how to use vi or emacs, a simple editor for Linux that is quick and easy to use is gedit. Open the files as follows:

```
# gedit /usr/etc/appdefs.trig
# gedit /usr/etc/appdefs.resp
```

The Amap trigger file specifies which data to send to discovered open ports. In the trigger file, notice the option of HARMFUL described near the top of the file. Some of the triggers could cause a target service to come down, so we must be careful. Within your editor, search for :1: to find the potentially harmful triggers.

The response file tells Amap what to look for in its responses to determine which services are in use. Note that the responses are actually searched using a Perl regular expression (run **man perlre** for more details on how such syntax works).

Running Amap

- Now, let's run Amap against target 10.10.10.10
- We'll have it perform its own port scan, of ports 1-150
- Don't forget to run tcpdump
- Then, invoke Amap as follows:

```
# amap -qv 10.10.10.10 1-150
```

  - The –q tells it to omit closed ports from its output
  - The –v means to be verbose
- Next, let's run it again with the –b flag to print out the banners it receives back

```
# amap -bqv 10.10.10.10 1-150
```

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 120

We will now run Amap against 10.10.10.10, configured to scan and check TCP ports 1-150. As Amap runs, make sure you have tcpdump running so that you can watch as your packets are sent to the target machine.

Invoke Amap as follows:

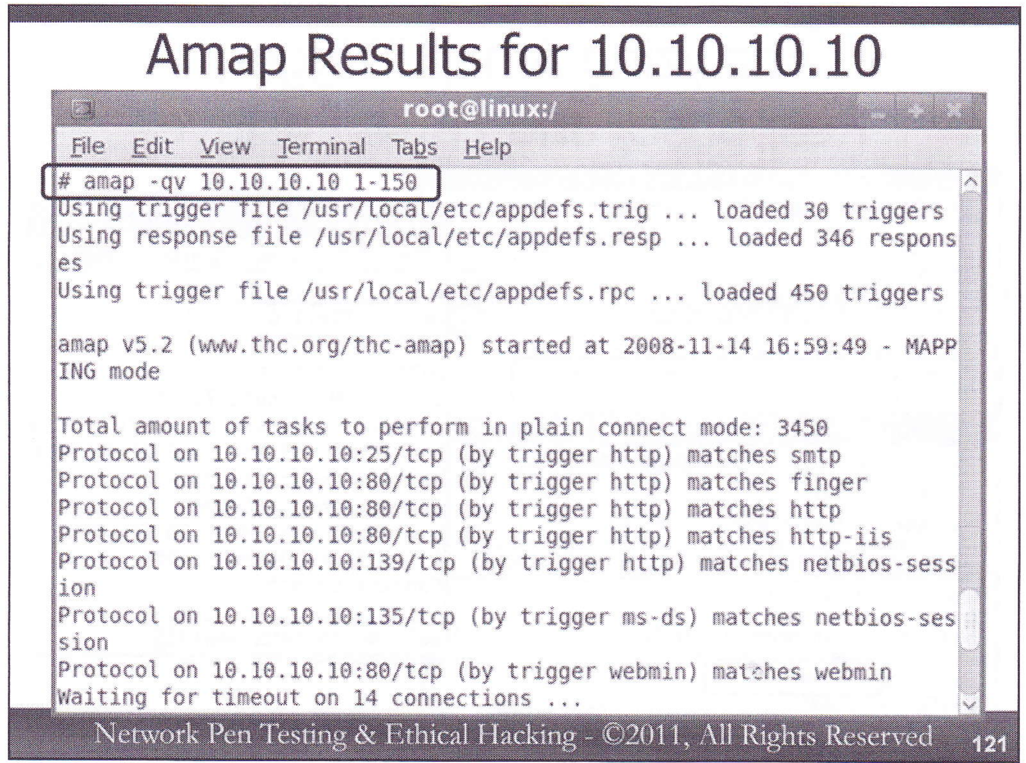
```
# amap -qv 10.10.10.10 1-150
```

This command tells Amap to omit closed ports from its output (-q) and to give us verbose results (-v) of the scan against 10.10.10.10, using ports 1-150 (TCP is the default). Look at its output, and compare with what Nmap told us.

To get more detail, let's have Amap print the banners it received on each port, by including the –b flag in its command:

```
# amap -bqv 10.10.10.10 1-150
```

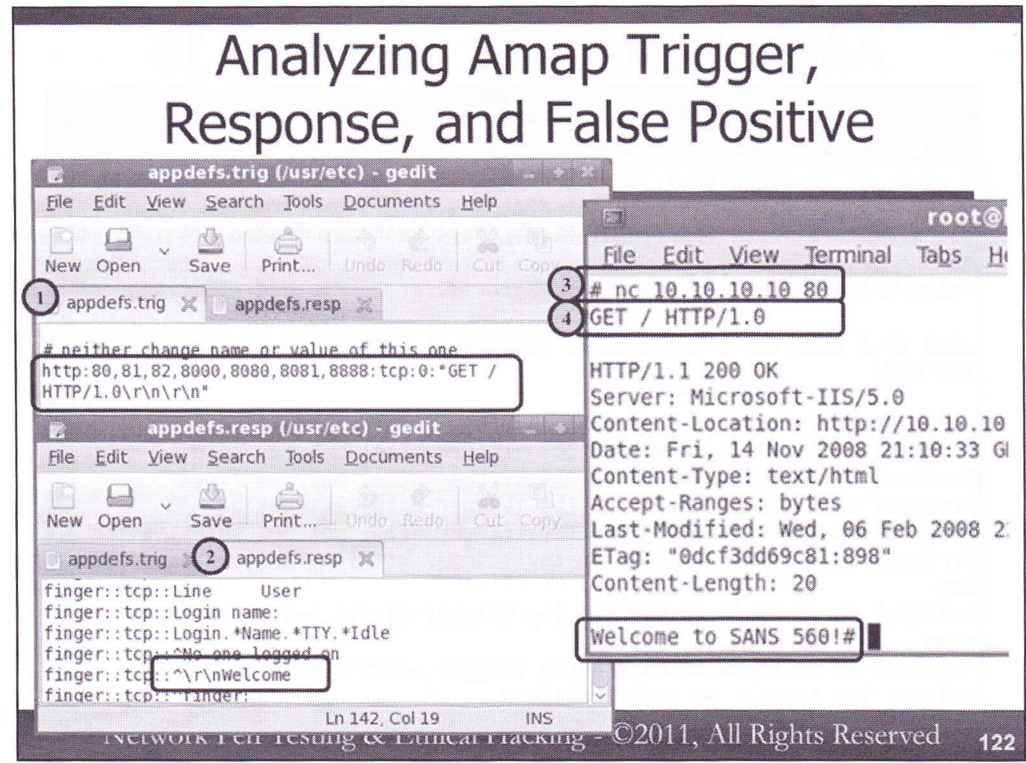
This provides a lot more detail, giving us results closer to what we got with Nmap.



Here are the Amap results (without the –b flag). Note how it tells us the number of triggers and responses it loads. The appdefs.rpc file contains an additional set of triggers associated with Remote Procedure Call (RPC) services.

We received similar findings to what we discovered with Nmap. We can see the trigger that elicited the given response (with the output text "by trigger"). We can also see the response that matched.

Note, however, that the TCP port 80 finding matched multiple responses, including http and http-iis as we'd expect. It also matches "finger", which seems strange. The Amap response associated with the finger service includes several different responses, and the text on the web page on 10.10.10.10 matches it. We've gotten a false positive! Let's see why.



In Step 1, let's look at the Amap trigger for http, by opening the /usr/etc/appdefs.trig file:

```
# gedit /usr/etc/appdefs.trig
```

Search for http. We can see that Amap sends the string: "GET / HTTP/1.0\r\n\r\n".

In Step 2, look in the **/usr/etc/appdefs.resp** file, searching for "finger". Just start browsing down the file, as the finger signatures are pretty easy to locate by hand. We can see that several possible matches appear for "finger", including:

```
^\r\nWelcome
```

In Step 3, let's manually simulate what Amap is seeing here. Use Netcat, the general purpose connection tool, to connect to 10.10.10.10 on port 80:

```
# nc 10.10.10.10 80
```

You will see no response, but the connection will be made. Now, in Step 4, let's manually type in the trigger used by Amap:

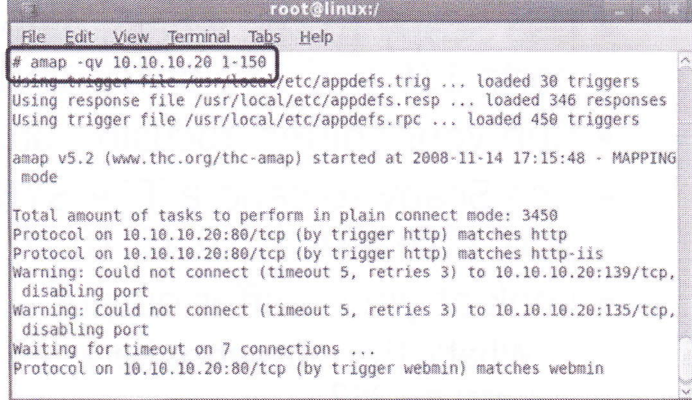
```
GET / HTTP/1.0
```

Hit the carriage return twice.

Look at the response from the target in Netcat's output. Do you see the "Welcome" text? That's why we got the Amap false positive for the "finger" response.

Now, Analyze 10.10.10.20

- Continue your Amap analysis on 10.10.10.20
- Run both with and without the –b option
- Notice the difficulty it has with TCP ports 135 and 139



Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 123

Next, let's continue our analysis by doing an Amap scan of 10.10.10.20. Run it both with and without the –b option, as follows, again scanning ports 1-150:

```
# amap -qv 10.10.10.20 1-150

# amap -bqv 10.10.10.20 1-150
```

In the output, look for the messages associated with TCP ports 135 and 139 (this is easier to see in the Amap run without the banner grabbing –b flag). Note that it sensed that these ports were open, but couldn't make a connection to them to perform version detection. What's happening with these ports? In the next component of the exercise (on the next page), we'll analyze them in more detail.

Note also that you may get false positives with triggers associated with ms-remote-desktop-protocol and smtp on port 80 on target 10.10.10.20. These are again anomalies, and they do not always appear during the scan, because some specific characters in the web response aren't always delivered back from the server.

Investigating the Different Ports

- Use Scapy to see what's different about ports 135 and 139 on 10.10.10.20
- Run your sniffer, focusing on TCP
- Use Scapy to send a TCP SYN packet to port 130-140 on 10.10.10.20
- Look at your sniffer output
  - What's the difference between 130-134 versus 135?

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 124

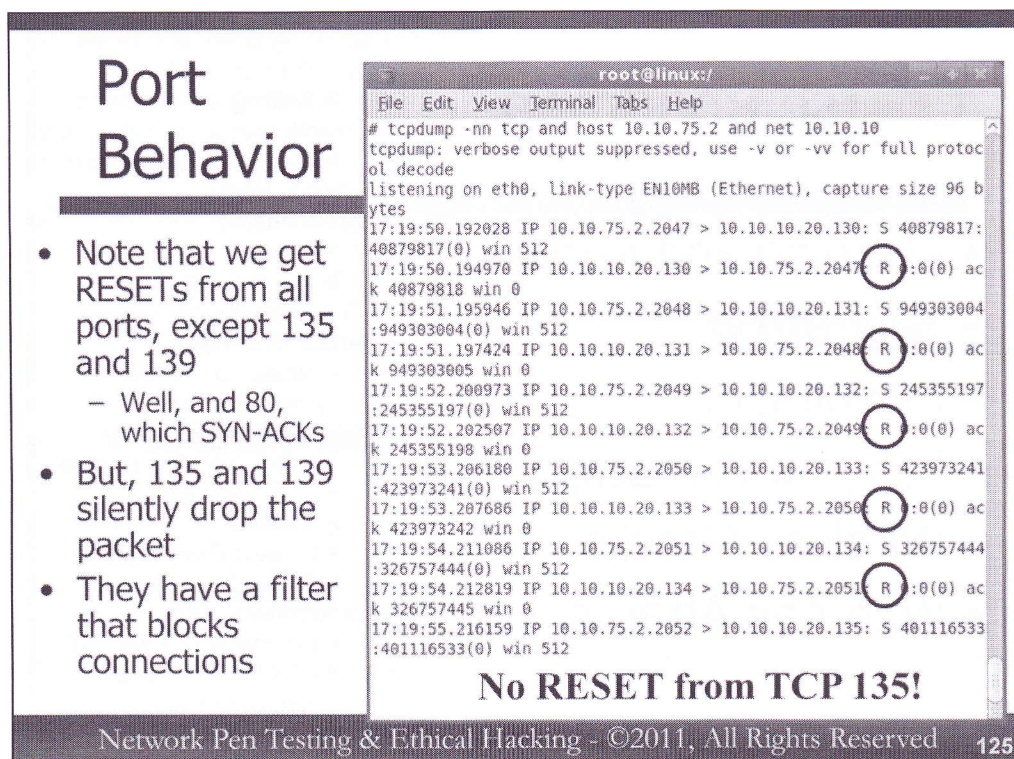
Next, we'll look at why we are getting different Amap behavior for port 135 and 139 on 10.10.10.20. Why does it sense that the port is open, yet it cannot make a connection? Our trusty friends Scapy and tcpdump will help us find out.

First, run tcpdump so that it is looking for all packets that your machine sends to the target network of 10.10.10. To cut down on clutter, have it gather only TCP packets.

Then, write a Scapy invocation that sends TCP SYN packets to 10.10.10.20 on all ports from 130 to 140.

Hint: Remember to put parents ( ) around your port range for Scapy.

Try to compose these commands yourself. If you are having trouble, you can refer back to the sections on Scapy and tcpdump we covered earlier. Or, if you'd prefer, flip to the next page for more info.



To achieve what we described on the previous slide, we can invoke tcpdump as follows:

```
# tcpdump -nn tcp and host [YourLinuxIPaddr] and net 10.10.10
```

Then, we can run Scapy as follows:

```
# scapy
>>> ans,unans=sr(IP(dst="10.10.10.20")/TCP(dport=(130,140)))
```

Hit CTRL-C after you see "Finished to send 11 packets".

Now, look in the tcpdump's output. Note that for ports 130, 131, 132, 133, and 134, we get a RESET (R) response. But, for port 135 and 139, we don't get anything back. It silently rejects our packet because of a packet filter.

Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

- Scanning Goals and Types
- Overall Scanning Tips
  - Sniffing with tcpdump
- Network Sweeping with Scapy
  - Scapy/tcpdump Exercise
- Network Tracing
- Port Scanning
  - Nmap
  - Nmap Exercise
- OS Fingerprinting
- Version Scanning
  - Nmap -O -sV and Amap Exercise
- ***Vulnerability Scanning***
  - Nmap Scripting Engine
  - NSE Exercise
  - Nessus
  - Nessus Exercise
  - Other Vuln Scanners
- Enumerating Users
  - Enumerating Exercise
- Netcat for the Pen Tester
  - Netcat Exercise

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 126

Now, we have reached the topic of Vulnerability Scanning. The goal of these kinds of scans is to find potential security flaws in the target environment. Discovering misconfigurations, unpatched services, architectural mistakes, and more are what this component of our test is all about.

Methods for Discovering Vulnerabilities

- How can we determine whether a given piece of software is vulnerable?
  1) Check software version number
     - Compensating controls might block exploitation (network- or host-based IPS, etc.)
  2) Check protocol version number spoken
  3) Look at its behavior – somewhat invasive
  4) Check its configuration – more invasive
     - Requires access to target
     - Or, requires configuration documentation from target environment personnel

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 127

Vulnerability scanning tools can determine whether a target system is vulnerable to attack in several different ways. The primary (but by no means exclusive) methods employed by today's vulnerability scanners for finding security flaws include:

1) *Checking version numbers:* If the software running on a target machine has a version number that is known to be flawed, we can have a reasonable expectation that the software is indeed vulnerable. There might be compensating controls in place that block exploitation, such as a network- or host-based IPS. However, even with compensating controls, most organizations strive to upgrade and patch out-of-date software.

2) *Checking protocol versions:* A related method for finding flaws involves checking which protocol versions a given piece of software speaks. Even if we cannot determine the version of the software itself, we might be able to determine that it speaks an older version of a network protocol, possibly indicating that it hasn't been patched or hardened.

3) *Looking at its behavior:* Even if software doesn't provide us a means for ascertaining its version number, a tester's tools can interact with the software across the network (or in certain circumstances locally), measuring whether it exhibits behavior consistent with a vulnerability. These behavior-discoverable vulnerabilities could be due to old software or misconfigurations. Measuring behavior of target programs could be somewhat invasive, as the tester has to interact with the target in various ways.

4) *Checking its configuration:* With local access to a machine, or even with remote access gained via some other mechanism (such as an exploit or password-guessing attack), a tester could analyze a system at a fine-grained level to determine whether the configurations of the programs on the machine exhibit weaknesses. Such tests tend to be even more invasive than the options above, as they requires the tester to gain access to a target or get a copy of the system configuration from an administrator.

More Methods for Discovering Vulnerabilities

5) Run exploit against it – potentially dangerous, but potentially very useful
   - Successful exploit shows the vulnerability is present
   - Helps lower false positives
     - Note that failed exploit does not indicate that the system is secure!

- Not all vulnerabilities lead to exploit
  - Some misconfigurations could be associated with information leakage
  - Others might indicate a concern, but without exploitation being possible

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 128

There is another method for finding vulnerabilities:

5) *Running an exploit against target:* This often most invasive form of vulnerability discovery involves actually trying to exploit the target, potentially taking over the system. Running an exploit could be dangerous, as it could bring down the target service or entire system. But, running an exploit can be very helpful for testers, as successful exploitation proves the presence of a vulnerability (false positive reduction). It should be noted that failed exploitation does not mean that the software is safe, however. It's possible that the tester's exploit failed for any number of reasons, but a different attacker might be able to get it working. So, actual exploitation can lower the number of false positives, but it doesn't really help us manage false negatives.

We'll analyze this issue of the safety of exploitation in further detail at the start of our 560.3 class.

It's also important to note that not all vulnerabilities lead to exploitation. Many vulnerabilities don't let an attacker take over a machine at all. Instead, they could be associated with information leakage or other problems. As penetration testers and ethical hackers, we are interested in all kinds of vulnerabilities in a target environment. Our jobs involve reporting on the issues we discover, whether they can be exploited or not. Obviously, exploitable vulnerabilities have higher importance than non-exploitable issues, but all discovered flaws should be reported.

Nmap Version Scan and Amap as Vulnerability Scanners?

- Couldn't Nmap's version scanning or Amap be used to find vulnerabilities?
  - Yes, by detecting an old version of some software...
  - ...or by formulating packets and pattern matching on the results
  - It is certainly possible... You'll have to look up and interpret those results yourself, by hand
  - Watch out for false positives
- But... Nmap version scanning and Amap are limited
  - They send a packet and scan the response for strings
  - They can't have meaningful communications with multiple back-and-forth messages
- However, the Nmap Scripting Engine can

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 129

As we have seen, both Nmap and Amap support version checking, which sends probe packets to given ports and matches specific strings in the response to determine the version of a service. With that functionality, couldn't we use those tools to find vulnerable systems? We certainly could, by researching the versions of the detected services on the target machines to see if they have a history of flaws. Currently, such research must be done manually by the user of the tool. Nmap and Amap do not tell you that the given target is vulnerable; they merely give you information about the service version, which you must look up.

It's important to note that, while the version scan outputs can give you insight into whether the target is vulnerable, Nmap version scanning and Amap are limited. They send a probe and scrape through its response looking for certain text. They don't have meaningful, complex back-and-forth interactions with targets to measure more complicated behaviors to determine if the given service is vulnerable. Thus, Nmap version scanning and Amap are prone to false positives. It might look like a given service is vulnerable based on its version number. However, it's possible that there are other compensating controls that prevent the issue from being exploited. Simple version checking cannot look for those compensating controls. A more complex back-and-forth interaction is required to measure whether the target has the behavior of a vulnerable service, not just its version.

However, outside of its version scanning functionality, Nmap has been extended to include a powerful feature to let it have complex interactions with targets using scripts to measure for vulnerabilities. This feature is called the Nmap Scripting Engine (NSE).

Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

- Scanning Goals and Types
- Overall Scanning Tips
  - Sniffing with tcpdump
- Network Sweeping with Scapy
  - Scapy/tcpdump Exercise
- Network Tracing
- Port Scanning
  - Nmap
  - Nmap Exercise
- OS Fingerprinting
- Version Scanning
  - Nmap -O -sV and Amap Exercise
- Vulnerability Scanning
  - ***Nmap Scripting Engine***
  - NSE Exercise
  - Nessus
  - Nessus Exercise
  - Other Vuln Scanners
- Enumerating Users
  - Enumerating Exercise
- Netcat for the Pen Tester
  - Netcat Exercise

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 130

Because the Nmap Scripting Engine (NSE) can be used to discover some vulnerabilities, let's zoom in on its functionality in more detail, running an exercise on some of the NSE scripts to see their capabilities.

Nmap Scripting Engine

- Goals of the Nmap Scripting Engine (NSE)
  - Allow for arbitrary messages to be sent or received by Nmap to multiple targets, running scripts in parallel
  - Be easily extendable with community-developed scripts
  - Support extended network discovery (whois, DNS, etc.)
  - Perform more sophisticated version detection
  - Conduct vulnerability scanning
  - Detect infected or backdoored systems
  - Exploit discovered vulnerabilities
- May someday rival Nessus and NASL as a general-purpose, free, open source vulnerability scanner

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 131

The Nmap Scripting Engine has numerous goals, which really extend the capabilities of Nmap beyond mere port scanning and OS fingerprinting. These goals include:

- Utilize Nmap's efficient multi-threaded architecture to send arbitrary messages and receive responses in parallel to and from multiple targets
- Create an environment so that a development community can write and release free scripts that can easily be incorporated into scans by all Nmap users
- Support network discovery options that augment Nmap's port scanning and OS fingerprinting features, including whois lookups, DNS interrogation, etc.
- Enhance version detection functionality beyond "probe and match" to look more deeply into the interaction with a target
- Perform vulnerability scanning of target systems to find configuration flaws and other issues
- Detect systems that have been infected with malware or backdoors based on their network behavior
- Support exploitation of given flaws to gain access to a target machine or its information, not supplanting the Metasploit exploitation framework, but offering some subset of exploit functionality integrated into Nmap

With these goals, the Nmap Scripting Engine could one-day rival Nessus and its Nessus Attack Scripting Language (NASL), if developers increasingly embrace the NSE.

Nmap Scripting Engine Scripts

- Written in the Lua programming language
  - Often used in games, Lua is fast, flexible, and free, with a small interpreter that works across platforms and is easily embedded inside of other applications
  - Described in detail at www.lua.org
- To invoke NSE:
  - To run all scripts in the category of 'default':

```
# nmap -sC [target] -p [ports]
```

  - To run an individual script:

```
# nmap --script=[all,category,dir,script...]
  [target] -p [ports]
```

  - Add "--script-trace" for detailed output from each script

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 132

Nmap scripts are written in the Lua scripting language, which is commonly used in computer games. Lua is widely regarded as a flexible and extremely fast scripting environment. It's interpreter is free, cross-platform, and has a very small footprint, making it ideal for incorporation into other applications, such as Nmap. Lua is named after the Portuguese word for "moon", and is described in detail at www.lua.org. The Snort network-based Intrusion Detection System (IDS) and Wireshark sniffer also offer Lua support.

To invoke the Nmap Scripting Engine, a user would invoke it either with the –sC option (to run all scripts in the 'default' category), or with the --script= option to choose specific scripts. When running specific scripts, a user could choose all (to run all scripts), script categories (which we'll describe shortly), a directory containing several scripts, or individual scripts by name. Alternatively, these different methods can be combined in a comma-separated list.

To get detailed, step-by-step output from a script as it runs, Nmap supports the `--script-trace` option, which operates rather like Nmap's `--packet-trace` option, but is focused on scripts.

NSE Script Categories

- Developers who create NSE scripts identify each script in one or more categories:
  - Safe: Not designed to crash targets, consume bandwidth, or exploit vulns
  - Intrusive: May leave logs, guess passwords, or otherwise impact the target
  - Auth: Test for issues associated with authentication
  - Malware: Detect network-accessible malware or backdoors
  - Version: Detect the version of target's services
  - Discovery: Info gathering about target environment
  - Vuln: Look for a given vulnerability in the target
  - External: Sends information to third-party for lookup (example: whois). Third party could record query, response, or IP address
  - Default: Run this set of scripts when Nmap is invoked just using -sC or –A without a category of individual script specified

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 133

The Nmap Scripting Engine supports several different categories of tests, with each script fitting into one or more script categories.

The first category is "Safe" scripts, which are designed to have minimal impact on a target, neither crashing it nor leaving any entries in its logs. Furthermore, these scripts should not utilize excessive bandwidth, nor should they exploit vulnerabilities.

The second category is "Intrusive" scripts, which may leave logs, guess passwords (which could lock out accounts), and have other impacts on the target machines.

The "Auth" category are tests associated with authentication, including some password guessing and authentication bypass tests.

The "Malware" category measures for the presence of an infection or backdoor on the target. Examples in this category include checks to see if a port used by a given malware specimen is open on the target and whether it responds as that malware would.

The "Version" category of scripts attempts to determine which versions of services are present on the target. These scripts can be more complex than the normal version checking of Nmap.

"Discovery" scripts determine more information about the network environment associated with the target, and include some whois and DNS lookups, among other functions.

The "Vuln" category includes scripts that determine whether a given target has a given security flaw, such as a misconfiguration or an unpatched service.

The "External" category includes scripts that may send information to a third-party database or other system on the Internet to pull additional data. Whois lookups fall into this category, because they send data to whois servers, which may record the query information.

And, finally, the 'Default' category includes scripts that are run when Nmap is invoked with just the -sC or –A syntax and no specific script category or individual script specified.

Some Example NSE Scripts

- Scripts are located in their own directory inside the Nmap data directory
  - Often /usr/share/nmap/scripts/
- The file script.db inventories and categorizes the various types
- Several dozen scripts look for a variety of different conditions:
  - Look for common SMB vulnerabilities on target Windows machines
  - Determine if an FTP server supports bounce scans
  - DNS servers supporting zone transfer
  - Tell if a Windows shell is on a given port
  - Test if SMTP server can be used as a relay
  - Many, many more

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 134

The scripts associated with NSE are found in their own directory called, appropriately enough, scripts, which is located by default in the Nmap data directory. On many Nmap installs (including the one associated with the VMware image for this class), they are located in /usr/share/nmap/scripts.

Inside this directory, there is a file called scripts.db, which inventories the several dozen scripts in the directory. This handy file simply associates the given script with its category. Thus, we can easily search for "safe" scripts by running:

```
# grep safe /usr/share/nmap/scripts/script.db
```

"Intrusive" scripts can be found by with:

```
# grep intrusive /usr/share/nmap/scripts/script.db
```

Note that the categories are in all-lowercase within this file.

In the scripts directory, there are several dozen scripts. Some of the more interesting include:

- A script to determine if an FTP server supports Nmap bounce scans
- A script to find common SMB vulnerabilities on Windows targets
- A script that attempts to do a DNS zone transfer from a target
- A script that looks for Windows shells on TCP port 8888, which could easily be altered to look for them elsewhere
- A script that analyzes whether an SMTP server can be used as a mail relay, thus leaving them open to abuse by spammers

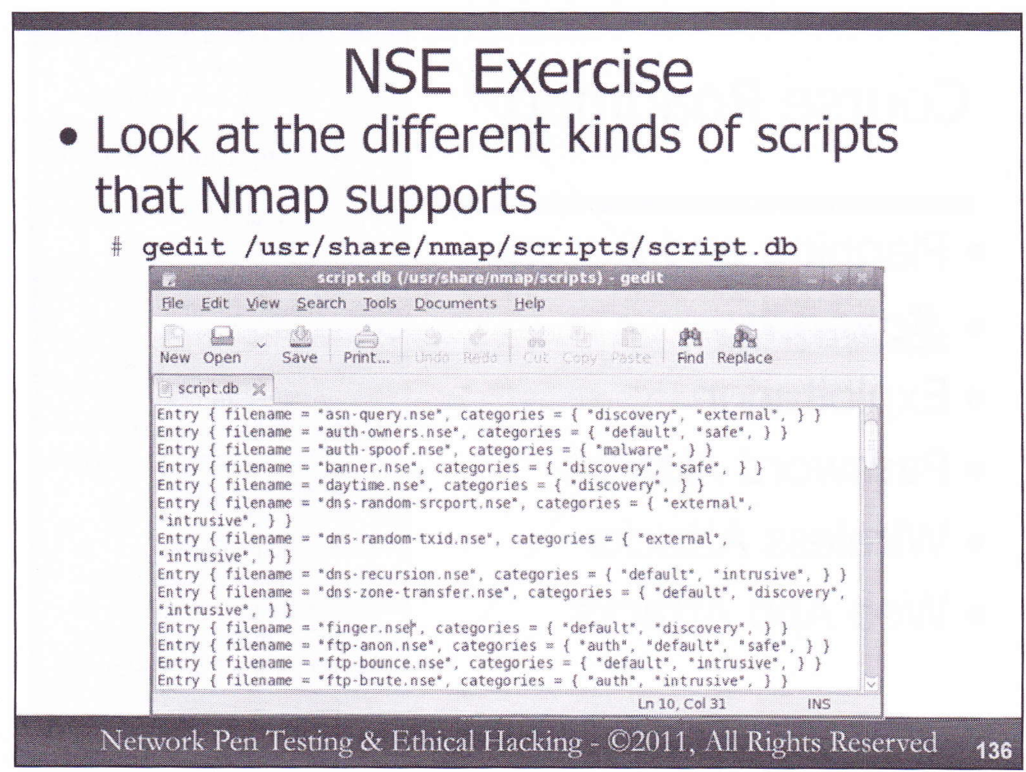
Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

- Scanning Goals and Types
- Overall Scanning Tips
  - Sniffing with tcpdump
- Network Sweeping with Scapy
  - Scapy/tcpdump Exercise
- Network Tracing
- Port Scanning
  - Nmap
  - Nmap Exercise
- OS Fingerprinting
- Version Scanning
  - Nmap -O -sV and Amap Exercise
- Vulnerability Scanning
  - Nmap Scripting Engine
  - ***NSE Exercise***
  - Nessus
  - Nessus Exercise
  - Other Vuln Scanners
- Enumerating Users
  - Enumerating Exercise
- Netcat for the Pen Tester
  - Netcat Exercise

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 135

To get a better feel for the various vulnerability scanning tools we've been discussing, let's run some of them against our test targets in the lab environment. For these exercises, we'll be experimenting with Nessus and the Nmap Scripting Engine.



We will now look at the functionality of the Nmap Scripting Engine. Start by opening up the file that contains the inventory of all of the scripts that have been defined for NSE:

```
# gedit /usr/share/nmap/scripts/script.db
```

If you don't like gedit, a simple WYSIWIG editor, feel free to use another Linux/Unix editor with which you are familiar, such as vi, emacs, nano, etc.

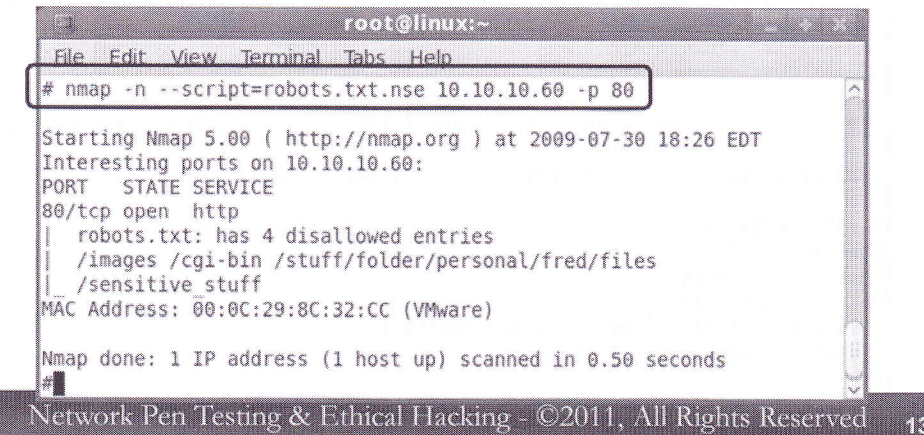
This scripts.db file has a very simple format, essentially just mapping script categories such as "safe", "intrusive", and "vulnerability" to the specific script file, which ends in .nse. Note that some scripts are in multiple categories, such as dns-zone-transfer.nse, which is in the default, discovery, and intrusive categories.

Let's count the number of scripts in some of the categories, by sending the script.db file through the wc (wordcount) command with the –l (where that lower-case L stands for linecount) option:

```
# cd /usr/share/nmap/scripts
# cat script.db | grep safe | wc -l
```

← NOTE: That is a dash lowercase L, not a dash one.

```
# cat script.db | grep discovery | wc -l
# cat script.db | grep intrusive | wc -l
```



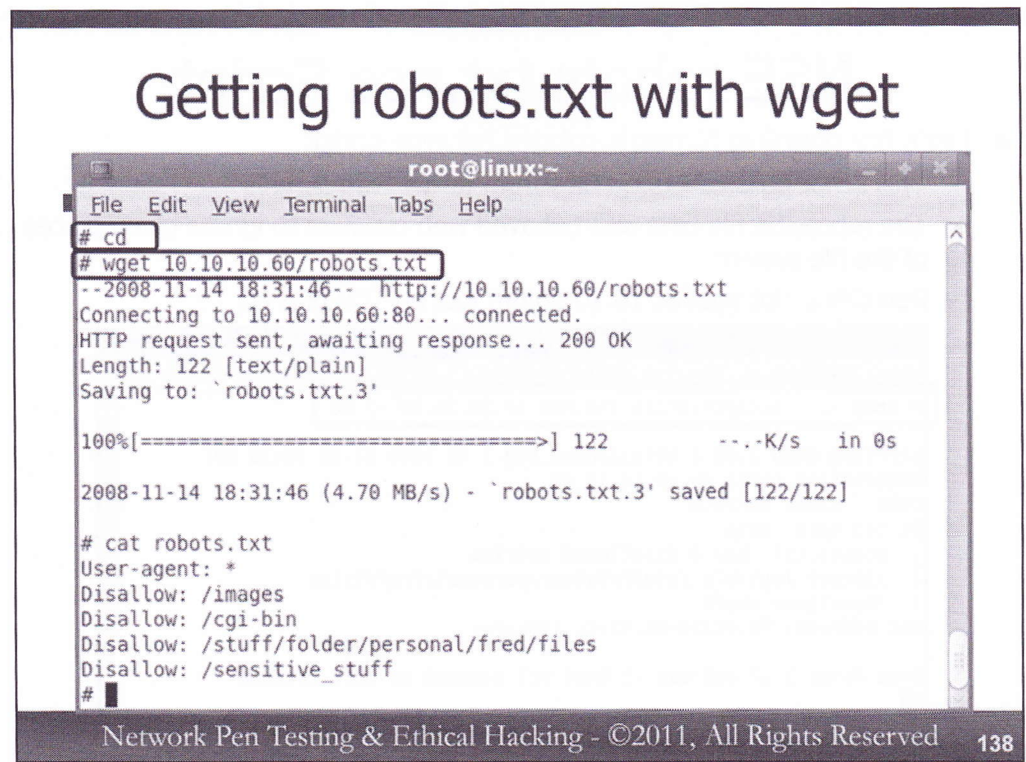
Let's experiment with the robots.txt.nse script. This script will pull the robots.txt file from target web servers. The robots.txt file tells well-behaved web crawlers (such as those from the major search engines that are attempting to find new pages on the world wide web) to ignore given directories or pages on a website, because they have information that the website owner doesn't want to be included in search engines. In other words, robots.txt tells well-behaved crawlers what to ignore, possibly because it is sensitive. Attackers often focus on the directories and files listed in robots.txt, because they may include some juicy information. As a penetration tester, we'd very much like to have a copy of the robots.txt files from all web servers in our target range. Note that robots.txt is a file readable by anyone who accesses the website and is usually included in the document root of the web server. Thus, it really isn't a security feature; it merely helps keep things out of search engines that shouldn't be there. But, it is also a red flag indicating where more interesting parts of a web site might be located in the file system structure.

The Nmap script robots.txt.nse pulls robots.txt files from target machines. Let's test it by invoking it as follows:

```
# nmap -n --script=robots.txt.nse 10.10.10.60 -p 80
```

Note that we are just having the script focus on TCP port 80 to save time. During a more comprehensive scan, we would have invoked it with –sV and possibly with scanning all target TCP ports 1-65535 (-p 1-65535).

In the Nmap output, you should see the directories that are listed in the robots.txt file of the target web site.



While the output from the Nmap robots.txt.nse information is helpful, note that the script doesn't display the full contents of robots.txt. It merely lists directories, without the "Disallow:" notation and any "User-agent" restrictions that help specify which browser types should get access to which data. To get that information, you'd have to surf to the website from a browser, going to 10.10.10.60/robots.txt. Let's do that using the wget tool, a helpful widget for fetching web pages via the command line. We'll start by changing into our home directory:

```
# cd
```

Then, let's get the robots.txt file from 10.10.10.60.

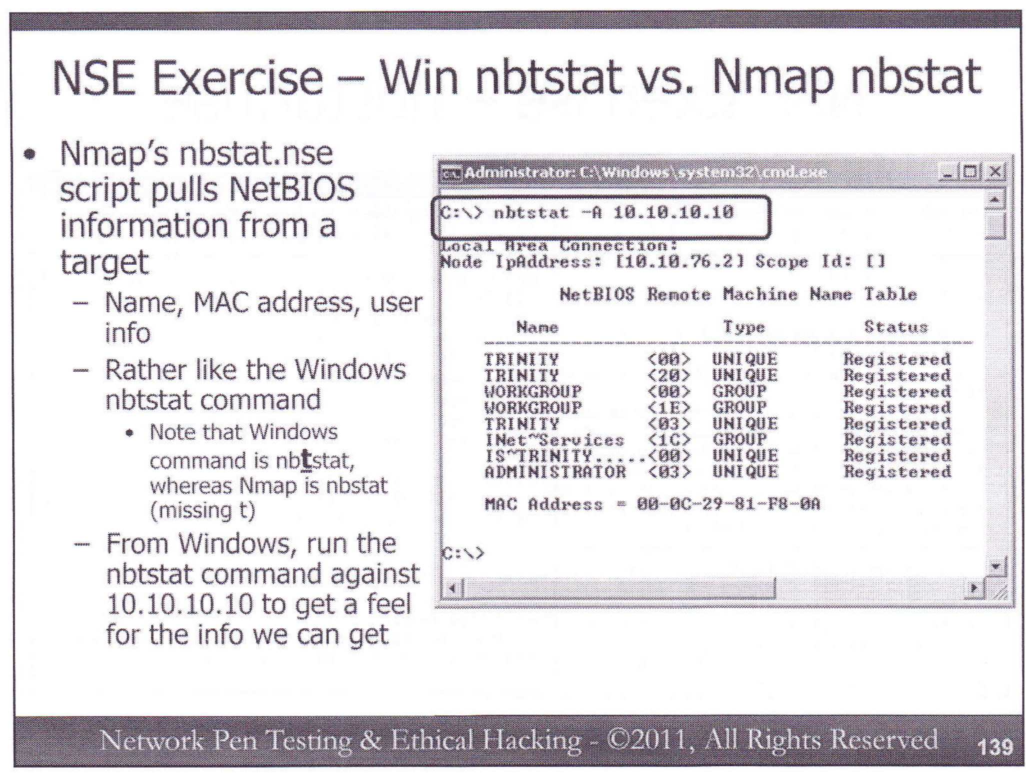
```
# wget 10.10.10.60/robots.txt
```

This should fetch the page. To display it, simply type:

```
# cat robots.txt
```

So, if we can fetch robots.txt files manually, then of what use is the robots.txt.nse script? It can be used in an Nmap scan of a large number of machines, pulling back information about robots.txt file restrictions and the tipping off a tester to go back and investigate a given robots.txt file on specific target machines in more detail. In fact, let's pull robots.txt files from all of our target machines:

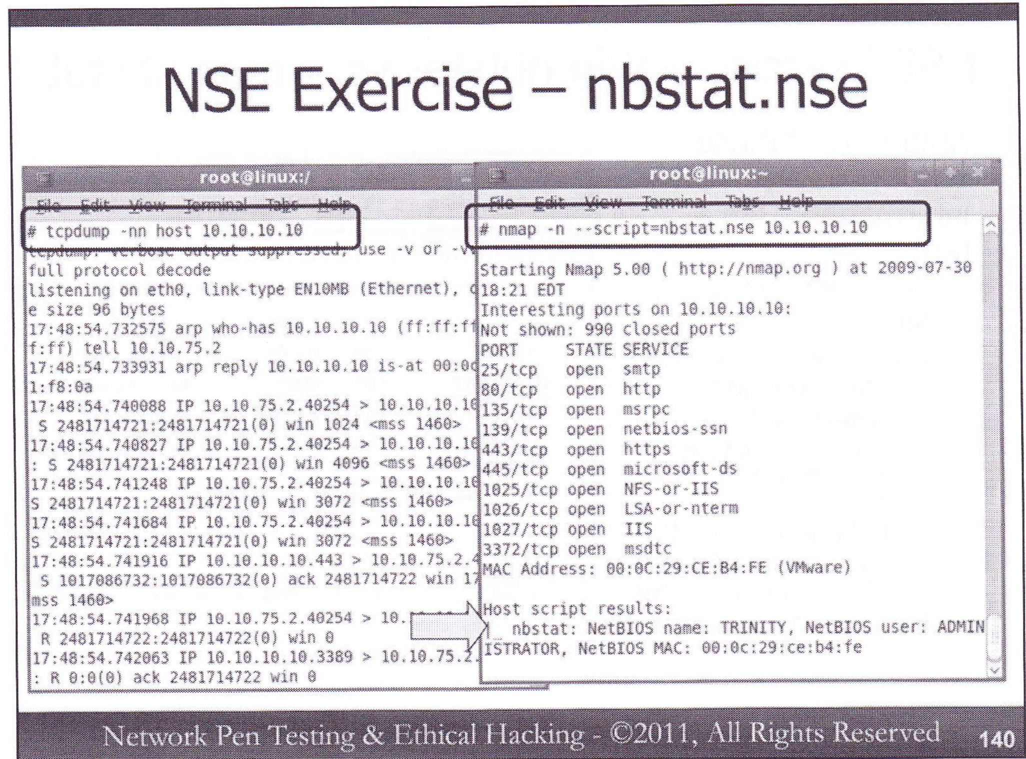
```
# nmap -n --script=robots.txt.nse 10.10.10.1-255 -p 80
```



Next, let's explore the Nmap nbstat.nse script. This script works like the nbtstat command in Windows, which pulls NetBIOS over TCP statistics, including machine names, MAC addresses, and user names. To get a feel for what nbtstat output looks like on Windows, bring up a cmd.exe and run it against the Windows target 10.10.10.10 as follows (the –A means that we want to use an IP address):

```
C:\> nbtstat -A 10.10.10.10
```

We can see the machine name, MAC address, and a user name of Administrator. While we can run this command from Windows, we can also gather similar information from our Nmap scans via the nbstat.nse script. Note that although the Windows command is nbtstat (with a T), the Nmap script is called nbstat, without the t between the b and the s characters.



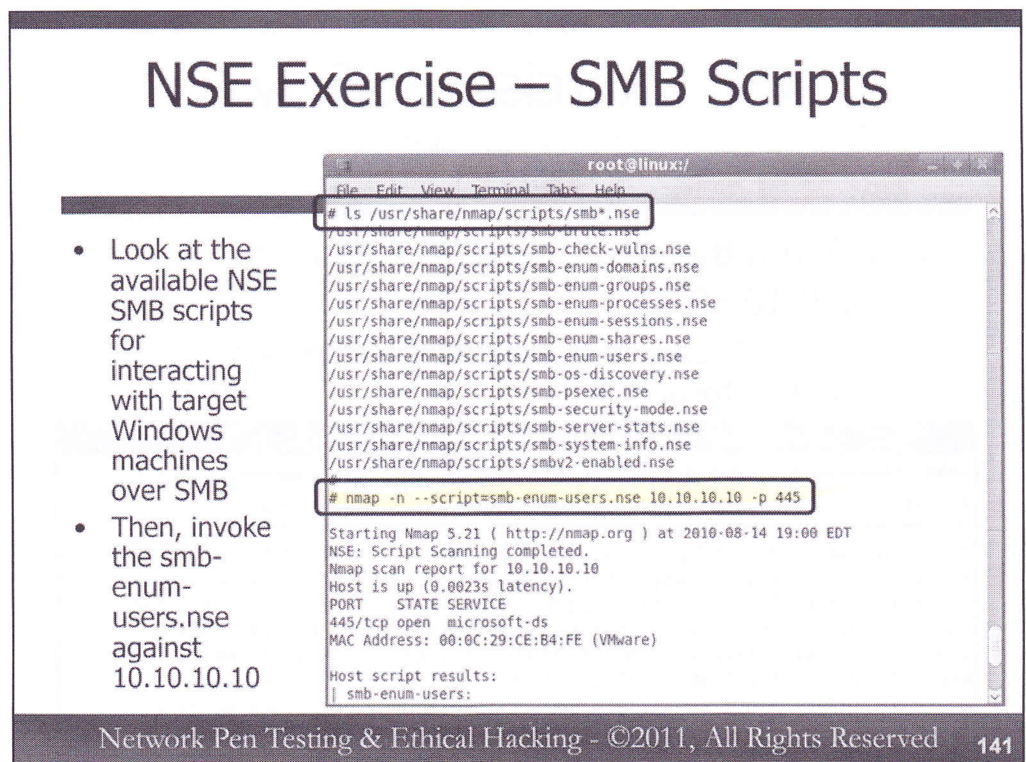
Next, move to Linux and run a sniffer so that we can see all traffic going to or from IP address 10.10.10.10, without resolving names or services. We want to get a feel for what Nmap does when invoked to run a script without specifying a target port.

```
# tcpdump -nn host 10.10.10.10
```

Then, run Nmap against 10.10.10.10, configured to run the nbstat.nse script:

```
# nmap -n --script=nbstat.nse 10.10.10.10
```

As Nmap runs, look at the output of your sniffer. Notice anything interesting? Nmap is doing a port scan of the target machine, analyzing the interesting ports on the box. Even though we told it to run only the nbstat.nse script, it does a port scan. Why? Because it needs to know which ports are open so that it can determine if the service(s) the script tests are available. A full three-way handshake scan (a "connect" scan) has been run. Then, if the appropriate ports are open, Nmap runs the nbstat.nse script against the target, showing the results in its output. You should see, at the bottom of your Nmap output, a line that says "NBSTAT: NetBIOS name:" and so on, with the results from the nbstat.nse script.



Next, let's look at the Server Message Block (SMB) scripts included with Nmap, many of which were written by Ron Bowes. First, we'll look at the name of all of the SMB NSE scripts included with this version of Nmap:

```
# ls /usr/share/nmap/script/smb*.nse
```

Here, you can see scripts that will let us perform brute force password guessing (smb-brute.nse), check for common vulnerabilities (smb-check-vulns.nse), and plunder the target for information (smb-enum-domains, groups, processes, etc.).

Additionally, the smb-psexec command allows us to provide a username and password in the administrators group (with --script-args=smbuser=[AdminUser],smbpass=[AdminPass],config=[ConfigFileName, stored in /usr/share/nmap/nselib/data/psexec]), as well as one or more commands we want to run in a configuration file, and this script will attempt to cause any targets that it discovers communicating using SMB to run the commands. It operates in a fashion similar to the Microsoft Sysinternals' psexec command.

Let's try the smb-enum-users.nse script:

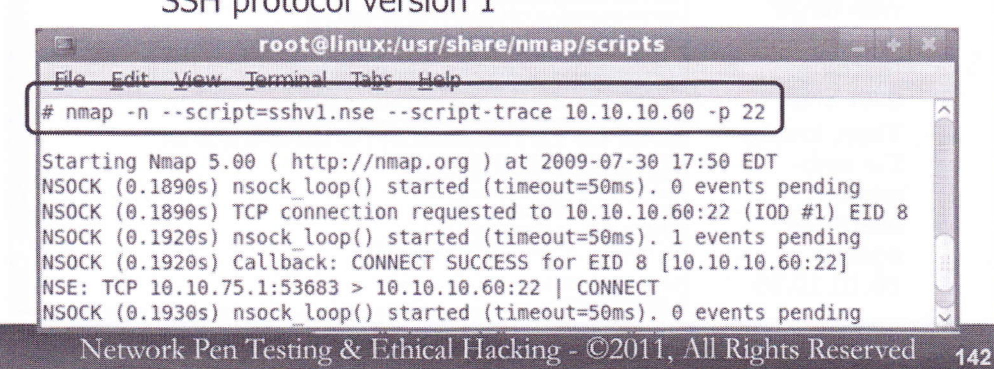
```
# nmap -n --script=smb-enum-users.nse 10.10.10.10 -p 445
```

In the output, you will see the results of the port scan, indicating that the given port is open. Then, we can see a list of users and their Relative Identifiers (RIDs), the unique portion of each user's Security Identifier (SID), in the output. We'll look at the technical mechanisms used by this script later in book 560.2 to iterate through a series of RIDs to find user names.

If you have extra time, you can try the other SMB.nse scripts in this directory.

NSE Exercise – SSHv1 Support?

- Let's run the sshv1.nse check against 10.10.10.60
  - This will tell us if it supports the older and weaker SSH protocol version 1



Next, we will use an NSE script to test whether machine 10.10.10.60 supports SSH protocol version 1, an older form of the Secure Shell protocol that is subject to man-in-the-middle attacks. SSH protocol version 2 is far stronger. We can measure whether the server has this issue by invoking Nmap as follows:

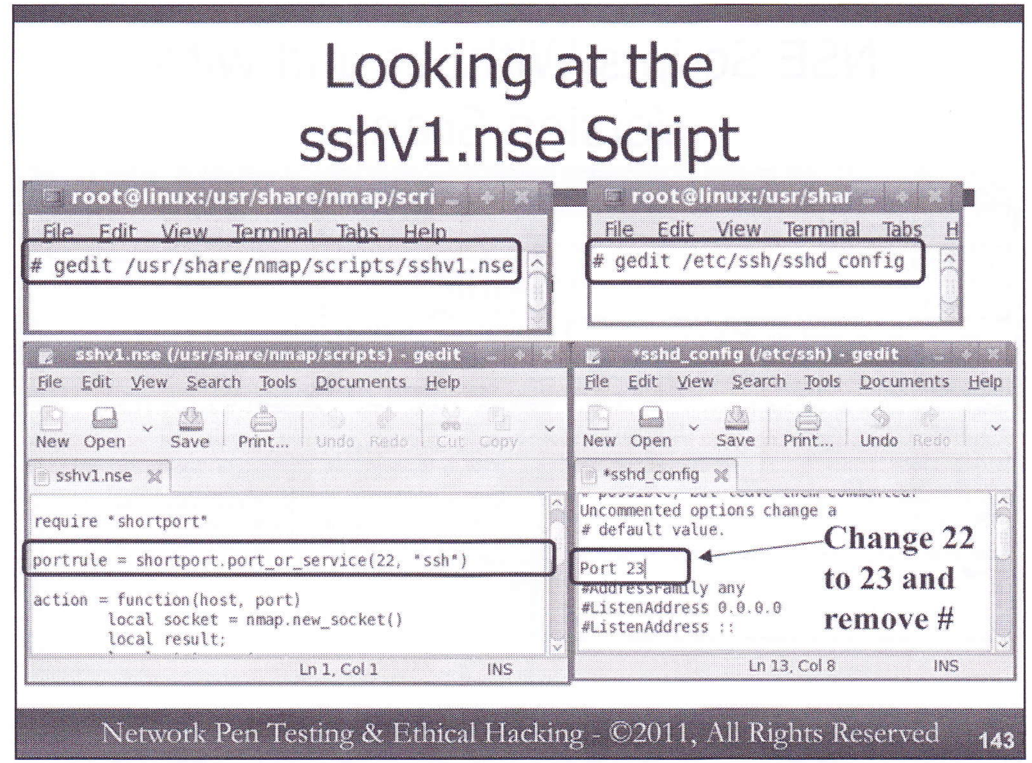
```
# nmap -n --script=sshv1.nse
--script-trace 10.10.10.60 -p 22
```

This command tells Nmap to run the script called sshv1.nse and to display the trace of the script's activity to the screen (--script-trace), against target 10.10.10.60, using TCP port 22. Note that we are only measuring TCP port 22 for this example, to keep things focused and quick. TCP 22 is the port commonly associated with SSH, of course.

Note that, because we have specified a given script with the "--script=" syntax, we do not have to specify –sC. Indicating a specific script implies that we want to invoke a script scan, so –sC is not required.

Once you've run this command, look through its output carefully. Can you get a sense of what the script is doing? Note that the --script-trace invocation makes Nmap put a lot of details on its output. Normally, you wouldn't run Nmap with this option. Still, for debugging, troubleshooting, or fine-grained analysis, this option is helpful.

So, does 10.10.10.60 support SSH protocol version 1? The answer should be yes.



Now that we've got a feel for what these scripts can do, let's look at them in more detail so that we can avoid some common mistakes in their usage. Let's return to the sshv1.nse, opening it in an editor to look at a very important setting in each script:

```
# gedit /usr/share/nmap/scripts/sshv1.nse
```

Now, look for `portrule = shortport.port_or_service(22, "ssh")`

This line tells Nmap that it should only run this script if it finds TCP port 22 listening on a target machine, or if a version scan finds that the ssh service is listening. That's good, but what happens if an sshd is listening on a port other than TCP 22? We need to know.

Let's reconfigure our sshd on our own Linux systems to listen on TCP port 23. You can do this by opening the file /etc/ssh/sshd_config:

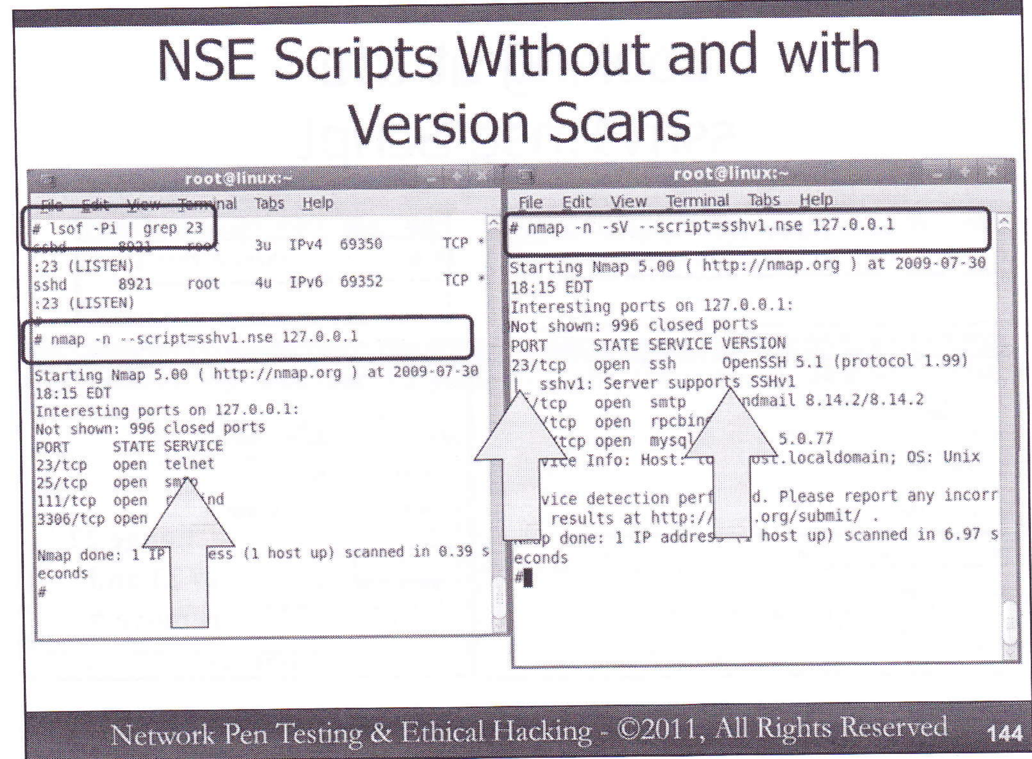
```
# gedit /etc/ssh/sshd_config
```

Find the line that says `#Port 22`. Edit that line so that it says:

```
Port 23
```

Make sure you remove the # from the front of the line. Save the file. Now, make your sshd re-read its configuration file by sending it the HUP signal:

```
# killall -HUP sshd
```



Verify that your sshd is listening on TCP port 23, by running:

```
# lsof -Pi | grep 23
```

The –i option indicates that we want to see network usage, while the –P modifier makes lsof display port numbers, not service names. If you see a line of output mentioning sshd and TCP 23, you are ready to go.

Now, run Nmap with the sshv1.nse script against your localhost:

```
# nmap -n --script=sshv1.nse 127.0.0.1
```

Do you see any output from the script (not Nmap overall, but the script itself) commenting on whether SSH protocol version 1 is in use? You likely do not, because Nmap only performed a TCP connect scan, discovering that TCP 23 was open, but not realizing that it spoke SSH. Instead, it just looked up the "normal" service associated with that port, which is telnet. It never measured whether a telnet services was listening there, because we didn't do a version scan. Instead, it just looked up that service in the nmap-services file. Also, the sshv1.nse script's portrule that checks for 22 or "ssh" service couldn't see that TCP 23 spoke ssh, so it didn't try to measure the SSH protocol version.

Let's try it again, but this time, telling Nmap to perform a version scan in addition to running the script:

```
# nmap -n -sV --script=sshv1.nse 127.0.0.1
```

Now, you should see in your output that the listener on TCP port 23 not only speaks SSH, but that it also uses SSH protocol version 1. Our script ran properly this time because the version scan was able to detect that TCP 23 spoke the secure shell protocol.

Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

- Scanning Goals and Types
- Overall Scanning Tips
  - Sniffing with tcpdump
- Network Sweeping with Scapy
  - Scapy/tcpdump Exercise
- Network Tracing
- Port Scanning
  - Nmap
  - Nmap Exercise
- OS Fingerprinting
- Version Scanning
  - Nmap -O -sV and Amap Exercise
- Vulnerability Scanning
  - Nmap Scripting Engine
  - NSE Exercise
  - ***Nessus***
  - Nessus Exercise
  - Other Vuln Scanners
- Enumerating Users
  - Enumerating Exercise
- Netcat for the Pen Tester
  - Netcat Exercise

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 146

While the NSE has great promise, and is starting to get more use in professional penetration testing and ethical hacking, it doesn't detect nearly as many flaws as other full-fledged vulnerability scanners. While NSE might someday catch up as a general-purpose vulnerability scanner, today, it is used mostly to focus in on a specific set of issues. That's not a knock against NSE. It's objectives center on augmenting Nmap and bringing more flexible analytic capabilities to the tool. But, it is a realization that, for now, Nmap with its NSE capabilities will not supplant traditional vulnerability scanners.

Most modern vulnerability scanners can measure for the presence of thousands of flaws in a target environment. One of the most full-featured vulnerability scanners available today is Nessus, our next major topic.

Tenable Network Security's Nessus Vulnerability Scanner

- Maintained and distributed by Tenable Network Security
  - www.nessus.org
- Free download
- Nessus 2, 3, and 4 all still supported
- Plugins measure flaws in target environment
  - Over 30,000 plugins, mix of open-source and commercial
  - As of August 1, 2008, commercial plugin subscription required for non-home use
- As new vulnerabilities are discovered, Tenable personnel release plugins
  - Available to paying customers immediately via Commercial Feed service
    - US $ 1,200 per year per Nessus scanner (includes tech support)
  - Free Home Feed, but only for Non-Commercial Use

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 147

The Nessus Vulnerability Scanner is maintained and distributed by Tenable Network Security. Available for free download from www.nessus.org, there are actually three versions of Nessus actively maintained today: Nessus 2, Nessus 3, and Nessus 4. Nessus 2 includes an open source scanning engine, supported by a development community and Tenable Network Security personnel. Nessus 3 and 4 have a closed source scanning engine, and is actively maintained by Tenable personnel.

The scanning engine is the component of Nessus that actually scans targets. But, those scans conducted by the scanning engine are based on plugins, individual small programs that tell the scanning engine what to do to measure for each individual security issue on a target machine. Some plugins are open source, while others (specifically the more recent ones) are commercial. Today, all plugins can run on either Nessus 2, 3, or 4. There are over 30,000 plugins available today, with new ones released on almost a daily basis. Most plugins are written by Tenable personnel and researchers, although a third-party development community does develop some.

Prior to August 1, 2008, Nessus supported a 7-Day Delayed feed, for all plug-ins on a free basis. As of this date, though, all recent plug-ins require either a Commercial or Home feed subscription. Tenable's Commercial Feed service makes new plugins immediately available to paying customers for a US $ 1,200 annual fee per Nessus scanner. This fee also includes tech support. Non-subscribers can get free access to all Nessus plugins but only for Home use. All commercial use requires a Commercial subscription feed as of August 1, 2008. Many professional penetration testers and ethical hackers who rely on Nessus do subscribe to the commercial service.

Nessus Architecture

- Nessus is a client-server architecture
  - Client: nessus
  - Server: nessusd
- Clients and servers available for Linux, MacOS X, Windows, Solaris, FreeBSD
- Nessus 2 versus Nessus 3 & 4
  - Nessus 2: Free and engine freely redistributable (some plugins free, others commercial)
    - Tenable claims, "`Tenable is committed to the open-source version of Nessus 2.x`" in the `Nessus FAQ`
  - Nessus 3 & 4: Commercial, 50% or more faster, with commercial plug-ins
  - The same plugins work in both, unless they use extended plugin functionality of Nessus 3 & 4

Tester

Nessus client

Nessusd

Targets

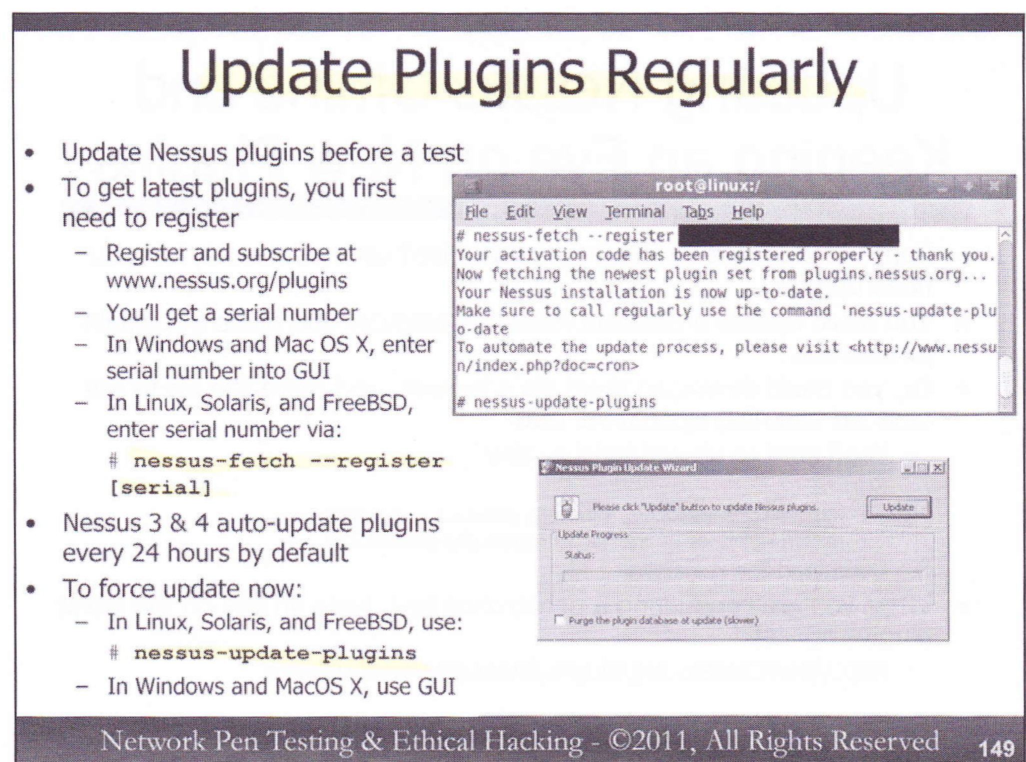
Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 148

Nessus itself is a client-server architecture. A user invokes the Nessus daemon (nessusd), and then uses a Nessus client to connect to it. The nessus client configures and manages things, while nessusd performs the scan. All reporting occurs at the nessus client. While nessus and nessusd can run on separate systems, they are often run on the same machine. Nessus clients and servers have been released for Linux, Mac OS X, Windows, Solaris, and FreeBSD.

So, should you use Nessus 2 or later versions as a professional penetration tester and ethical hacker? Nessus 2, 3, and 4 are in widespread use right now by professionals around the world. Nessus 2 is free and can be redistributed on a free, open-source basis. That's why we'll run our exercise on it in this class. Furthermore, Tenable has claimed in the Nessus FAQ that they are, "Committed to the open-source version of Nessus 2.x." So far, they have held up this commitment, continuing to make it available for download and supporting it with the latest plugins. Some penetration testers and ethical hackers rely on Nessus 2 and utilize the open-source nature of the tool to add their own tweaks to the underlying engine.

Nessus 3 and 4 are commercialized versions of Nessus. Furthermore, their plugins require a Commercial Feed subscription. Their scanning engine is faster than the Nessus 2 scanning engine, with most scans taking half the time with Nessus 3 or 4.

Currently, the vast majority of plugins work for Nessus 2, Nessus 3, and Nessus 4. Tenable has introduced some extensions to the language in which some plugins are written (the Nessus Attack Scripting Language, or NASL for short), that only work on Nessus 3 and 4, however. According to Tenable's web site, "Starting with Nessus 3.0.2, NASL scripts can write directly to the ethernet level, handle non-blocking sockets, etc. While engineers at Tenable will try to be backward compatible with Nessus 2 most of the time, these functions will be used to improve the results of the scan or to speed it up."



You should update your Nessus plugins on a regular basis to make sure you are testing against the latest set of known vulnerabilities. To get updated plugins, you'll first need to register with Tenable. Upon subscribing, registering, and providing your e-mail address, you will be e-mailed a serial number for use in downloading the plugins. In the Windows and Mac OS X versions of Nessus, simply enter the serial number into the Nessus GUI. In Linux, Solaris, and FreeBSD, you need to run the nessus-fetch program to register your serial number with your given Nessus install.

Once your serial number is registered, you can then use it to download plugins from within Nessus. Nessus 3 and 4 automatically update plugins every 24 hours by default. You can shut this off by altering the Nessus configuration to require manual plugin updates, a helpful option if you want to have control over the plugin update process. Nessus 2 requires manual intervention to update plugins.

Once you've gotten a serial number and registered your version of Nessus to use it, you can update plugins manually right away. In Linux, Solaris, and FreeBSD, this is accomplished by running the nessus-update-plugins script. On Windows and Mac OS X, you simply invoke plugin update via the GUI.

Updating Nessus Offline and Keeping an Eye on New Plugins

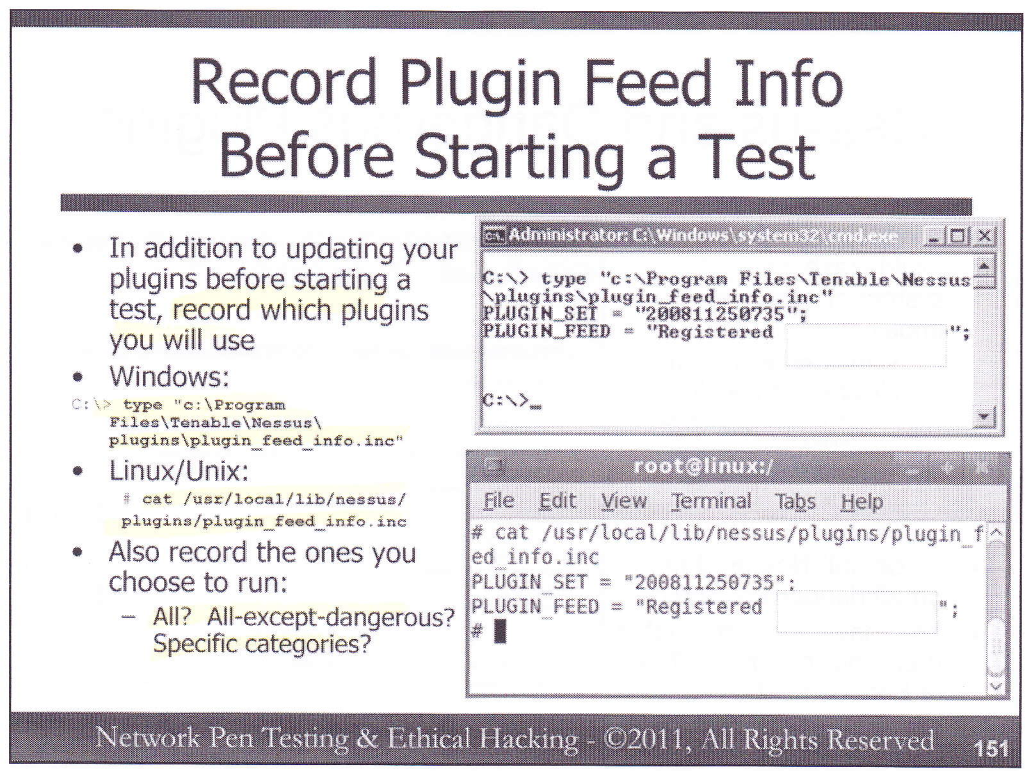
- Suppose your system running nessusd isn't on the Internet, but you need updates
- You could update a different nessusd computer and move the plugin directories...
- Or, you could download them via a browser and move the file to the nessusd scanning system via USB
  - You'll need an unused serial number
  - Surf to
    - http://plugins.nessus.org/offline.php (Nessus 3.x and later)
    - http://plugins.nessus.org/manual-register.php (Nessus 2.x)
  - Download the revisions
- While you are conducting a penetration test, keep an eye on the latest plugins released
  - http://www.nessus.org/plugins/index.php?view=newest

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 150

Sometimes, we are faced with a situation where we cannot update our Nessus plugins from the system running nessusd. Perhaps the nessusd computer isn't able to access the Internet, yet we need it to get an up-to-date set of plugins. We can accomplish this in two ways. First, we could put another system with nessusd on the Internet and update its plugins. Then, we'd need to copy the plugin directory to our Internet-shielded nessusd machine. The directory on Linux is /usr/local/lib/nessus/plugins by default. On Windows, it's c:\Program Files\Tenable\Nessus\plugins.

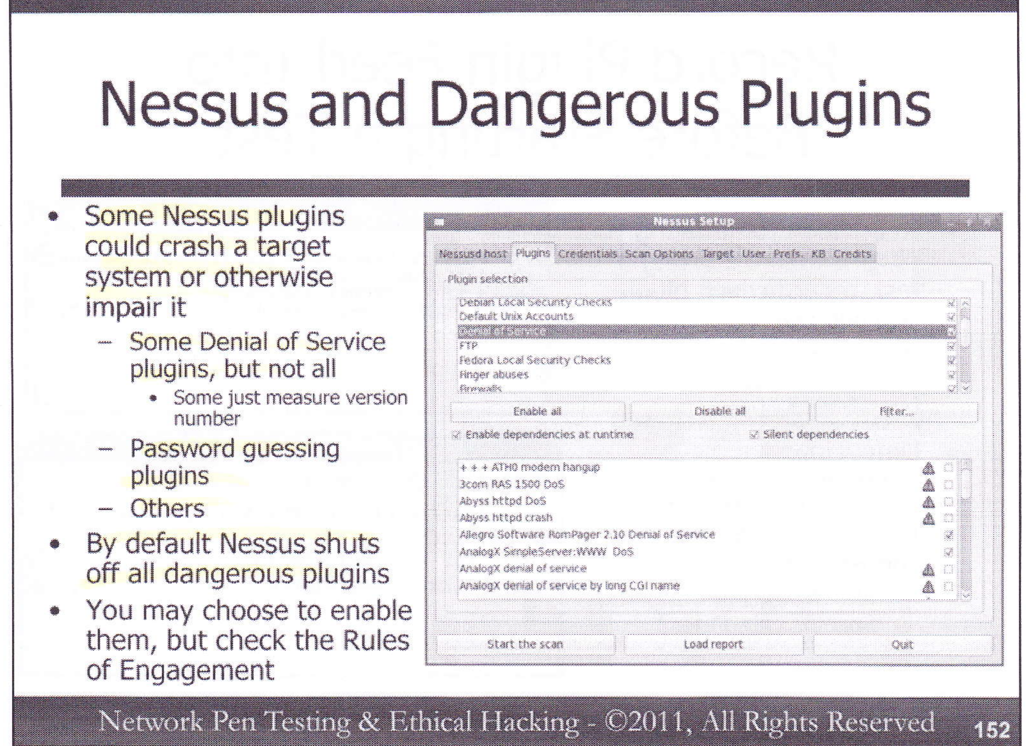
Tenable provides another option. We could use a browser to surf to the appropriate URL at nessus.org, and download the plugins directly, without using Nessus at all, but instead relying on our browser for doing the download. We'll need a serial number that hasn't been used for a Nessus install yet. We enter that serial number into the web form, and then we can download the plugins in our browser. We can copy the file to a USB token, and move it to the machine running nessusd.

It's also important for us to keep an eye out for new plugins while we are conducting a test. If a vitally important new plugin is released while a test is underway, we need to analyze whether we should run another scan using just those very new plugins against the target so we can have the latest, up-to-date results. Of course, there is risk in this approach, because those new plugins haven't been carefully scrutinized yet, either by our testers or the community. Still, we need to keep this option open and discuss it with the people who formulated the Rules of Engagement for the given test.



In addition to updating your plugins before a test, you should also record the plugin feed info you are running the test from. Nessus maintains this information in a file stored with the plugins. The file is called plugin_feed_info.inc, and records the PLUGIN_SET number, essentially a date and timestamp of when that set of plugins was released by Tenable. Make a copy of this file and store it with the results of any scans you conduct.

You should also make a note of the particular plugin configuration you use for the test. Are you enabling all plugins? All plugins except the dangerous ones? Are there specific plugins that you are shutting off? Are there categories you are choosing to run or not to run? Make sure you write down the specifics of the plugin groups you choose in your testing notes.

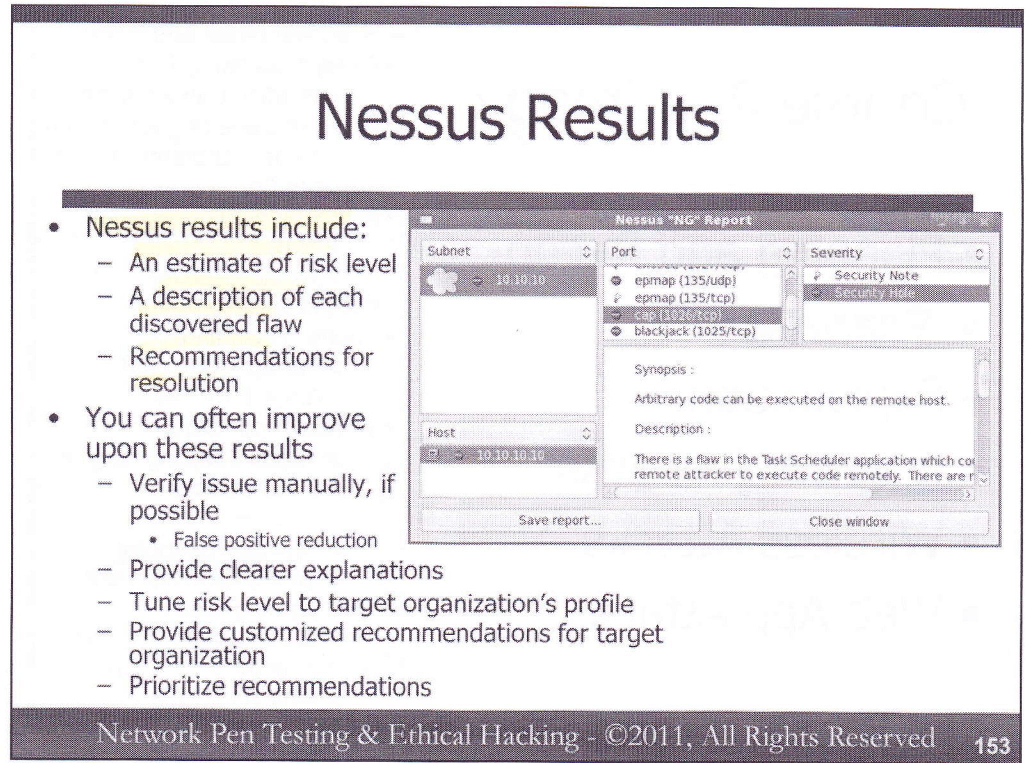


The authors of Nessus plugins have characterized some of the plugins as dangerous, meaning that they could impair a target system.

Some, but not all, of the Nessus denial of service plugins are dangerous. Some of the denial of service plugins merely measure the version number of a target service; that is typically not dangerous. Others actually launch malformed packets at the target service, which could cause it to crash, a dangerous circumstance. Some password guessing plugins are dangerous because they could lock out accounts in a target environment. Other plugins formulate benign exploit code for a target, which could crash a service running on it, again illustrating a potentially dangerous circumstance.

By default, Nessus disables all dangerous plugins when it is first run. You have to enable individual plugins by hand if you want to run them. You'll note that the denial of service *category* is enabled by default, but some of the plugins *within the category* are shut off.

So, should you run the dangerous plugins during a penetration test or ethical hacking exercise? Consult the Rules of Engagement. In most environments, you will not be allowed to run these plugins.



Nessus results include an estimate of the risk level associated with each finding (High, Medium, or Low), a brief description of each discovered flaw, and recommendations for resolution. Note that most professional penetration testers and ethical hackers use this Nessus output as a starting point, refining it and providing value-added analysis. Don't just throw the Nessus results at target personnel as your entire final report. Instead, help them focus on the most vital issues. The Nessus report might be an appendix of your final report, but it should not be its centerpiece.

Instead, provide value-added services by verifying the Nessus results manually if possible, researching each discovered issue and trying to see if the given target machine really exhibits that problem, or if we've got a false positive. You may need to review the configuration of the target with the system administrator, or research methods for using tools like Netcat (which we'll cover in more detail later in this class) to interact with the target manually. Furthermore, tune the risk level to the target organization's risk profile, as well as the importance of the machine on which the vulnerability was discovered. Even though Nessus says that a given issue is High risk, for a given target in a given environment, it may be Medium or Low risk. Of course, the opposite could also apply.

You should also strive to provide clearer explanations of issues than those offered by Nessus results. Describe the issue in the context of the given target environment, using examples of how the given threat could be exploited within their industry, if possible. Also, tailor your recommendations to the target organization, based on your understanding of their environment and motivations. And, finally help prioritize recommendations to focus on those findings that are most urgent.

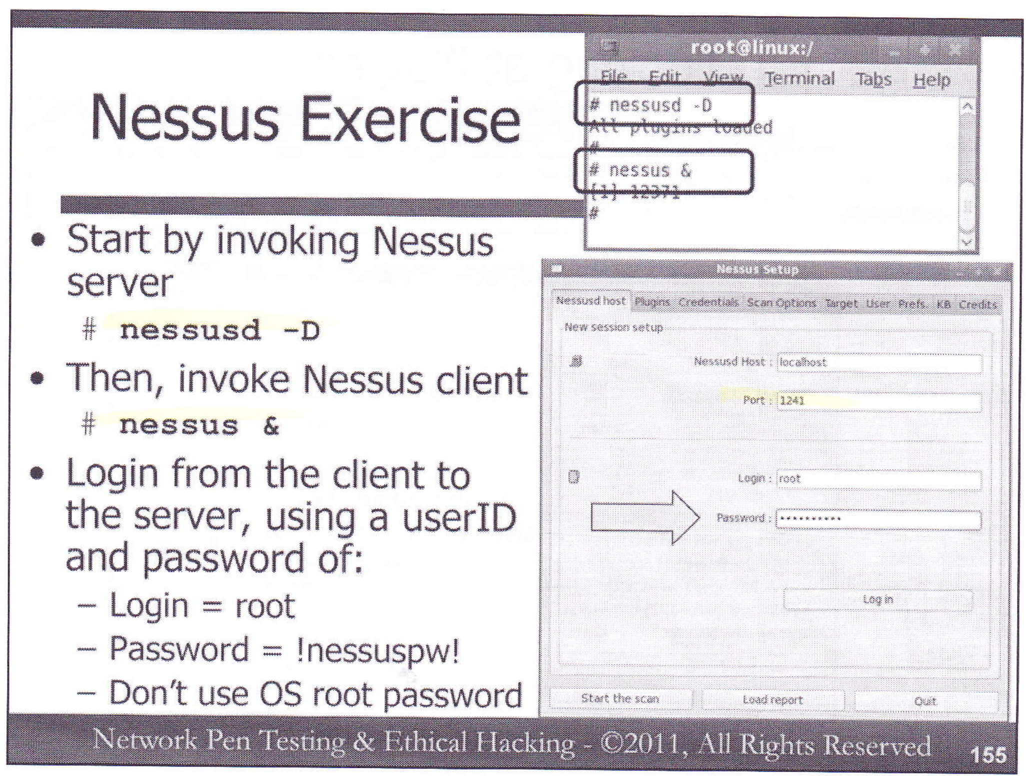
Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

- Scanning Goals and Types
- Overall Scanning Tips
  - Sniffing with tcpdump
- Network Sweeping with Scapy
  - Scapy/tcpdump Exercise
- Network Tracing
- Port Scanning
  - Nmap
  - Nmap Exercise
- OS Fingerprinting
- Version Scanning
  - Nmap -O -sV and Amap Exercise
- Vulnerability Scanning
  - Nmap Scripting Engine
  - NSE Exercise
  - Nessus
  - ***Nessus Exercise***
  - Other Vuln Scanners
- Enumerating Users
  - Enumerating Exercise
- Netcat for the Pen Tester
  - Netcat Exercise

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 154

Now that we've gotten an overview of Nessus functionality, let's tour its configuration in-depth during a hands-on exercise. In your Linux machine, get ready to run Nessus against our target environment.



First, we need to invoke the nessusd server, running it with a –D option for daemon mode, running in the background. This will make nessusd load its plugins, which may take some time.

```
# nessusd -D
```

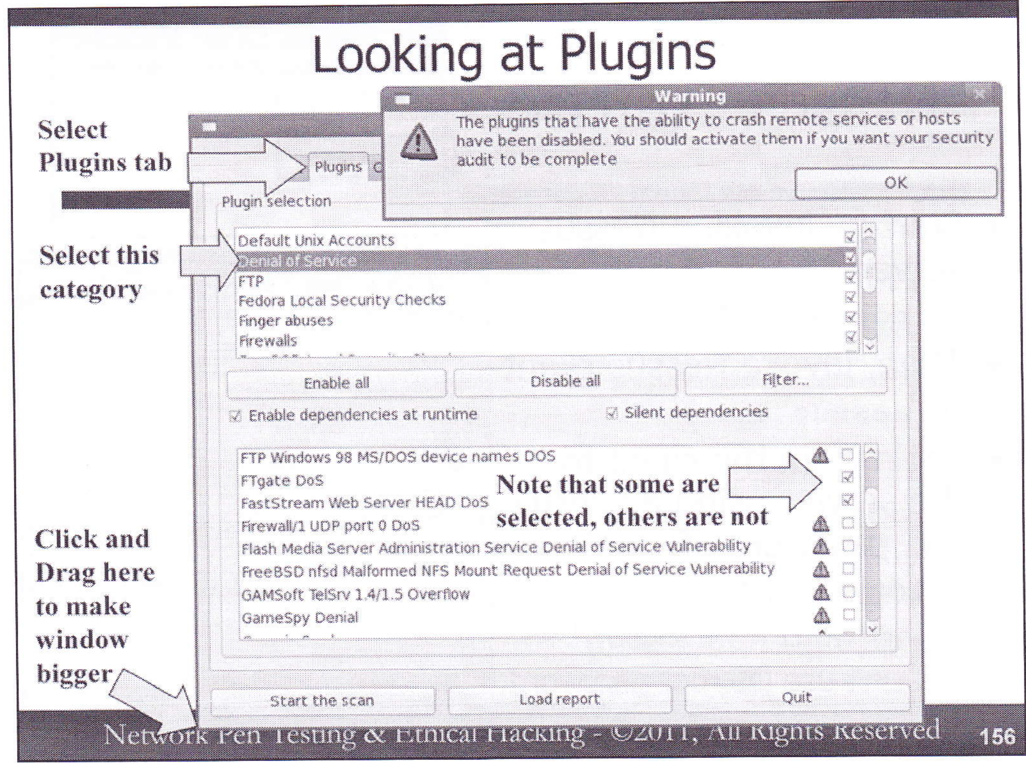
Once nessusd is ready, you will get your command prompt back. If you get an error saying that nessusd can't bind to the port, that is likely because you already have nessusd running, using that port. You can likely just connect to it if it is already running, or kill it with the killall command and an argument of nessusd.

Once the nessusd server is running, we can invoke the Nessus client, by typing:

```
# nessus &
```

We use the & here to kick the Nessus client into the background so that we can get our terminal back.

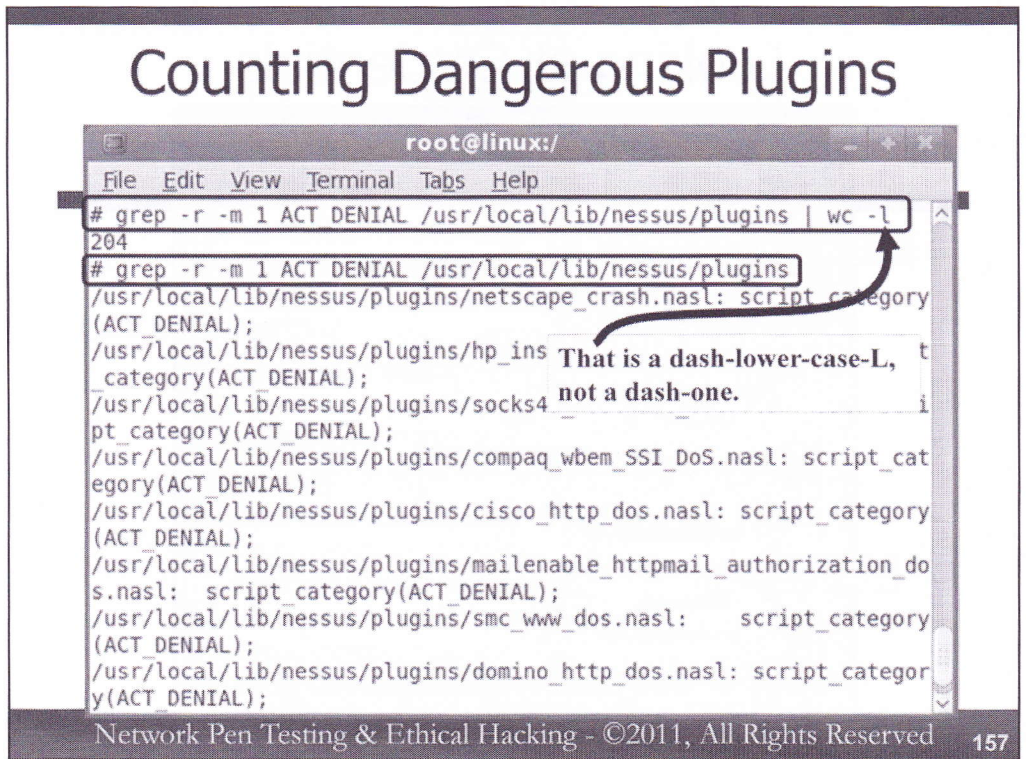
The Nessus client GUI will ask us for a Login name and Password to access nessusd. Use the default Host and Port, and type in a login name of root and a password of !nessuspw!. Please note that you are typing in the name and password of the root user we created in Nessus, not the overall root user for the operating system. The operating system root password is different from this password within Nessus.



By default, the Nessus dangerous plugins are disabled. Upon successful login, Nessus tells you this with a Warning message on the screen. Click OK.

Let's explore these dangerous plugins in more detail. Go to Plugins tab, and select the Denial of Service category under "Plugin selections". On the bottom area of the screen, you should see the individual plugins in this category. Make your Nessus client screen bigger by dragging the bottom corner of it so that you can see the detail on the right hand side of the individual plugins. Wherever you see a yellow triangle with an exclamation point (!), these are dangerous plugins. Note that by default, the dangerous ones are off. Also, note that some denial of service plugins are not dangerous, and are turned on.

Let's look at another category to show that some dangerous scripts are located outside of the Denial of Service category. At the top, click on the Web Servers category. Within this category, look at the CERN httpd CGI name heap overflow. It is dangerous, and off by default, but it is located in the Web Servers directory.



Double clicking on a plugin shows details about it, what is measures, recommendations for addressing it, and so on. Alternatively, you can view the individual plugin scripts in the directory /usr/local/lib/nessus/plugins.

All dangerous NASL scripts contain a line that says "script_category(ACT_DENIAL)" in the script code. Thus, we can count the number of dangerous plugins by running the following command:

```
# grep -r -m 1 ACT_DENIAL /usr/local/lib/nessus/plugins | wc -l
```

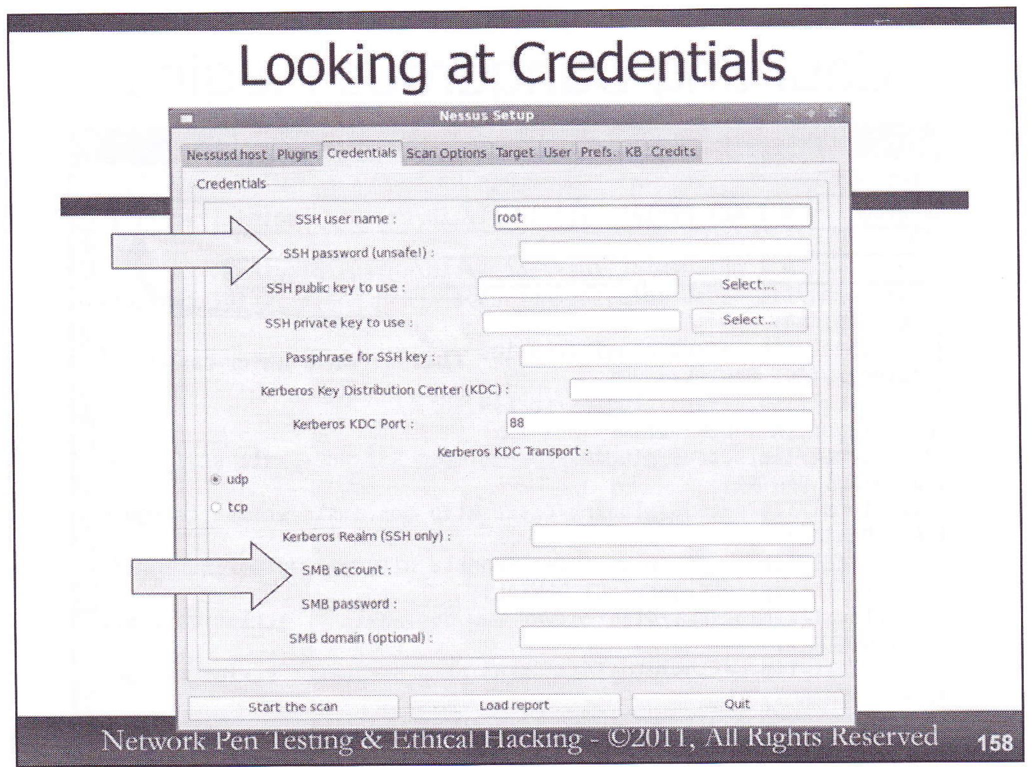
Again, please note that we are using wc with a dash-lower-case-L, not a one.

We are using a `grep -r` here instead of a `grep *` because the number of plugin scripts is often too large for `grep *` by itself to handle. But, with the –r option for recursing that directory instead of relying on the shell to expand the * wildcard, the command works just fine. The –m 1 indicates that we want to count files that have one or more occurrences of the string "ACT_DENIAL". After finding the string once in a given plug-in file, grep moves to the next file because of the –m 1 invocation. Running this command might take a half a minute or more, because there are many thousands of plugins to check. Just leave off the `| wc -l` to get a list of the dangerous plugins.

```
# grep -r -m 1 ACT_DENIAL /usr/local/lib/nessus/plugins
```

By the way, to do a similar thing on the Windows version of Nessus (not included on the course DVD), you could run:

```
C:\> cd "c:\Program Files\Tenable\Nessus\plugins\scripts"
C:\> findstr ACT_DENIAL *
```



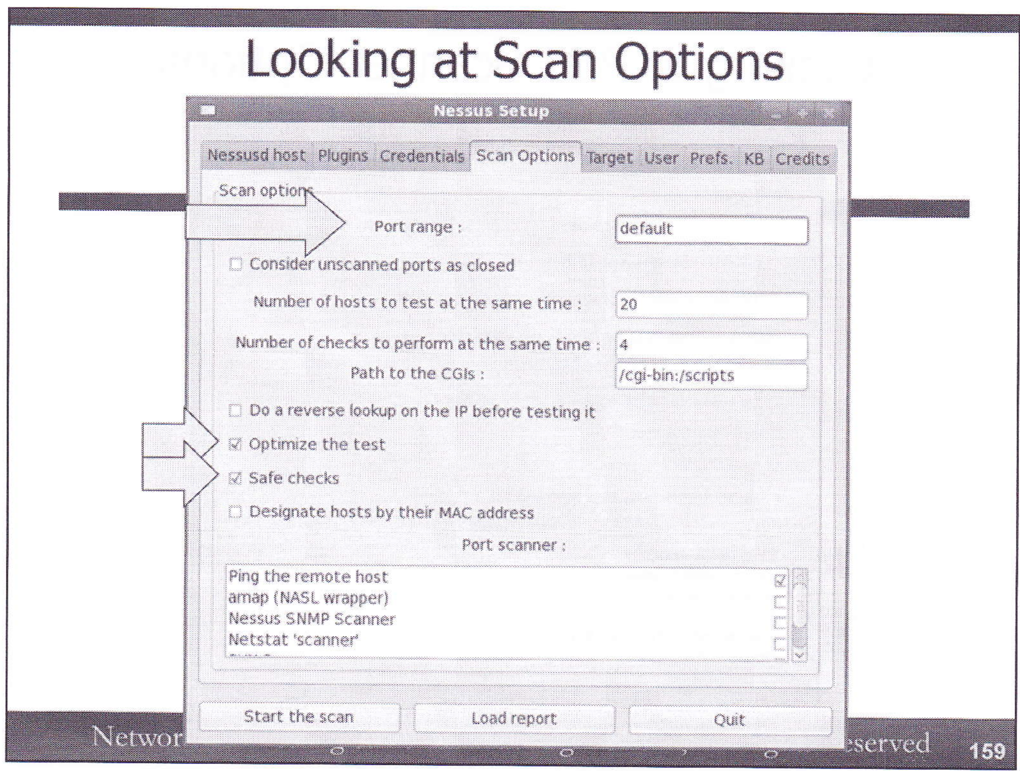
Click on the Credentials tab of the Nessus GUI. Nessus allows testers to enter userIDs and passwords for a target environment, which Nessus will use with various plugins that can supply user credentials to target machines. Some of these plugins actually try to login to various target systems and measure them for vulnerabilities.

Most professional penetration testers and ethical hackers do not use these options, instead relying on Nessus scans for vulnerabilities that can be measured without any user credentials at all. Some pen testers and some auditors do use these options, however, to gain more in-depth insight into security vulnerabilities of target machines that can only be measured using valid authentication credentials.

In this tab, we can enter Server Message Block (SMB) credentials, used for Windows file and print sharing services and domain authentication, as well as Linux and Unix Samba. Several exploits for Windows require a username and password of a limited privilege account, but can deliver local SYSTEM-level access with the exploit. We can provide an account name, a password, and a domain name.

We can also configure Nessus with an SSH user name and password, as well as public and private keys for authenticating to machines in the target environment. The SSH password option is listed as "unsafe!", because leaving copies of passwords for ssh access to the targets inside of Nessus is a security risk. Of course, entering SMB passwords into Nessus provides a similar area of risk as well.

Additionally, you can see fields for configuring Kerberos credentials, including the Key Distribution Center (a Kerberos server that distributes keys) and networking options.



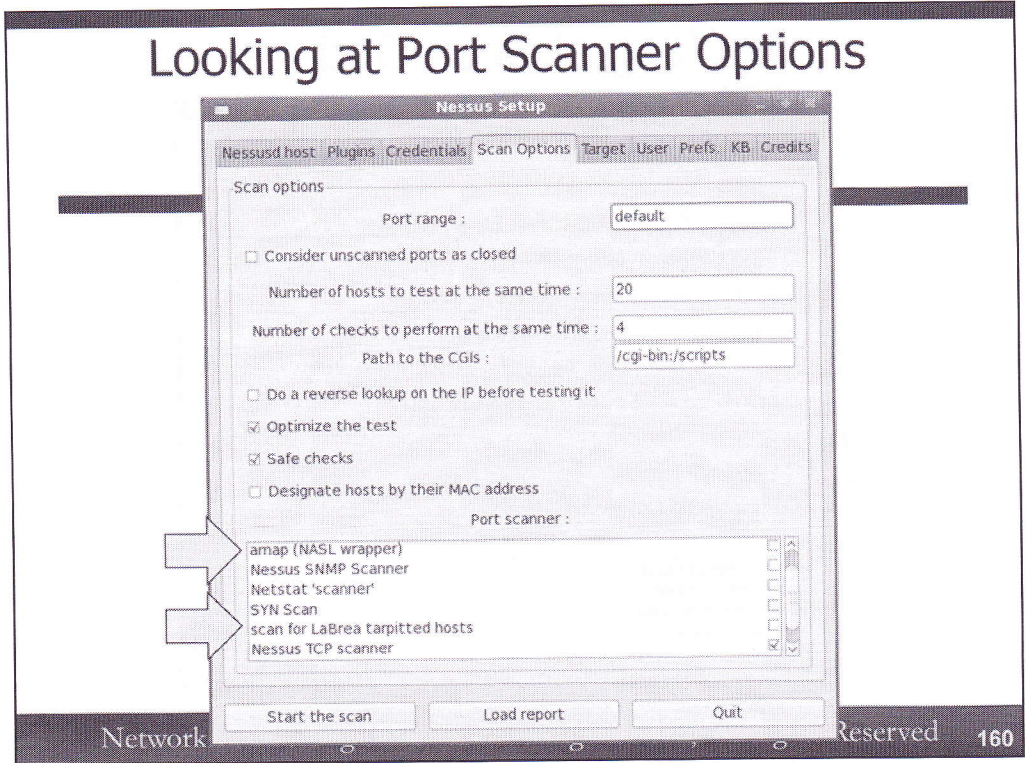
Now, click on the Scan Options tab. Here we can see the port range that Nessus will scan. You could configure Nessus to scan all ports, a range of ports, no ports (by entering –1), or the default set of ports. By default, Nessus scans the ports listed in the Nessus service file, which is located in /usr/local/var/nessus/nessus-services. This file contains about 9,000 ports, about half TCP and half UDP. Open this file and look through it:

```
# gedit /usr/local/var/nessus/nessus-services
```

The Scan Options tab also lets us configure Nessus parallel scans, setting the number of tests and target hosts that Nessus will scan simultaneously. If we are scanning a web server, we can also tell Nessus where the CGI scripts directory on the server is, so that it can look for well-known vulnerable scripts.

The Optimize the test option tells Nessus to only run a given test if the port associated with the service is listening, or some other test determined that the service might be running. Unfortunately, it's possible that the port scan or other test may give us a false negative, which would make Nessus set to "Optimize the test" miss a vulnerability. Still, this option does make tests significantly faster, as many plugins are skipped when the target doesn't seem to be running a given service.

The "Safe checks" option is another method for forcing Nessus to avoid functionality that could disable or crash a target service. By default Nessus turns off dangerous plugins. Going further, some plugins have two options for measuring whether a vulnerability is present: 1) by checking its banner or 2) by interacting with a system in a way that might cause problems. Enabling "Safe checks" makes these plugins use only method #1, resulting in even more safety on top of disabling the dangerous plugins. In the version of Nessus we use in this course, "Safe checks" is enabled by default.

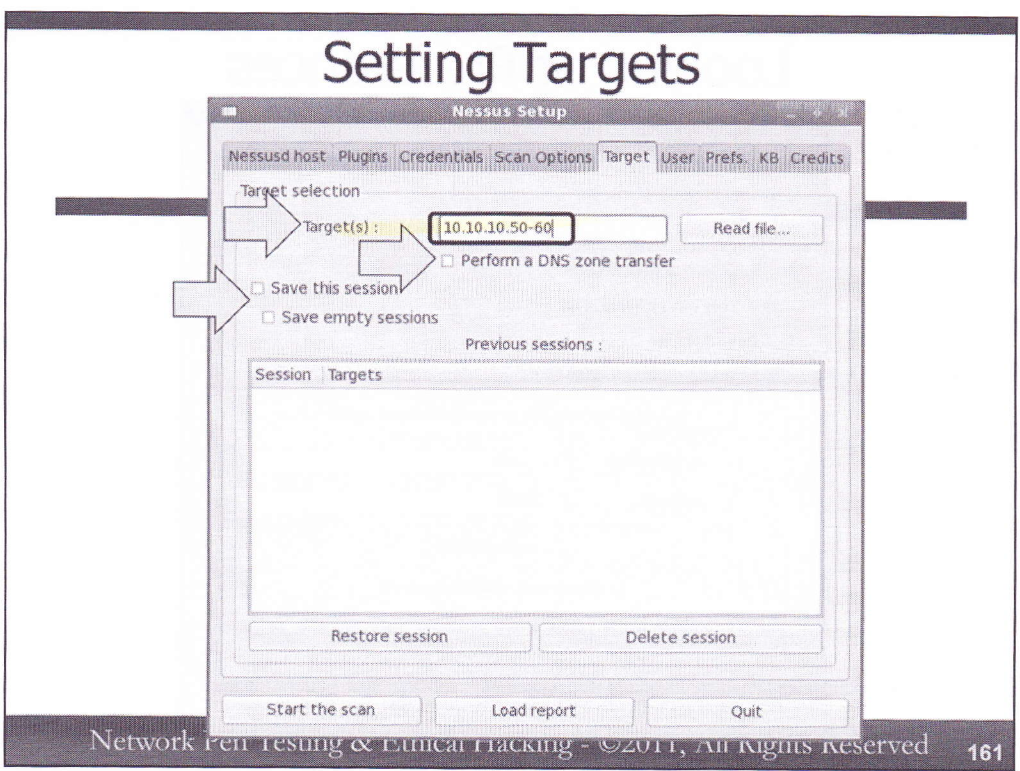


Staying in the Scan options tab, at the bottom, we can choose our port scanner, using the built-in Nessus scanner, which most testers do. Nessus also includes a special scanning function to try to find a LaBrea tarpit, a tool by Tom Liston that slows down aggressive scans by responding very slowly to session requests. Nessus will identify LaBrea based on its timing characteristics and tell the Nessus user, helping to explain why a scan is going so slowly.

Nessus also can invoke amap, the scanner we discussed earlier, to perform port scans and service identification. And, Nessus includes a scanner for pulling information via the Simple Network Management Protocol (SNMP).

Also, as you can see in this Scan Options tab, by default, Nessus pings hosts before scanning them, but this feature can be turned off.

Please note that for many of the options in the Nessus GUI, you can hover your mouse over the option to make Nessus display a brief help statement summarizing the functionality of the given option.

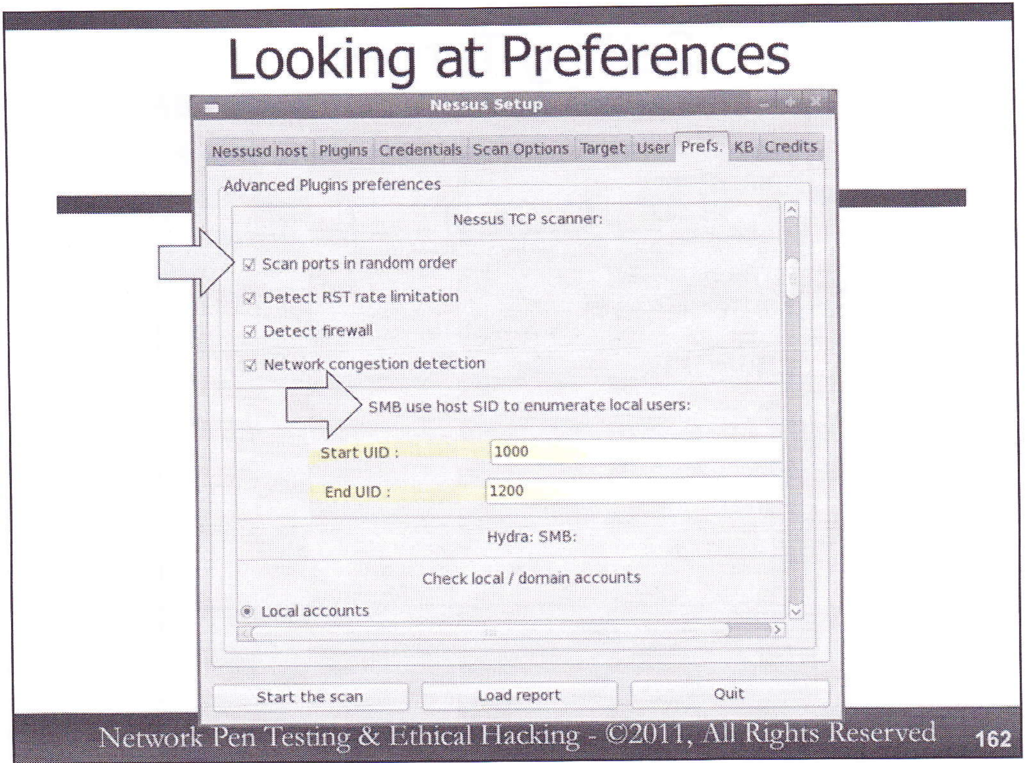


Note that we left the default for the Plugins, Credentials, and Scan Options tabs. In the Target tab, though, we're going to enter a target for our scan. We could specify one host, a range of hosts, a list of hosts, or a file from which lists of hosts can be read.

We'll scan target IP address range 10.10.10.50-60, so enter that into the Target(s) field.

We have an option to tell Nessus to perform a zone transfer from the DNS server configured for the operating system. Nessus will then scan all hosts referred to in the results of the zone transfer. While that sounds convenient, it can be dangerous, because there may be hosts in the zone transfer results that are outside of the scope of the test. Thus, we recommend that you not use this option.

We can also have Nessus save its session state so that we can resume a scan that was begun before.



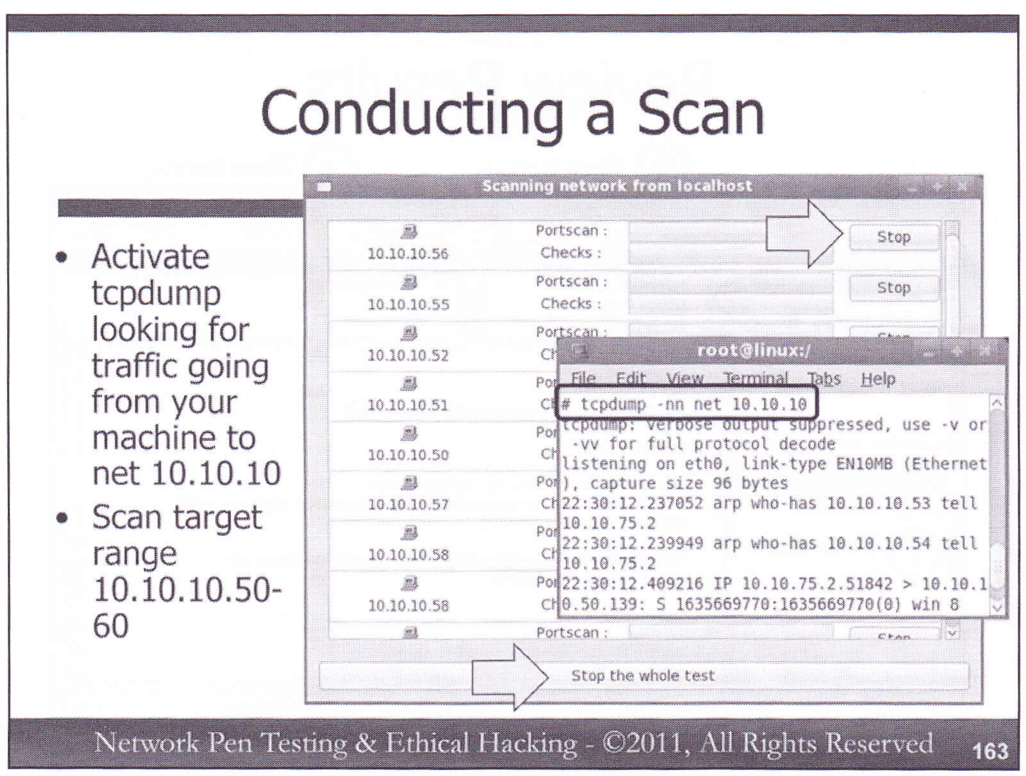
We'll skip over the User tab, because most penetration testers and ethical hackers don't use it. That tab allows for configuring certain accounts within Nessus that have limitations, in that they cannot use certain plugins or scan options. Some auditors might need access only to a limited set of Nessus scanning functionality, and making restricted user accounts within Nessus can help support such needs. But, most professional penetration testers need the ability to run any combination of plugins.

Move on to the Prefs. Tab. Here we can set some detailed configuration options for various components of Nessus. We can configure the port scanner to scan ports in random order. Nessus can detect network congestion indications based on slower responses over time, and try to compensate by throttling back its scan.

Nessus can also remotely enumerate accounts on Windows target machines by iterating through the Security IDentifiers (SIDs) of accounts, a technique we'll cover using different tools (SID2user and user2SID) in more detail in an exercise later in this session, 560.2.

Many of these preferences are associated with configuring the amap scanner, the services identification tool we discussed earlier. Another large set of them are associated with configuring Nessus to launch THC Hydra, a flexible password-guessing tool that we'll look at in depth in Section 560.4 when we cover password guessing.

Briefly review these options. The default settings for them are quite reasonable for most tests.



The KB tab is used to configure the Nessus Knowledge Base, a database in which Nessus stores information about scans in progress. By saving the Knowledge Base, you can optimize the speed of later scans by having them rely on information gathered from earlier scans.

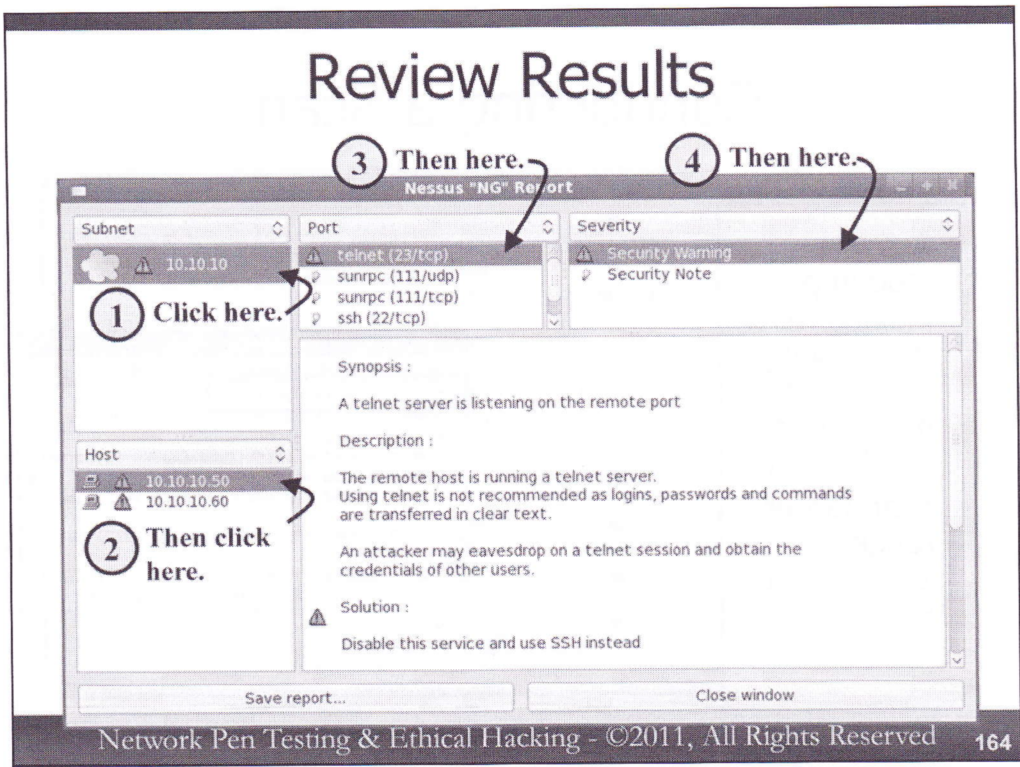
Next, let's start our scan of target range 10.10.10.50-60. But, before you start the scan, run the tcpdump sniffer, configured to display all traffic associated with our target network, 10.10.10, but without resolving names or looking up ports. We can do this with:

```
# tcpdump -nn net 10.10.10
```

Then, click on "Start the scan" at the bottom of the Nessus GUI.

You will see the Nessus scanning window appear. Each target host undergoing a scan will get a progress bar in this window. Note that you could stop an individual host by clicking on the Stop button next to its progress bar. Or, you could click on "Stop the whole test" at the bottom of the screen to abandon the test. Even if you do abandon the test, the interim results will be displayed. Note that at the onset of your test, each IP address in the range 10.10.10.50-60 will briefly get a progress bar, but the only ones that will remain are those that respond to the pings sent by Nessus.

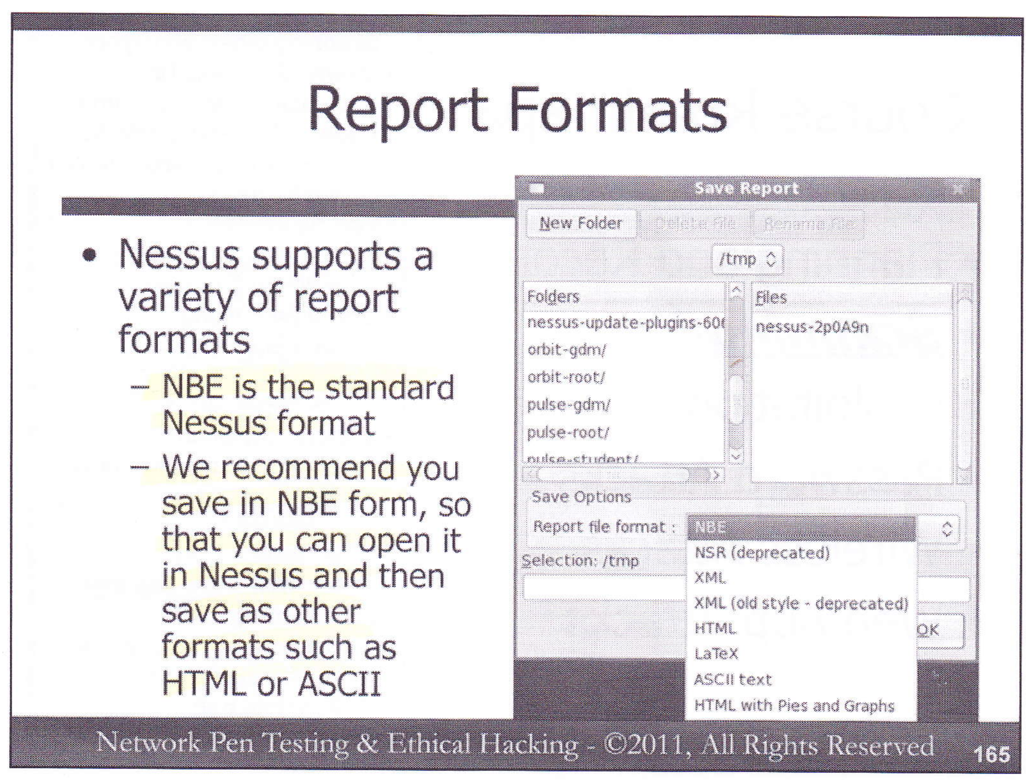
Let the test run without stopping for a few minutes. If it hasn't completed at that time, click on "Stop the whole test". We want to look at our interim or final results.



Next, let's look through our results. On the left-hand side of the screen near the top, we see a summary of the subnet(s) that we scanned. We only scanned a single network, 10.10.10, so click on it. Then, below the subnet list, we'll see the Host list on that network. Nessus should have found and scanned 10.10.10.50 and 10.10.10.60 in our target range. Click on any one of these hosts, and the Port pane shows us the ports that were found to be open on that host. We can then click on the ports to determine which Security Holes, Security Warnings, and Security Notes Nessus found for each target on each port.

A red stop sign icon indicates a potential Security Hole, possibly a high-risk flaw. The yellow yield sign indicates a Security Warning, possibly a medium-risk flaw. And, the light bulb indicates a Security Note, which may or may not be a risk. Note that we must analyze these issues rather than just labeling them as High-, Medium-, or Low-Risk issues in our final report.

Look through the findings of the two targets you scanned.



In the Nessus report window, you can click on "Save results". Nessus allows us to save scan results in a variety of formats. The default format understood by Nessus itself is NBE. Other formats include XML, HTML, and ASCII.

We recommend that you save results immediately in NBE format. Then, if you need to generate any XML, HTML, or ASCII reports, you could simply open the NBE file in the Nessus client, and then save the results in another format. Keeping the original in NBE format, though, offers more flexibility for conversion at a later time.

Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

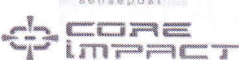
- Scanning Goals and Types
- Overall Scanning Tips
  - Sniffing with tcpdump
- Network Sweeping with Scapy
  - Scapy/tcpdump Exercise
- Network Tracing
- Port Scanning
  - Nmap
  - Nmap Exercise
- OS Fingerprinting
- Version Scanning
  - Nmap -O -sV and Amap Exercise
- Vulnerability Scanning
  - Nmap Scripting Engine
  - NSE Exercise
  - Nessus
  - Nessus Exercise
  - ***Other Vuln Scanners***
- Enumerating Users
  - Enumerating Exercise
- Netcat for the Pen Tester
  - Netcat Exercise

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 166

While Nessus is very popular, there are other vulnerability scanning tools on the market. Let's briefly survey some of them that you might want to consider using, and then zoom into one of them to explore its capabilities in more detail.

Other Vulnerability Scanning Tools

- Commercial solutions:
  - Rapid7 NeXpose and Metasploit Express: www.rapid7.com
  - Saint: www.saintcorporation.com
  - Retina Network Security Scanner: www.eeye.com
  - Lumension PatchLink Scanner (formerly Harris Stat): www.lumension.com
  - BiDiBLAH – www.sensepost.com
  - Core IMPACT – www.coresecurity.com - exploitation tool, but limited scanner
- Scanning services / appliances:
  - Foundscan: www.foundstone.com
  - Qualys: www.qualys.com
- Free solutions:
  - Sara: www-arc.com/sara, free, but not as comprehensive as others
  - SuperScan – www.foundstone.com, free, but limited to port scans and Windows information pulling



Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 167

Besides Nessus, there are numerous other commercial and free vulnerability scanning solutions available today. From a commercial perspective, products include Rapid7's NeXpose, a comprehensive vulnerability scanning and management solution. Rapid7 also sells Metasploit Express, a product that provides a GUI for Metasploit and integration between its scanning and exploitation components, with automation of numerous common tasks performed by penetration testers and a step-by-step process organized around the workflow of pen testers. Saint, a product derived from the Security Administrator's Tool for Analyzing Networks (SATAN), is one of the original vulnerability scanners. eEye Digital Security has a comprehensive scanner called the Retina Network Security Scanner. The Lumension PatchLink Scanner was built on the Harris Stat scanner, and is used by US government and military agencies as well as some commercial companies. The BiDiBLAH scanner by Sensepost offers some very interesting features, with integration into Nessus and Metasploit.

Some penetration testers and ethical hackers consider using their exploitation frameworks as vulnerability scanners. For example, Core IMPACT, a commercial exploitation tool, can scan for some vulnerabilities, specifically the vulnerabilities for which the tool offers exploit code to compromise a target system. While the scanning features of these exploitation tools are useful, they are not as comprehensive as other commercial scanners. You will miss some flaws, and potentially pretty serious vulnerabilities, if you rely exclusively on your exploitation tool for scanning. Thus, exploitation tools do not supplant vulnerability scanners; they augment vulnerability scanners.

Some companies offer subscription scanning services, which can be configured to scan across the Internet on a regular basis, such as monthly, weekly, or even daily. For intranet scans, these companies often ship an appliance that sits on the internal network scanning regularly, with reports accessible to authorized personnel via a web portal running on either the appliance or on the service provider's website. Foundstone and Qualys offer such subscription-based scanning solutions.

And, let's not overlook some additional free scanners. Sara is a free scanner also built on the foundations of the SATAN scanner. However, the number of vulnerabilities it can scan for is lower than for tools like Nessus or the commercial solutions cited above. SuperScan by Foundstone provides some helpful scanning capabilities, but is very limited. It focuses on port scanning and pulling information from Windows machines.

Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

- Scanning Goals and Types
- Overall Scanning Tips
  - ➢ Sniffing with tcpdump
- Network Sweeping with Scapy
  - ➢ Scapy/tcpdump Exercise
- Network Tracing
- Port Scanning
  - ➢ Nmap
  - ➢ Nmap Exercise
- OS Fingerprinting
- Version Scanning
  - ➢ Nmap -O -sV and Amap Exercise
- Vulnerability Scanning
  - ➢ Nmap Scripting Engine
  - ➢ NSE Exercise
  - ➢ Nessus
  - ➢ Nessus Exercise
  - ➢ Other Vuln Scanners
- ***Enumerating Users***
  - ➢ Enumerating Exercise
- Netcat for the Pen Tester
  - ➢ Netcat Exercise

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved

An important component of many penetration testing and ethical hacking projects involves getting a list of account names for target systems, a process sometimes called "enumerating users." Next, we'll discuss several tactics for enumerating user accounts in a target environment so that we can use those account names during our exploitation and password attacks as we move forward with the test.

Methods for Getting Account Names

- We often need account names for our tests
  - We pull them during our scans
  - We may use them later for our password guessing attacks
- We have numerous methods for getting account names
- Public sources of information:
  - Look at e-mail addresses, blog postings, newsgroup postings, etc.
  - Most organizations use e-mail addresses that contain account names:
    - [account_name]@[target_domain_name]
  - Not every organization does this, but enough of them do to make it worthwhile to try
  - Pull potential user names from document metadata
- Alternatively, you may want to ask target personnel for account names for the test
  - Such information helps to perform a more thorough test
  - Assume the worst case – the attacker knows an account name... can we get in then?

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 169

During the scanning phase of a test, it is helpful to build an inventory of account names for a target organization that we can use throughout the rest of the attack. Later, we may need a valid account name to make an exploit work. Or, for password guessing, we need account names against which we can use automated password guessing tools. These lists of account names should be carefully documented and guarded throughout a test.

There are numerous methods for getting account names. One method involves doing research on the Internet in various public sources of information to pull potential account names. A tester can look at e-mail addresses in newsgroup postings, mailing list archives, blog posts, and social networking sites. Many (but not all) organizations formulate their e-mail addresses so that they contain user account information, simply because it is easier for users to remember their account name and e-mail address if they both contain the same information. In other words, many organizations have e-mail addresses of the form:

[account_name]@[target_domain_name]

Some enterprises are more careful and separate the e-mail address from account names, possibly using e-mail aliases. But, because the practice of keeping these names in synch is so common, it is worthwhile for us to try the first part of an e-mail address as an account name.

Additionally, as we saw in 560.1, we could try to pull user names from document metadata.

Another option for getting account names is simply to ask target organization personnel for them. They may provide them to help you do a more thorough test. Such tests model a worst-case situation: an attacker knows account names because he or she shoulder surfed them from a legit user at an airport or cyber café.

Methods for Pulling Account Names from Linux/Unix and Windows

- Linux / Unix:
  - Local, with login on the box
    - Get list of all accounts: `$ cat /etc/passwd`
    - See who is currently logged in: `$ finger`
    - Another way to see same thing: `$ who`
    - See what they are doing: `$ w`
  - Remotely, across the network:
    - Try finger, but almost always off now: `$ finger @[targetIP]`
    - If NIS is in use, pull user names with: `$ ypcat passwd`
      - Pull group names and user membership with: `$ ypcat group`
    - If LDAP is in use, query user names with: `$ ldapsearch [criteria]`
- Windows:
  - Pulling user lists from Null SMB sessions
  - Automating enumeration via User2sid and Sid2user conversion tools

student@linux:/

File Edit View Terminal Tabs H

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologi
n
daemon:x:2:2:daemon:/sbin:/sbin
/nologin
adm:x:3:4:adm:/var/adm:/sbin/no
login
lp:x:4:7:lp:/var/spool/lpd:/sbi
```

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 170

We have other, more technical methods for getting lists of account names. From a Linux or Unix environment, we can try to pull them either locally or across the network. If we have a local account to login to a Linux or Unix machine, we could simply look at the /etc/passwd file, in which each user account for the operating system is defined. Because /etc/passwd is readable by any user on the machine, this is a handy way of getting a list of all user names for the system:

```
$ cat /etc/passwd
```

Alternatively, we could run the `finger` command locally, which will show us who is currently logged into the system (even if the finger service isn't active, the local finger command still works). The finger command provides less information than we can get from looking at /etc/passwd (which shows us all users regardless of whether they are logged in our not), but it still might be interesting. Additionally, we could run the `who` command to show us who is logged in, giving us pretty much the same information as finger, in a very similar format. The `w` command gives us more info, showing us what the user is doing (that is, it will display the command each user is running on a given terminal).

Remotely across a network, some older or less secure Linux and Unix systems may have the finger service running providing finger information remotely on TCP port 79. If that port is listening, we can at least try to see who is logged in by running:

```
$ finger @[targetIP]
```

Alternatively, if there is an Network Information Service (NIS) server, we can use the Linux and Unix ypcat command to query it for users and groups using the syntax shown on the slide. If LDAP is in use, the ldapsearch command built-into some Linuxes can be used to formulate queries for usernames against it. For specific syntax of the ldapsearch command, please consult the man pages.

From a Windows perspective, we have two really useful options for remotely harvesting user accounts: pulling user lists from Null sessions and using User2sid/Sid2user tools. Let's explore each of those approaches in more detail.

Windows: Pulling Account Names via Null Sessions

- Windows Null session:
  - SMB session with no userID, no password, no domain membership
- If tester has SMB access of a target Windows system (via TCP port 135-139 or TCP 445), and the machine is configured to support Microsoft file and print sharing...
  - The attacker can set up a Null session
- To test if you can establish a Null session by hand:

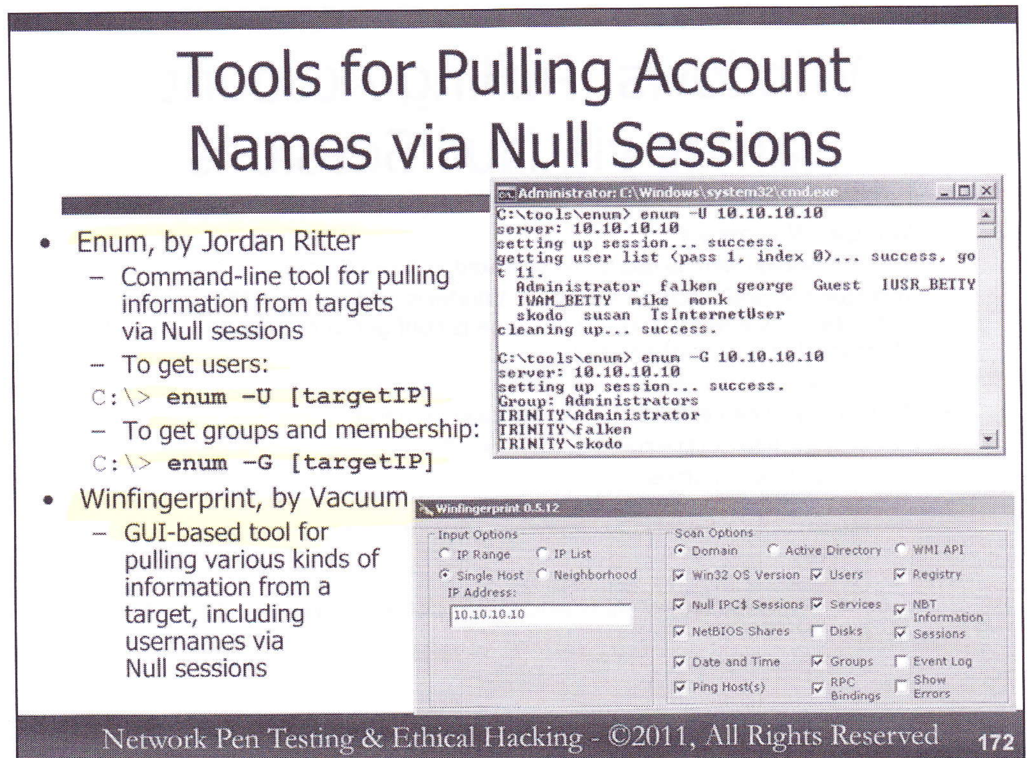
  `C:\> net use \\[targetIP] "" /u:""`
- We can pull user names:
  - On Windows 2000 targets, if HKLM\System\CurrentControlSet\Control\Lsa\RestrictAnonymous = 0 (the default)
  - On Windows 2003, XP, and Vista targets, if HKLM\System\CurrentControlSet\Control\Lsa\RestrictAnonymousSAM = 0 (not the default)

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 171

On Windows machines, a Null session is a Server Message Block (SMB) connection with a blank userID, a blank password, and a blank domain. Literally, the information associated with who sets up such an SMB session is Null. If a tester can connect to the SMB over NetBIOS ports (TCP ports 135-139) or SMB port (TCP 445) associated with a target Windows machine, and the target has been configured to support Microsoft file and print sharing, a tester can establish a Null session. To set up a Null session by hand to a target machine, the tester could run:

```
C:\> net use \\[targetIP] "" /u:""
```

The information that can be pulled from a target using the Null session depends on the settings of various Registry keys. Of most interest to us as ethical hackers and penetration testers are the settings associated with getting a list of user names on the target machine. We can pull user names via a Null session from a Windows 2000 target machine if the Registry key HKLM\System\CurrentControlSet\Control\Lsa\RestrictAnonymous has a value of 0. That's the default for Windows 2000 machines, and is seldom changed because many applications designed to run on Windows 2000 expect to be able to get user information via this method. On Windows 2003, XP, and Vista machines, the ability to pull user names via a Null session is controlled by the Registry key HKLM\System\CurrentControlSet\Control\Lsa\RestrictAnonymousSAM. If this value is set to 0, we can pull names via a Null session. The default setting for this key is 1, which prohibits pulling user names. However, on some target machines, this setting has been configured to 0 by administrators to support compatibility with a given application that requires such a configuration.



Two tools that pull information from a target machine using Null sessions are Enum and Winfingerprint, both included on the course DVD in the Windows directory. Both Enum and Winfingerprint establish their own Null sessions as they run, so there is no need for an attacker to set up a Null session before activating either tool.

Enum is a command-line tool released by Jordan Ritter, which can pull lists of users (when invoked with the –U option), lists of groups and their membership (-G), and other information from targets. Additionally, Enum can pull password policy information (-P), such as the maximum allowed password age and the minimum length password. It can also get a list of available shares from a target (-S). Enum also supports dictionary-based password guessing for NetBIOS over TCP connections on a target Windows machine or SAMBA file server via its –D option, but we'll go over far more powerful password guessing tools in our 560.4 session later in this course. Enum runs against only a single target when it is invoked.

Winfingerprint is a GUI-based tool that can pull information from one target, ranges of targets, lists of targets, or everything available in the network neighborhood. When run with both its "Domain" and "Null IPC$ Session" options, Winfingerprint pulls information from a target machine using Null sessions, and can get a list of users and groups. It can also pull information from Active Directory and the Windows Management Instrumentation (WMI) API, other methods for extracting information from target Windows machines.

Enumerating SIDs

- On Windows, each group and account has a unique Security Identifier (SID)
  - Unique number for that system
  - Consists of S-[X]-[Y]-[domain/computer]-RID
    - X is the revision level (typically 1)
    - Y is an authority level (typically 5 for users and groups)
    - Domain is a unique number for the given machine or domain
  - Last component is RID
    - Well-known accounts have common RIDs:
      - Original administrator account has RID of 500 (regardless of name)
      - Guest account has a RID of 501
      - Users created on the machine have RIDs 1001 and up
      - Documented by Microsoft at http://support.microsoft.com/kb/243330

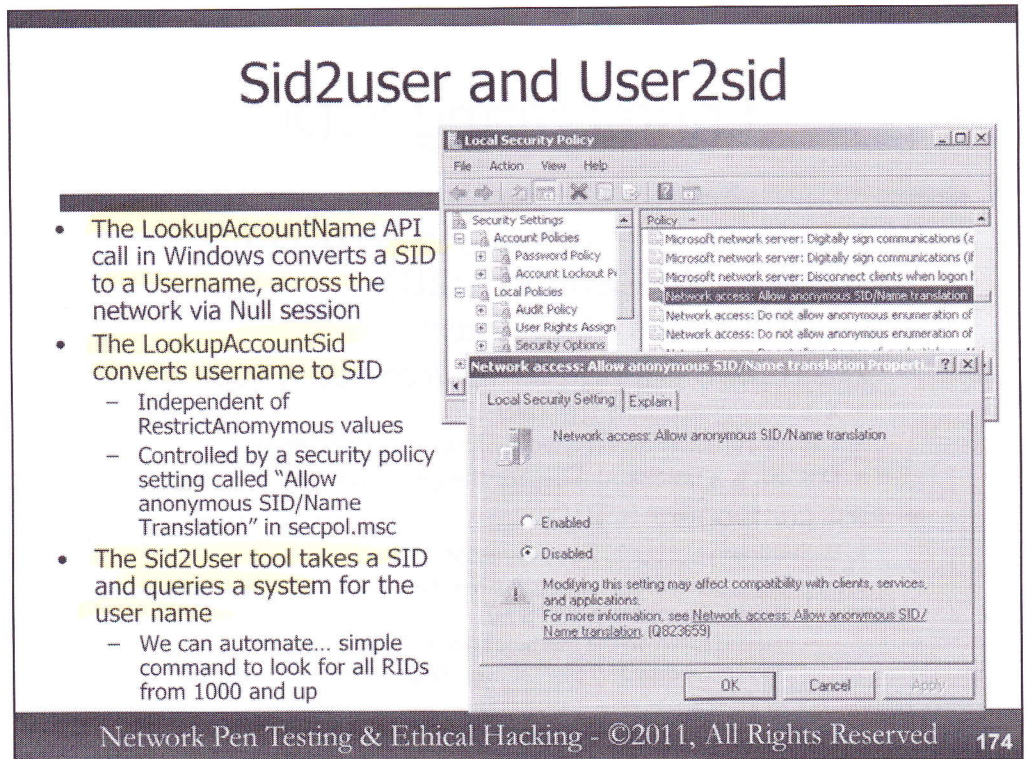
Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 173

Besides pulling user and group names via Null sessions, we have another related method for pulling information based on the Security Identifier (SID) for each account. Windows assigns a unique SID for each user account and group defined on each system. The SID consists of several components, and is typically displayed in the format of:

```
S-[X]-[Y]-[domain/computer]-RID
```

The S up front merely indicates that this is a SID. The X is the revision level, typically given a value of 1. The Y indicates the authority level of the SID, and is typically set to 5 for user accounts and groups. Next comes a unique number associated with either the individual machine on which the account was created, or, for accounts that are defined on a domain, a unique number indicating that domain. Then, at the end, we have the Relative ID (RID), which makes a unique number for the given account or group.

Various important accounts have known RIDs, with a comprehensive list of well-known SIDs and RIDs defined by Microsoft in an article at http://support.microsoft.com/kb/243330. The administrator account has an RID of 500, regardless of the name of the account. It is possible to rename the original administrator account on Windows, but its SID still remains with a suffix (RID) of 500. The Guest account has an RID of 501. Individual user accounts and groups are assigned RIDs by the system when they are created, starting at 1000 and moving up by one for each new entity created.



Windows includes two API calls associated with mapping user names and SIDs. The LookupAccountName API allows an anonymous user via a Null session to convert a SID to a username, remotely across the network. The LookupAccountSid API goes in the opposite direction, converting a username to a SID. Again, these API calls can be made remotely across a Null session, providing tremendously useful information to an attacker.

Also, their functionality is independent of the RestrictAnonymous and RestrictAnonymousSam registry keys that control the ability to extract user names from a target machine via tools like Enum and Winfingerprint. Regardless of the settings of RestrictAnonymous or RestrictAnonymousSam, an attacker can still pull information. The conversion of SID to username and vice versa is controlled by a separate setting, accessible via the local security policy (viewable and editable using secpol.msc). In Local Policies, under Security Options, there is a setting called "Network access: Allow anonymous SID/Name translation". By default, almost all Windows machines allow User-to-SID and SID-to-User translation, except for Windows 2003 servers that are configured as domain controllers.

Two tools, Sid2user and User2sid take advantage of these Windows APIs to pull information from Null sessions about users and SIDs across the network. We can automate these calls to harvest user names from a target machine by querying SID after SID, looking for those that successfully resolve into a username.

Using User2sid and Sid2user

- Goal: Use Sid2user to harvest names from a target
- Start by establishing a Null session

```
C:\> net use \\[targetIP] "" /u:""
```

- Then, ask the target for its domain/computer component of the SID

```
C:\> user2sid \\[targetIP] [machine_name]
```

- Then, with the domain/computer component of SID, we can lookup potential users based on their RIDs:

```
C:\> for /L %i in (1000,1,1010) do @sid2user
  \\[targetIP] [SID without RID] %i
```

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 175

To harvest user names from a target Windows machine with Sid2user, we could apply the following steps. First, we need to open a null session with the target. Unlike Enum and Winfingerprint, User2sid and Sid2user do not establish their own Null sessions. We have to create one manually before running the tool.

```
C:\> net use \\[targetIP] "" /u:""
```

Now, we want to run Sid2user to ask the target machine about various SIDs. However, to do this, we need to know the [domain/computer] portion of the SID for the target machine. We can pull this by running User2sid against the target, with a user name of the machine name itself:

```
C:\> user2sid \\[targetIP] [machine_name]
```

This command will tell us the overall SID for the target machine, a value of something like S-1-5-[some series of digits]. It's those series of digits we want, because they are the unique numbers from which SIDs are built for that target machine. Once we have those digits, we can then run an automated loop around them, asking for SID-to-username conversion for RIDs 1000 and up. We can accomplish this with a FOR loop as follows:

```
C:\> for /L %i in (1000,1,1010) do @sid2user \\[targetIP]
 [domain/computer] %i
```

This FOR loop tells Windows that we want a counter (/L) that will iterate the variable %i through a series of integers, starting at 1000, counting by 1, and going up through 1010 (1000,1,1010). At each iteration through the loop, we'll run the sid2user command against the target machine with a SID (consisting of the number 5 followed by the unique domain/computer string, but not including the RID) followed by our RID guess (%i). The system will display the result each time, showing us which SIDs are valid and giving us the associated user name. We'll do an exercise on this later, and cover Windows FOR loops in more detail in session 560.3.

Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

- Scanning Goals and Types
- Overall Scanning Tips
  - Sniffing with tcpdump
- Network Sweeping with Scapy
  - Scapy/tcpdump Exercise
- Network Tracing
- Port Scanning
  - Nmap
  - Nmap Exercise
- OS Fingerprinting
- Version Scanning
  - Nmap -O -sV and Amap Exercise
- Vulnerability Scanning
  - Nmap Scripting Engine
  - NSE Exercise
  - Nessus
  - Nessus Exercise
  - Other Vuln Scanners
- Enumerating Users
  - ***Enumerating Exercise***
- Netcat for the Pen Tester
  - Netcat Exercise

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 176

In our next exercise, we'll look at methods for enumerating users on a target Windows machine. Specifically, we'll use the enum tool to get a list of users and groups via a Null session. Then, we'll explore really useful techniques for applying User2sid and Sid2user to extract user names from systems that have even enabled the RestrictAnonymous and RestrictAnonymousSam registry keys.

Preparing Enum

- Unzip Enum onto your hard drive
- Your anti-virus tool may not like Enum
  - You may need to shut down your AV tool to use Enum
  - Don't just kill the AV processes or stop their services
    - They will still protect you
    - You need to turn them off using their admin GUI
    - You must have access to that GUI to disable the tool
- Extract enum.exe to c:\tools\enum\

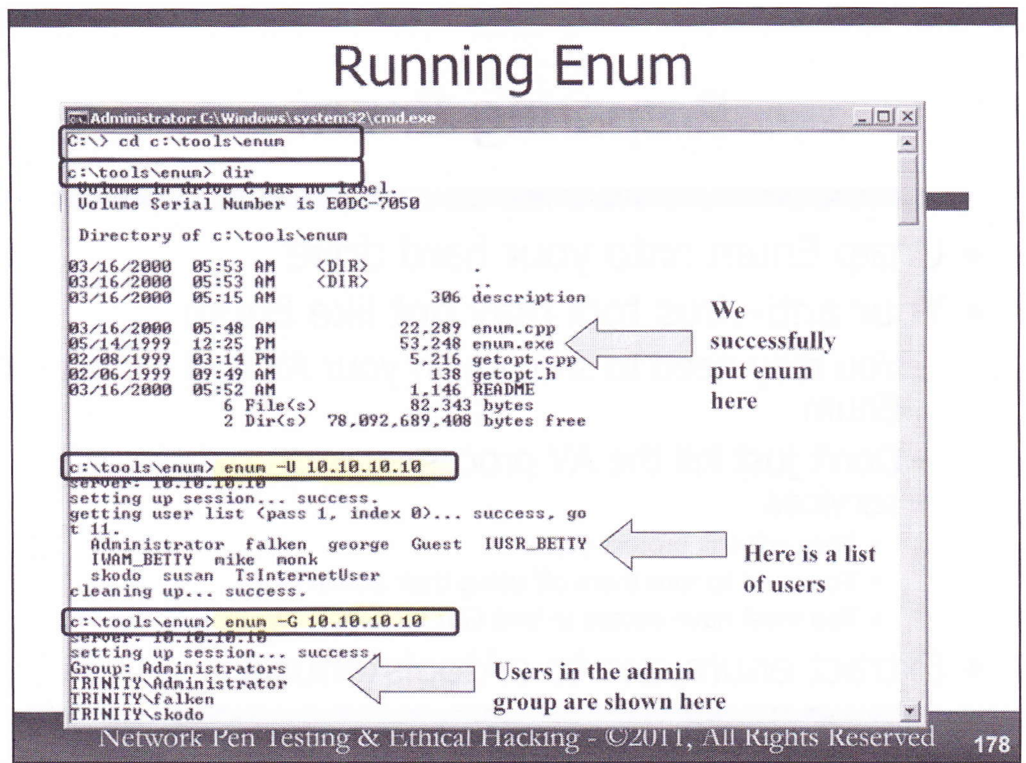
Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 177

For this component of the exercise, you'll need to unzip the enum tool from the course DVD. It is located in the Windows directory. Unzip it to your hard drive, putting it in a directory called c:\tools\enum.

Your anti-virus tool may have a signature that detects enum as malware. Enum is not malware; it is a tool used for pulling configuration information from machines remotely using Null sessions. But, because some computer attackers have abused systems with enum, some anti-virus vendors have written signatures for it. Thus, if you have such an anti-virus tool, you must first disable it before you can unzip and run enum.

Disable your anti-virus tool using the anti-virus admin GUI. DO NOT DISABLE YOUR ANTI-VIRUS TOOL BY KILLING ITS PROCESSES IN TASK MANAGER OR DISABLING ITS SERVICES IN THE SERVICES CONTROL PANEL! Most anti-virus tools will still protect you even if you kill them using those methods. To disable anti-virus protection, you must use the anti-virus administrative GUI.

Make sure that you've successfully extracted enum.exe into c:\tools\enum, the directory from which we'll run the first component of this exercise.



Now, with Enum on your hard drive, change directories to it:

```
C:\> cd c:\tools\enum
```

Verify that you are in a directory with enum.exe:

```
C:\> dir
```

Now, run enum against 10.10.10.10, configured to extract users:

```
C:\> enum -U 10.10.10.10
```

Then, run it to extract groups:

```
C:\> enum -G 10.10.10.10
```

Finally, get password policy information:

```
C:\> enum -P 10.10.10.10
```

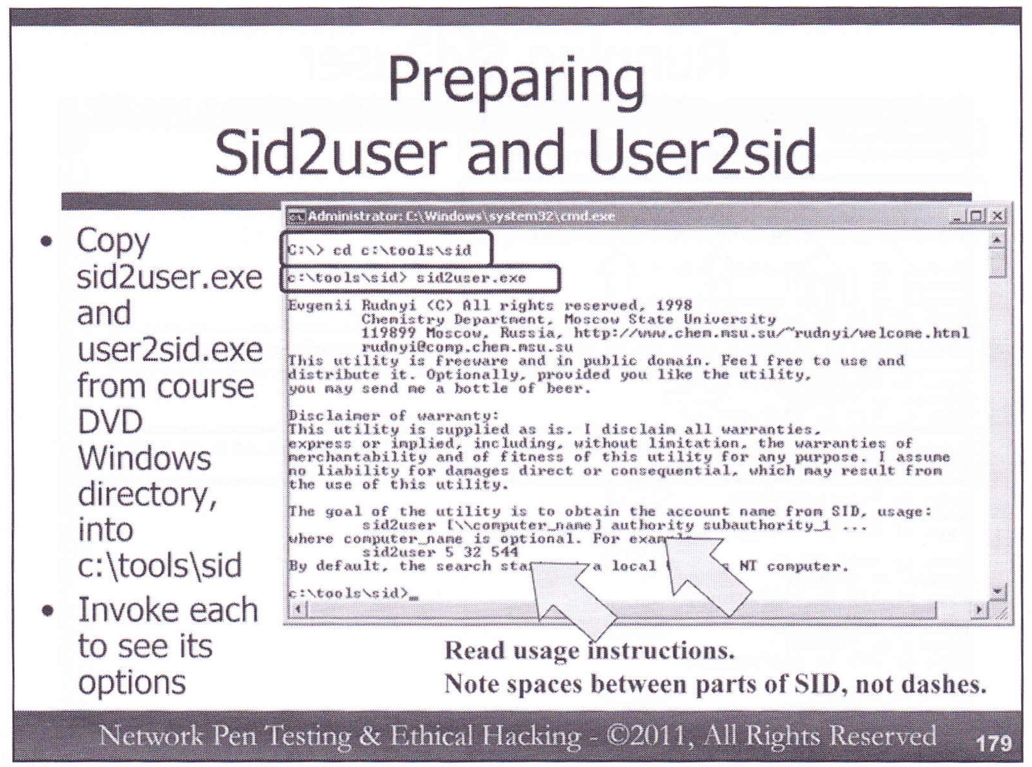
Record your findings here:

Users:

Groups:

Users in Admin Group:

Password settings:



Enum worked, but we have other options that are more widely applicable to machines that even have blocked extracting user and groups via Null sessions using the RestrictAnonymous and RestrictAnonymousSam Registry keys. We could rely on Sid2user and User2sid instead. To start this part of the exercise, copy each of these tools from the course DVD Windows directory onto your hard drive, into a folder called c:\tools\sid.

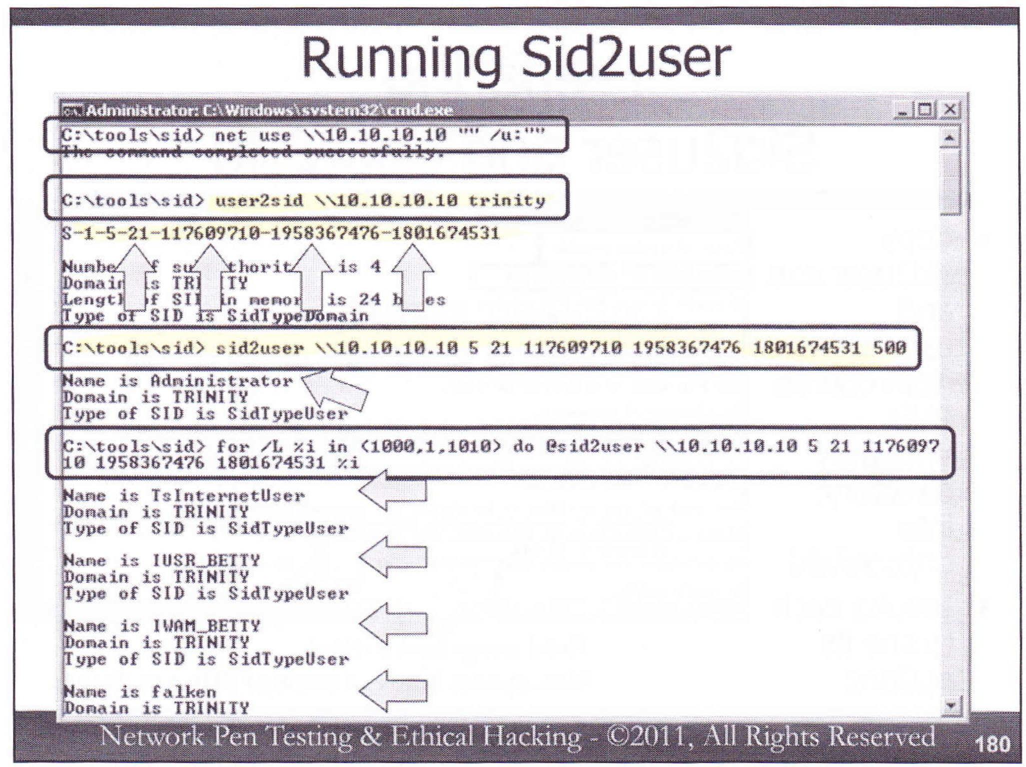
Then, change directories into c:\tools\sid:

```
C:\> cd c:\tools\sid
```

Now, invoke the sid2user tool without any options, and read its usage instructions:

```
C:\> sid2user.exe
```

Note specifically that we can run the tool, followed by a remote computer name with \\[computer_name]. We then give it the SID of the given account, starting with 5 and then a space, followed by the remaining elements of the SID, separated by spaces. Please note that it is asking for spaces between the components of the SID, and not dashes. Windows displays SIDs with dashes, but we need to convert them into spaces when we run this tool.



Now, let's try using the Sid2user method for getting a list of users. First, establish a null session with the target:

```
C:\> net use \\10.10.10.10 "" /u:""
```

Then, run the User2sid command to determine overall domain/computer component of the SID by providing it with hostname of target (we could get hostname from an `nslookup` or `ping -a` command):

```
C:\> user2sid \\10.10.10.10 trinity
```

Then, find out the administrator's name:

```
C:\> sid2user \\10.10.10.10 [domain number, starting with 5 followed by
space, followed by 21, followed by space, followed by 3 sets of digits] 500
```

Don't forget to put the 500 on the end, to specify the administrator's SID.

Then, enumerate users, starting at 1000 and going up through 1010:

```
C:\> for /L %i in (1000,1,1010) do @sid2user \\10.10.10.10 [5 followed by
space, followed by 21, followed by space, followed by 3 sets of digits
separated by spaces] %i
```

This FOR loop is a counter (/L), starting at 1000, counting by intervals of 1, up through 1010 (1000,1,1010), running sid2user on the given domain SID at each iteration through the loop. You should see a series of usernames in the output. Don't worry if you don't understand the details of the FOR loop right now. In 560.3, we have a whole section on Windows command line capabilities, including FOR loops, for professional penetration testers and ethical hackers.

Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

- Scanning Goals and Types
- Overall Scanning Tips
  - Sniffing with tcpdump
- Network Sweeping with Scapy
  - Scapy/tcpdump Exercise
- Network Tracing
- Port Scanning
  - Nmap
  - Nmap Exercise
- OS Fingerprinting
- Version Scanning
  - Nmap -O -sV and Amap Exercise
- Vulnerability Scanning
  - Nmap Scripting Engine
  - NSE Exercise
  - Nessus
  - Nessus Exercise
  - Other Vuln Scanners
- Enumerating Users
  - Enumerating Exercise
- ***Netcat for the Pen Tester***
  - Netcat Exercise

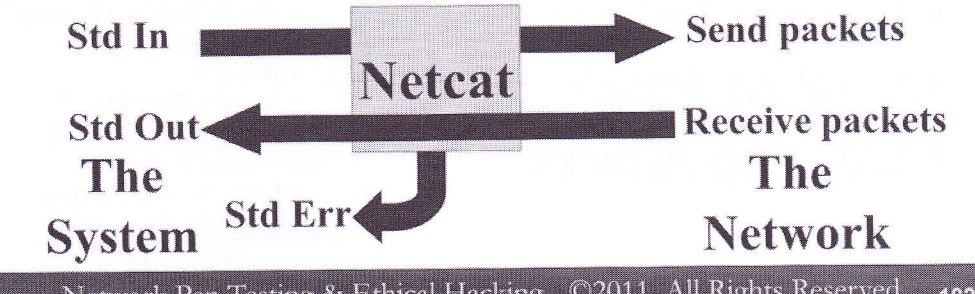
Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 181

In our next section, we'll look at the incredibly flexible tool Netcat, specifically as applied to penetration testing and ethical hacking. Some of you may be Netcat fanatics, while others aren't... yet. As a professional penetration tester or ethical hacker, you'll likely use Netcat on a regular basis in your job. We'll use it throughout the rest of the course, so let's get familiar with it now.

For those of you who already know Netcat, we'll go over some specific uses that are important for penetration testers and ethical hackers, so pay careful attention. And, if you already know Netcat, start brainstorming about how you can use this amazingly flexible tool in other creative ways for penetration testing and ethical hacking. For those new to Netcat, don't worry. We'll describe how the tool works, and then apply it directly to several important tasks.

Netcat for the Pen Tester

- Netcat: General-purpose TCP and UDP network widget, running on Linux/Unix and Windows
  - Built-in to many Linuxes, available for Windows
  - Recent versions of Nmap include ncat – a re-implementation of many Netcat features, plus SSL encryption
  - We'll focus on standard Netcat, given that it is built-in to so many Linuxes
    - Most concepts we'll cover here map directly to Nmap's ncat as well
- Netcat takes Standard In, and sends it across the network
- Receives data from the network, and puts it on Standard Out
- Messages from Netcat itself put on Standard Error



Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 182

Netcat is a general-purpose TCP and UDP network widget for Linux/Unix and Windows, sending data to or from a given TCP or UDP port, or listening for data to come in on a given TCP or UDP port. That's really it from a functionality perspective. But, with those essential capabilities, we can use Netcat for all kinds of network-related tasks that penetration testers and ethical hackers may face every day. Netcat is available in many forms. The most common form is the one installed by default on many variants of Linux, which we'll cover in this class. There is also a great version of Netcat for Windows, which we will also be covering and using in this class. The Nmap development team re-implemented most of Netcat's features in their tool called ncat, which includes SSL encryption capabilities.

Netcat takes whatever comes in on Standard Input and sends it across the network. Standard Input could be the keyboard, redirection from a file (using < for a redirect of Standard Input, as in `nc [options] < [file]`), or piped from another program (using | for piping, as in `[program] | nc [options]`).

When Netcat receives data from the network, it places it on Standard Output. Standard Output could be the screen, redirected to a file (using > for a redirect of Standard Output, as in `nc [options] > [file]`), or sent to another program's Standard Input. To send Netcat's Standard Output to another program's Standard Input, we have two options. We could first simply pipe it using the | symbol, as in `nc [options] | [program]`. That would start streaming the output of Netcat immediately to the program, which would be executed right away. Alternatively, we could use Netcat with the `-e [program]` option, which tells Netcat to execute a program only after a connection is made (for TCP) or data arrives (for UDP). Also, -e has the effect of not only passing whatever Netcat receives on the network to Standard Input of the program, but it also sends Standard Output of the program back across the network via Netcat. A very important property of Netcat involves its use of Standard Error. Any messages from Netcat itself associated with what it's doing on the network are sent to Standard Error. Being able to read and interact with this form of Netcat commentary is useful, as we shall see.

Netcat Command Flags

```
nc [options] [targetIP] [remote_port(s)]
```

-l: Listen mode (default is client)
-L: Listen harder (Windows only) – make a persistent listener
-u: UDP mode (default is TCP)
-p: Local port (In listen mode, this is port listened on. In client mode, this is source port for packets sent.)
-e: Program to execute after connection occurs
-n: Don't resolve names
-z: Zero-I/O mode – don't send any data, just emit packets
-wN: Timeout for connects, waits for N seconds
-v: Be verbose, printing when a connection is made
-vv: Be very verbose, printing when connections are made, dropped, etc.

Netcat
Clients initiate connections

Netcat
Listeners wait for connections

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 183

These are the most important command-line options for Netcat. While there are (many) others, knowing these will help you diagnose Netcat's use in about 95 % of circumstances. The format is:

```
nc [options] [targetIP] [remote_port(s)]
```

The target_system is simply the other side's IP address or domain name. It is required in client mode, of course (because we have to tell the client where to connect), and is optional in listen mode.

-l: Listen mode (default is client).

-L: Listen harder (supported only on Windows version of Netcat). This option makes Netcat a persistent listener, which starts listening again after a client disconnects.

-u: UDP mode (default is TCP).

-p: Local port (In listen mode, this is port listened on. In client mode, this is the source port for all packets sent.)

-e: Program to execute after a connection occurs, connecting Std In and Std Out to the program.

-n: Don't perform DNS lookups on names of machines on the other side.

-z: Zero-I/O mode (Don't send any data, just emit a packet without payload).

-wN: Timeout for connects, waits for N seconds. A Netcat client or listener with this option will wait for N seconds to make a connection. If the connection doesn't happen in that time, Netcat stops running. If a connection does occur, Netcat sends or retrieves data. Then, after Standard In has been closed for a total of N seconds, Netcat stops running.

-v: Be verbose, printing out messages on Standard Error, such as when a connection occurs.

-vv: Be very verbose, printing even more details on Standard Error.

Some Netcat Uses for Penetration Testers and Ethical Hackers

- Right now, we'll use Netcat for a variety of tasks:
  - Connection string gathering from servers or clients
  - Port scans
  - "Service-is-alive" heartbeats
  - "Service-is-dead" notification
- These aren't the only uses of Netcat for a penetration tester or ethical hacker
- We'll cover additional uses as we need them throughout the rest of the course
  - Moving files between systems
  - Setting up relays to forward connections
  - Creating backdoor listeners

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 184

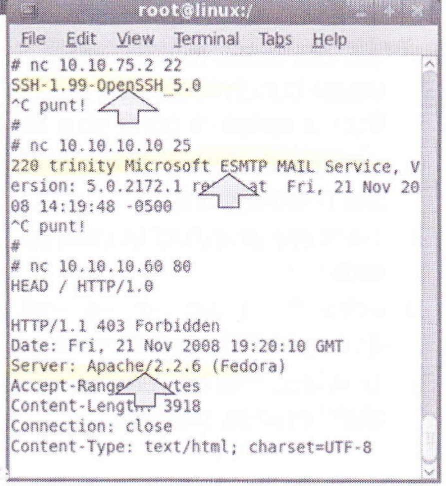
Right now, we'll build up our Netcat skills focusing on various Netcat uses to help penetration testers and ethical hackers. Specifically, we'll look at using Netcat to gather connection strings from servers or clients, which can provide us insights about the software types, version numbers, and protocols they speak. We'll do a hands-on exercise with Netcat as a port scanner and automated connection string grabber from services. We'll look at using Netcat to monitor a target system's services, providing us a heartbeat when a service is alive, or giving us a warning message when a service has gone down.

Please note that throughout the rest of this class, we'll be using Netcat in numerous other ways beyond the ones we're covering in this section. At this point in the course, we wanted to emphasize how Netcat works and how it can help in some penetration testing and ethical hacking job tasks. But, as we move forward to other sections of the course, we'll cover additional uses of Netcat, including moving files, setting up forwarding relays, and creating backdoors.

- A Netcat client can connect to a target service, and pull back its service info

$ **nc [targetIP] [remote_port]**

- You may need to enter a connection string to elicit a response from the target
  - Enter Enter
  - **HEAD / HTTP/1.0**, followed by Enter Enter
  - Others



Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 185

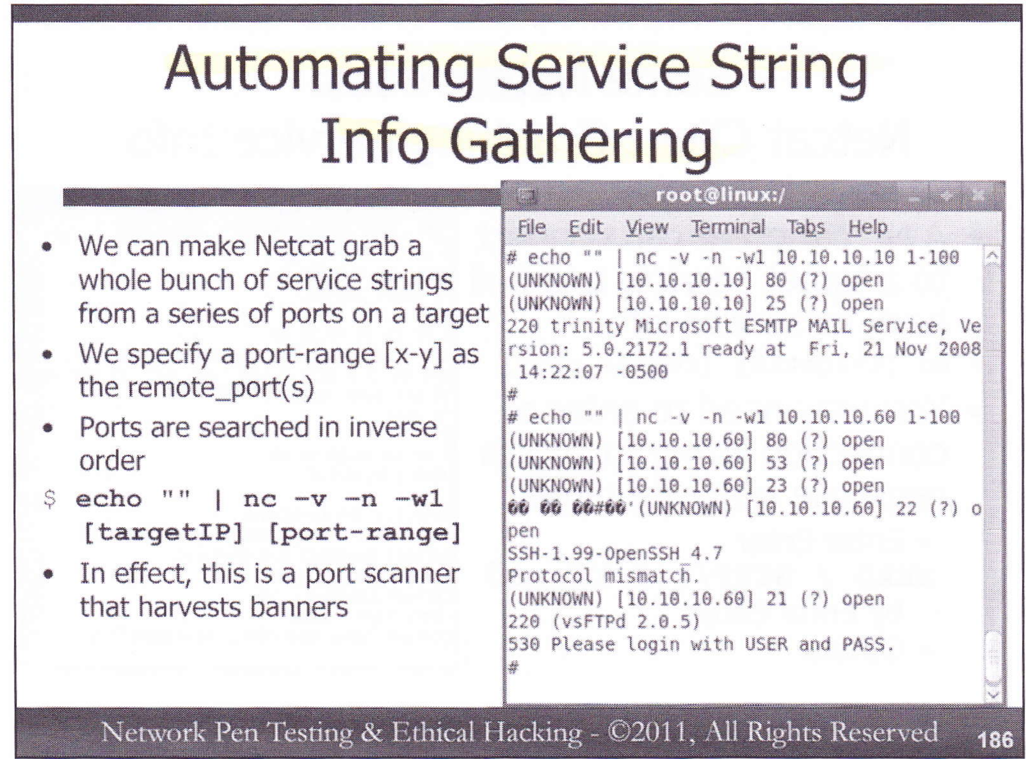
Now that we've had a brief discussion of those command flags, let's look at some practical uses of Netcat for penetration testers. You can harvest a connection string presented by services at connection by simply using a Netcat client to connect to the target service with the following syntax:

$ **nc [targetIP] [remote_port]**

Some services will present a banner including their service type, version number, and protocol immediately upon connection. Other services require some string to elicit a response with this information. For some services, simply hitting Enter Enter will elicit a response. If the target service speaks HTTP, you can get its connection string by typing:

HEAD / HTTP/1.0 followed by Enter Enter.

In the screenshot above, we've used Netcat to connect to 10.10.75.2 on TCP port 22, the port commonly associated with Secure Shell. Upon connection, without any solicitation, the target tells us its version of SSH. We hit CTRL-C to make Netcat drop the connection, which causes the Linux/Unix version of Netcat to print out a message that says, " punt!" on Standard Error, displayed on the screen. We next use Netcat to connect to 10.10.10.10 on TCP port 25. The target tells us that it is running the Microsoft mail service. We connected to 10.10.10.60 on TCP port 80. Nothing was immediately displayed, so we entered **HEAD / HTTP/1.0** followed by Enter Enter. The system told us that it was running Apache, along with its version number and underlying operating system type. While these connection strings can be altered to fool an attacker, they usually tell the truth.

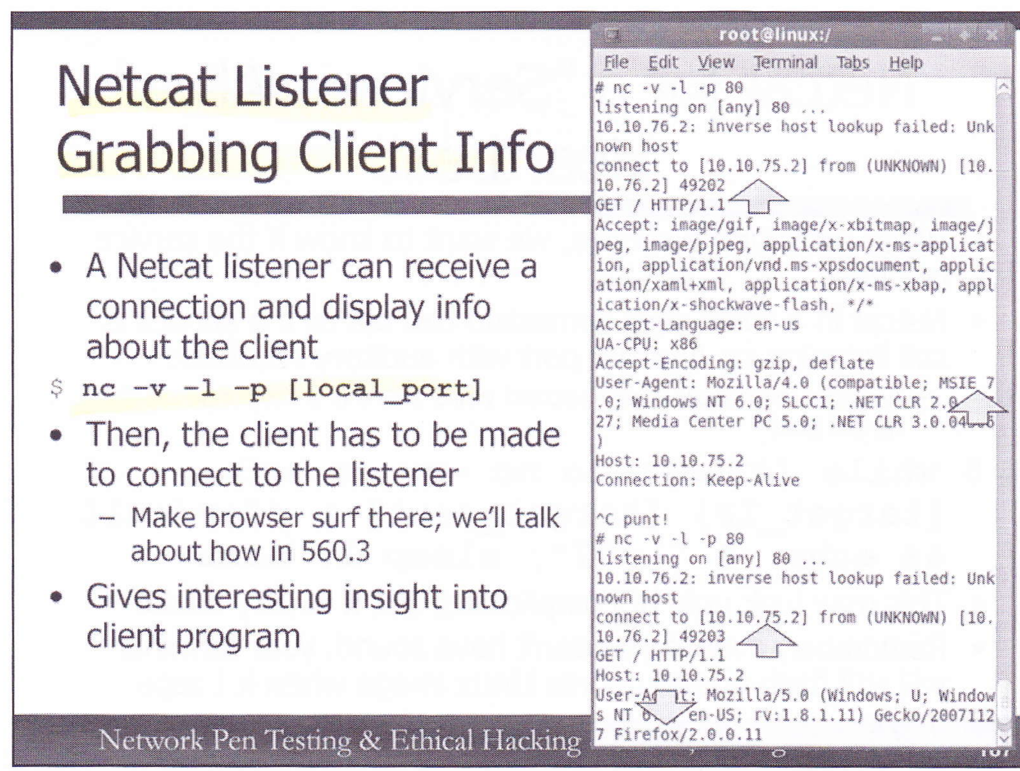


Of course, testing one target machine on one port is helpful, but we might want to automate this over a range of target ports. Netcat supports such functionality, with the [remote_port(s)] option taking a range of numbers, specified as [x-y]. This setting will make Netcat try to connect to the ports, starting at port y, and then decrementing by 1 going down until it tries to connect to port x. The –r flag will make Netcat work through ports in this range randomly, but is only used if you want to be just a little more stealthy.

We can harvest connection strings from a range of ports using this command:

```
$ echo "" | nc -v -n -w1 [targetIP] [port-range]
```

This will echo nothing onto Standard Output, piping that through a Netcat client. We echo nothing to force the closure of Standard Input. Remember, the wait option in Netcat (-wN) will wait for N seconds on an open port after there is no information on Standard Input. If we don't do this echo "", our Netcat client will hang on the first open port, waiting forever for Standard Input from the keyboard, so we purposely echo nothing to close off Standard Input. You'll see how this works in an exercise shortly. We echo our nothing into a Netcat client (nc), verbosely printing output (-v) so we can see when a connection is made, not resolving names (-n) to keep clutter out of our output, waiting no more than 1 second to make a connection or after a connection is made (-w1), of the target IP address on the target range of ports. In the screenshot above, you can see that we directed the scan at 10.10.10.10 and 10.10.10.60, finding some interesting listening ports that didn't return data (TCP 80), and some that did (TCP 25, 22, and 21).



Just as a Netcat client can grab connection strings from a service on the network, we can also have a Netcat listener grab connection strings from clients such as browsers and other network tools. That client connection string provides us insight about the client program.

We can make a Netcat listener wait on a given port as follows:

```
$ nc -v -l -p [local_port]
```

Note that this command includes a dash-lower-case-L, not a dash-one.

Then, we have to direct the client to access the machine on which Netcat is running, on that given port. We'll talk about how to get the client to access the tester's machine on that port in our 560.3 section, when we address client-side exploits.

In the example in the screenshot above, we show a Netcat (nc) listener (-l) running verbosely (-v) on local TCP port 80 (-p 80). A client connected to this Netcat listener. Because we invoked Netcat with a –v for verbose output, we can see the IP address the client had come from displayed on Standard Error. We note that our first connection appears to have come from an Internet Explorer 7 browser, given the User-Agent string, the method a browser can use to tell a server its type and version number. We hit CTRL-C and started another listener. Now, we've got another connection coming in from the same source IP address, but with a User-Agent string that says it is a Firefox browser, with its detailed version number.

Netcat for a "Service-is-Alive" Heartbeat

- While exploiting a service, we want to know if the service crashes
- Netcat in a small shell command can tell us if a service is still listening on a target port with auditory feedback
  - A digital heartbeat every second while there is a response on the target port

```
$ while (true); do nc -vv -z -w3
  [target_IP] [target_port] > /dev/null
  && echo -e "\x07"; sleep 1; done
```

- This may look ugly or complicated, but it is very useful
- Remember, even if you don't have sound, your terminal will still flash on the course Linux image when it beeps

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 188

As a professional penetration tester, while you are exploiting a network service on a target system, you really want to know if and when the service crashes. One way to determine that a service may have crashed is to see if the target system still completes the TCP three-way handshake on the port where the service should be listening. If it doesn't, the service has come down. We can use Netcat in a small shell command to measure whether a service is alive on a regular basis, such as every second or every ten seconds. Our command can provide auditory feedback, beeping if the service is still alive, and going silent if the service stops. In effect, such a command gives us a remote digital heartbeat for the target service. We can implement this functionality with the following command:

```
$ while (true); do nc -vv -z -w3 [target_IP] [target_port] > /dev/null
&& echo -e "\x07"; sleep 1; done
```

This command starts a while loop, which will run continuously. At each iteration through the loop, Netcat is invoked as a client (there is no –l), being very verbose (-vv), sending no data (-z), and waiting no more than 3 seconds to make a connection (-w3) to the target_IP address on the remote target_port. Anything that comes back from the other side is dumped into /dev/null, because we don't really care what the target is telling us, just that it is alive. As long as this Netcat client can make a connection successfully (&&), we want to print the BEL character on Standard Output by making echo evaluate its hexadecimal code (echo –e "\x07"). We wait for 1 second (sleep 1) and the loop starts again. When Netcat cannot make a connection, the echo –e is skipped, and the sound stops.

This may look ugly or complicated, but it is very useful.

Also, please remember, even if you don't have sound support on your system, your terminal border will still flash visibly on the course Linux image when the system tries to beep.

Netcat for "Service-is-Dead" Notification

- Sometimes, you might want to reverse that logic
  - That is, print a message that the service is OK...
  - ...but beep when it dies

```
$ while `nc -vv -z -w3 [target_IP]
[target_port] > /dev/null` ; do echo
"Service is ok"; sleep 1; done; echo
"Service is dead"; echo -e "\x07"
```

- If you really want it to freak out when the service dies, replace `echo -e "\x07"` with `while (true); do echo -e "\x07"; done`

Sometimes, a tester may want different behavior from a service-monitoring command. Instead of a heartbeat showing that the service is alive, we might want a warning saying that the service is dead, beeping when it goes down. We can do that with the following shell command using Netcat:

```
$ while `nc -vv -z -w3 [target_IP] [target_port] > /dev/null` ; do
echo "Service is ok"; sleep 1; done; echo "Service is dead"; echo -e
"\x07"
```

PLEASE MAKE SURE THAT YOU USE A BACKTICK (AN UNSHIFTED TILDE AT THE UPPER LEFTHAND CORNER OF A U.S. ENGLIGH KEYBOARD) JUST BEFORE THE nc AND JUST AFTER THE /dev/null.

Here, we've moved the Netcat invocation itself into the while command to evaluate. In a while loop, we enter the command to evaluate in backticks (`) which are typed with the unshifted tilde on most keyboards. The while loop kicks off a Netcat client (nc), which very verbosely (-vv) sends no data (-z) waiting no more than 3 seconds (-w3) to connect to the target IP address on the target port. Any response that comes back is sent to /dev/null. As long as Netcat makes a connection successfully (the while loop evaluates to positive), our command will print a happy message saying that the "Service is ok", and then sleep for 1 second before going another round in the while loop. If the while loop ends (because Netcat couldn't make a connection), we print out a sad message that the "Service is dead" and ring the bell (echo –e "\x07"). If you want the machine to really beep a lot when the service dies (an emergency warning to be sure!), you could replace **`echo -e "\x07"`** with **`while (true); do echo -e "\x07"; done`**. Such a change will ring the bell until someone hits CTRL-C. That's annoying, but certainly attention getting.

Course Roadmap

- Planning and Recon
- ***Scanning***
- Exploitation
- Password Attacks
- Wireless Attacks
- Web App Attacks

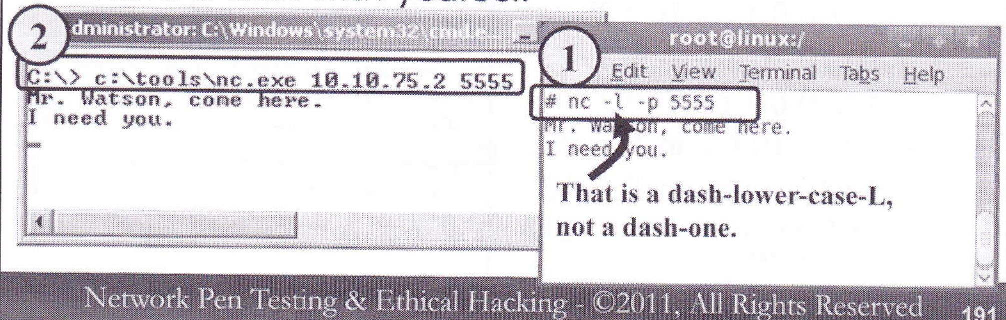
- Scanning Goals and Types
- Overall Scanning Tips
  - Sniffing with tcpdump
- Network Sweeping with Scapy
  - Scapy/tcpdump Exercise
- Network Tracing
- Port Scanning
  - Nmap
  - Nmap Exercise
- OS Fingerprinting
- Version Scanning
  - Nmap -O -sV and Amap Exercise
- Vulnerability Scanning
  - Nmap Scripting Engine
  - NSE Exercise
  - Nessus
  - Nessus Exercise
  - Other Vuln Scanners
- Enumerating Users
  - Enumerating Exercise
- Netcat for the Pen Tester
  - ***Netcat Exercise***

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 190

Now, let's apply some of these Netcat techniques in hands-on exercises. If you finish the written, explicit exercises early, use these exercises as starting points, and feel free to explore further, using them to learn more about the software on your local Windows machine and the Linux virtual machine we've provided. However, please do not make changes to our target machines across the network. Feel free to explore your own Windows and Linux machines to your heart's content.

Playing with Netcat Clients and Listeners

- Start by creating a simple Netcat listener on Linux that does nothing but listen
- Then, on Windows, use a Netcat client to connect to it
- Have a chat with yourself



We'll start this exercise by experimenting with a plain Netcat client communicating with a plain Netcat listener, so we can get a feel for how they are moving Standard Input and Standard Output across the network. In our analysis, we'll look at moving information between a Netcat listener on Linux and a Netcat client on Windows. Start by unzipping Netcat from the course DVD Windows directory (netcat.zip) into c:\tools on your Windows box. Then, run a listener on Linux:

```
# nc -l -p 5555
```

← *Note: That is a dash-lower-case L, not a dash-one.*

This listener will simply wait for a connection to arrive on local TCP port 5555. When it comes in, it will display the data on Standard Output.

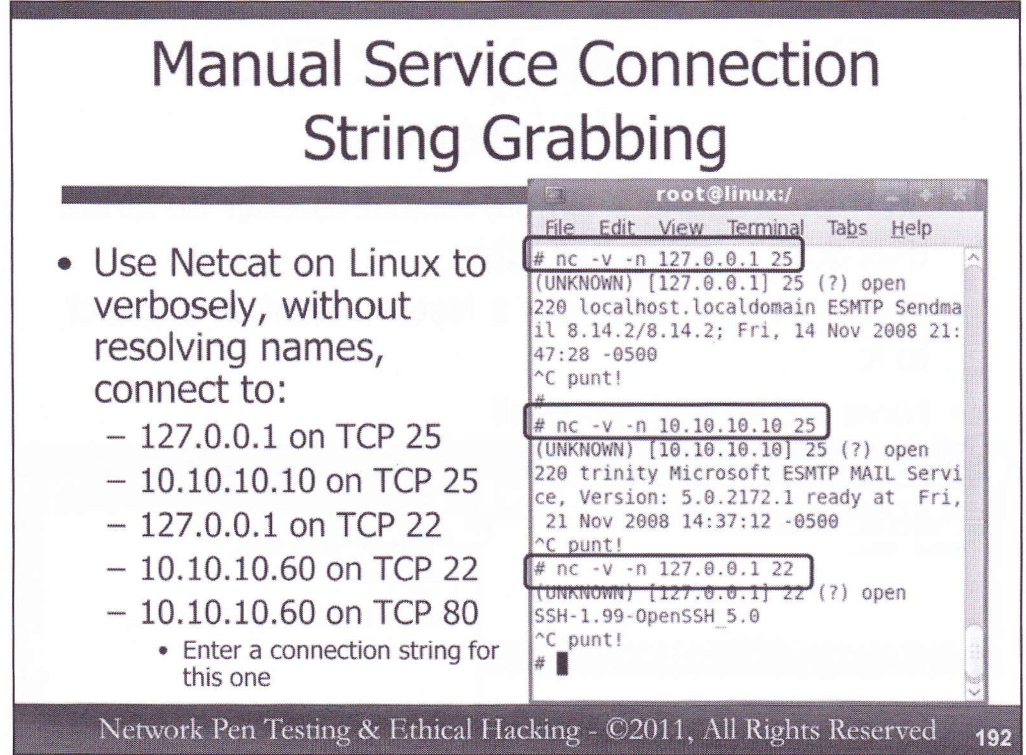
On your Windows machine, initiate a connection from Windows to Linux with Netcat as follows:

```
C:\> c:\tools\nc.exe [YourLinuxIPaddr] 5555
```

When the connection is made, start typing information into either the client or listener. When you hit Enter, the data will be sent to the other side. Type into each side and make sure data is flushed back to the other side. Drop the connection with a CTRL-C.

If the connection is not successful, your Linux firewall may be blocking it. Disable the Linux firewall with:

```
# service iptables stop
```



Now that we have seen how data is exchanged using Netcat clients and listeners with Standard Input and Standard Output, let's try some manual service connection string gathering. From your Linux machine, we'll pull information from various locations. We want to run a Netcat client, verbosely (-v) without resolving names (-n) to connect to our localhost (127.0.0.1), connecting to TCP port 25. Try that, using an IP address of 127.0.0.1. Also try it with a target machine name of localhost. Why doesn't the latter work?

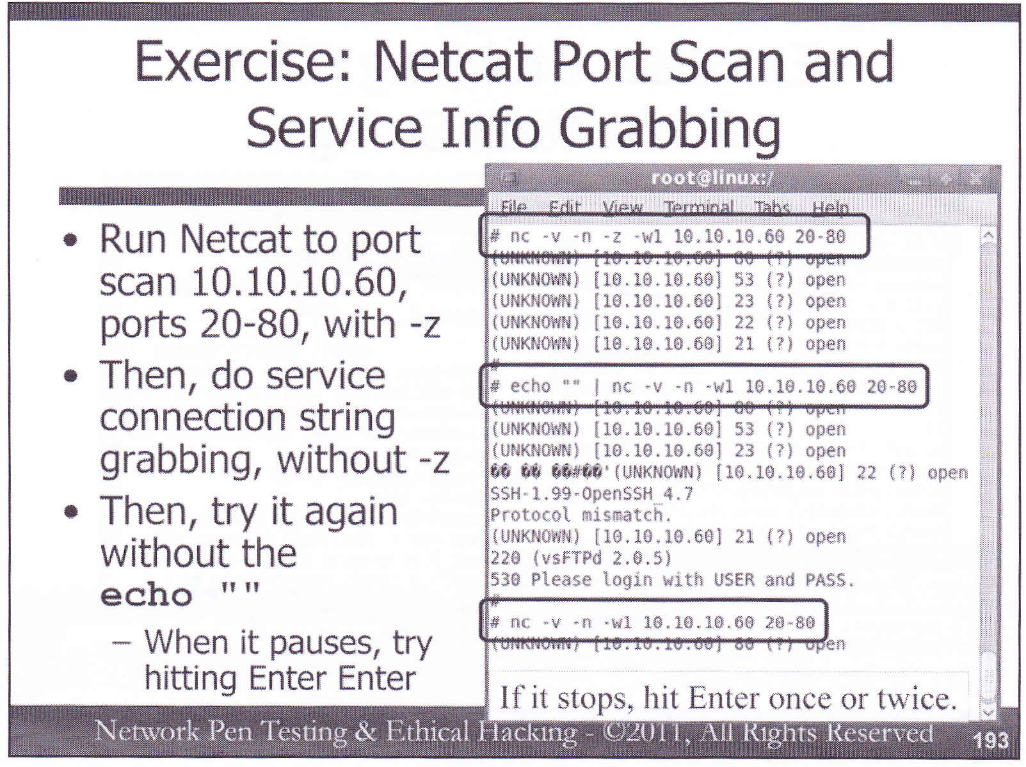
```
# nc -v -n 127.0.0.1 25
# nc -v -n localhost 25
```

Hit CTRL-C to drop any connections you make. Now, try pulling connection strings from the following targets, comparing the results and trying to determine the service, its version, and anything the target tells us about the operating system type:

```
# nc -v -n 10.10.10.10 25
# nc -v -n 127.0.0.1 22
# nc -v -n 10.10.10.60 22
# nc -v -n 10.10.10.60 80
```

For that last one, type in the appropriate HTTP connection string to elicit a response:

HEAD / HTTP/1.0 (Followed by Enter Enter)



Next, we will explore the different behaviors Netcat has with and without –z, and with and without `echo ""`, when port scanning and pulling service connection strings from a target machine. Start the exercise by using your Linux Guest machine connected to our network to conduct a port scan of target 10.10.10.60, with ports 20 through 80:

```
# nc -v -n -z -w1 10.10.10.60 20-80
```

This will tell Netcat to run verbosely (-v, printing when a connection is made), not resolving names (-n), without sending any data (-z), waiting no more than 1 second for a connection to occur (-w1) on target 10.10.10.60, TCP ports 20 through 80. You should see a series of open ports. *But please note that you don't see any strings that come back from the services. You only get an indication of which ports are open, but not the connection string.*

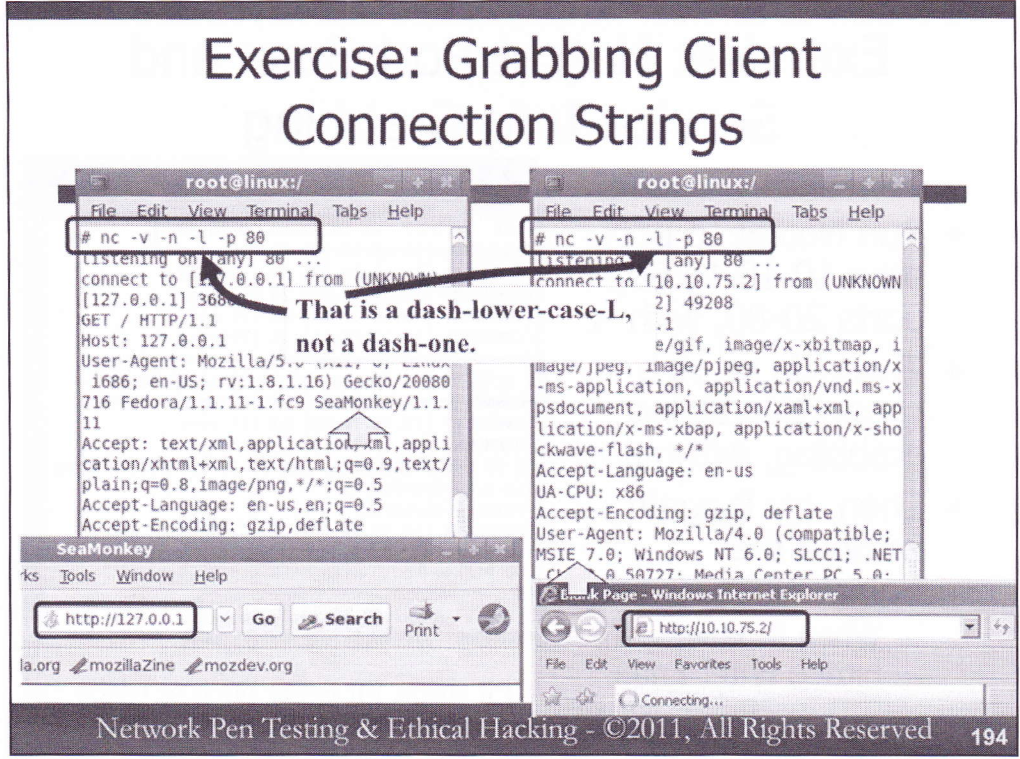
Then, let's do our connection string grabbing. Make sure you omit the –z from this command! If you include –z, you won't see the connection strings, because Netcat will move on before it gets any data back. The –z and –w used together have that impact.

```
# echo "" | nc -v -n -w1 10.10.10.60 20-80
```

You should see the open ports, as well as connection strings from some (but not all) of the services.

And, finally, try running this again, but without the `echo ""`. You'll see that it pauses on the first open port, waiting for Standard Input from you on the keyboard. Because Standard Input stays open without the `echo ""`, Netcat pauses. Hit Enter once or twice to nudge it along.

```
# nc -v -n -w1 10.10.10.60 20-80
```



Next, on your Linux machine, set up a Netcat listener that will verbosely listen on local TCP port 80, not resolving names of systems that connect there:

```
# nc -v -n -l -p 80
```
← *Note: That is a dash-lower-case L, not a dash-one.*

Then, from another terminal on your Linux machine, run the Mozilla browser, kicking it into the background with &:

```
# mozilla &
```

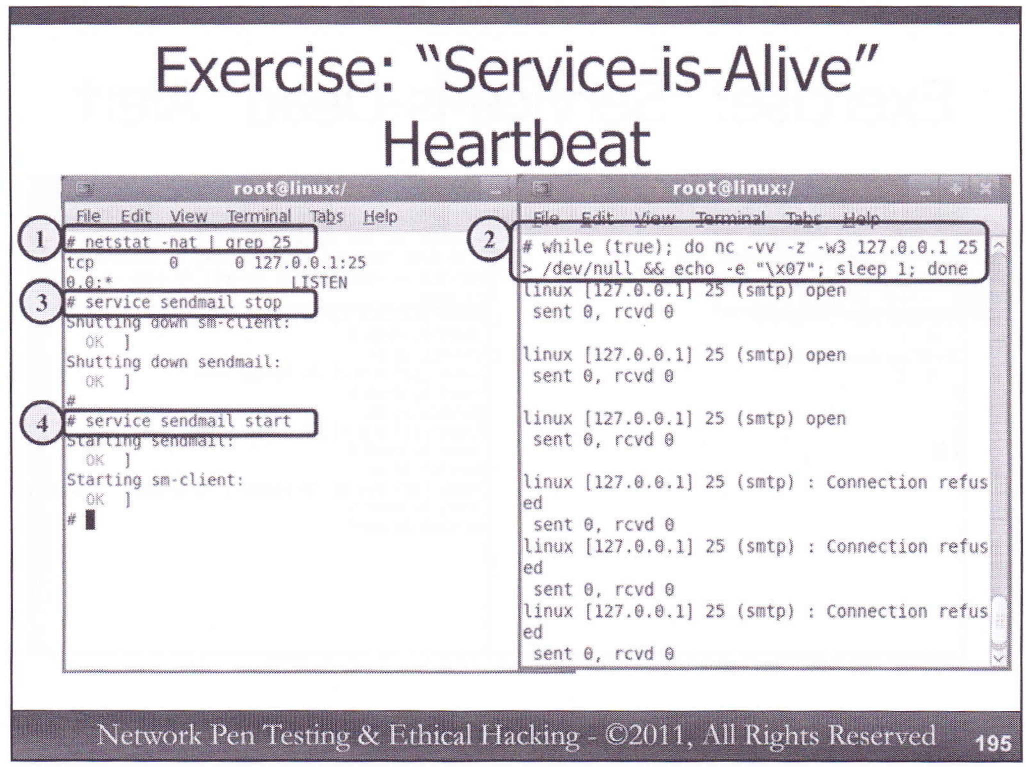
When the browser comes up, enter a URL for it to surf to http://127.0.0.1

Look at your Netcat output, specifically the User-Agent string. It tells you the kind of browser that just accessed the Netcat listener.

Now, hit CTRL-C in your Netcat window on Linux, and then restart your Netcat listener, again on TCP port 80:

```
# nc -v -n -l -p 80
```
← *Note: That is a dash-lower-case L, not a dash-one.*

Now, from Windows, run Internet Explorer, and have it surf to a URL of http://[LinuxIP]. Note its User-Agent string. Try other Windows client programs that you might have, such as Firefox, RealPlayer, and others, having them surf to [YourLinuxIPaddr]:80. Most of these programs have options for opening a URL,typically by going to File→Open… and typing in a URL of the form http://[IPaddr]:[port] or simply [IPaddr]:[port]. Make a note of the various User-Agent strings you identify for IE, RealPlayer, QuickTime, etc.



Let's try a "service-is-alive" heartbeat checker with Netcat. On our Linux machines, we have configured the system with a listening Sendmail server on TCP port 25.

In Step 1, you can see this listening port with the netstat command, invoked to show us numbers (not names) of all port and socket usage (-a) of TCP ports (-t), scraping output for the number 25:

```
# netstat -nat | grep 25
```

For Step 2, in a separate window, let's set up a Netcat heartbeat to check that port:

```
# while (true); do nc -vv -z -w3 127.0.0.1 25 > /dev/null && echo -e
"\x07"; sleep 1; done
```

You should hear the heartbeat.

In Step 3, go back to your first window and stop the sendmail service:

```
# service sendmail stop
```

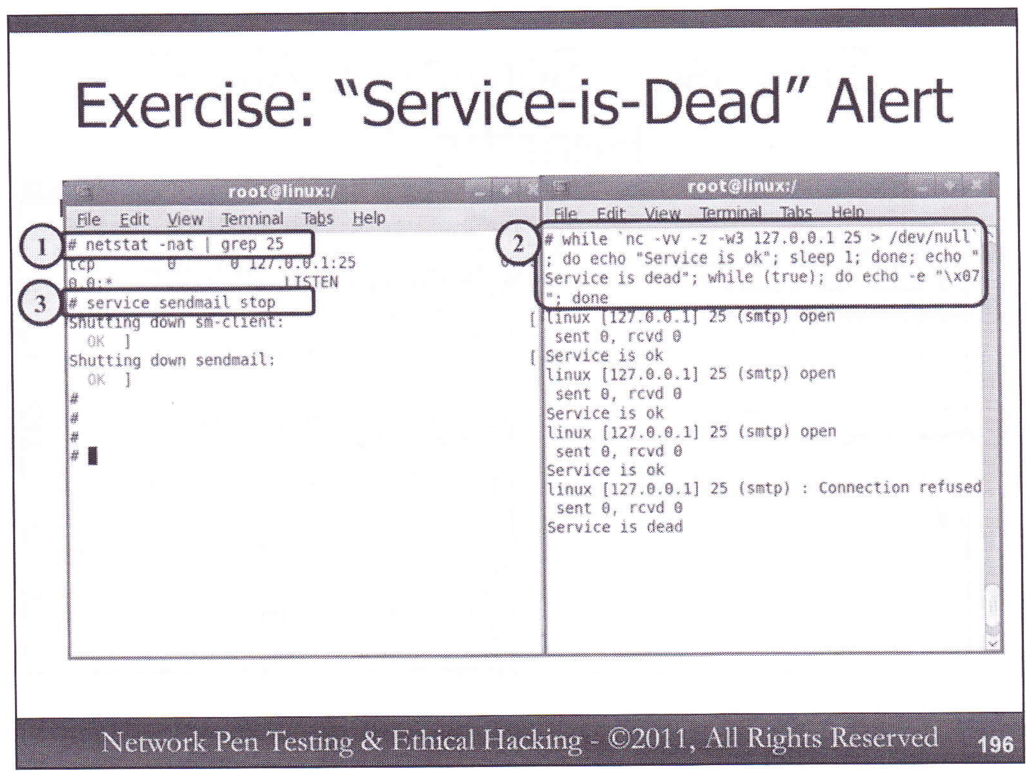
The Netcat service-checking heartbeat should go silent. The service is down!!!

In Step 4, when you bring your sendmail back, the heartbeat starts again:

```
# service sendmail start
```

To end the heartbeat monitor, simply hit CTRL-C in its window. If that doesn't stop it, hit CTRL-Z, followed by:

```
# killall -9 nc
```



Next, let's create our "service-is-dead" red-alert message with Netcat, again monitoring our Sendmail service.

In Step 1, verify that the port is listening:

```
# netstat -nat | grep 25
```

If the port isn't listening, start your Sendmail service using the command on the previous slide. For Step 2, in a separate window, let's set up a Netcat monitor to check that port, printing happy messages when the port completes a connection, and making lots of noise when it doesn't:

```
$ while `nc -vv -z -w3 127.0.0.1 25 > /dev/null` ; do echo "Service is ok";
sleep 1; done; echo "Service is dead"; while (true); do echo -e "\x07";
done
```

PLEASE MAKE SURE THAT YOU USE A BACKTICK (AN UNSHIFTED TILDE AT THE UPPER LEFTHAND CORNER OF A U.S. ENGLIGH KEYBOARD) JUST BEFORE THE nc AND JUST AFTER THE /dev/null.

When the command is running, you shouldn't hear anything for now... But get ready.

In Step 3, go back to your first window and stop the sendmail service:

```
# service sendmail stop
```

Your system should now make lots of noise. The service is down! Ouch. Stop it by hitting CTRL-C in the window running Netcat. Note that the nature of the monitor command we used this time does not stop the noise when the service comes back up. It just keeps making noise. You could alter the command to make it a while (true) loop with that kind of functionality if you have extra time. Also, if you want, you can start your Sendmail service again.

Conclusion for 560.2

- That concludes the 560.2 session
  - We've gathered information about target system types, open ports, available services, and other useful information
  - At this stage of a project, the tester has completed scanning, and is poised to perform exploitation
- In 560.3, we'll look at exploitation in depth

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 197

This will bring our 560.2 section to a close. Throughout the scanning phase, penetration testers and ethical hackers gather very useful information about the target environment that will be critical in the ongoing stages of a test. We've analyzed methods for determining many things about the target environment, including operating system types, open ports listening on the network, available services, and other useful information about target machines.

The next phase of the testing process will focus on exploitation, the topic we'll address in depth in 560.3, and continue through the first part of 560.4.

Optional Appendix:

Hping

Review if you have extra time.
Not covered in class.

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 198

If you have extra time, you may want to review this Appendix, which covers the Hping tool. Hping can be used to craft packets and send them to a target system, like Scapy. Although it isn't as flexible as Scapy, Hping is more likely to be installed on target systems, and can therefore be helpful to know. In this optional appendix, we'll review Hping configuration, with a particular focus on how Hping can be used to sweep a target environment.

Network Sweeping with Hping

- Inspired by ping, but goes much further
  - Originally Hping, then Hping2... latest is Hping3
  - From man page: "Send (almost) arbitrary TCP/IP packets to network hosts"
- Free at www.hping.org, runs on Linux, *BSD, MacOS X, and Windows
- The latest version, Hping3, supports TCL scripting
- By default, sends TCP packets with no control bits set to target port 0 continuously, once per second
  - Possibly getting RESETs back
- Example: # **hping3 10.10.10.20**

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 199

Hping is a general-purpose packet generation tool, useful for a variety of scan types. It can be used to conduct a network sweep with several different protocols. Furthermore, it supports tracerouting in a flexible fashion, as well as port scans. And, Hping's flexible protocol and payload options make it useful even beyond the scanning phase, with useful abilities for exploitation as well.

Originally, the tool was called Hping. Hping2 supplanted that version, and was itself superseded by Hping3, the latest version. We'll refer to the tool as Hping, realizing that we are discussing the latest version from this point on.

Hping3 offers all of the same functions as Hping2, but with a lot of bug fixes. Furthermore, Hping3 has been expanded to support Tool Command Language (TCL) scripting. Thus, instead of using standard command line options or shell scripts to control Hping, scripters can use TCL. Our use for this course will continue to be the most common method of invoking Hping: at the command line.

Hping was created for Linux and Unix (including Mac OS X), but has also been ported to Windows. By default, Hping merely "pings" a target IP address by sending TCP packets with no control bits set (SYN, ACK, FIN, RST, PSH, and URG are all set to zero) to the target machine on port 0. Hping will continuously send one packet per second, until it is stopped. Most systems respond with a RST packet, indicating that there is a system there. Admittedly, this is an unusual ping, but it can be effective in some network environments.

Hping Protocol Selection

- Default protocol is TCP, but can easily be switched using the following flags

--udp: Send UDP packets

--icmp: Send ICMP packets

--rawip: Send raw IP packets, with no TCP or UDP component

- Example:

```
# hping3 --rawip 10.10.10.20
```

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved

By default, Hping uses TCP packets. However, it can easily be switched to use other protocols, such as UDP (with the --udp option), ICMP (with the --icmp option), or raw IP packets (with the --rawip invocation). The raw IP mode will send packets without a TCP or UDP header in them. Later, we'll see how to put the contents of a specific file as a payload on such packets.

For an example, consider this invocation:

```
# hping3 --rawip 10.10.10.20
```

This command will make Hping send raw IP packets to 10.10.10.20. Most systems will silently reject a raw IP message.

Setting TCP Control Bits

- By default, Hping sets all TCP Control Bits to zero
- But, Hping supports a simple syntax to choose control bits:

  --syn --push
  --fin --ack
  --rst --urg
- Numerous other TCP, UDP, ICMP, and IP settings are configurable as well

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 201

As we've discussed, Hping sends a TCP packet by default, with all of the TCP Control Bit values set to zero. These items can be turned on and off independently by simply using the command line options of --syn, --fin, --rst, --push, --ack, and/or --urg when invoking Hping.

Other TCP, UDP, ICMP, and IP settings are configurable as well, such as TCP sequence numbers, TCP and UDP checksums, IP Time-to-Live values, IP identification numbers, and so on. The hping3 man page includes these details.

Hping Target Selection

- As we've seen, Hping can send packets to a single target by merely specifying its IP address or domain name
- Alternatively, we can specify:
  - --rand-dest IP_addr: Will send packets to random targets wherever an x is included in the IP address (e.g., 10.10.10.x will send packets to targets from 10.10.10.1 to 10.10.10.255). Note that targets are repeated randomly.
  - --interface [Int]: When using the random destination option, you must specify which interface to send the packets on

Example: # **hping3 --rand-dest 10.10.10.x --interface eth0**

- Unfortunately, Hping doesn't support a range of targets, unless we use a shell or TCL script

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 202

By default, Hping sends packets to a single target destination, as we have seen. Alternatively, it supports a rather crudely tuned option to choose targets randomly within a target network environment. We can specify --rand-dest followed by dotted-quad IP address with one or more x characters in it (such as 10.10.10.x or 10.x.20.30 or 10.20.x.x). Hping will then generate packets by creating a random number between 1 and 255 for each x, and then sending packets there. This is a crude tuning because it cannot be used to hit anything smaller than a single class-C sized network (/24), nor can it be used to hit fine-tuned selections of target networks.

Also, when using the --dest-rand option, Hping requires the user to specify which interface the packets should be sent through, with the --interface directive. For example, consider this command:

```
# hping3 --rand-dest 10.10.10.x --interface eth0
```

This invocation will make Hping send packets to random targets in the 10.10.10 network, using interface eth0 to transmit the packets. Note that Hping does not maintain state of the targets it has already tested; each one is selected randomly each time, making it likely that we'll have repeats of a single target in potentially short periods of time.

Unfortunately, Hping does not provide options at the command line for doing a sequential walk through a target network range at the command line by itself. Instead, a user would have to write a shell script at the command line or a TCL script inside of Hping to make it walk through a range of targets sequentially.

Hping Source Selection

- --spoof [IPaddr]: sets spoofed source IP address of all packets sent

Example: `# hping3 --spoof 10.10.10.10 10.10.10.20`

- --rand-source: Randomly selects a source address for all packets
  - No way to specify range
  - Still useful for stress testing stateful firewalls
  - May fill up a state table, causing additional packets for other users to be dropped

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 203

Hping provides some options for specifying the source IP address of the packets that it creates. By default, it uses the IP address of the machine running Hping, as you might expect.

For example, consider this command line:

```
# hping3 --spoof 10.10.10.10 10.10.10.20
```

This syntax will cause Hping to send packets from a source address of 10.10.10.10, directing them to a destination address of 10.10.10.20. The tool will generate its default TCP packets with zero for all of the control bits to destination port 0.

Alternatively, Hping supports the --rand-source option, which will generate packets with a completely randomized source address. Unfortunately, there are no options to narrow down the range of random source addresses. Still, this technique can be used to stress test a firewall, generating a large number of packets from a large number of source addresses. Such actions can fill up the state table of some firewalls, causing them to drop packets for other users, so be careful.

Hping Port Selection

- --destport [port]: Use this destination port
  - If preceded by a +, port is incremented by 1 for each response *received*
  - If preceded by a ++, port is incremented by 1 for each packet *sent*
- --scan [port_range/list]: Scan this target range or list of ports (x-y,z,known)
- --baseport [port]: Start with this source port, incrementing for each packet sent
  - No + and ++ supported for source port (behavior is effectively like ++ for destport)
  - Default is to use random baseport
- --keep: Use only a single source port for all packets

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 204

In choosing destination and source ports, Hping supports several options. For destination port, we can use the --destport or --scan options as follows:

--destport [port]: This tells Hping to generate packets for this target port. If the target port number is preceded with a +, the port is incremented by 1 for each *response* packet that Hping receives. A ++ before the port number tells Hping to increment the destination port number by 1 for each packet that Hping *sends*, something like the default behavior of the Unix and Linux traceroute command (which, as you may recall, increments through high UDP ports for each packet that it sends).

--scan [port_range/list]: This invocation tells Hping to send packets to a range of destination ports (with a – to indicate the range) or to a comma-separated list of target ports. We can mix and match ranges and lists, using syntax such as 1-100,135,139,445,700-800 to test ports from 1 to 100, plus 135, 139, and 445, as well as ports 700 to 800. The "known" keyword here makes Hping send packets to the list of ports in /etc/services.

For source port selection, Hping uses the "--baseport [port]" syntax, which will cause Hping to start sending packets with a source port of N, incrementing by 1 for each packet sent. Unfortunately, Hping does not support the + or ++ notation for source ports. For source port, by default it behaves in the fashion of a ++ for destination port; that is, it increments by one for each packet sent. If no source port is specified, Hping uses a randomly selected port number higher than 1024.

The --keep syntax tells Hping to use a fixed source port for all packets that it sends.

Hping: Some Helpful Options

- Hping supports numerous additional useful options
  - --count [N]: Send only N packets
  - --beep: Beep when a packet is received
  - --file [filename]: send contents of file as a payload, must be used with --data
  - --data [N]: Length of payload to send, in bytes (if no --file, payload is X's)

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved

Here are some useful miscellaneous options for Hping.

Specifying a --count [N] at the command line will limit the number of packets Hping generates to N. By default, Hping sends packets continuously, but, for example, with a "--count 4", it will only send 4 packets.

The --beep option makes the tool audible, triggering the system beep sound when a response is received back. This can act as a audible heartbeat to verify that the tool is still running and getting responses.

Users can specify "--file [filename]" to indicate a specific file whose contents should be used as the payload of all packets sent to the target. This option requires the user to also specify a "--data [N]", which indicates the length of the payload to send. If --file is used, a --data *must* be provided. If a --data is used, and a --file is not provided, the payload is padded with the character X.

Hping: Speed Options

- By default, hping sends one packet per second, but this can be changed

--fast: Send ten packets per second

--faster: Send 1,000,000 packets per second (if possible)

--flood: Send packets as fast as possible, perhaps even faster than they can be displayed

--interval [N] (or u[N]): Send packets every N seconds (or every uN microseconds)

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 206

By default, Hping sends one packet per second. We can speed this up considerably using various command-line options. The --fast option makes Hping send ten packets per second, while the --faster option sends one million packets per second if the interface can keep up with that pace. The --flood option sends packets as quickly as possible, likely exceeding the system's ability to let hping display any responses that come back.

A much more reasonable speed option is the "--interval [N]" or "--interval u[N]", which makes Hping send one packet every N seconds or every uN microseconds.

Using Hping to Iterate through an Address Space

- We can use Hping to iterate through an address space using some of the features of the Linux shell:

```
# for i in `seq 1 255`; do hping
--count 1 10.10.10.$i; done
```

- Or, if you only want to focus on the systems that respond, grep output for "ip=", because that string is included in responses:

```
# for i in `seq 1 255`; do hping
--count 1 10.10.10.$i 2>/dev/null
| grep ip=; done
```

Network Pen Testing & Ethical Hacking - ©2011, All Rights Reserved 207

We have seen how we can use Hping to move through a series of ports with the + and ++ notation. To make it iterate through a series of target IP addresses, however, we can use a bit of Linux shell functionality as follows:

```
# for i in `seq 1 255`; do hping --count 1 10.10.10.$i; done
```

Here, we start a for loop, which will iterate over the variable i, changing its value at each iteration through the loop. We iterate over the output of the seq command, which is set to generate a sequence of digits from 1 to 255 on its output. The backticks (`) on either side of the seq 1 255 command ensures that that command will be executed so that we can iterate on its output. At each iteration through the loop, we run hping, configured in whatever way we want. Here, we've set it to send a single packet (--count 1) with default settings to the target address of 10.10.10.$i. The $ in front of the i will make the shell expand it to the current value of the i variable. While this loop will work to help identify which hosts are in use, its output will be very cluttered. Each packet sent will generate a couple of lines, which might make our responses difficult to locate.

To help focus our command on those instances where we get a response, we can use the fact that, when hping receives a response, it displays the text "ip=" followed by the IP address on the screen. We can scrape through the output of the hping command to find only those hosts that respond by simply taking Standard Error messages and throwing them away (2>/dev/null), and then piping (|) our Standard Output to the grep command, looking for the string "ip=". The resulting command is:

```
# for i in `seq 1 255`; do hping --count 1 10.10.10.$i 2>/dev/null |
grep ip=; done
```