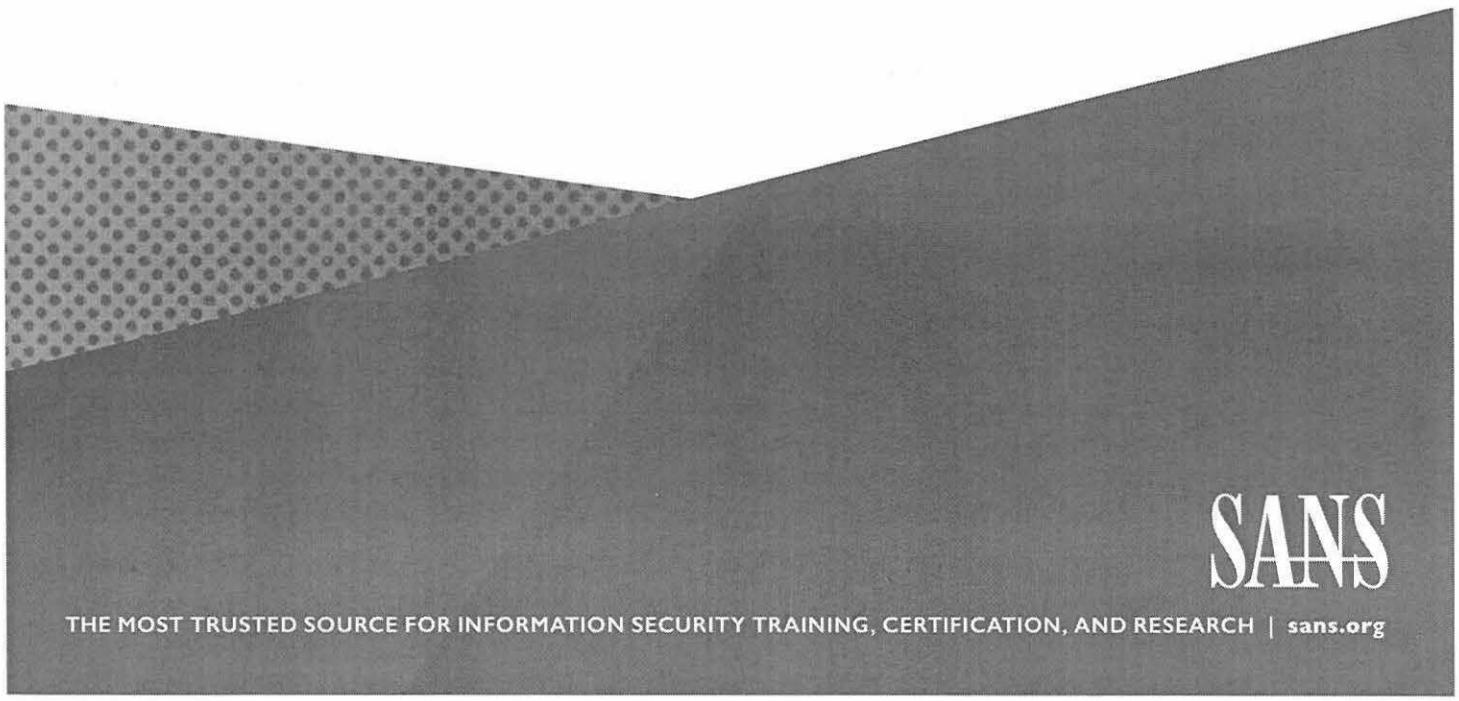
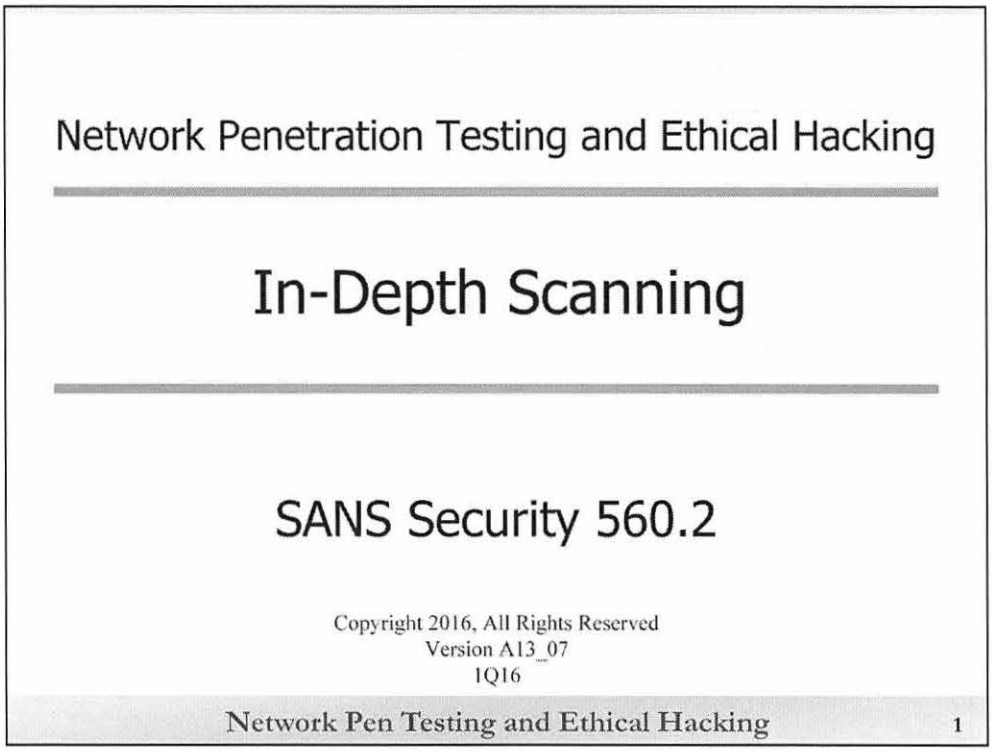
SEC560 | NETWORK PENETRATION TESTING AND ETHICAL HACKING

560.2
In-Depth Scanning



SANS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org



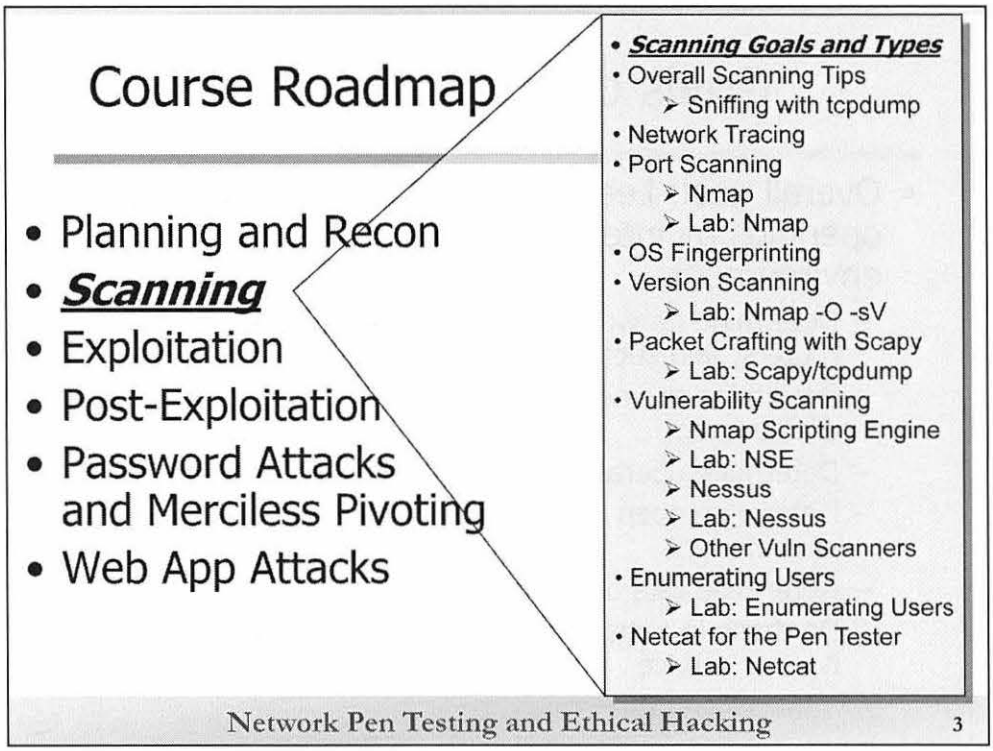
Hello, and welcome back. Today's section is called 560.2, Scanning. This component of the course focuses on the vital task of scanning a target environment, creating a comprehensive inventory of machines, and then evaluating those systems to find potential vulnerabilities. We'll look at some of the most useful scanning tools freely available today, experimenting with them in our hands-on lab. Our hands-on labs 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.

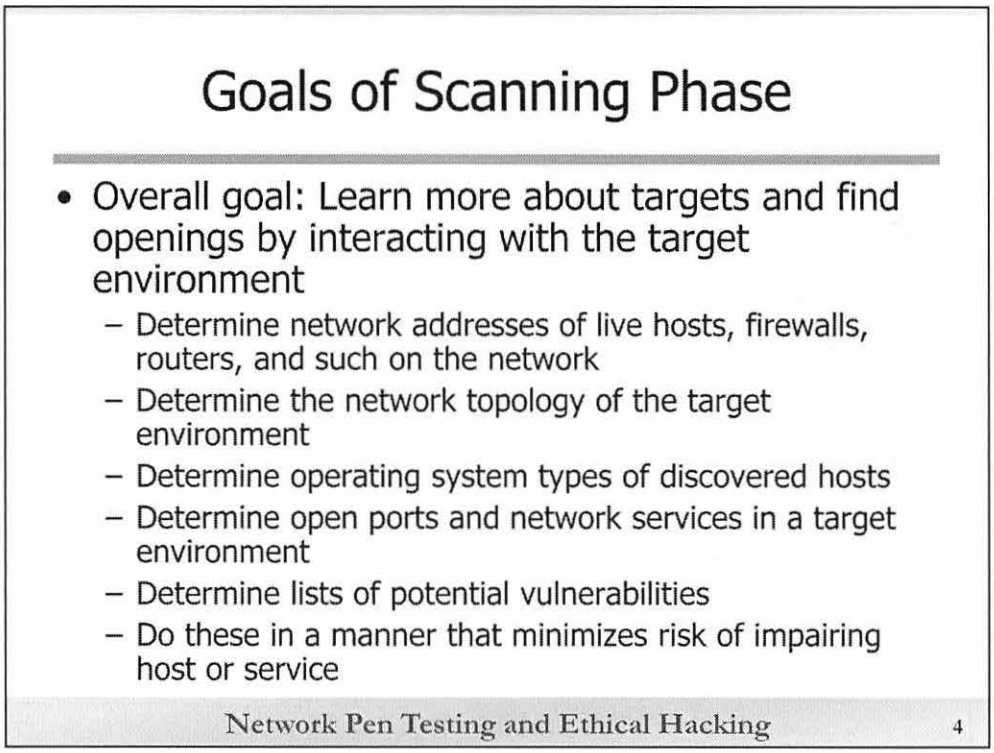
560.2 Table of Contents

Slide #

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



We'll start this section by discussing the goal of scanning and the different kinds of scans. We'll then review 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.



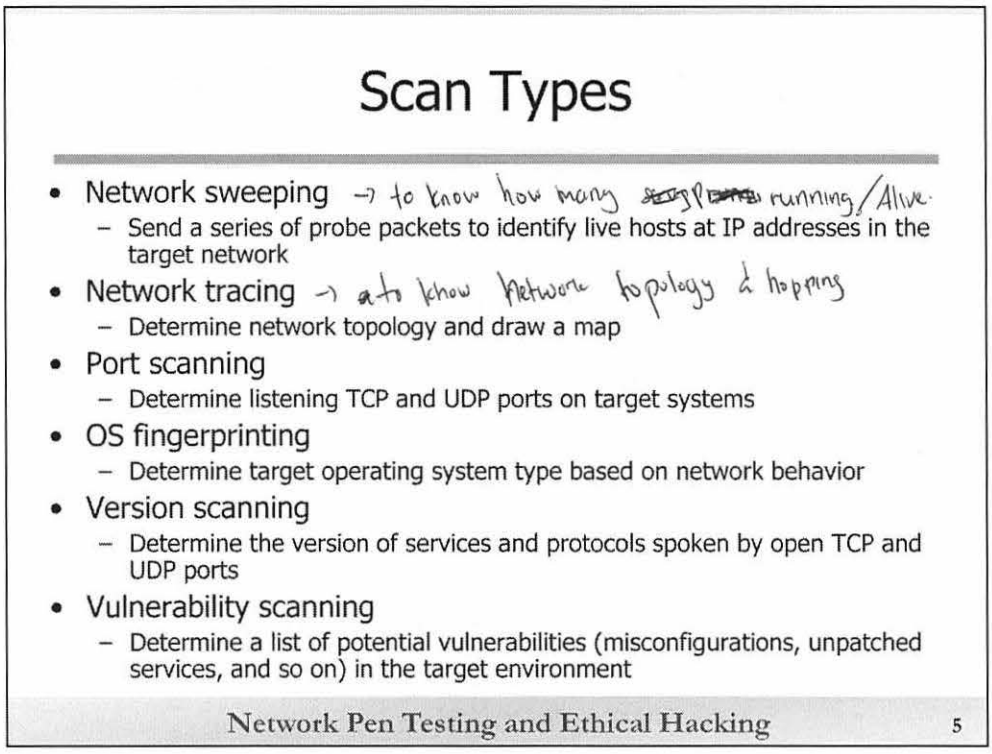
The overarching goal of the scanning phase is to learn more about the target environment and find openings by directly interacting with the target systems. Particular objectives under this goal include determining the addresses used by systems on the target environment, including hosts (servers and clients), network equipment (firewalls, routers, and 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 (for example, HTTP version, SMTP version, and SSH protocol version) that it speaks.

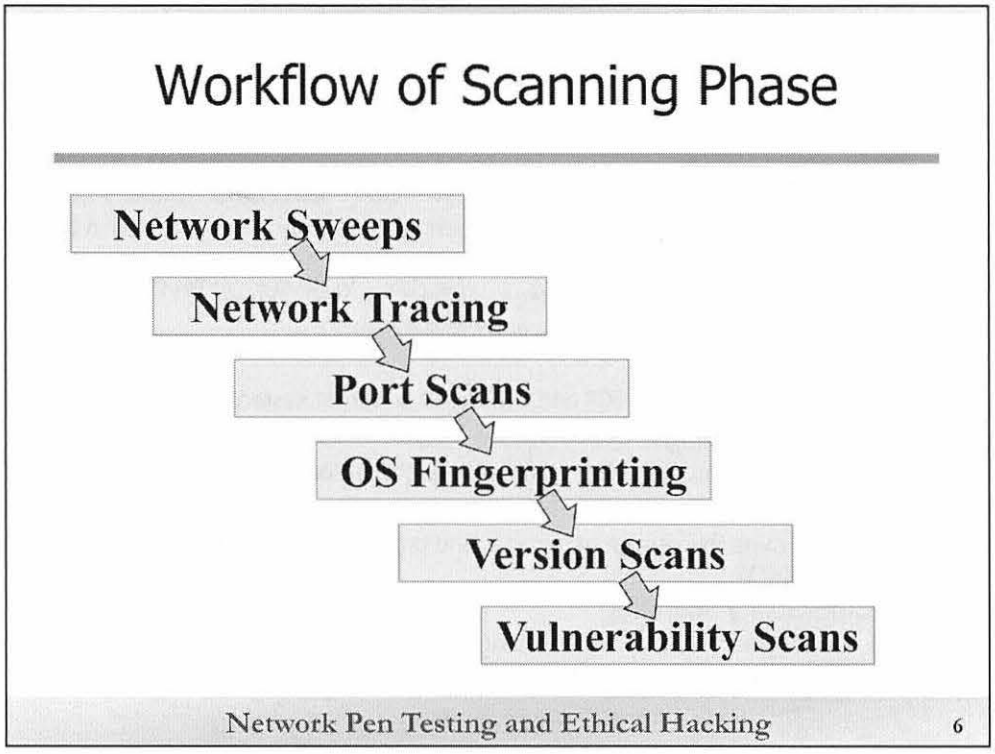
We then want a list of potential vulnerabilities, which may be determined from the version numbers determined earlier or based on the behavior of the target system in light of certain kinds of network interactions.

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



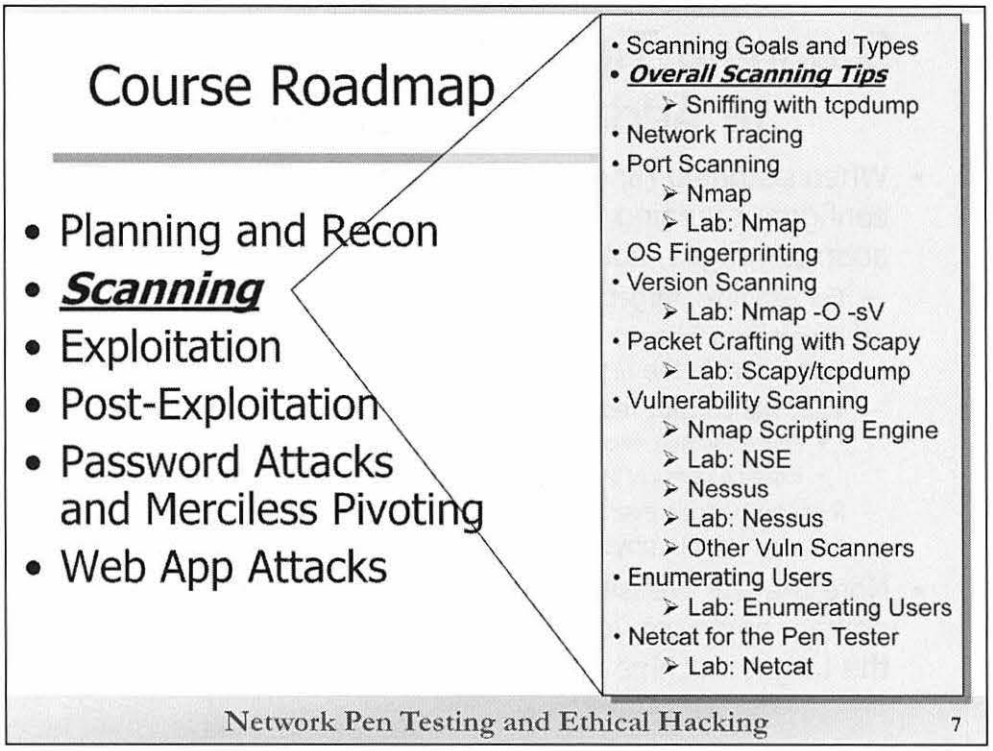
To achieve our goals, we'll perform several types of scans during the test, including:

- **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 (for example, 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.

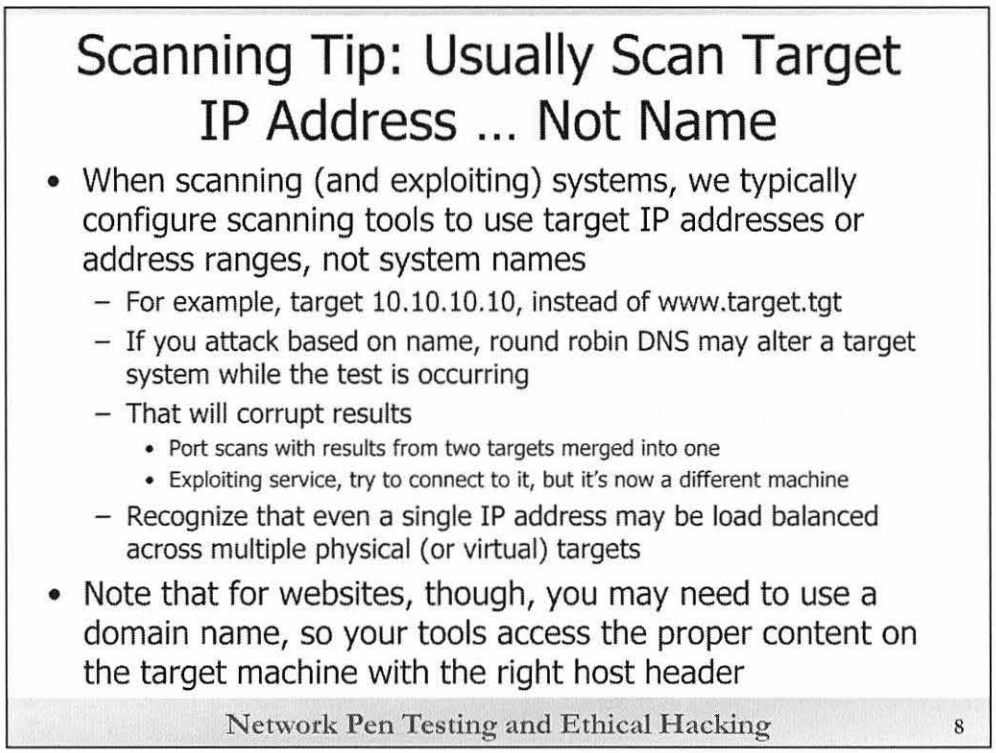


The workflow of a tester during the scanning phase generally progresses through the different kinds of scans indicated on this slide. We start with network sweeps to identify potential targets and the addresses they use. We then try to discern the network architecture to see how these targets are connected together. Next, we move 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 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 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 actually happening on target systems. Let's go over some of those tips now.



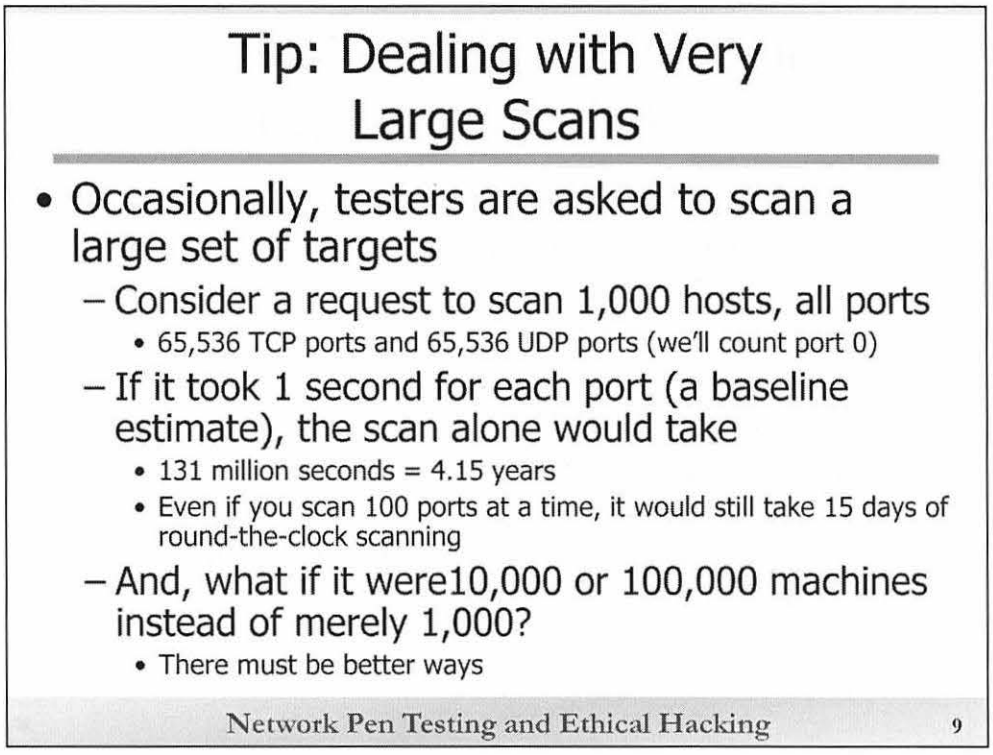
When conducting scanning or exploitation of a target system, we recommend that you indicate the target network or machine in your tools based on its destination IP address and not its domain name. For example, if you want to conduct a port scan or launch an exploit against a machine called www.target.tgt 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. You may want to ask target organization personnel whether this is the case. A crystal box test regimen can help identify such issues.

Finally, for some specific types of tests, especially web application penetration tests, you need to use domain names for your targets so that your tools access the proper content on target sites. With a shared web server hosting multiple target applications, using an IP address as your target causes your web scanning tools to omit the proper host header to access the right target content. Thus, in the situation of web applications and web server tests, you may want to utilize a domain name.

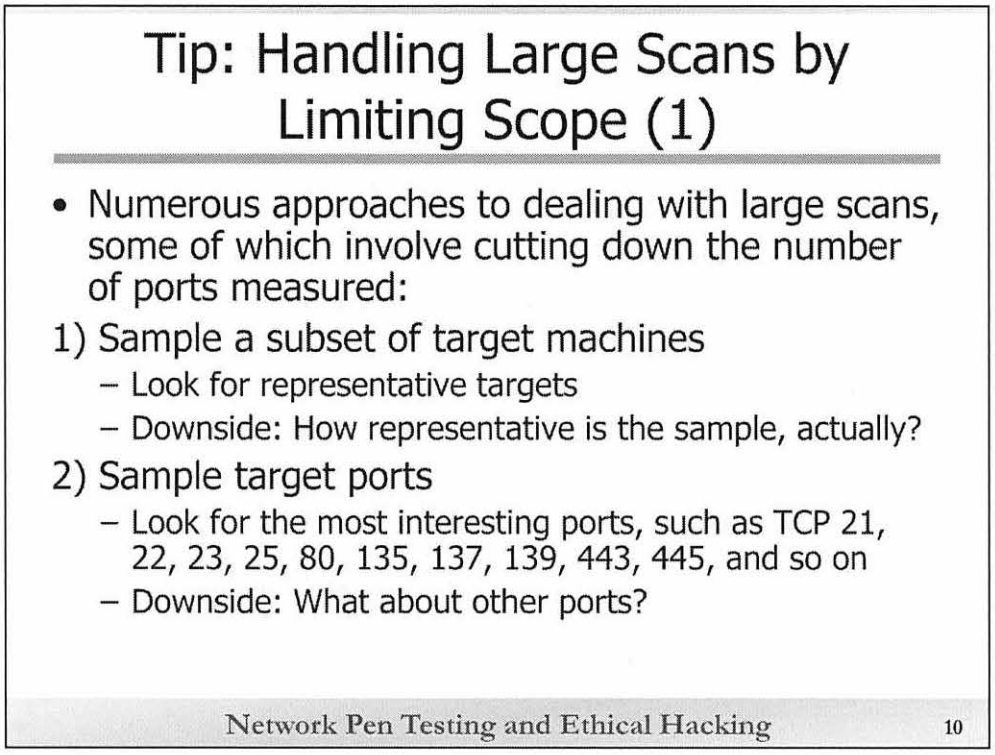


Occasionally, penetration testers and ethical hackers are asked to conduct comprehensive scans of large environments. An expansive scope could mean a huge, almost impossibly large, amount of work, and the numbers grow quicker than many people assume.

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 its 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 approximately131 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 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.

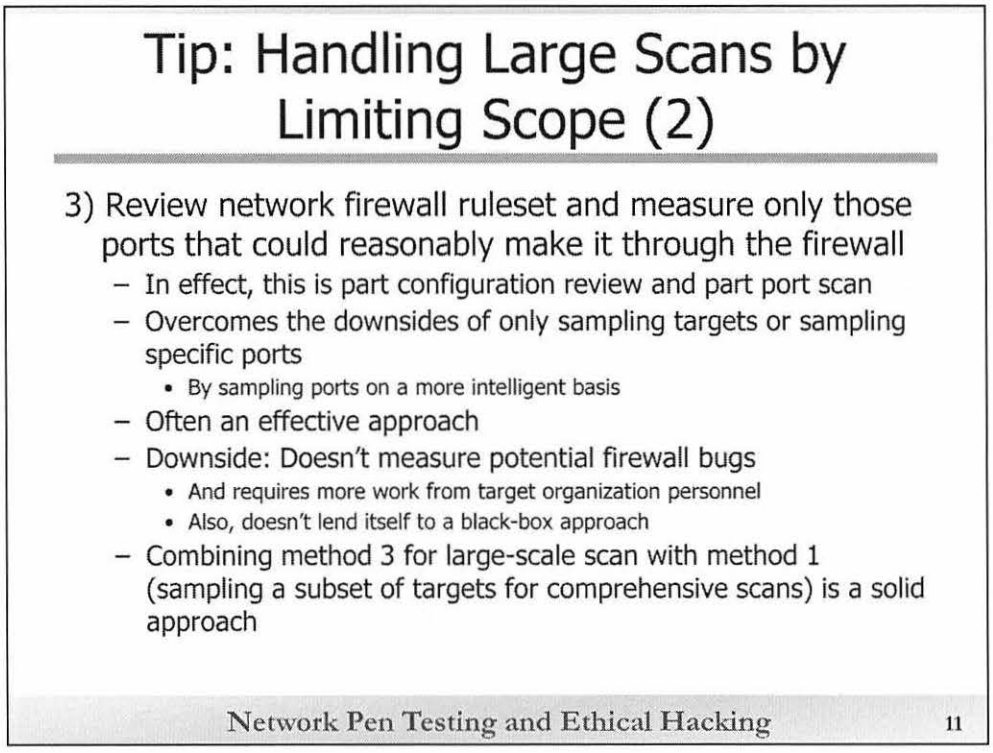
Clearly, there must be a better way.



We actually have many different approaches to dealing with requests for large scans. The specific approach chosen for a given test is 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:

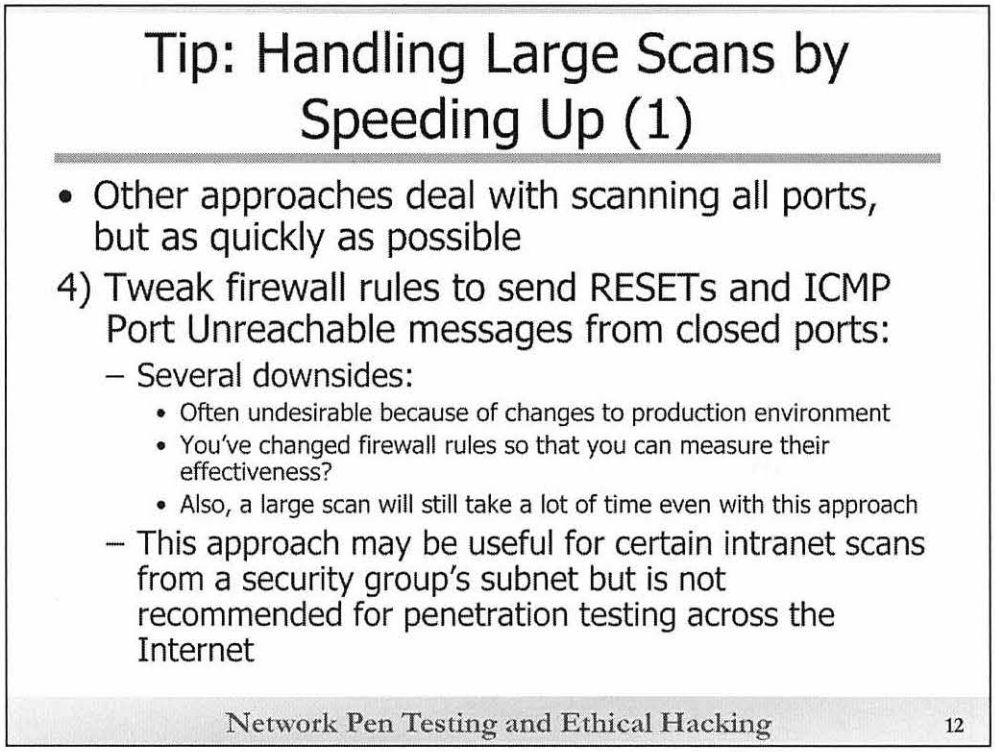
- **Sample a subset of target machines**: Instead of scanning the entire target environment, some organizations are comfortable narrowing the scope by selecting a representative sample of machines in the target environment. For example, instead of scanning all desktop machines, the testers could choose 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.
- **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 measures only a set of ports, leaving the organization unaware of the status of other ports.



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:

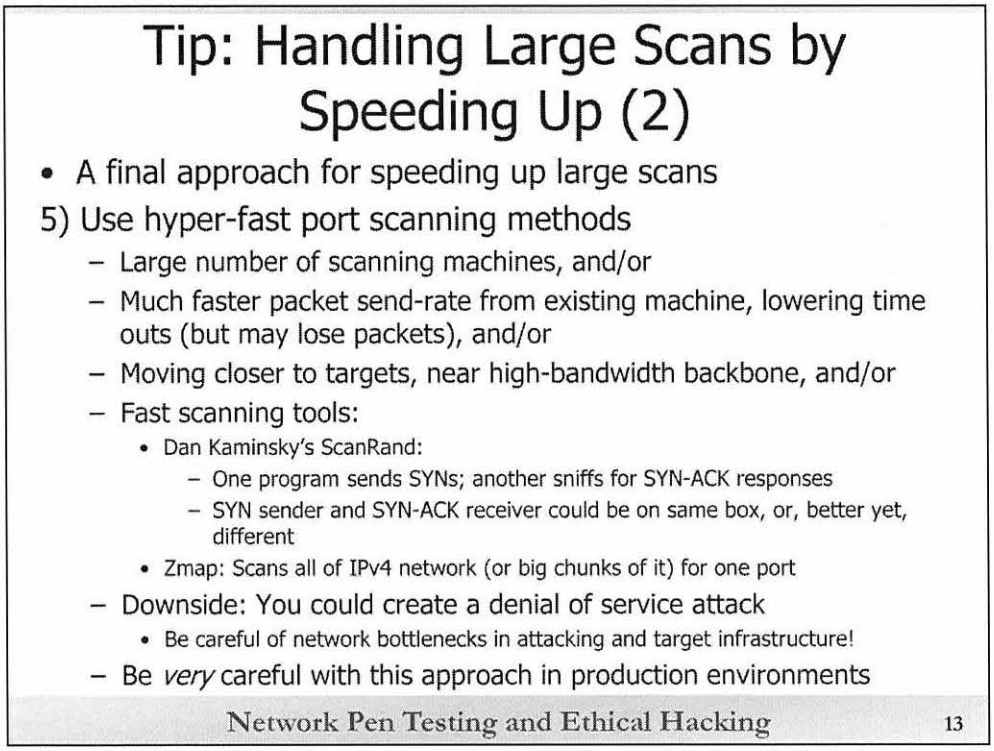
- **Review network firewall ruleset**: Target organization personnel could provide the testing team with a set of network firewall configuration rules. The testing team could then perform the scan on only those ports that would be allowed through the firewall ruleset. In effect, this approach bundles a focused configuration review with a scan to help make the scan more efficient. Although 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 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 in which 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.



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

- **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 prevent most scanning tools from waiting for a timeout 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. Although this technique can be helpful (for intranet scans from the security team's subnet), it has some 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 speeds up scans, it will still take time to measure each port. That time quickly adds up, and the scan still likely has a long duration.



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

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

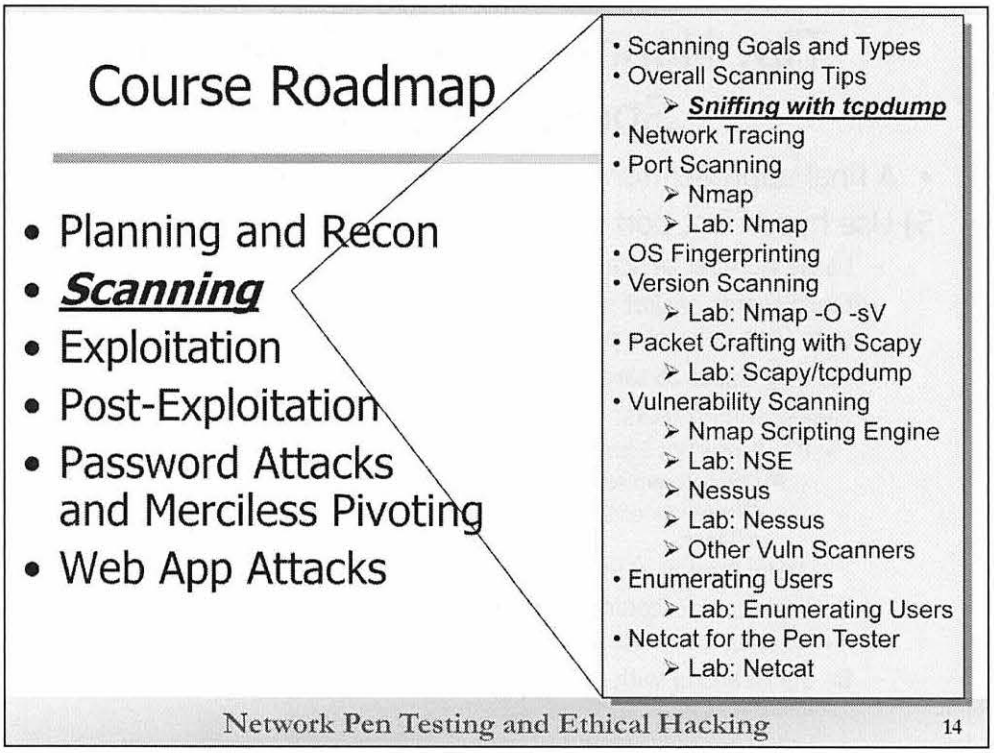
First, the attacker could use a large number of scanning machines. Instead of one or two, the tester could rely on 10, 20, or more machines distributed at various locations to conduct the scan.

Secondly, the tester could configure machines to send packets quicker 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 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. ScanRand 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. To improve performance, the SYN sender and SYN-ACK receiver programs could even be on separate machines, as long as the sender uses a spoofed source address of the receiver machine. The target responds to where it thinks the SYN came from, namely the spoofed address of the receiver. An alternative tool is Zmap, a port scanner designed to scan through all the IPv4 Internet (or major portions of it) to look for systems listening on a single TCP port.

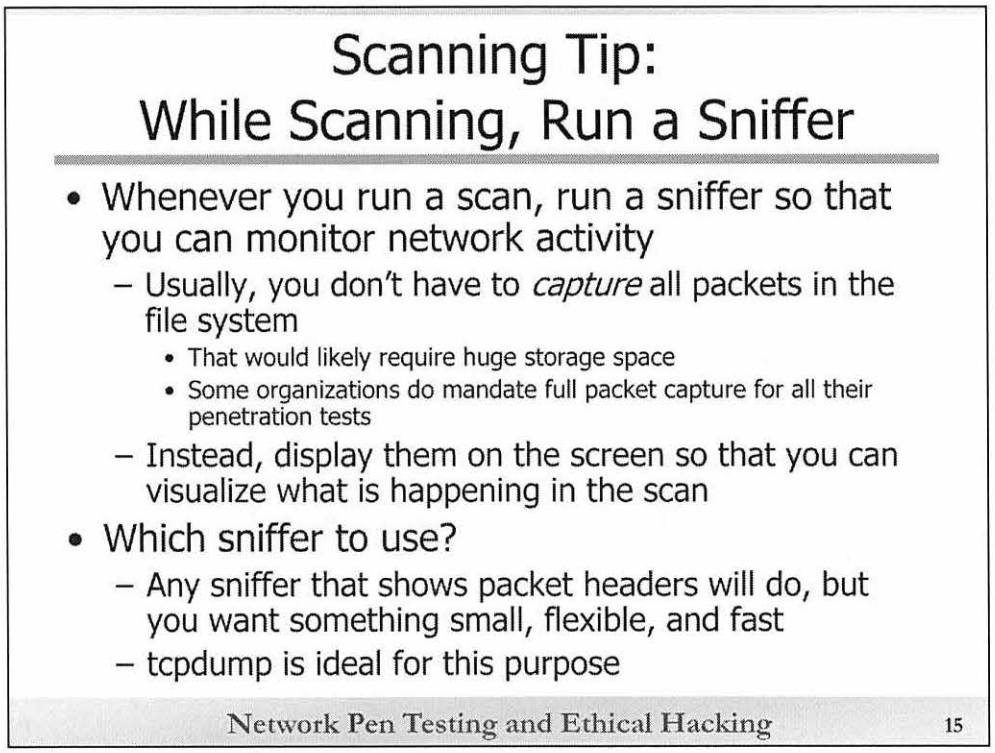
Any of these mechanisms for option 5, however, consume a lot of bandwidth. Thus, testers have to be careful of inadvertently causing a denial of service on the testing network and the target infrastructure. When using these approaches, the testers should carefully measure target systems to ensure that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



Our next set of tips focus on sniffers, specifically on the 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 can reveal
- If the Rules of Engagement allow for it, to sniff useful and interesting information from the target environment, possibly including user IDs and passwords or other sensitive information passing by the machines that the tester has compromised during a project

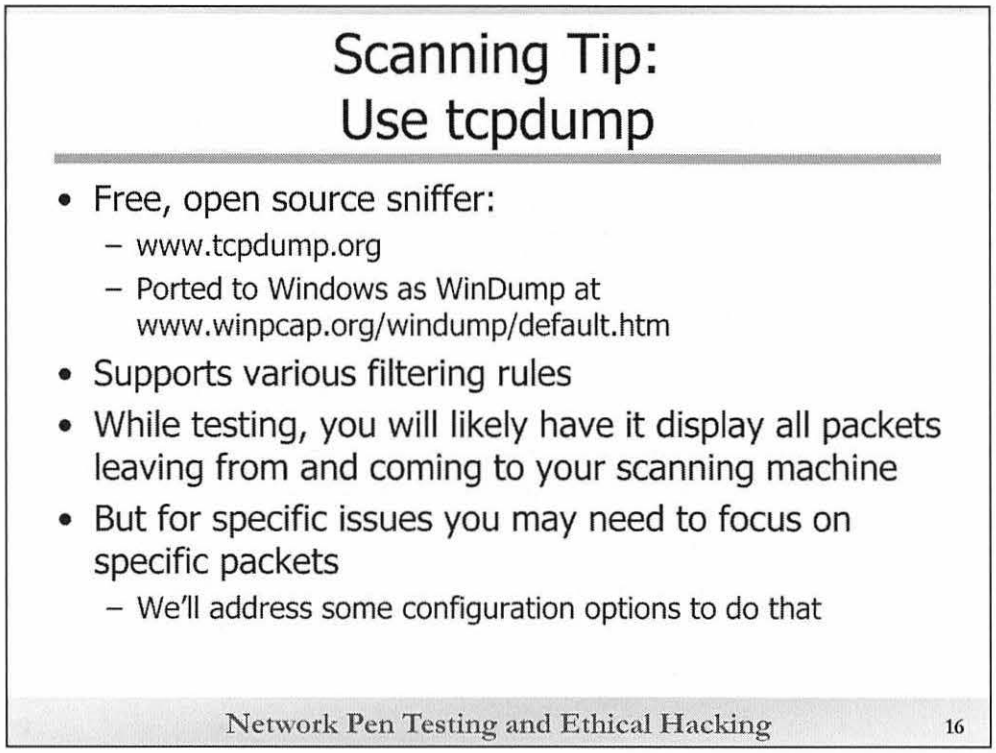
Although sniffers can be useful for all these items, you need to understand how to configure a sniffer so that it focuses on specific packets that interest you during a test. This section provides tcpdump configuration advice specifically targeted at penetration testers and ethical hackers.



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 runs 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 runs, you can verify that the scanning tool functions properly. If the packet display stops, either the tool has finished or encountered some sort of problem.

For most penetration tests, 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. (A few organizations do require their penetration testers to capture all packets, but the practice is relatively rare.) Regardless if you capture full packets to the file system, just displaying the packets on standard output in real time while a scan runs is quite useful.

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



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 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 the filtering options in this class. (They are covered in detail in the 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.

wpad => system in window proxy configuration

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, and so on)
  - -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 captures only first 68 bytes for most OSs
    - For those versions of tcpdump, you had to specify -s 0 to get whole packets
    - On more recent versions of tcpdump, default snaplength of zero grabs entire packets automatically

Most commonly, penetration testers simply run tcpdump without any special options, which by default shows 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.

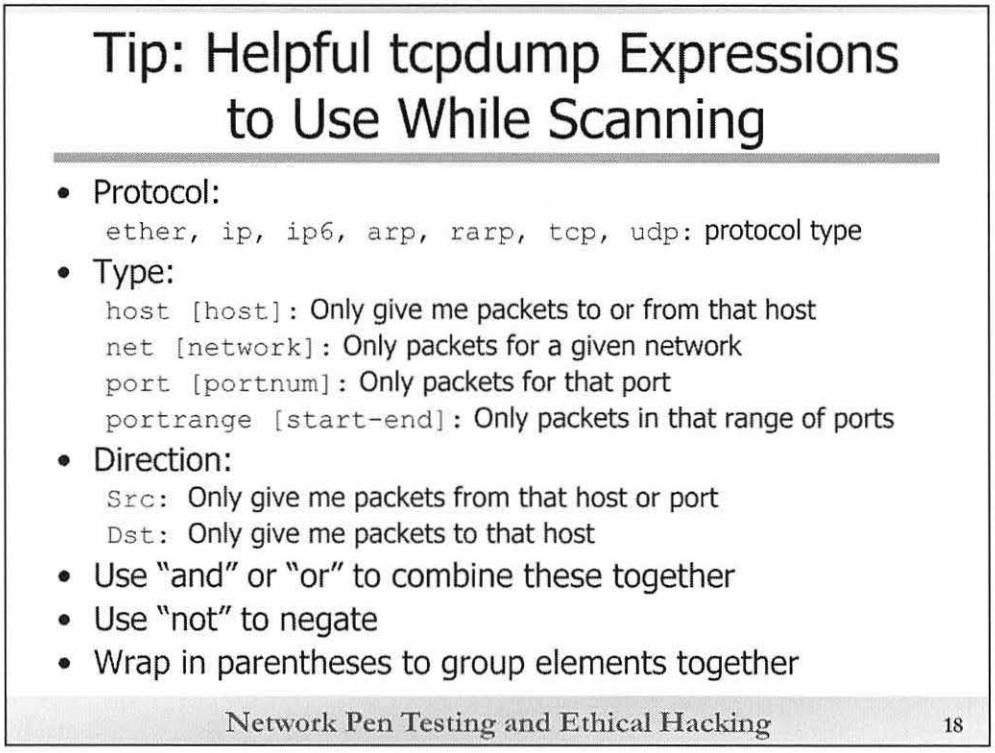
One relatively safe way to invoke a tool with root privileges is to use the sudo command, as follows:

```
$ 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 grab only 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.



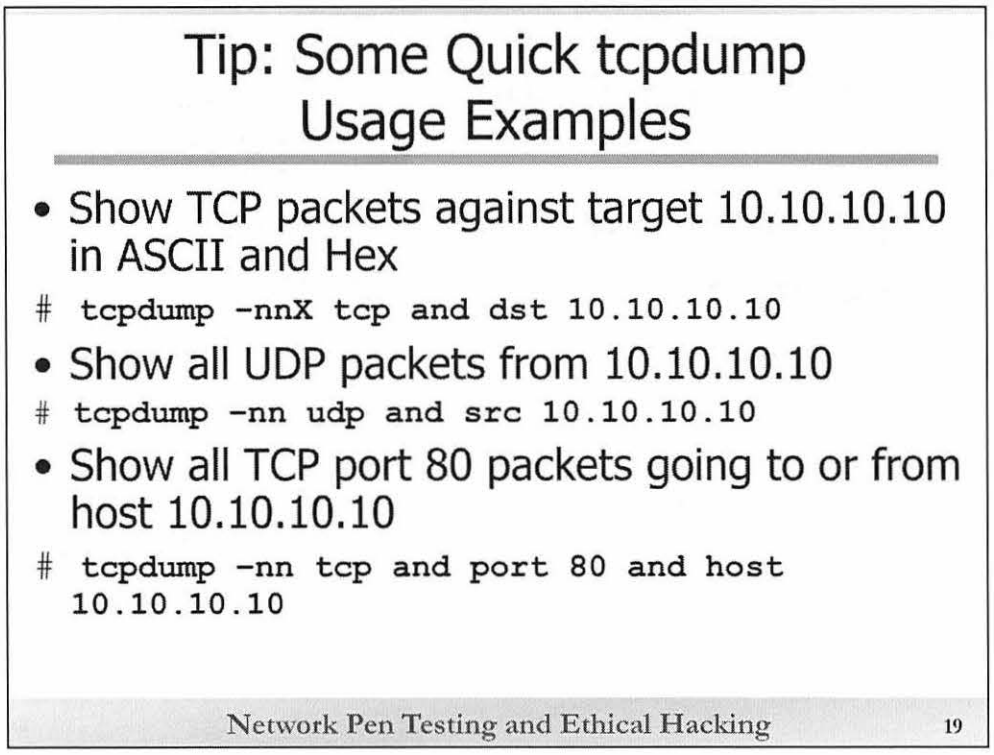
Sometimes, however, you want to run tcpdump to focus on specific packets, such as those associated with certain protocols, ports, or addresses. You can use several primitives to formulate an expression, which enables you to focus only on some specific packets.

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

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

Direction primitives enable 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. You could also add "not" to negate a given element, looking for everything *except* a particular item. 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.



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

If you want to view all TCP packets 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
```

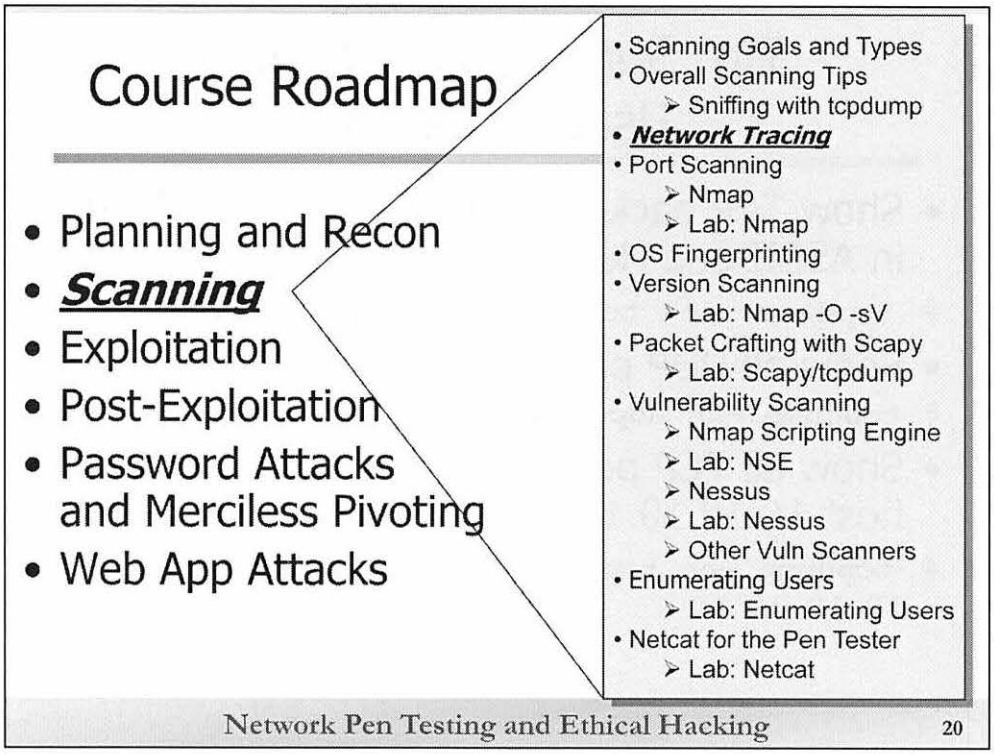
To see UDP packets with a source address of 10.10.10.10, you could run:

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

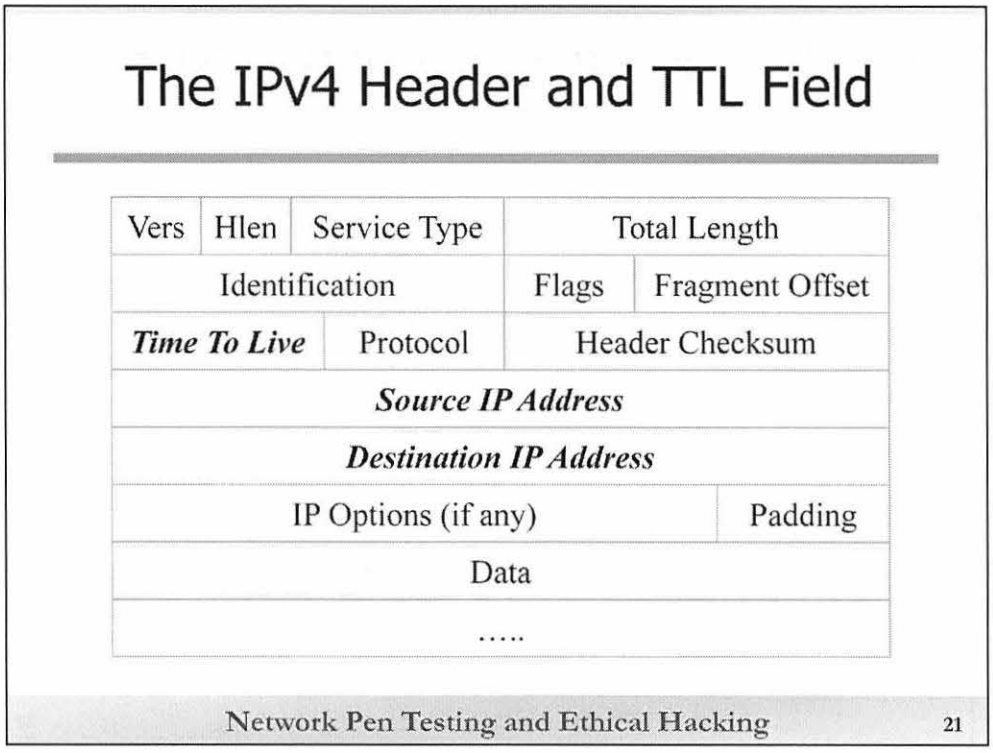
To see all packets associated with TCP port 80 going to or from host 10.10.10.10, you could run:

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

As you perform labs over the next several days, you can formulate tcpdump expressions to focus on the most interesting packets associated with the scan.



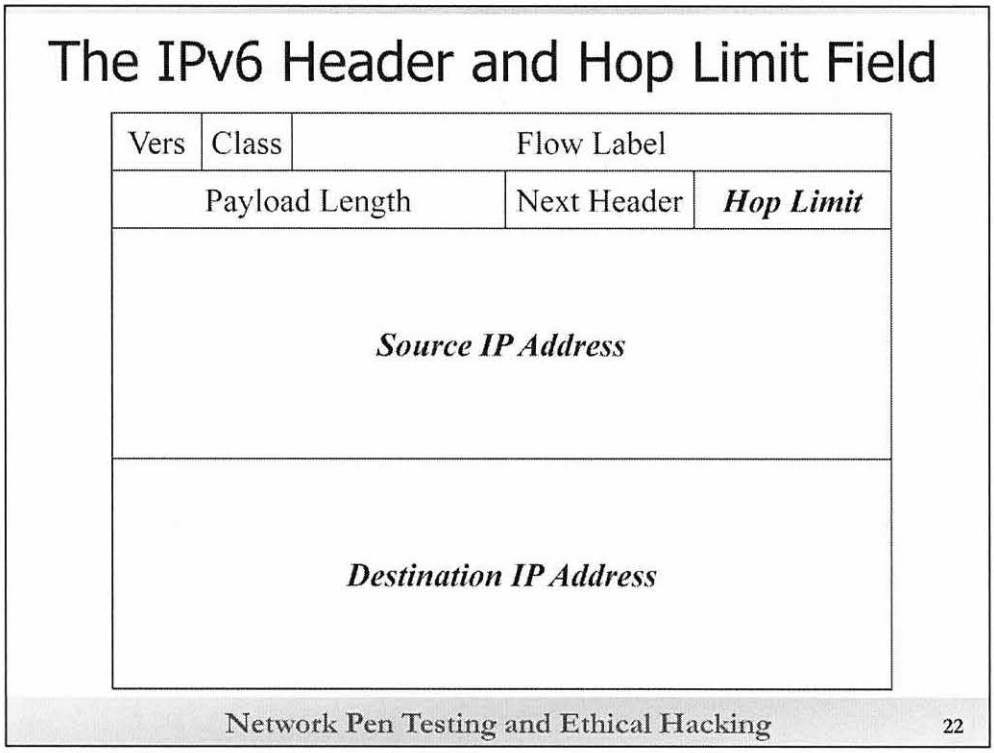
The next topic is network tracing, figuring out the paths that packets take as they traverse the network. These methodologies and tools will be instrumental in composing a network diagram of the target environment.



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 can use to determine the overall network topology. The source IP address is a 32-bit field indicating where the packet originates. This field is usually set to the address of the machine running the scanning tools, unless we use 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 to different destination addresses.

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 1. 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.

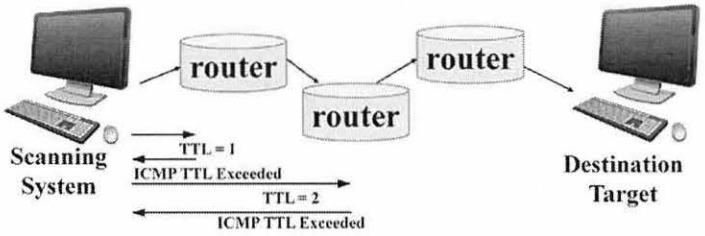


Here is the IPv6 header. First, note the massive size of the source and destination IP addresses, with each 128 bits in length. Further, notice that this packet structure is actually in many ways simpler than IPv4. For example, the fields associated with fragmentation (the IP Identification field, the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

But, most important right now, there is a Hop Limit field, which behaves in a 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



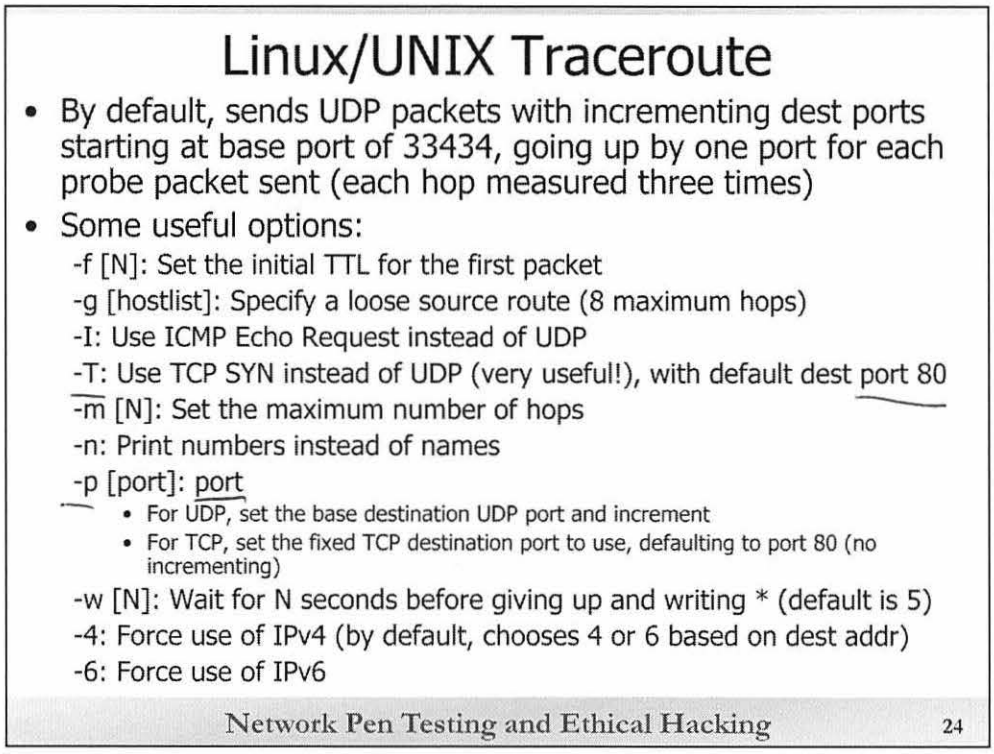
Network Pen Testing and Ethical Hacking 23

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 small: a value of 1is 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.

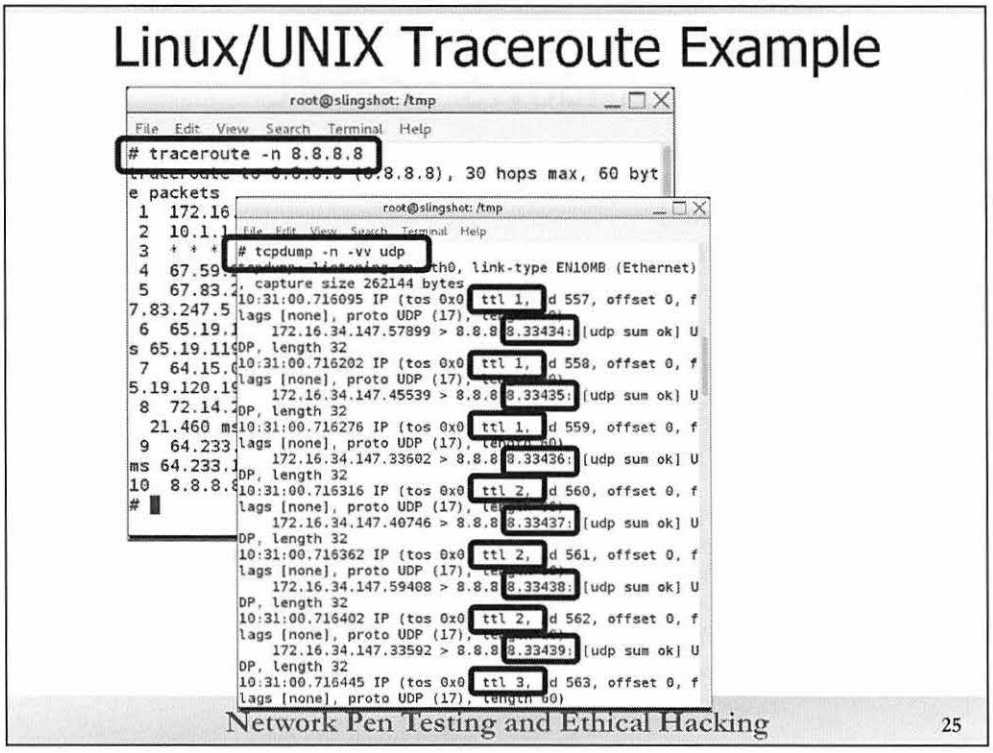
The traceroute tool proceeds in this fashion, measuring hop after hop, until it reaches the target. The target's response depends on the type of packet used by the traceroute tool. If a given hop doesn't return an ICMP TTL Exceeded in Transit message back (because it is configured to filter the inbound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



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's default behavior 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. 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 eight router hops.
- **-I**: Use ICMP Echo Request messages as probes instead of UDP packets.
- **-T**: Use TCP instead of UDP for the probe packets (with a fixed destination default TCP port of 80).
- **-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 destination port for probes. For UDP, this sets the base (that is, starting) UDP port, which subsequent packets will increment, instead of the default of 33434. For TCP, this sets a fixed (that is, non-incrementing) destination port. If no port is specified for TCP tracerouting, the default of 80 is used.
- **-w [N]**: Wait for an ICMP response for up to N seconds. (The default is 5 seconds.)
- **-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.

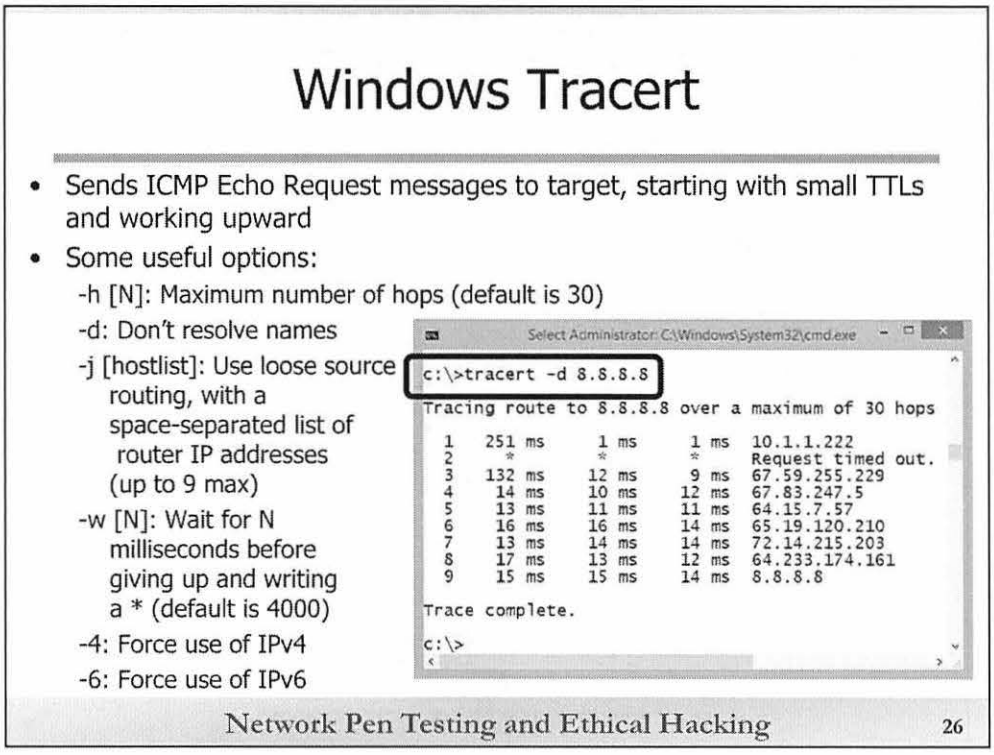


In this example of a Linux traceroute, we first started the tcpdump sniffer to verbosely (-v) use numbers instead of names (-nn) while printing out UDP packets (udp). We ran it verbosely so that tcpdump shows us the TTLs of packets.

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 to 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 or –T to send TCP packets to destination port 80.



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 nine 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, whereas 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.

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.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 this target

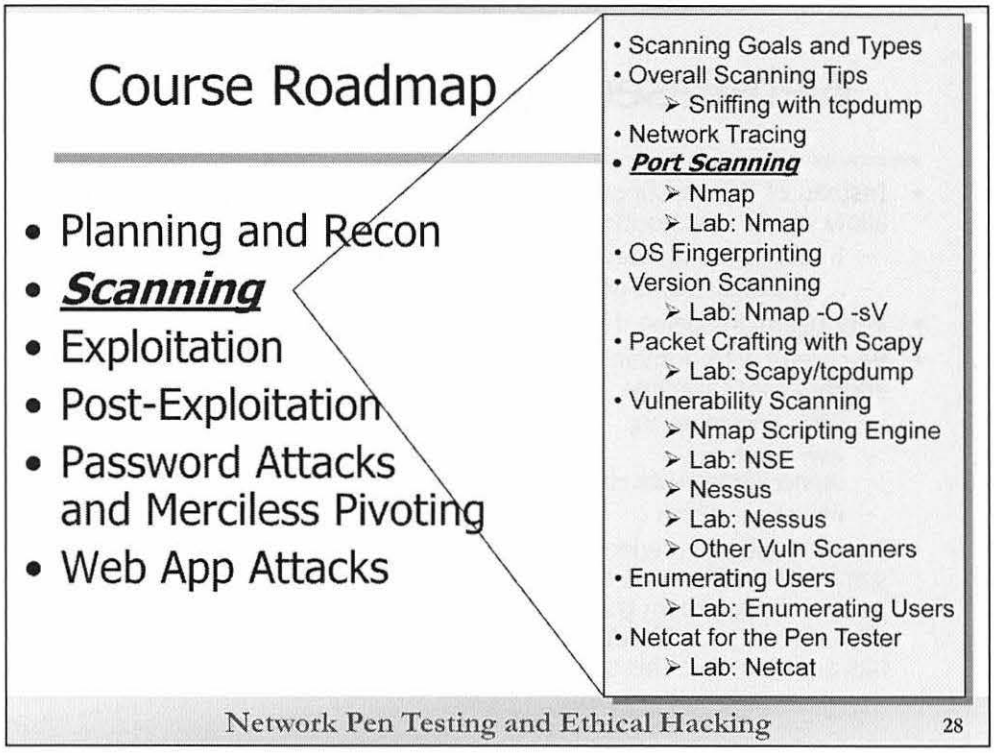
Network Pen Testing and Ethical Hacking 27

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.

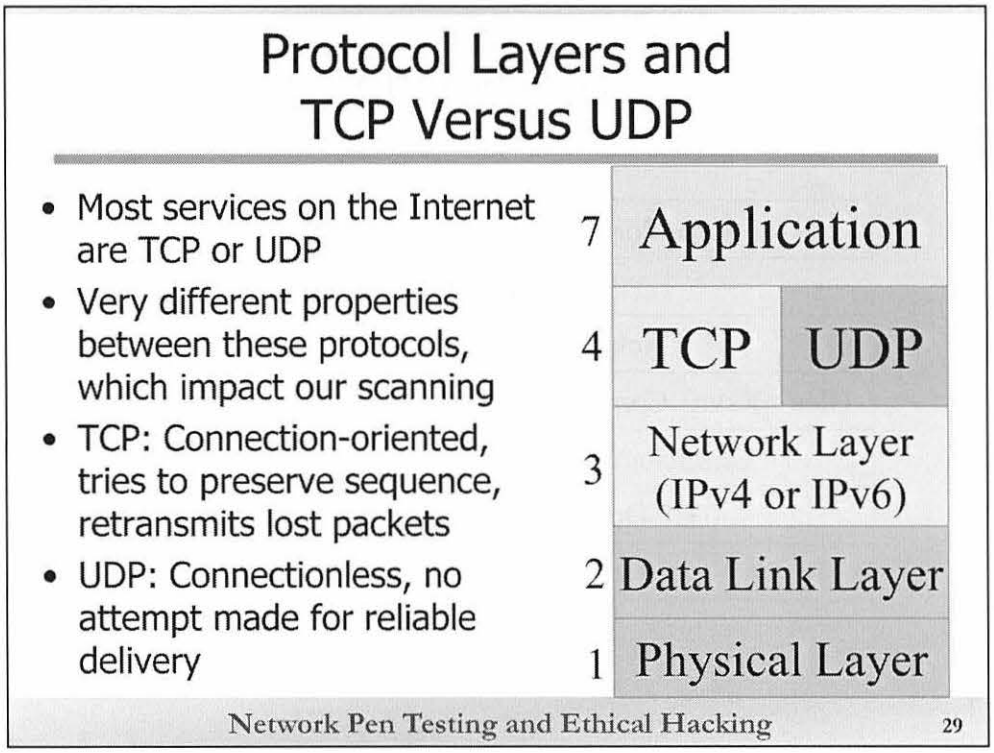
In this way, a tester can see what a traceroute against a target will look like from other parts of the world. These services are also helpful in determining if there is a localized outage or blockage on the network, or if the target system itself has gone down. For penetration testers and ethical hackers, 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. 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 sensitive information, but other kinds of external lookups should be considered carefully so that you can avoid violating your non-disclosure agreements.



The 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.



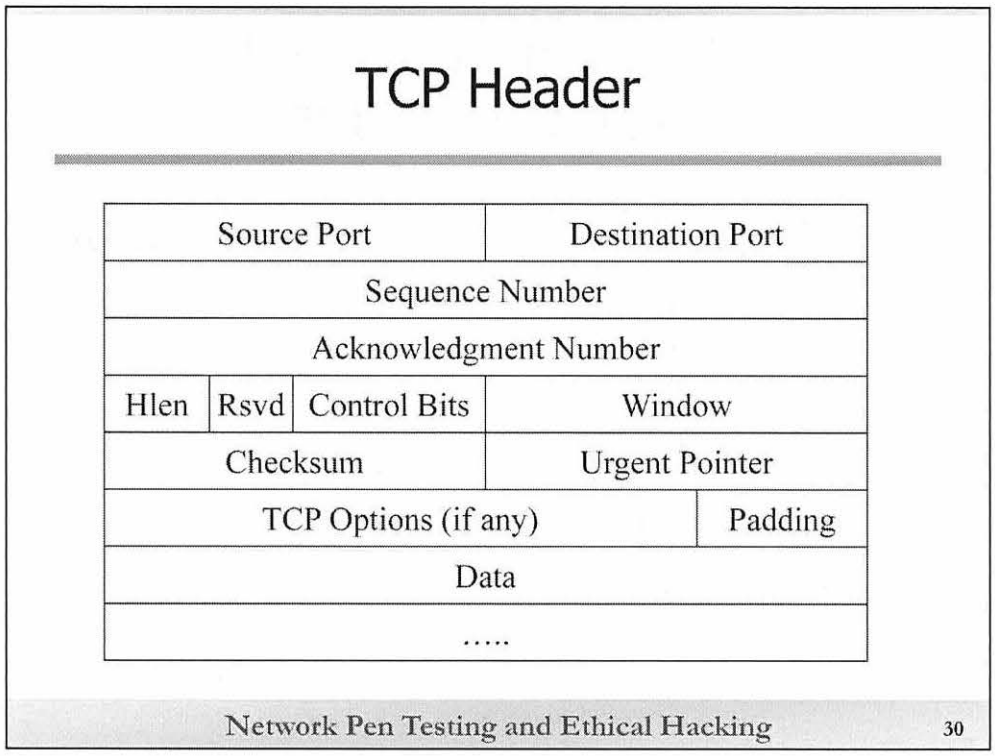
To understand port scanning, we first need to discuss some protocol issues. Layer 7, the application layer, formulates data to send across the Internet, and hands it to Layer 4, the transport layer (typically TCP or UDP). The transport layer hands data down to Layer 3, the network layer (typically IPv4 or IPv6) to carry the packet end-to-end across the Internet. The network layer gives the packet to the data link layer, Layer 2, to carry it across a single hop. And, finally, the packet is given to Layer 1, the physical layer, which is made up of the physical medium itself and the electronics that control it.

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 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 resequences them before handing them up to an application.

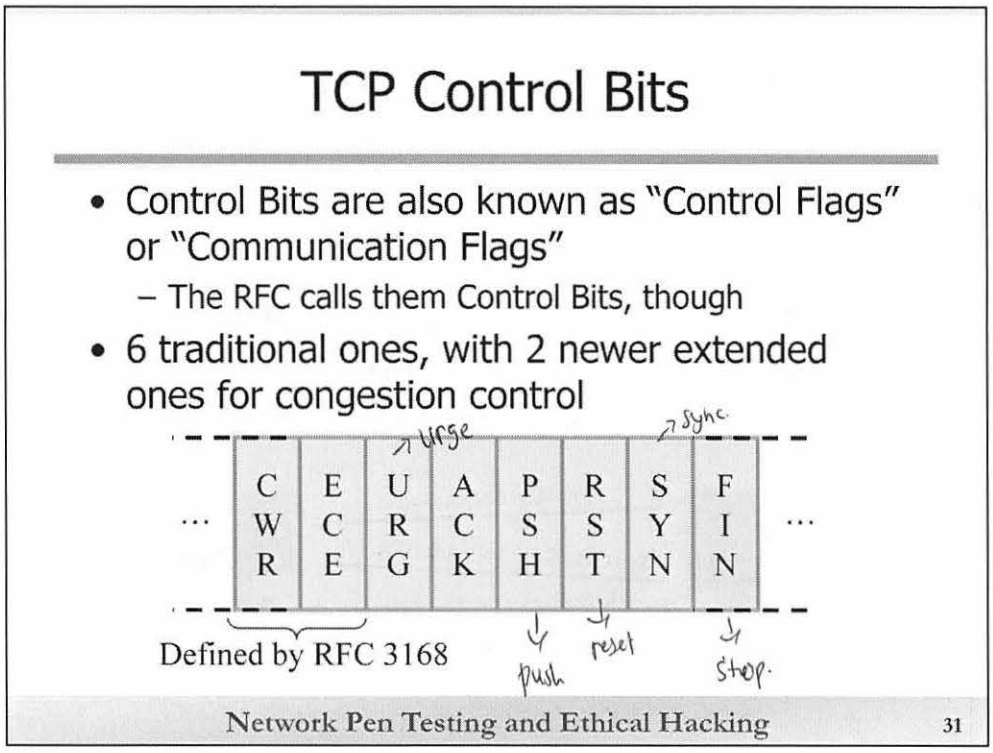
UDP is connectionless. The UDP software makes no attempt to associate streams of packets together. As far as UDP is concerned, each packet is completely independent, unrelated to other packets. No attempt is made by UDP for retransmission or resequencing. If a packet it lost via UDP, it's up to the higher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



Here is the TCP header. Note that it includes a source port and destination port; each is 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.



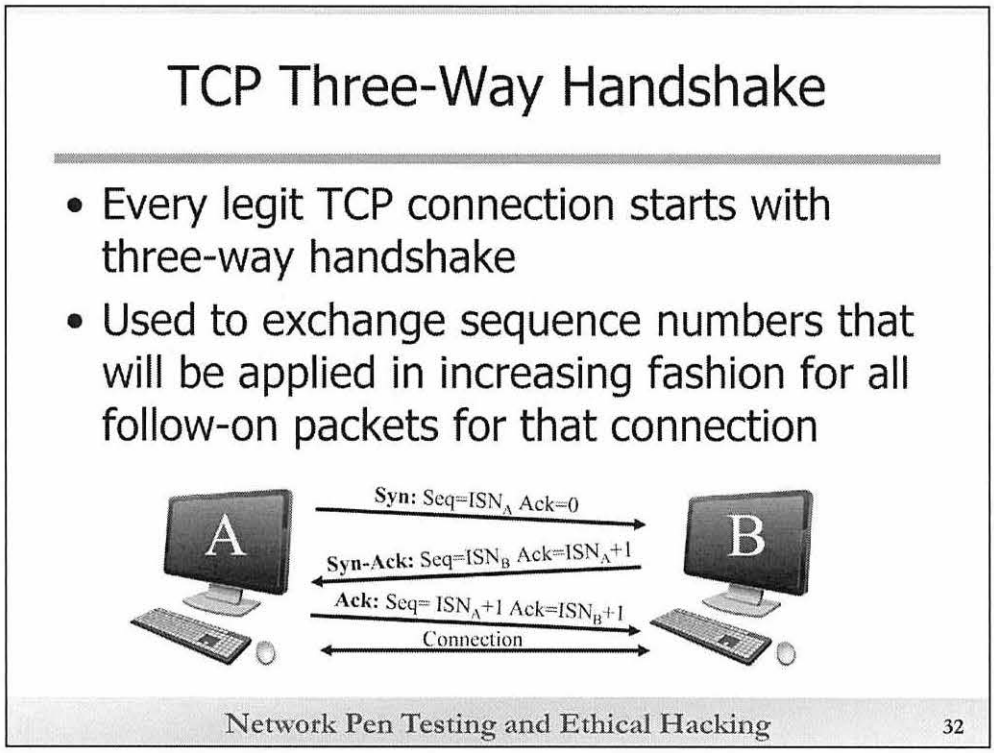
The TCP Control Bits are sometimes called the Control Flags or Communication Flags, but the RFC refers to them as the Control Bits. These bits in the TCP header help identify the state of the TCP connection and which components of the TCP connection the given packet is associated with. There are six traditional TCP Control Bits, with two 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 1-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.



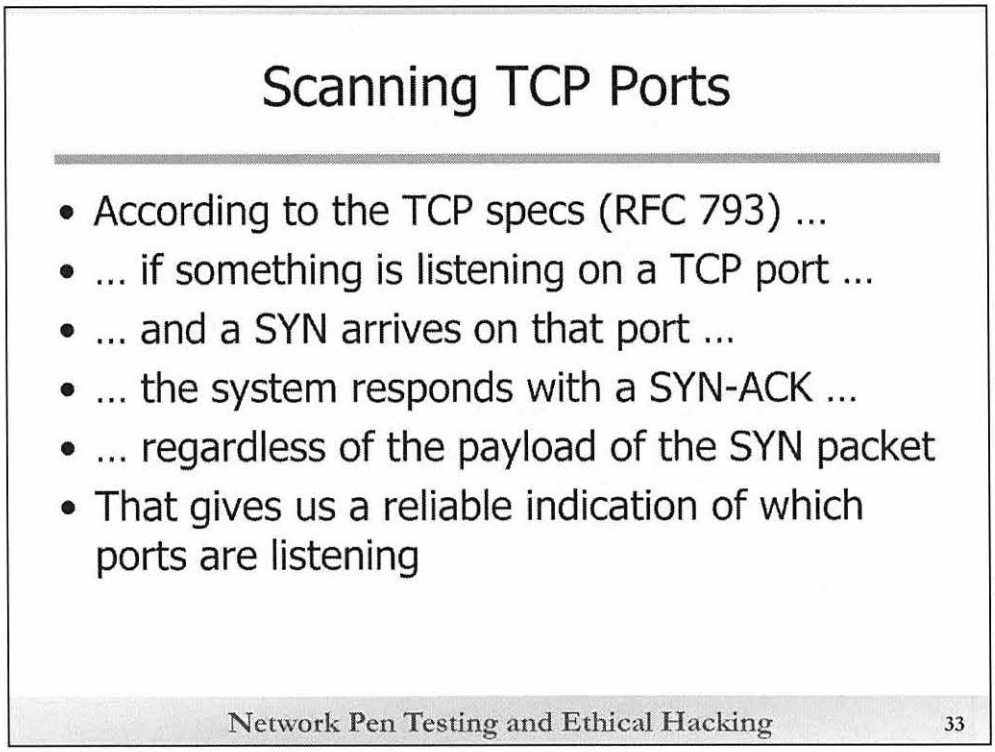
Every legitimate TCP connection begins with the TCP three-way handshake, which is used to exchange sequence numbers so that lost packets can be retransmitted and packets can be placed in the proper order.

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 nonzero. 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.

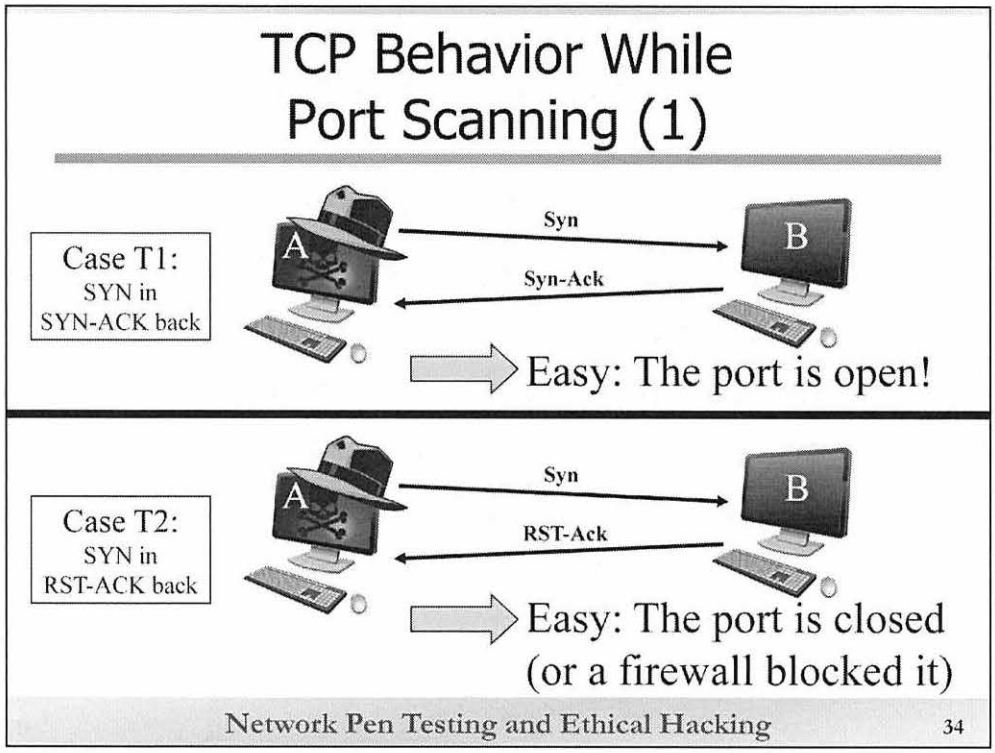
To complete the three-way handshake, machine A responds with an ACK packet, which has a sequence number of ISNA+1. (It's the next packet, so the sequence number has to change from the value in the original SYN packet.) The acknowledgment number field is set to ISNB+1, thereby acknowledging the SYN-ACK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



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.

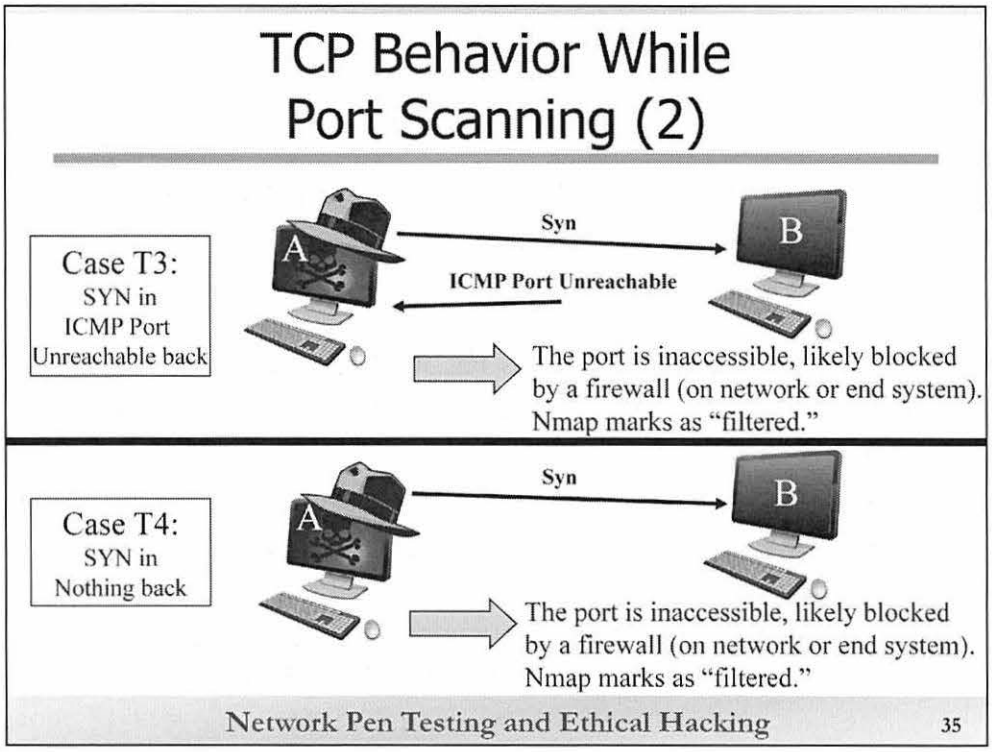
Thus, even if we don't know what service is listening on the target port, we can still measure whether it is open by simply sending it a SYN packet. That gives us a reliable method for determining whether a TCP port is open or closed.



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 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 likely open. There is a 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 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.

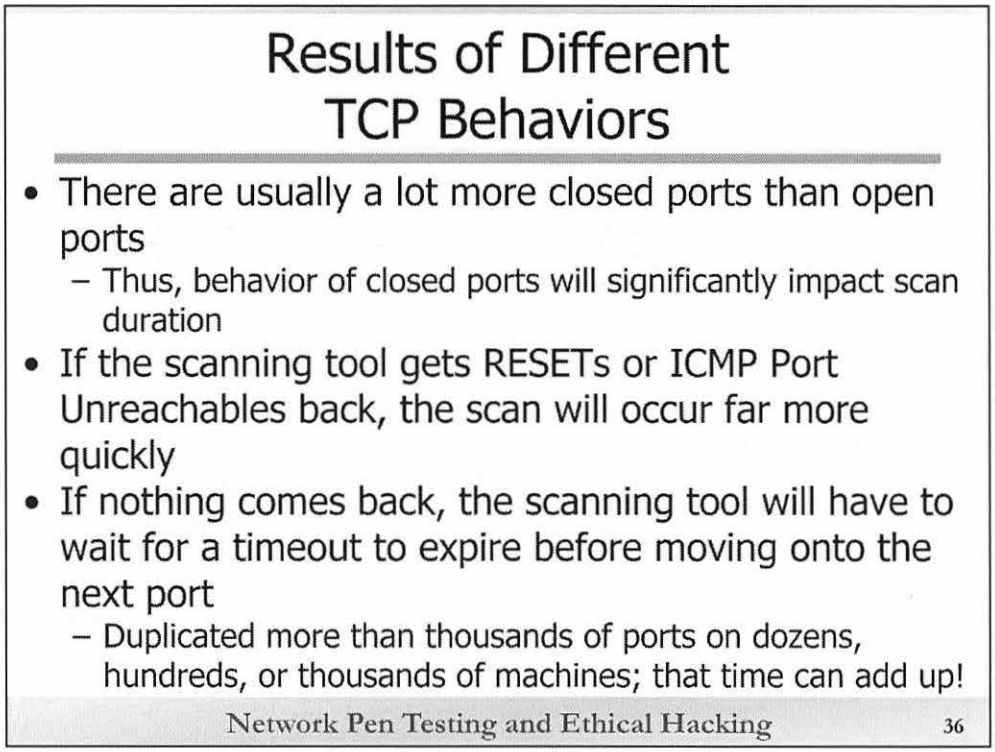
As a penetration tester or ethical hacker, we like to see packets with the RST Control Bit set to 1 coming back from closed ports during our scan because they make the scanning process significantly faster. Rather than having to wait for a timeout before we can move on to another port, we know quickly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

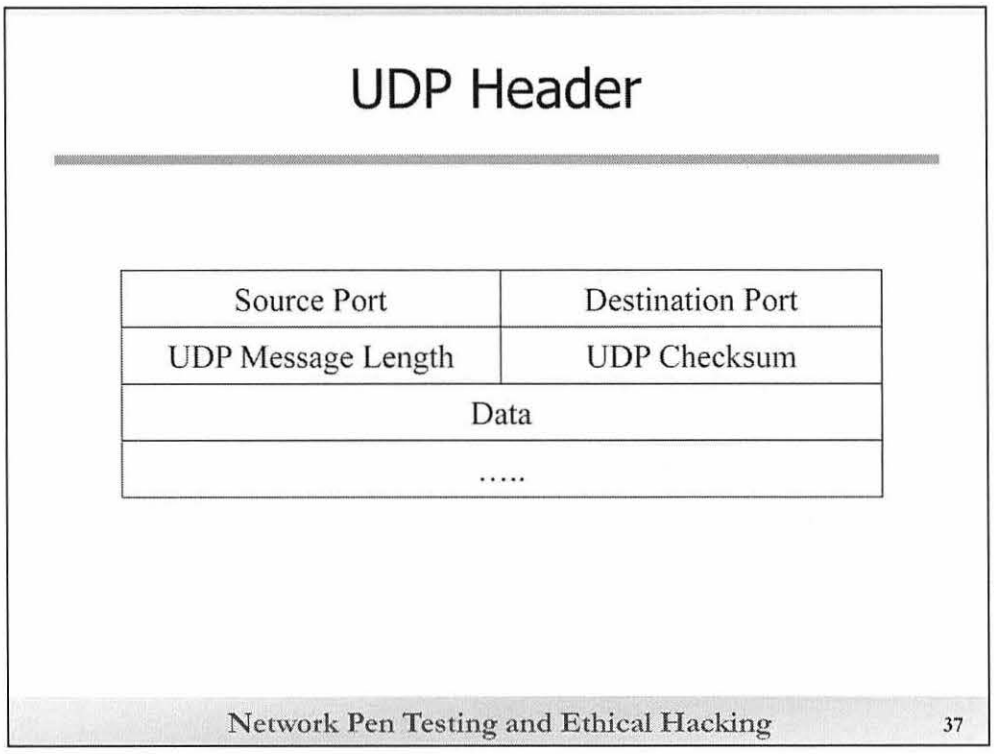
Each of these four cases is summarized well in the Nmap man page, which states

"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."



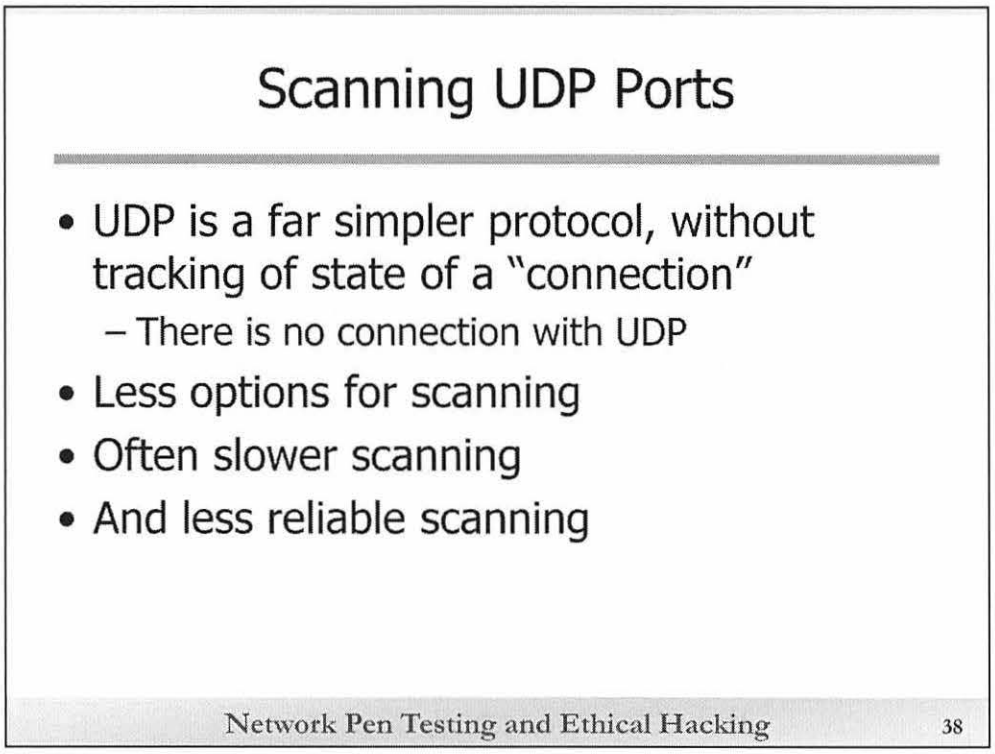
When doing a port scan, you usually find far more closed ports than you do open ports. There are 65,536 possible TCP ports, and most systems have only a handful of ports open. Therefore, from a timing perspective, the behavior of the tens of thousands of closed ports could seriously slow down a scan. If the target sends back RESETs or ICMP Port Unreachables, our scan can occur quicker because 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.



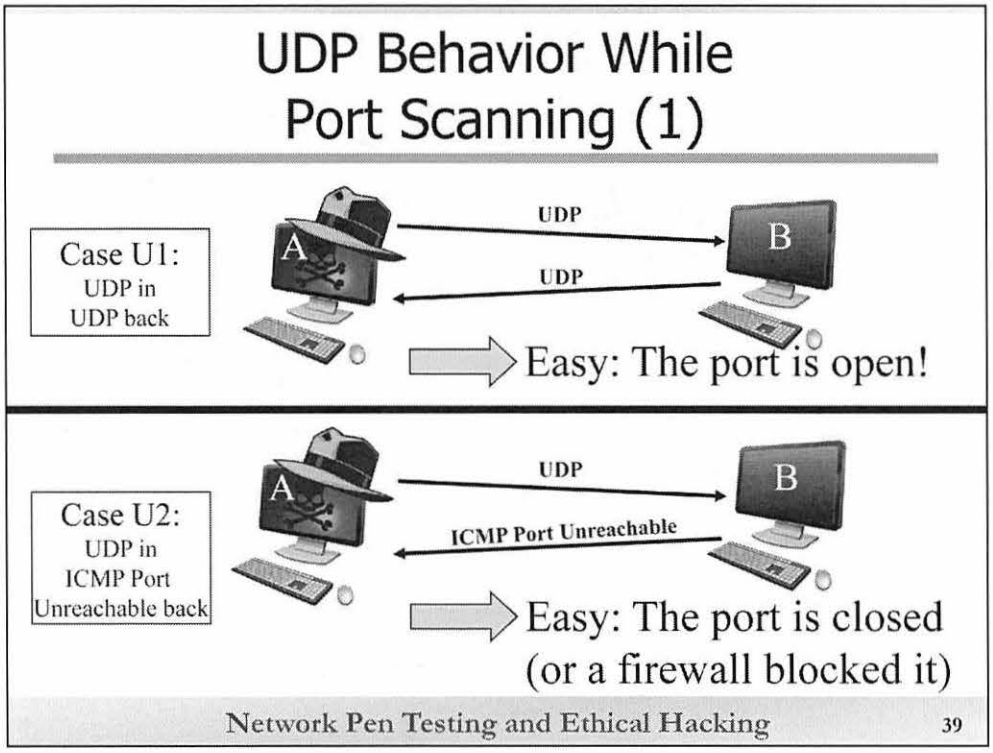
Here is the UDP header. Note its relative simplicity when compared to the TCP header. We have a source port and destination port (each 16 bits in length, giving us potential values of between 0 and 65,535). We also have a message length and a checksum.

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.



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.



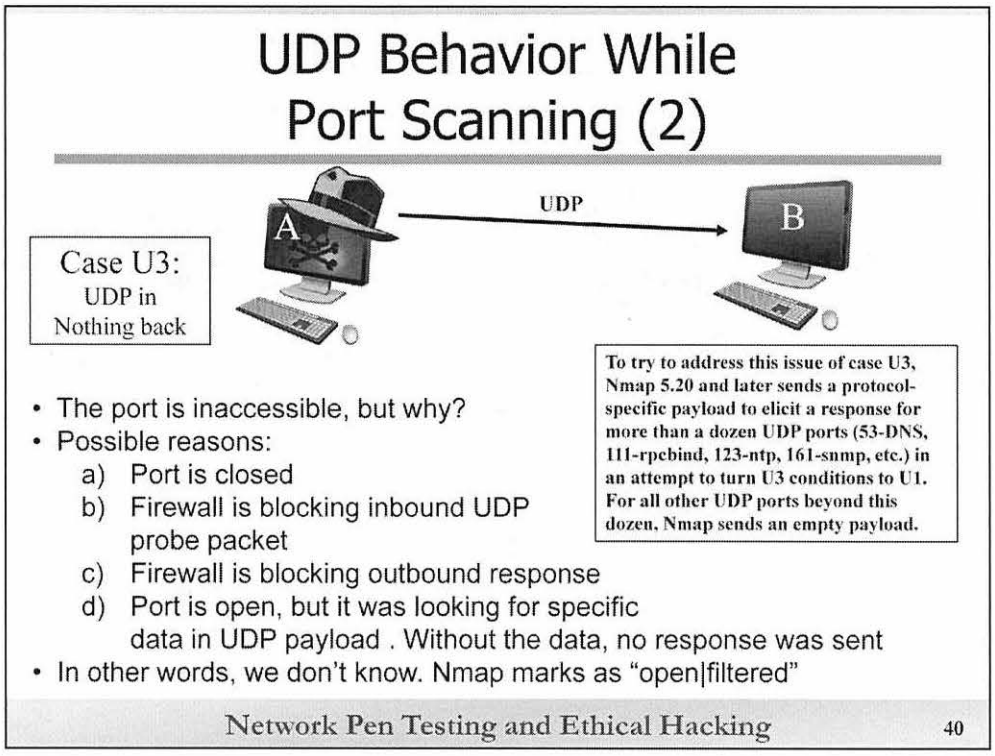
To see why UDP scanning is less reliable and often slower than TCP scanning, consider the cases that could occur when we perform a UDP port scan.

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 send only 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."

TFTP
Voice protocol.
DNS
NTP
SNMP



Now we get to the hard case.

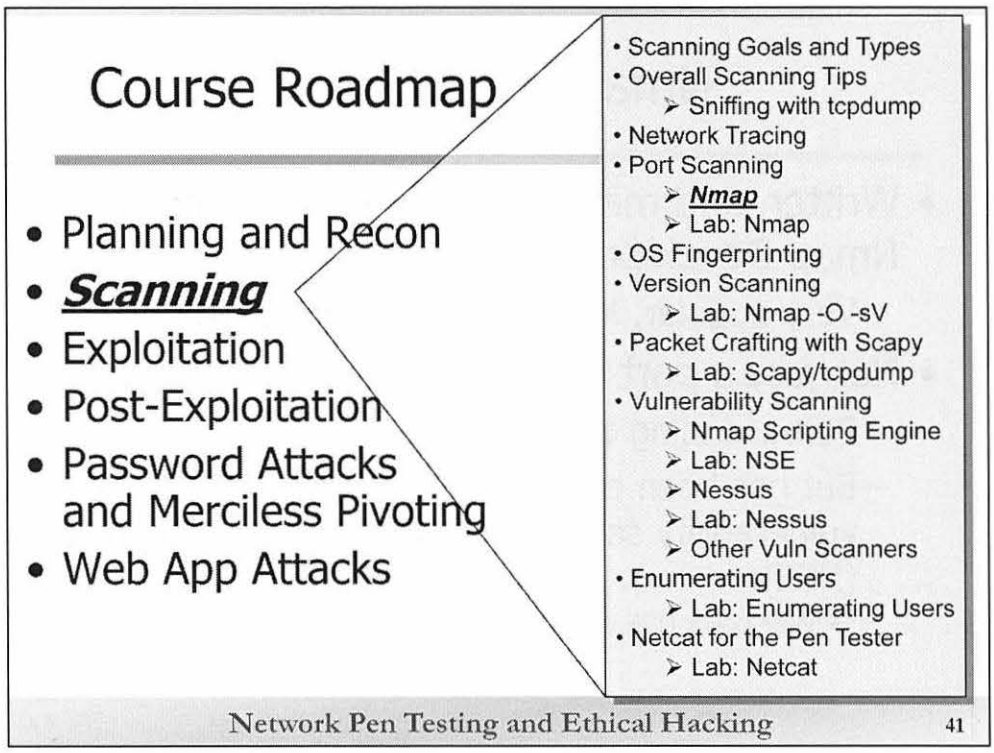
Case U3: We send in a UDP packet, and we get nothing back. There are numerous possible reasons for this, including:

- 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.

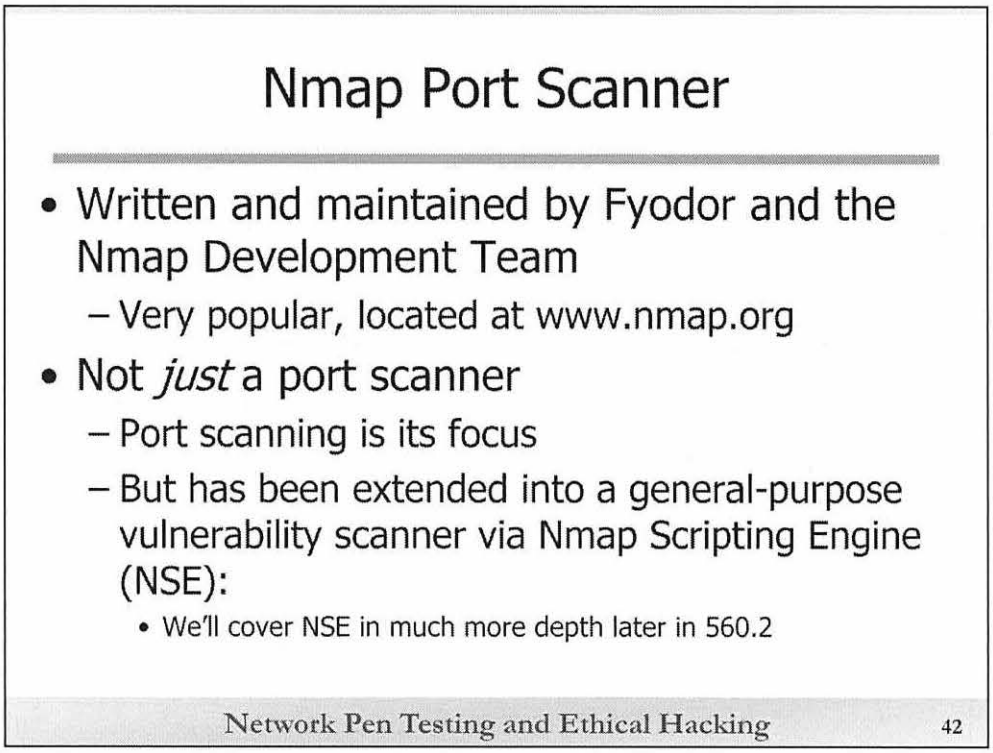
And for that reason, UDP port scanning is less reliable than TCP port scanning. With TCP, according to the protocol spec, if we send a SYN packet to an open port, the target system must respond with a SYN-ACK, regardless of the payload of our SYN. That behavior gives us the assurance that the TCP port 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 issue 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 more than a dozen UDP ports (53-DNS, 111-rpcbind, 123-ntp, 161-snmp, and so on) 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 good feature for identifying UDP-based services using their standard port.



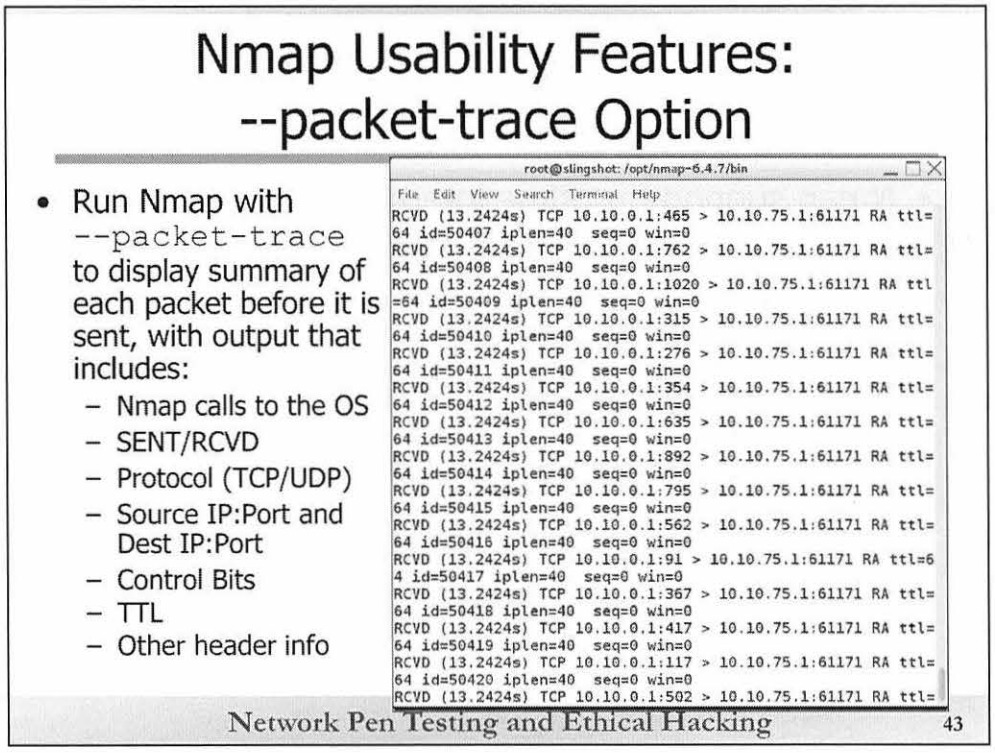
The most popular port scanner in the world is Nmap. Professional penetration testers and ethical hackers around the globe rely on this incredibly flexible and high-quality tool. In this section, we discuss some of the most useful features of Nmap for penetration testers and ethical hackers.

Even if you've run Nmap before, pay special attention to some of the new and more subtle features of Nmap that we address. In the past year, Nmap has been going through rapid changes, 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.



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 a lab that analyzes the capabilities and results of NSE.

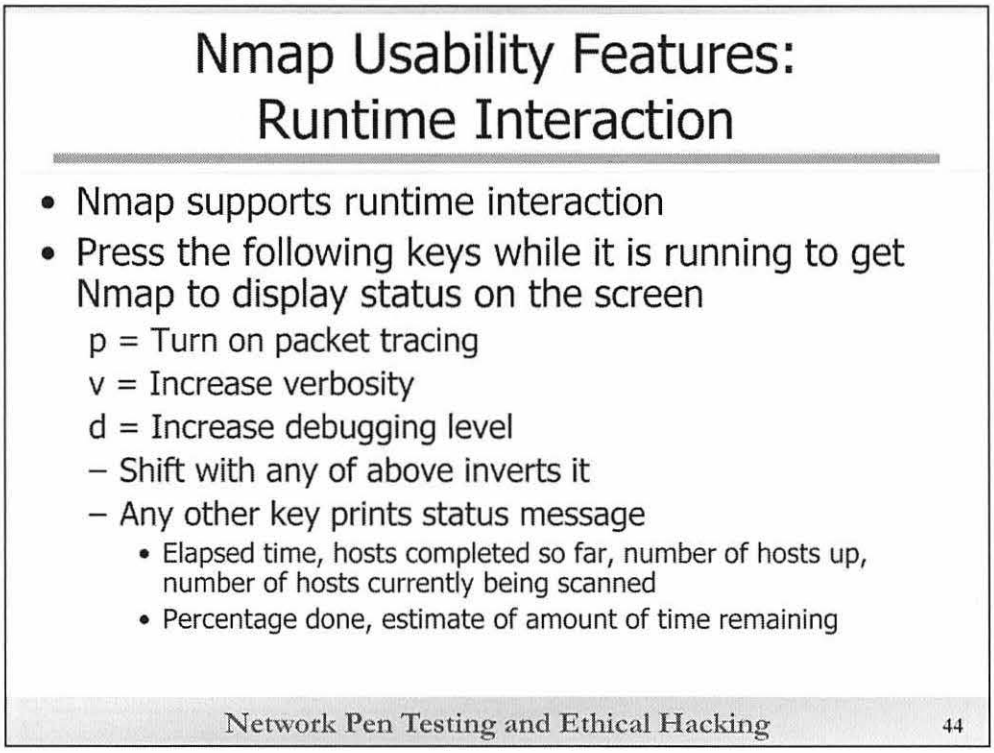


When using Nmap, it can be helpful to have the tool display a summary of the packets that it sends in real time. When invoked with the --packet-trace feature, Nmap does just that. It displays various status messages on its output, including some of the calls it makes into the operating system, such as 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 screen shot in the slide indicates a SYN packet.) It also displays other header information, such as the IP Time-To-Live (TTL), the TCP Sequence number, and so on.

The Nmap command-line sequence that resulted in the screen shot on this page was

```
# ./nmap -Pn -sS 10.10.0.1 -p 1-1024 --packet-trace
```

- **-Pn**: Indicates that we don't want to ping the target system; we just want to scan it.
- **–sS**: Indicates that we want a SYN scan (also known as a Stealth Scan or a Half-Open Scan).
- **–p 1-1024**: Tells Nmap to scan ports 1 through 1024 only.
- **--packet-trace**: Makes Nmap display the status and packet summary information.

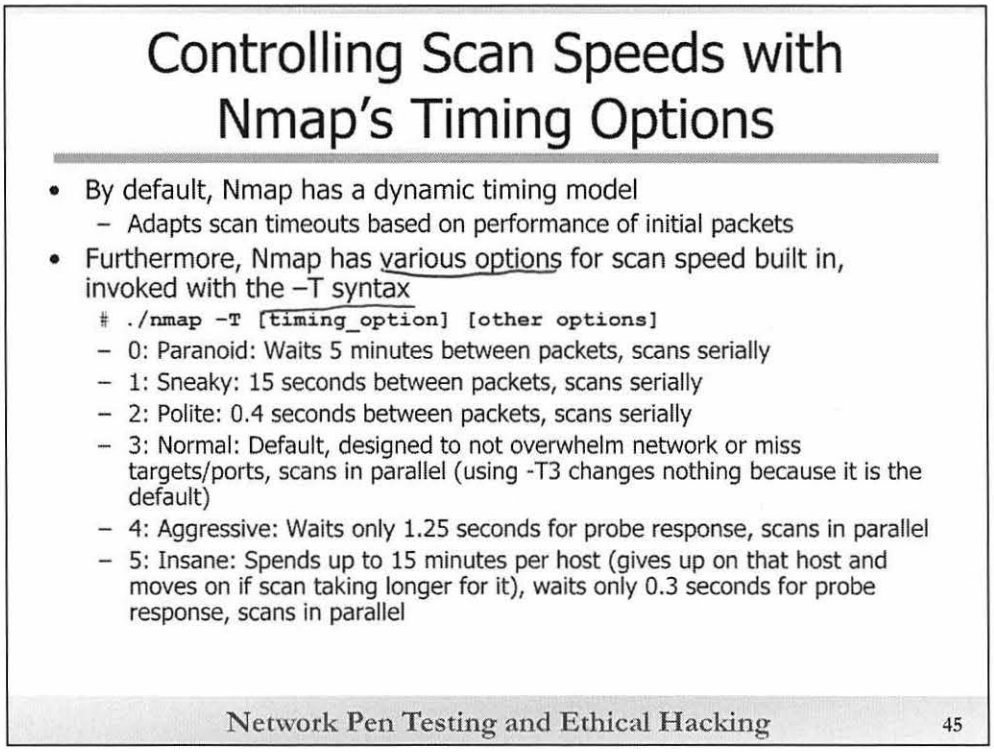


If a user forgets to invoke Nmap with the --packet-trace option, they can turn it on after invoking Nmap. While Nmap is running, the user can press the p key to turn on packet tracing. Nmap will print on the screen a summary of each packet sent.

Furthermore, the user can press 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.

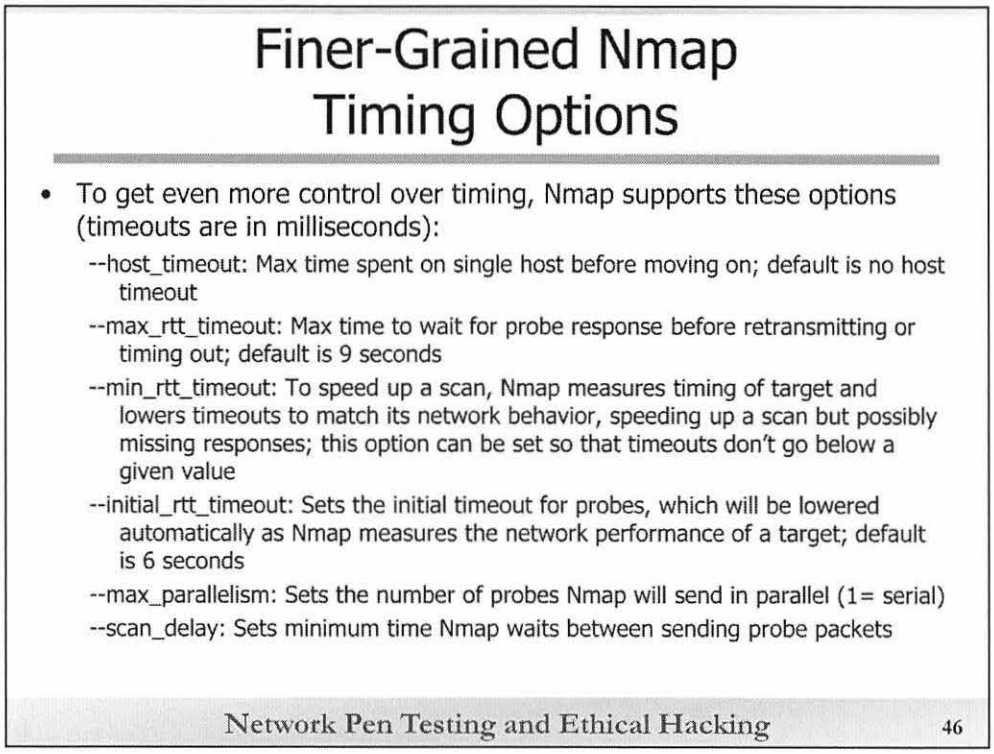
In addition, the v and d keys increase the verbosity and debug modes, respectively.

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



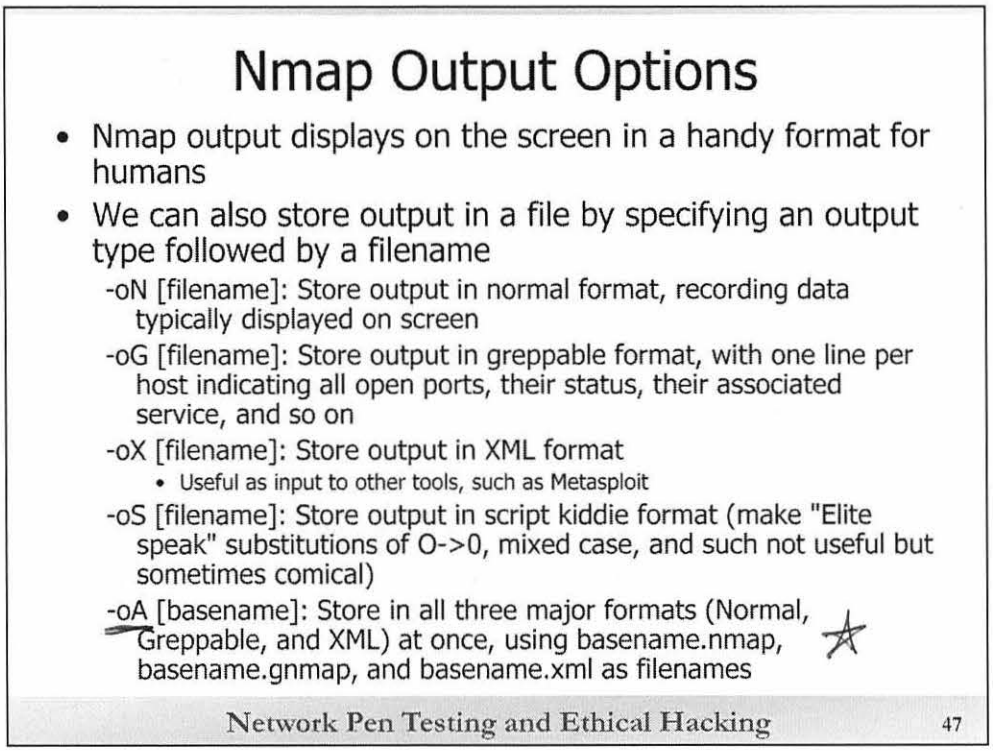
Nmap supports a variety of scanning speeds built in. They can be invoked at the command line by adding a –T <Paranoid|Sneaky|Polite|Normal|Aggressive|Insane> to the Nmap invocation. Alternatively, they can be referred to by numbers, with 0 meaning Paranoid (-T0) and 5 meaning Insane (-T5).

- **0**: 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.
- **1**: Sneaky mode sends packets every 15 seconds. As with Paranoid, no parallel sending is used with the Sneaky option.
- **2**: 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.
- **3**: 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.
- **4**: 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.
- **5**: 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 be used only on a fast network. Furthermore, sending packets at that clip could impact the target system or network equipment between the scanning machine and the target.



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.

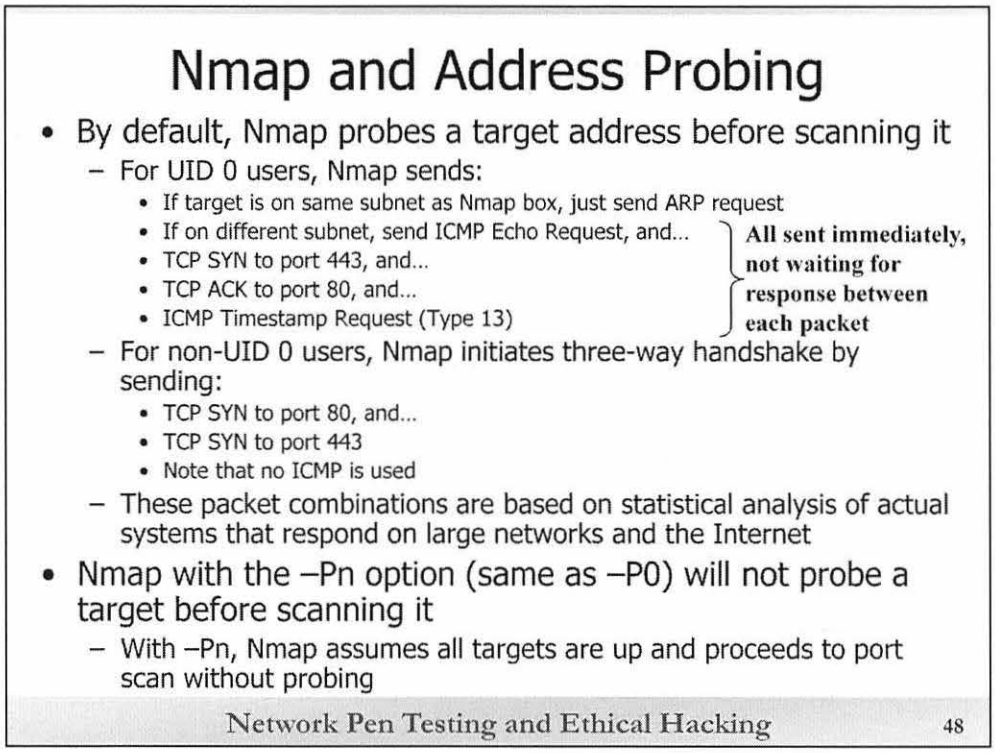


Nmap displays results to the screen in an easy-to-read, human-friendly format, indicating which hosts were scanned and the open ports and services associated with each host. In addition, we can have Nmap store its results in a file by specifying various options at the command line prefaced with a dash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

- The -oN [filename] option stores 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.
- 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. For example, Metasploit imports Nmap's XML format.
- The -oS [filename] option creates script-kiddie style output, which can be fun for laughs but isn't terribly useful. Os become zeros, Ses 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.

Using ARP Request always get response.
If not then the pc cant communicate with other



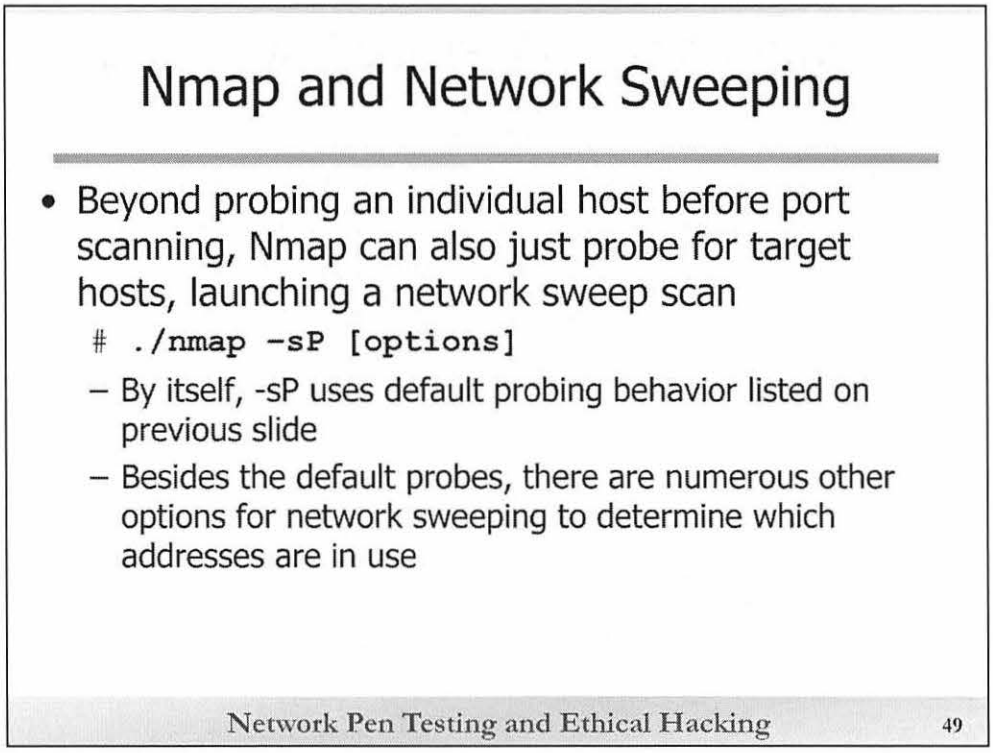
Nmap is not just a port scanner' although that is one of its primary purposes. The tool does offer numerous other features, such as identifying which addresses are in use on a target network. In other words, Nmap can be used for network sweep scans.

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 were 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 were 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 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.

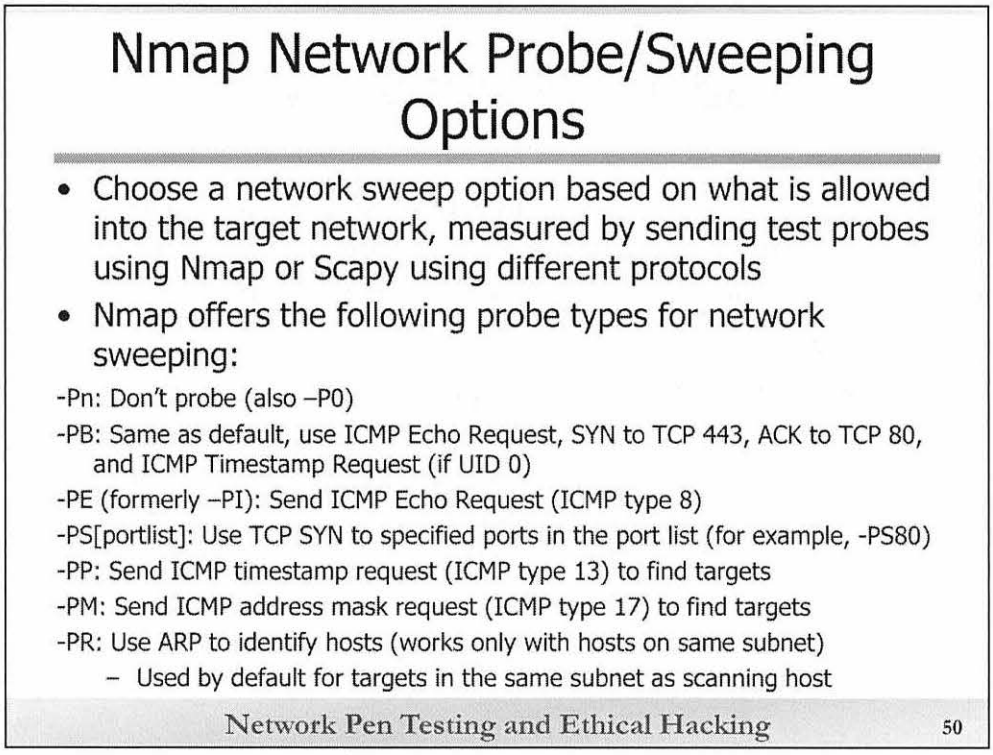
The specific probe packets were chosen by the Nmap development team based on statistical analyses of scans of large networks, focusing probes on those packets most likely to get a valid response.

By default, Nmap will scan only the target if it gets a response to the messages described here. If it doesn't get a response, Nmap gives up on that address. To make Nmap skip this probe 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 probing) and –PO (the uppercase 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).



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.



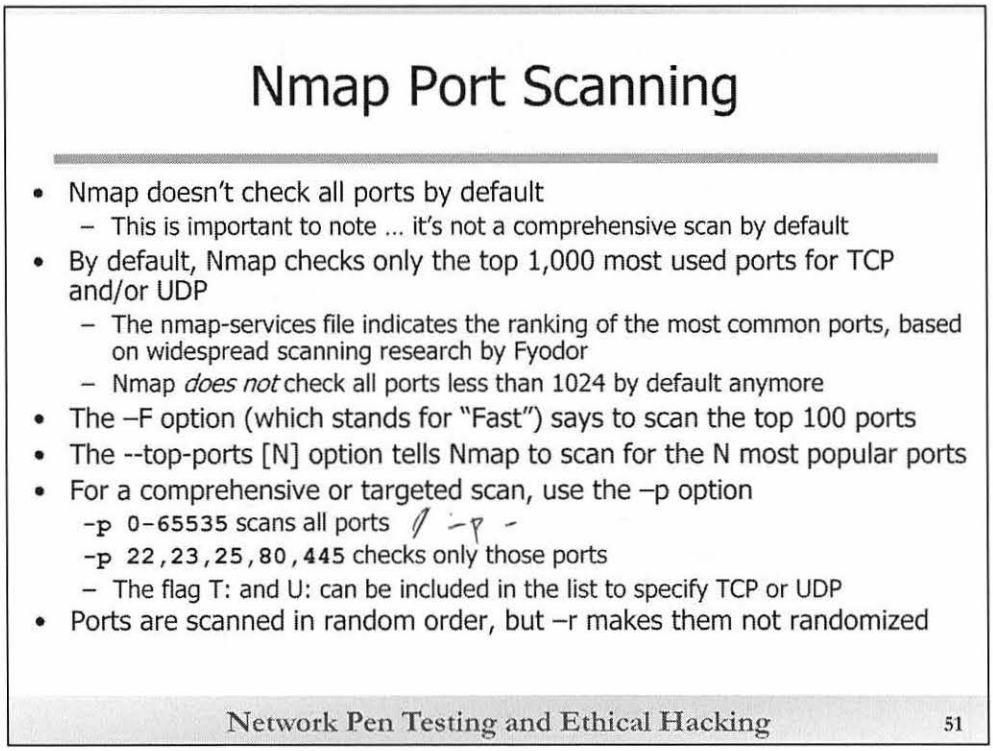
Here are the other probe options in Nmap for network sweeping. The attacker will choose an appropriate option based on what is allowed into the target network. If a network firewall blocks some ICMP types but not others, we might still identify hosts on the other side.

As we've discussed, the –Pn option tells Nmap not to probe at all. Some of the other useful probing options for a network sweep include

- **-PB**: 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**: Sends only an ICMP Echo Request message (formerly –PI).
- **-PS[portlist]**: 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**: Sends ICMP Timestamp query messages.
- **-PM**: Sends ICMP Address Mask queries.
- **-PR**: 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.



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 checks only 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.

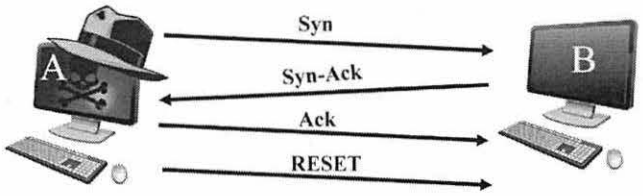
However, a given target environment may have a specialized application listening on a port that is not in the top port listing in the nmap-services file. Thus, if testing time permits, you should consider doing a comprehensive port scan, checking all possible ports. The –p flag can indicate a port, port 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 and Ethical Hacking 52

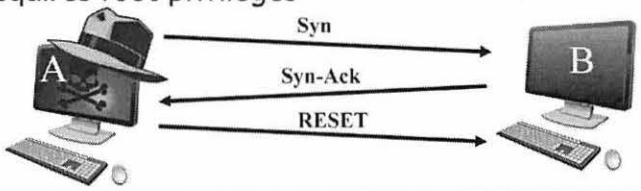
Nmap offers numerous types of TCP port scan options, most of which are based on triggering the behavior of target machines with various TCP Control Bits set.

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, the DEFAULT scan
  - 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

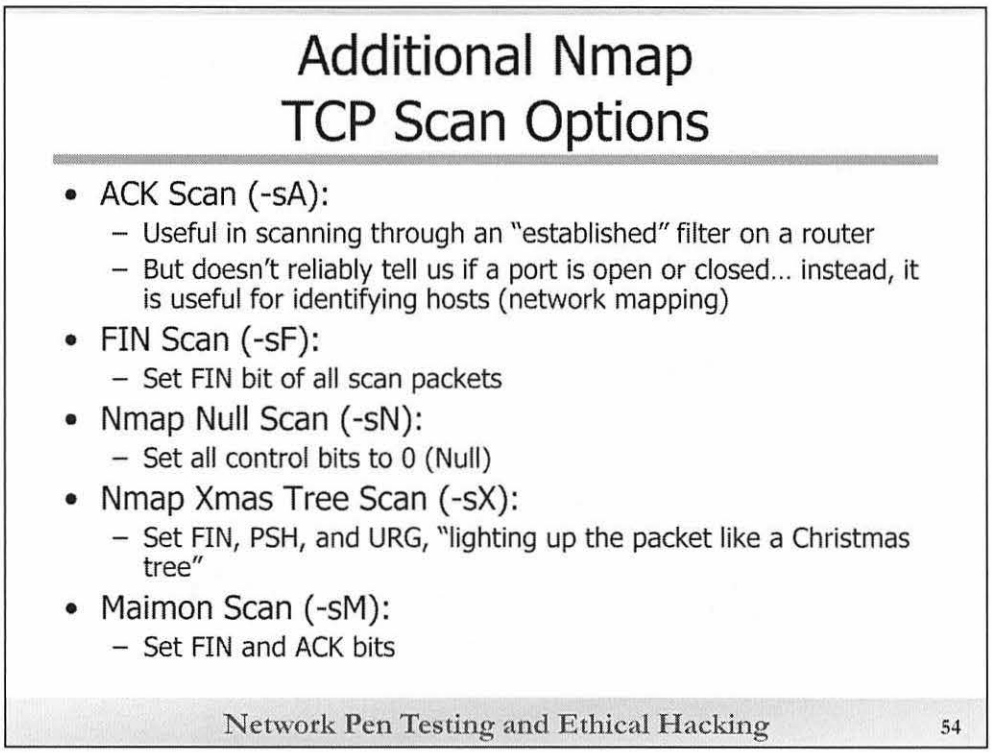


Network Pen Testing and Ethical Hacking 53

A SYN scan, also known as a "half-open" or "SYN Stealth" scan, doesn't complete the three-way handshake. Instead, it starts 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. If you don't specify an Nmap scan type and are running Nmap as root, the –sS SYN-Stealth scan is the most common kind of scan that Nmap defaults to.

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.

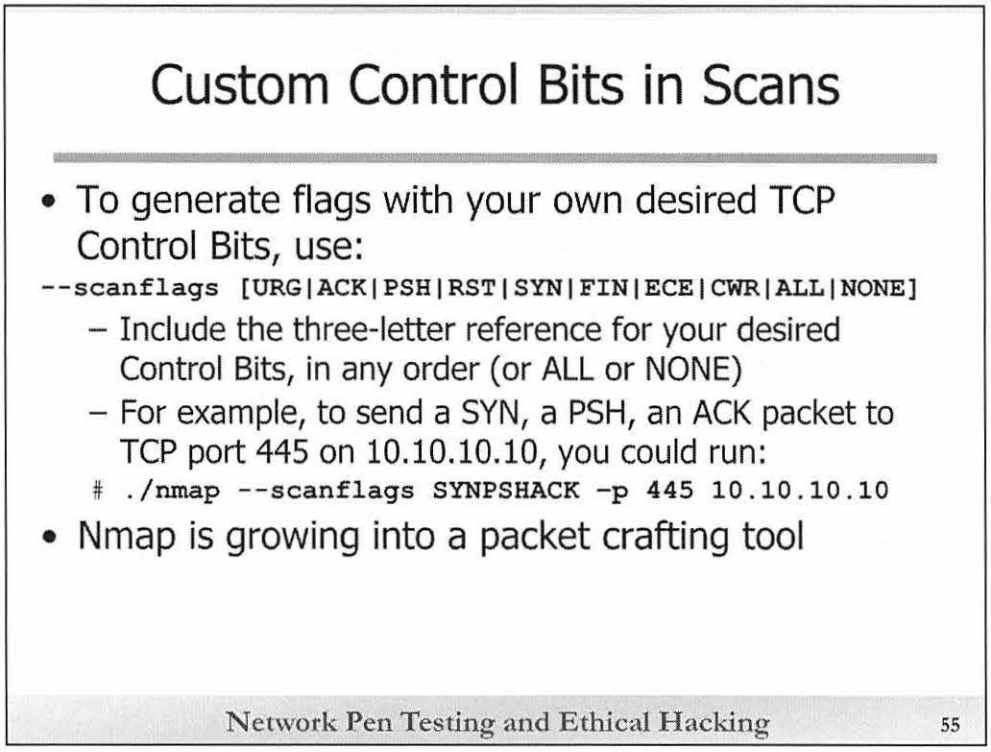


Nmap also supports ACK scanning to help get through certain kinds of packet filters. A router may have access control lists configured to allow outgoing SYNs from a protected network and their incoming responses (that is, established connections with the ACK Control Bit set). These filters also block 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.



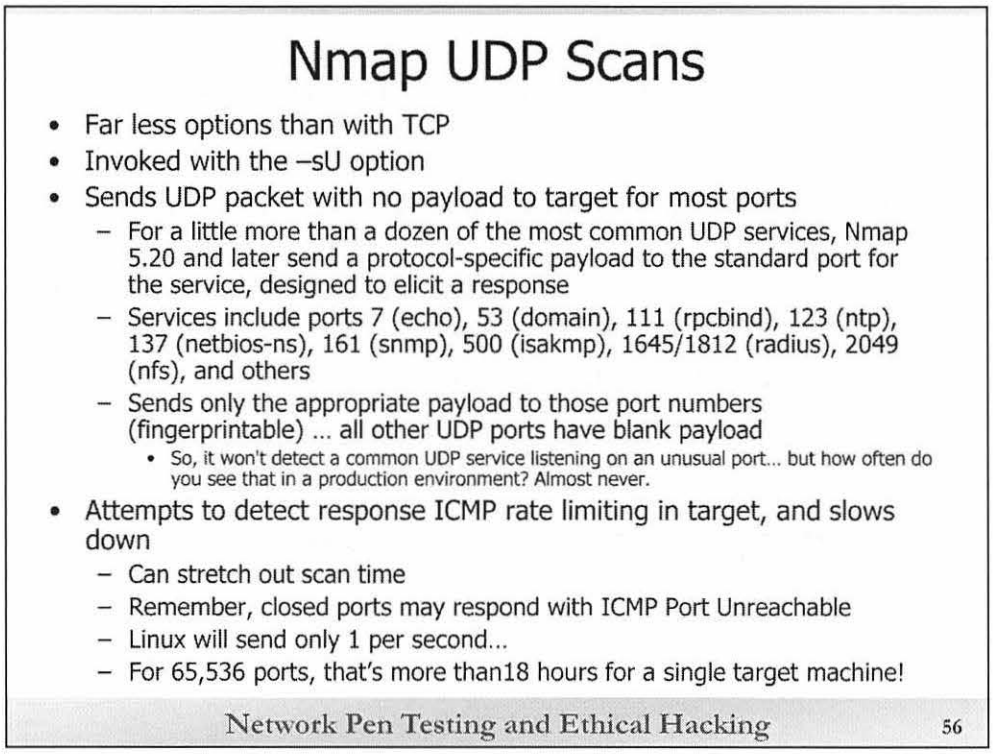
Beyond the prebaked Control Bit scans (Xmas, Maimon, and so on), Nmap users can also specify arbitrary Control Bit settings, using the --scanflags option, followed by a list of the wanted Control Bits. Control Bits are indicated based on three-letter abbreviations of URG, ACK, PSH, RST, SYN, and FIN and can be specified in any order. Note that even the extended Control Bits (ECE and CWR) are supported now. Specifying ALL sets all 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 445 10.10.10.10
```

That syntax will invoke Nmap to scan target IP address 10.10.10.10, sending a TCP packet with the SYN, PUSH, and ACK Control Bits set to destination port 445.

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 using Scapy.



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 sends UDP packets with no payload to the target machine for most ports.

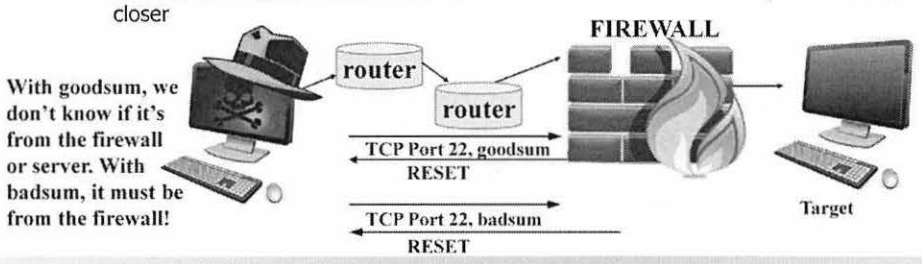
Starting with Nmap 5.20, for a little more than a dozen specific UDP ports associated with the most common UDP services, Nmap sends a protocol-specific payload in its UDP packet for each service, designed to get the target service to send a UDP response back. That way, we can get more reliable UDP port 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 sent only 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 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 sends 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 takes over 18.2 hours, a long time.

to know whether the server behind the firewall or not

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 and Ethical Hacking 57

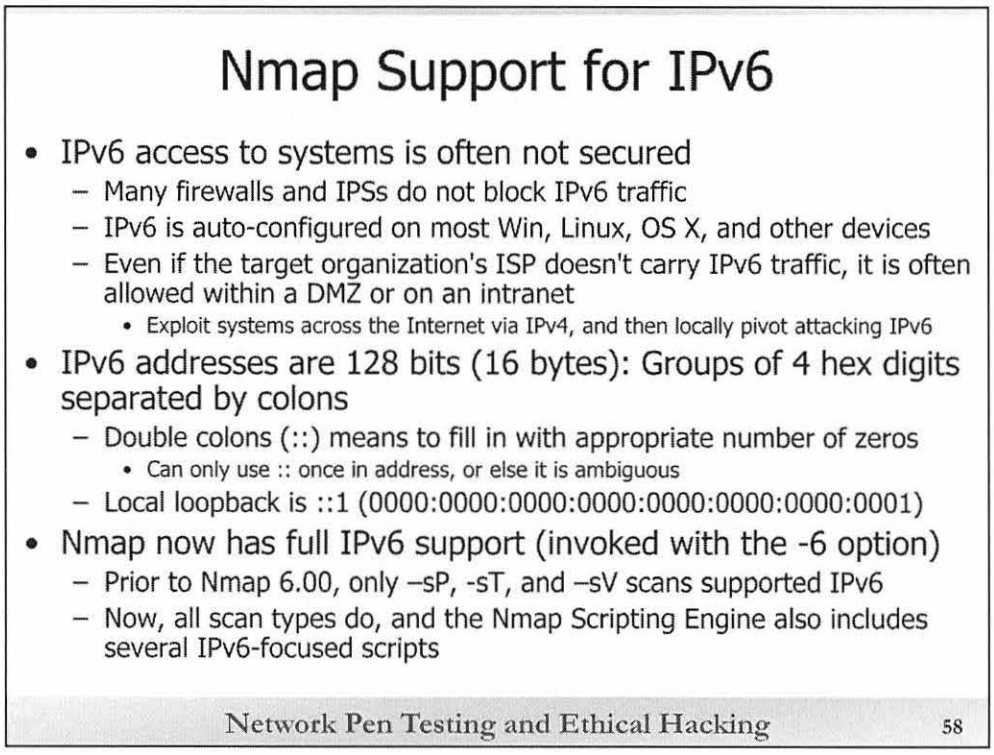
Another interesting Nmap feature involves sending packets with bad TCP or UDP checksums, calculated incorrectly on purpose. The resulting packets are bogus and should be ignored by any target operating system.

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. In addition, 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.

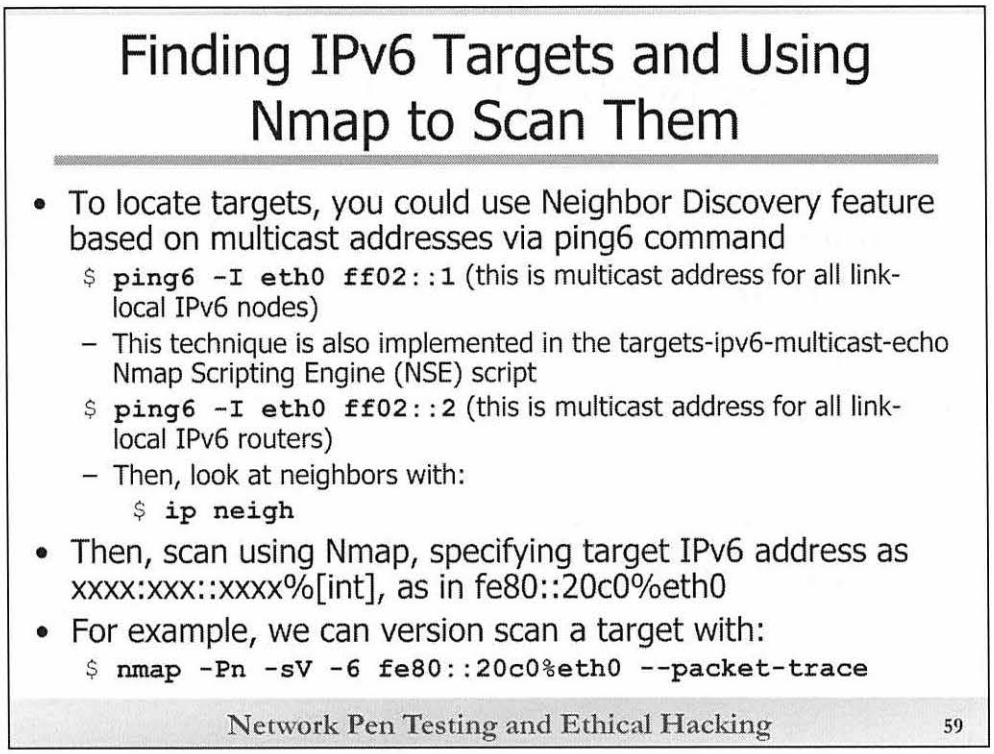
Another trick for performing firewall spotting is to look at the TTL values in the responses coming back from the target environment. If the TTL values from allowed services (say, SYN-ACKs from the web server on port 80) are different from the TTL values of blocked services (indicated by RESETs coming 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 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 use :: very 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 0000:0000:0000:0000:0000:0000:0000:0001.

Starting with version 6.00, Nmap now fully supports IPv6 for all scan types, by simply adding a "-6" to its command-line invocation. Earlier versions of Nmap supported IPv6 only for network sweeps (-sP), TCP Connect Scans (-sT), and version scans (-sV). If you are running an older version of Nmap often included in some stock Linux distributions, be aware that you may not have IPv6 support for other types of scans unless you upgrade to a post-6.00 version. With these more recent versions, all scan types (along with the Nmap Scripting Engine) support IPv6.



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 in to many Linux variations and Mac OS X to send a message to the multicast address of a local subnet looking for neighbors, a feature of IPv6 known as *neighbor discovery*. The multicast 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
```

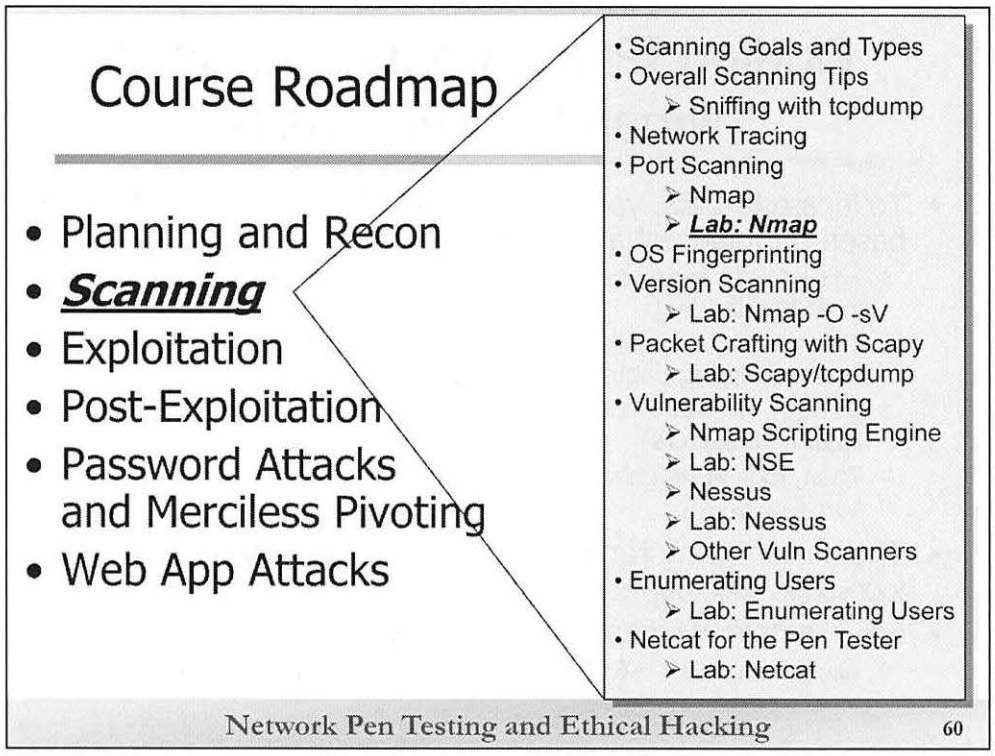
Note that the first of these items (multicast ping of link-local IPv6 nodes) is also supported by a script included with Nmap called targets-ipv6-multicast-echo. We'll cover Nmap scripts later in 560.2.

After you ping the target multicast addresses (using ping6 or an Nmap script), you could look at the output of your 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.

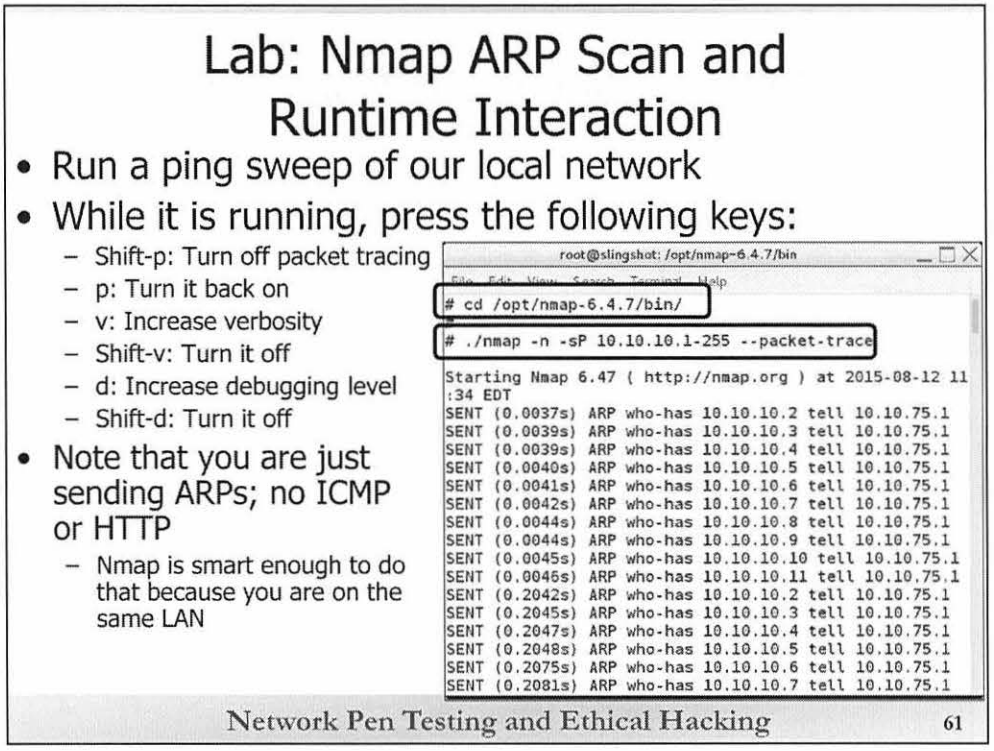
Then, to run Nmap to launch a TCP connect scan and/or version scan against discovered targets, you need to specify the IPv6 address followed by a %[int] to indicate the interface the packets should be sent on. For example, you may scan an address such as fe80::20c0%eth0 to send packets on your eth0 interface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



Now, we'll do some labs with Nmap, exploring its runtime interaction capabilities, ARP scanning, and more features useful to penetration testers. Get your Linux machine ready to go, logging in with root-level privileges, which Nmap needs to formulate most of the unusual packets we'll generate.



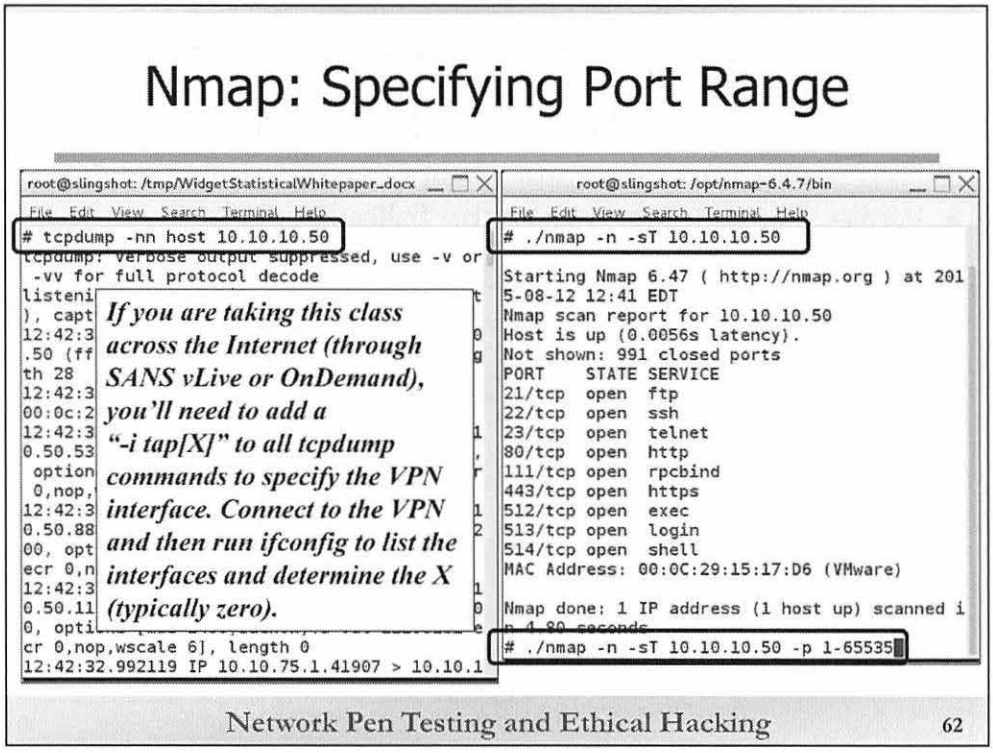
Let's run a scan of the target subnet. We'll change into the Nmap 6.4.7 directory and then run Nmap from that directory. (Use ./nmap to invoke it so that you are using the proper version.)

```
# cd /opt/nmap-6.4.7/bin
# ./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, pressing Shift-p turns this off, whereas pressing the p key toggles 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 pressing 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.
Start tcpdump, configured to show traffic associated with host 10.10.10.50 (not resolving names).
LAUNCH A NEW TERMINAL WINDOW SO YOU CAN RUN A SNIFFER IN IT TO OBSERVE YOUR PACKETS:

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

NOTE: IF YOU ARE TAKING THIS CLASS ACROSS THE INTERNET THROUGH SANS vLive or OnDemand, you need to specify the VPN interface in all the tcpdump commands for the class. Connect through the VPN and then run the ifconfig command to list interfaces, looking for an interface called tap[X], where X is an integer (typically zero). Then, add "–i tap[X]" (without the quotation marks, but with the appropriate X, and without the [] brackets) to all tcpdump commands.

Next, back in your original Nmap terminal window, invoke Nmap to scan that host, doing a TCP connect scan (full three-way handshake):

```
# ./nmap -n -sT 10.10.10.50
```

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

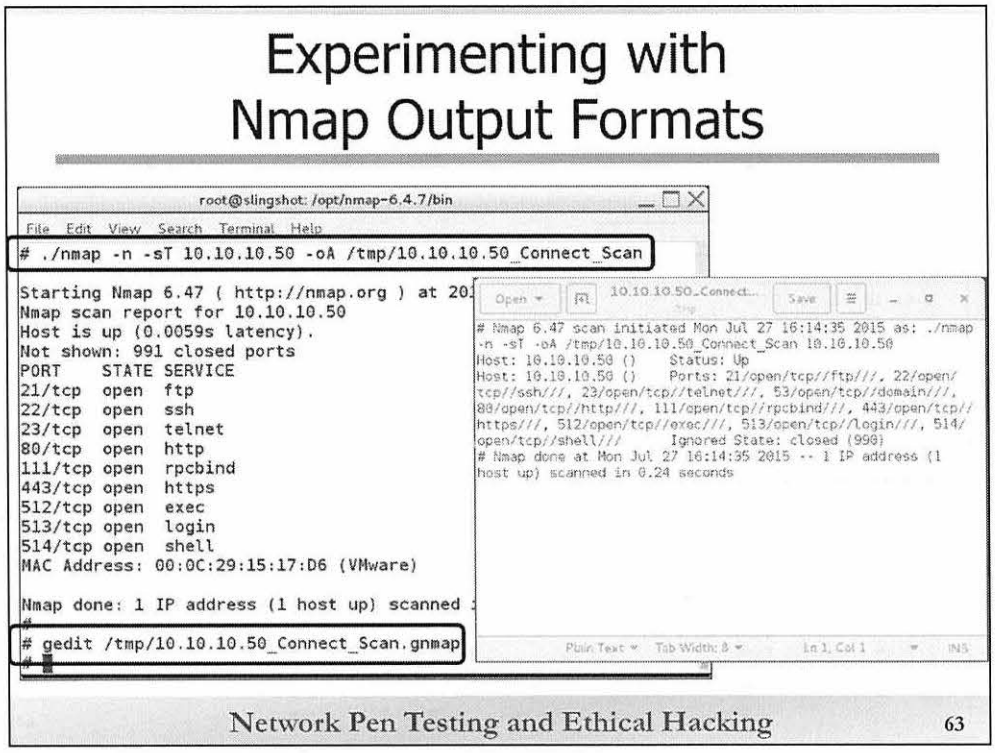
_____

Nmap did not scan all TCP ports with that invocation, however. It scanned only 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 a lot of SYN packets (S) going from your machine to the target, as well as a lot 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, look at the output format files that Nmap can create via the -oA option. Rerun your -sT scan with the default port, storing your results in all the major format styles (-oA to indicate Normal, Greppable, and XML output). Store your 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:

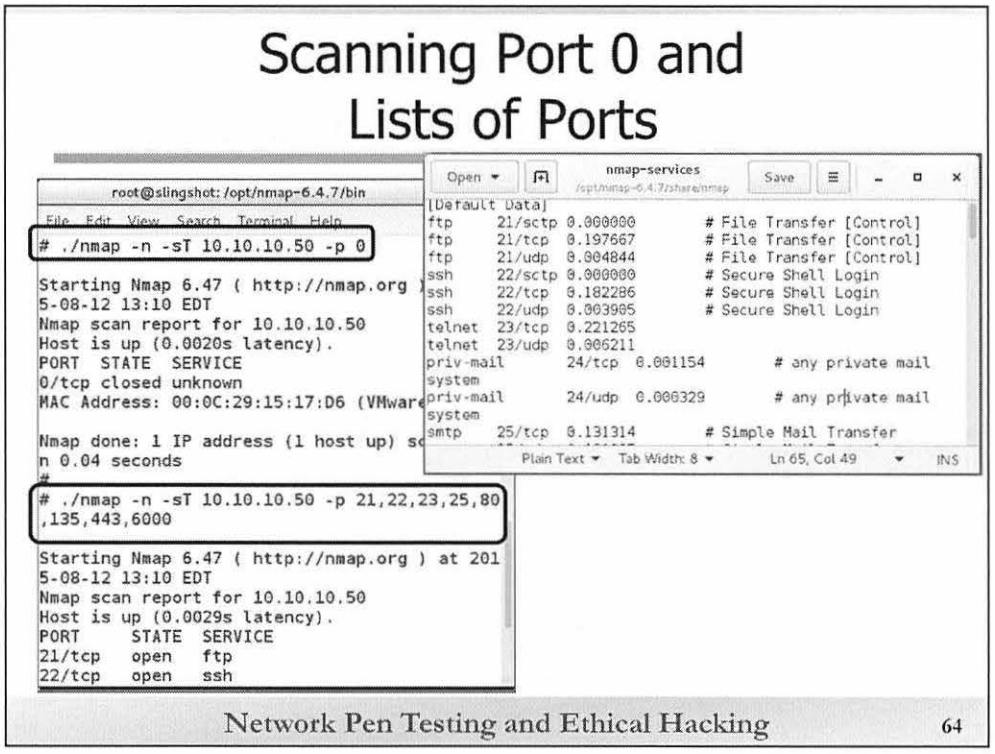
```
# ls /tmp/10.10.10.50*
```

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 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
```

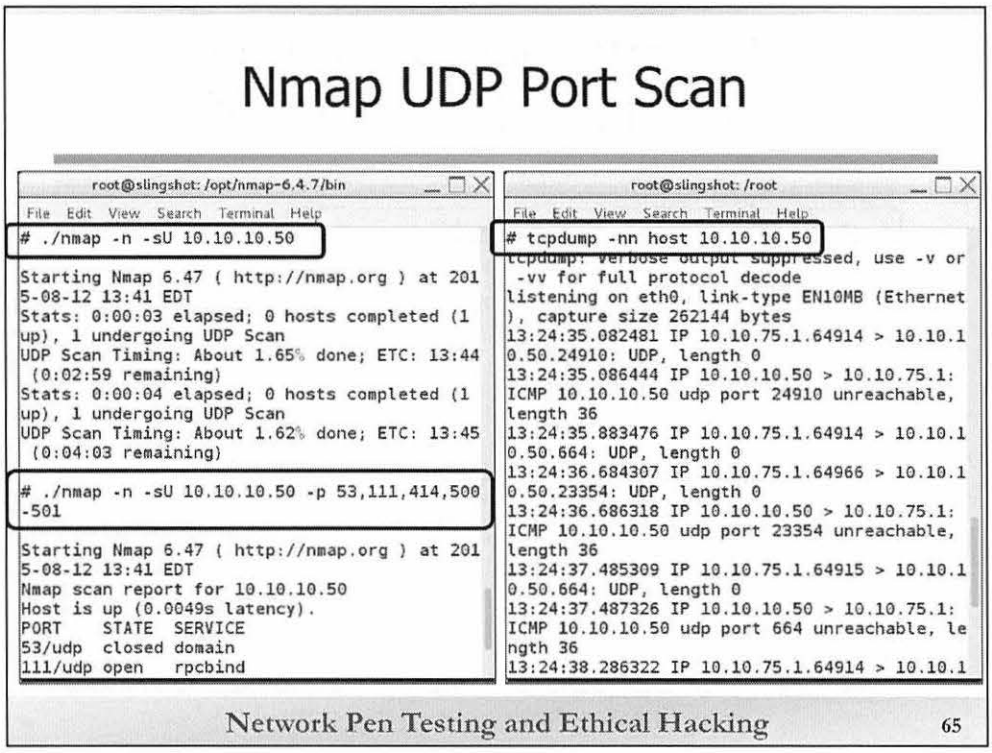
As we've seen, we can scan individual ports by just specifying –p [X] (where [X] is the port number we want to scan). We can do ranges of ports by specifying –p [X-Y]. And we can do individual sets of ports by using a comma-separated list. Try the last one by scanning:

```
# ./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 /opt/nmap-6.4.7/share/nmap/nmap-services
```

The format of this file includes the service name (for example, ftp), the associated port and protocol (for example, 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 you've looked at TCP port scanning with Nmap, try UDP port scanning. Remember we discussed earlier that Linux kernels throttle ICMP port unreachable responses so that they send only 1 every second? You'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:

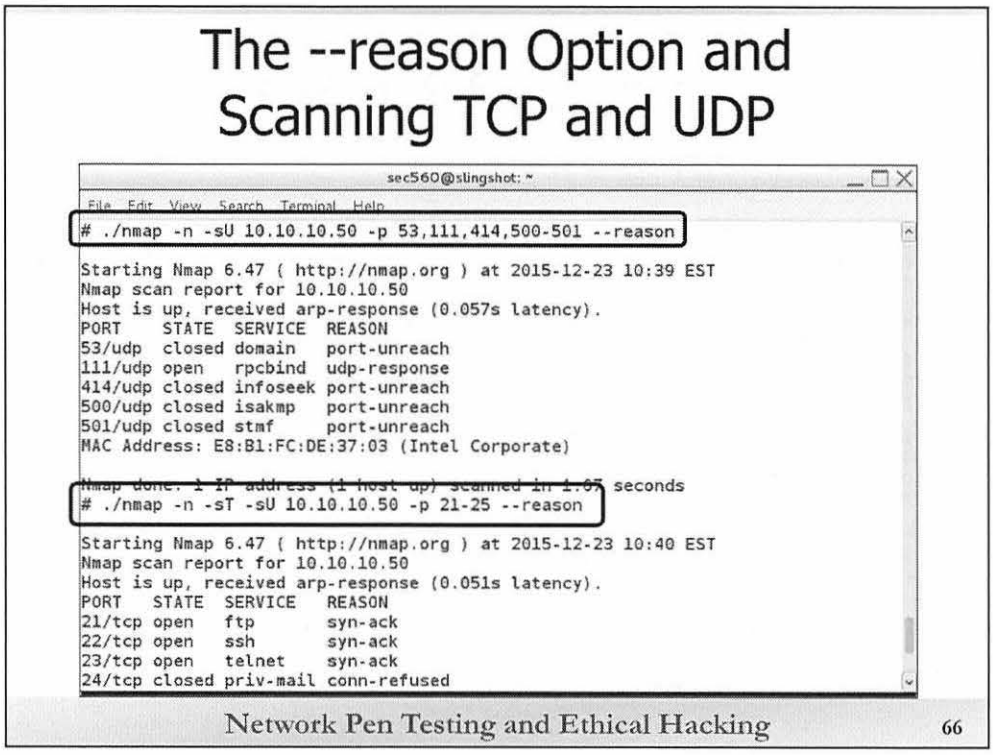
```
# ./nmap -n -sU 10.10.10.50
```

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

In your Nmap window, press 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 you have time for here, perhaps more than an hour or more, depending on your system speed and the network speed. You can't wait, so press CTRL-C to stop Nmap before the scan completes.

Now, rerun 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 you why Nmap classifies a given port's open/closed/filtered state as it does. Rerun your previous scan but with the --reason option:

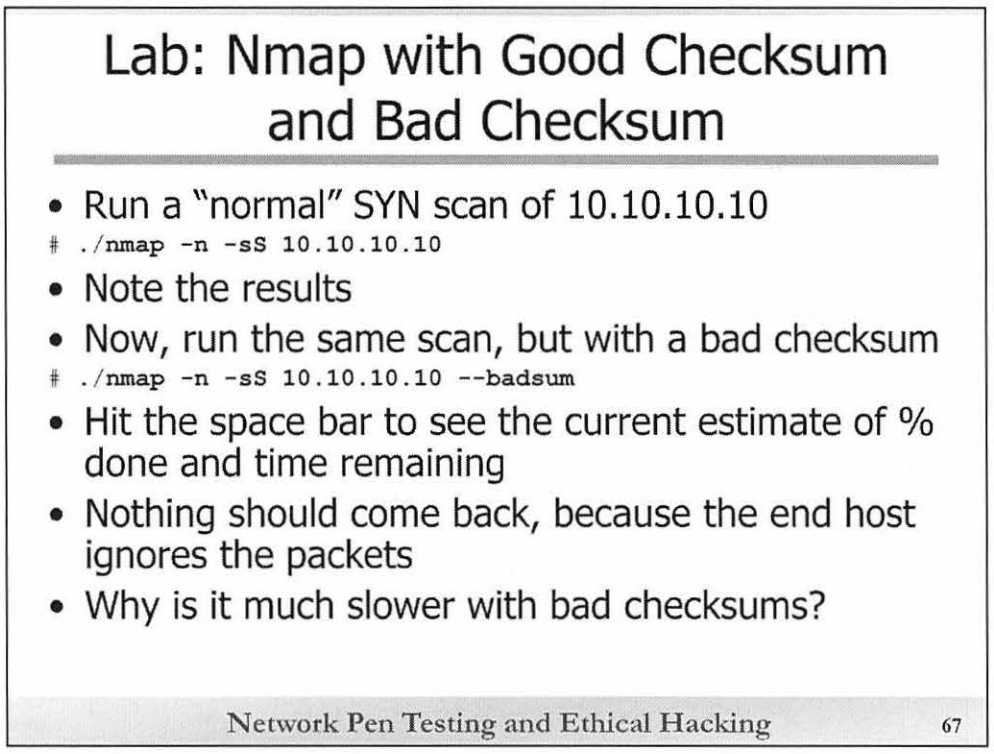
```
# ./nmap -n -sU 10.10.10.50 -p 53,111,414,500-501 --reason
```

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, 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.



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

```
# ./nmap -n -sS 10.10.10.10
```

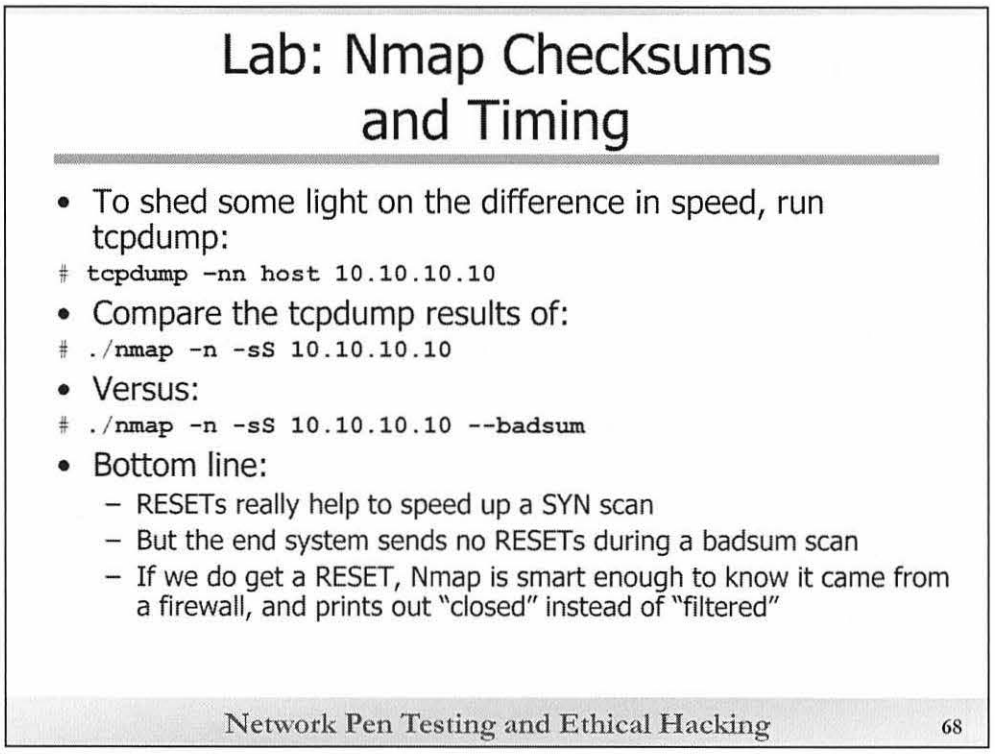
You should see some open ports on the target.

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

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

This takes a lot longer. To see what your current status is, press any key (such as the space bar) to see the 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.



To determine why it is slower with bad checksums, try running tcpdump IN ANOTHER TERMINAL WINDOW:

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

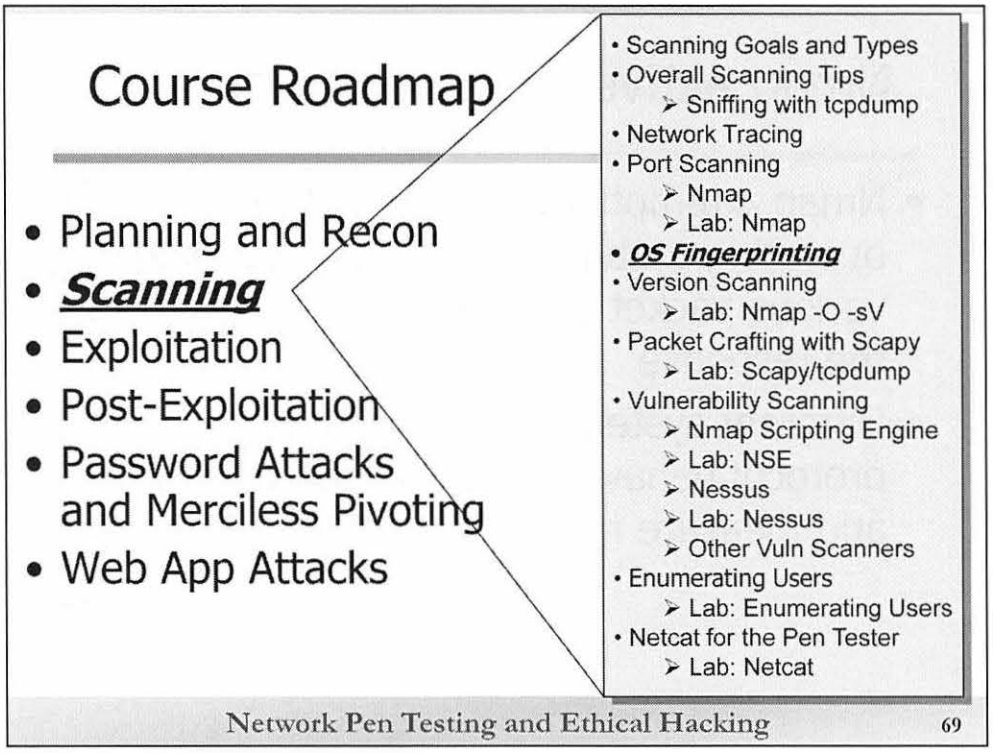
Compare the tcpdump results when running the following back in your Nmap terminal window:

```
# ./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?



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, but we can also begin researching known flaws (such as common misconfigurations and unpatched services) on those types of devices.

Nmap Active OS Fingerprinting (1)

- Nmap attempts to determine the operating system of targets by sending various packet types and measuring the response
- Different systems have different protocol behaviors that we can trigger and measure remotely

Network Pen Testing and Ethical Hacking 70

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.

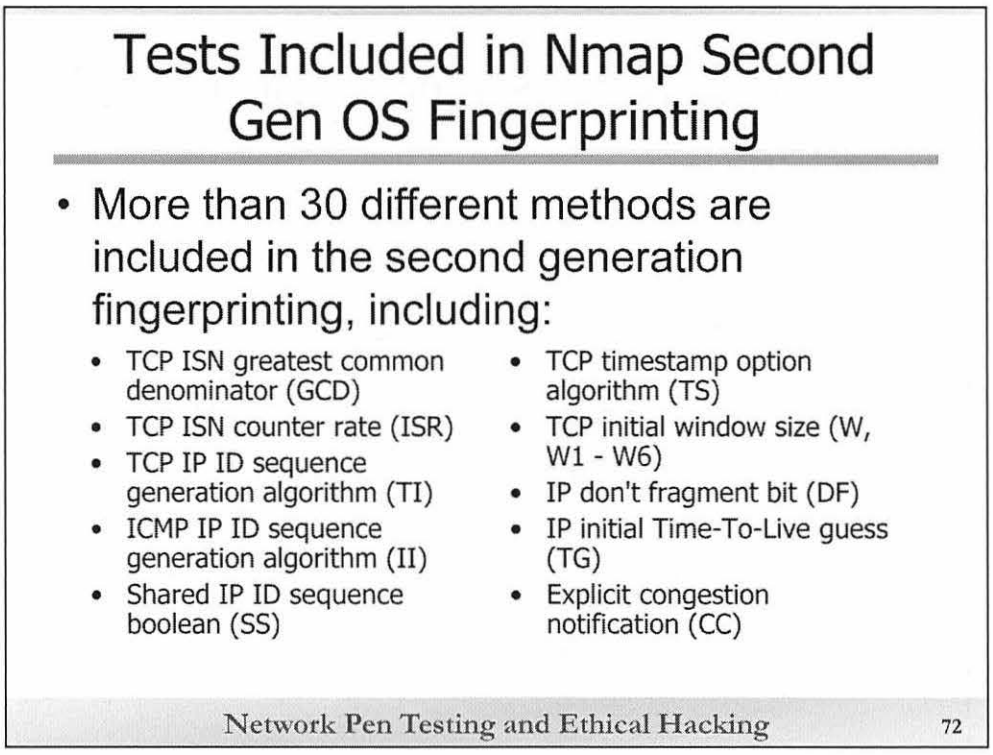
Nmap Active OS Fingerprinting (2)

- Modern versions of Nmap have dropped the first generation OS fingerprinting capability built in to 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

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.

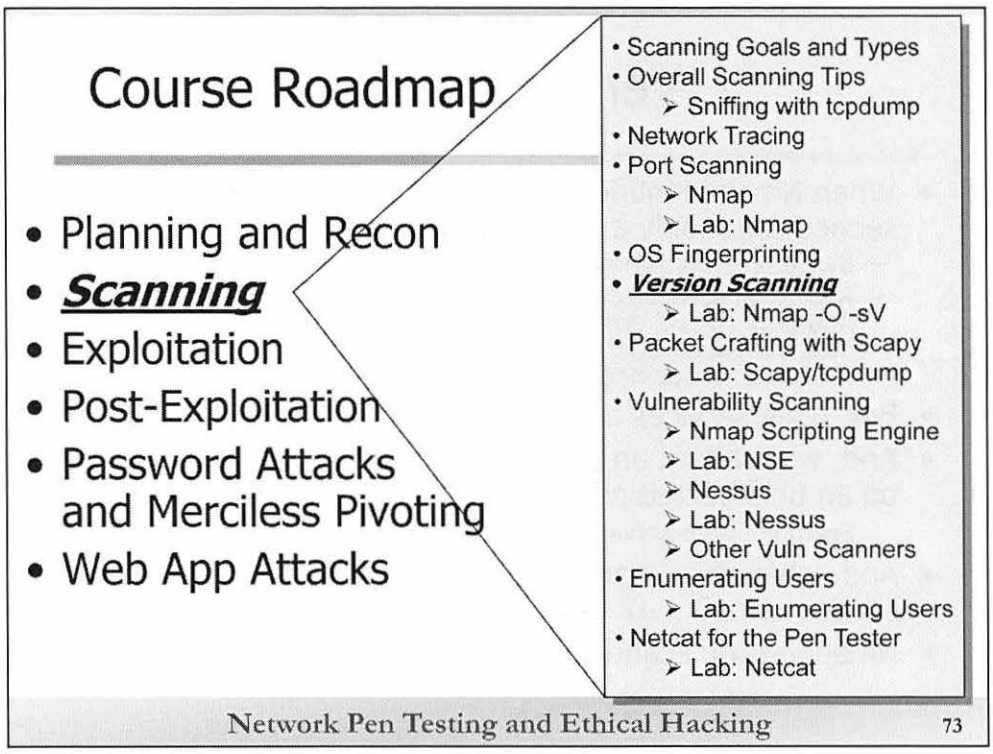
Recent versions of Nmap have a "second generation" active OS fingerprinting capability. A huge number of new active fingerprinting techniques have been added in this suite. Currently, the second generation tests are more accurate than the first. The most recent Nmap releases have dropped support altogether 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.



The second generation active OS fingerprinting of Nmap includes more than 30 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, whereas 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.



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

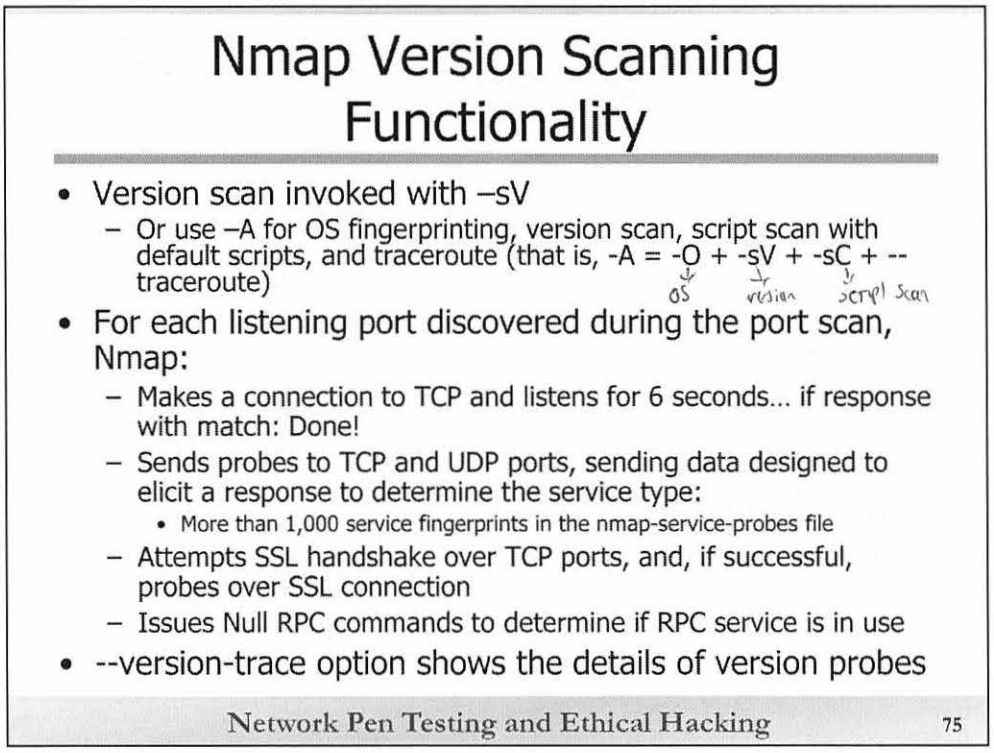
- When Nmap identifies an open port, it displays the default service commonly associated with that port
  - Based on nmap-services file, which lists approximately 2,200 services
  - Additional services are searchable at the Internet Assigned Numbers Authority (IANA) port assignments at http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xml
- 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? Use -sV command.
- Nmap version scanning has the answer

Network Pen Testing and Ethical Hacking 74

When you have a list of open ports, you 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. 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 not 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 useful Nmap feature known as Version Scanning.

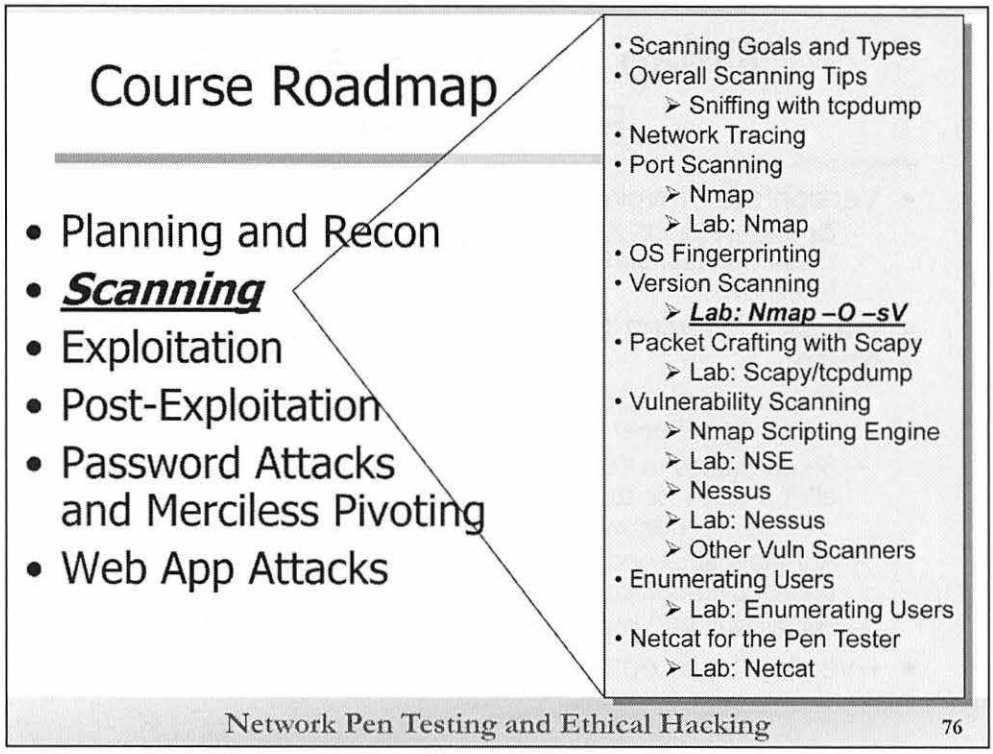


To invoke Nmap with its Version Scanning functionality, use the –sV option. Alternatively, Nmap executed with the –A option will conduct OS fingerprinting, Version Scanning, script scanning, and Nmap's traceroutng. In Nmap's algebra, it appears that –A = -O + -sV + -sC + --traceroute. In other words, running Nmap with the –A option is the same as running it with the –O (OS fingerprinting), –sV (version scan), -sC (run Nmap Scripting Engine scripts in the "default" category), and --traceroute (use Nmap's traceroute feature) options.

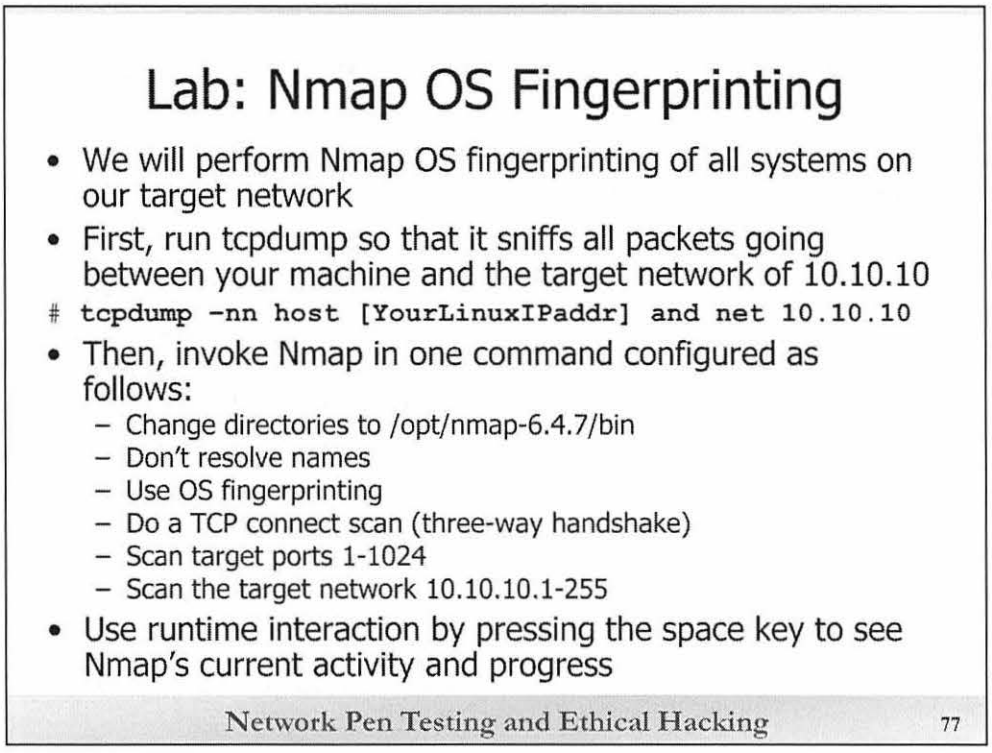
The Nmap version scan functionality is executed after Nmap finishes conducting a port scan of the target. For each discovered open port, Nmap will probe the port to try to determine what is listening there. For TCP ports, Nmap connects to the port with a three-way handshake and waits for a response. If a response comes across the connection, Nmap looks up that response in its nmap-service-probe file. 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 more than 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.



We'll now perform a lab featuring Nmap OS fingerprinting and version scanning. We'll also see some unusual port behavior in our Nmap analysis, and in a later lab, we'll attempt to discern its network behavior using Scapy and tcpdump.



For this lab, 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
```

Then, in ANOTHER terminal window, change into the nmap-6.4.7 bin directory:

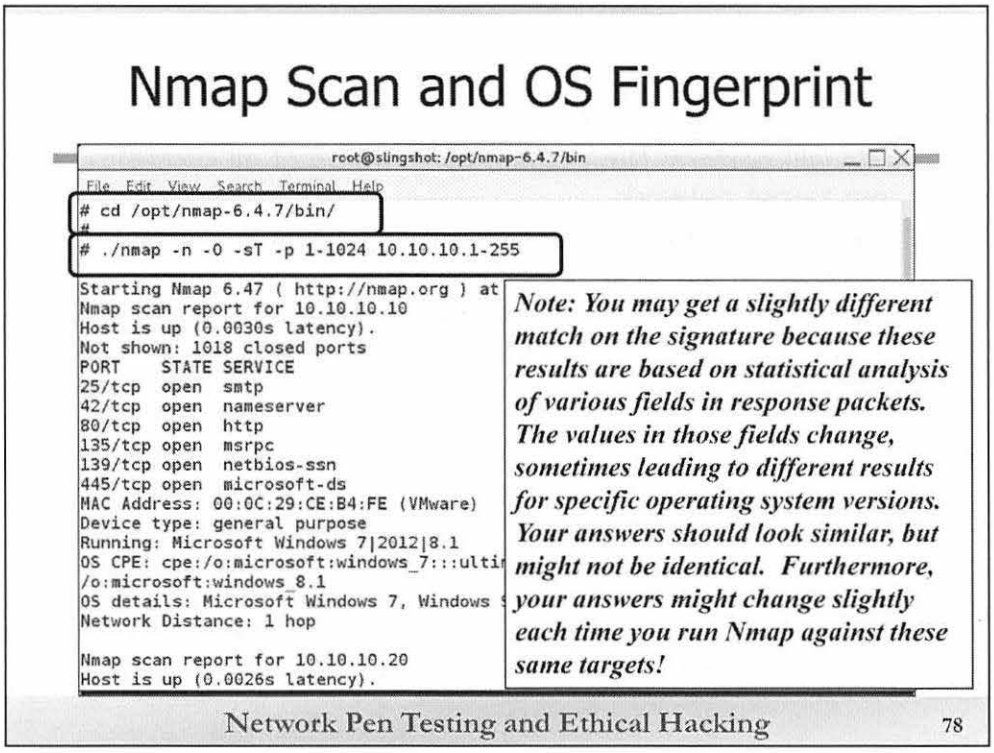
```
# cd /opt/nmap-6.4.7/bin
```

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. Then, flip the page to see the recommended Nmap command to verify your planned usage.

While Nmap is running, periodically check on its progress by looking at your sniffer output. Also, press the space key every once and a while to see what Nmap is up to.



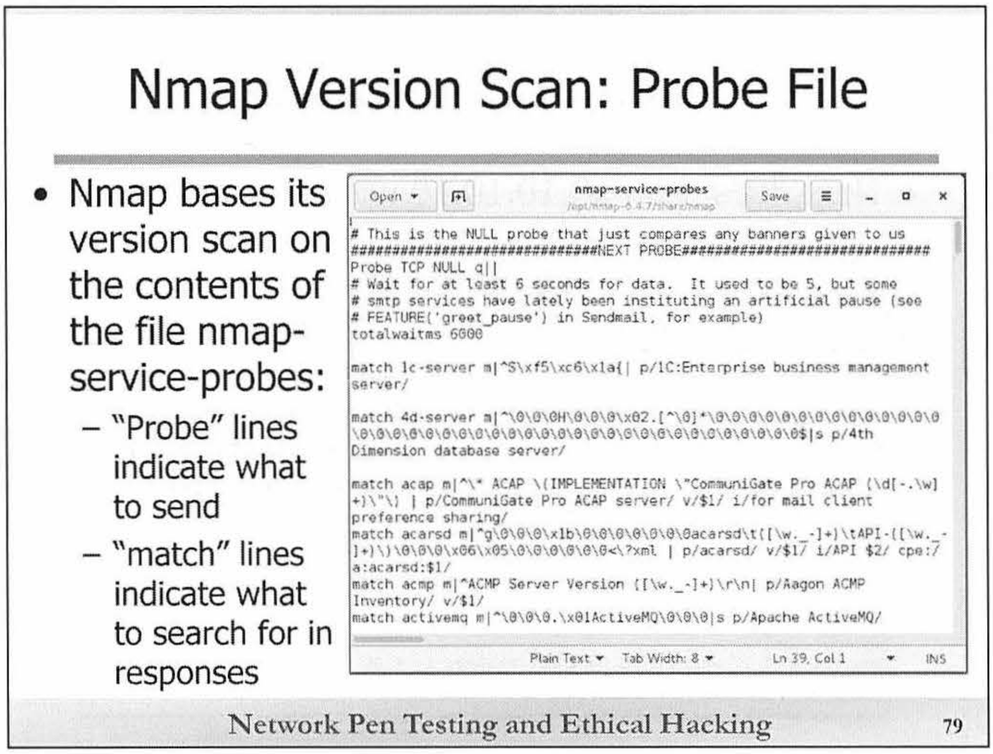
To make Nmap perform the scan described on the previous slide, we invoke it as follows:

```
# cd /opt/nmap-6.4.7/bin
# ./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. Did it identify the operating system types of 10.10.10.10, 10.10.10.20, 10.10.10.50, and 10.10.10.60?

Note that you may get a slightly different match on signature from what you see on this slide 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!

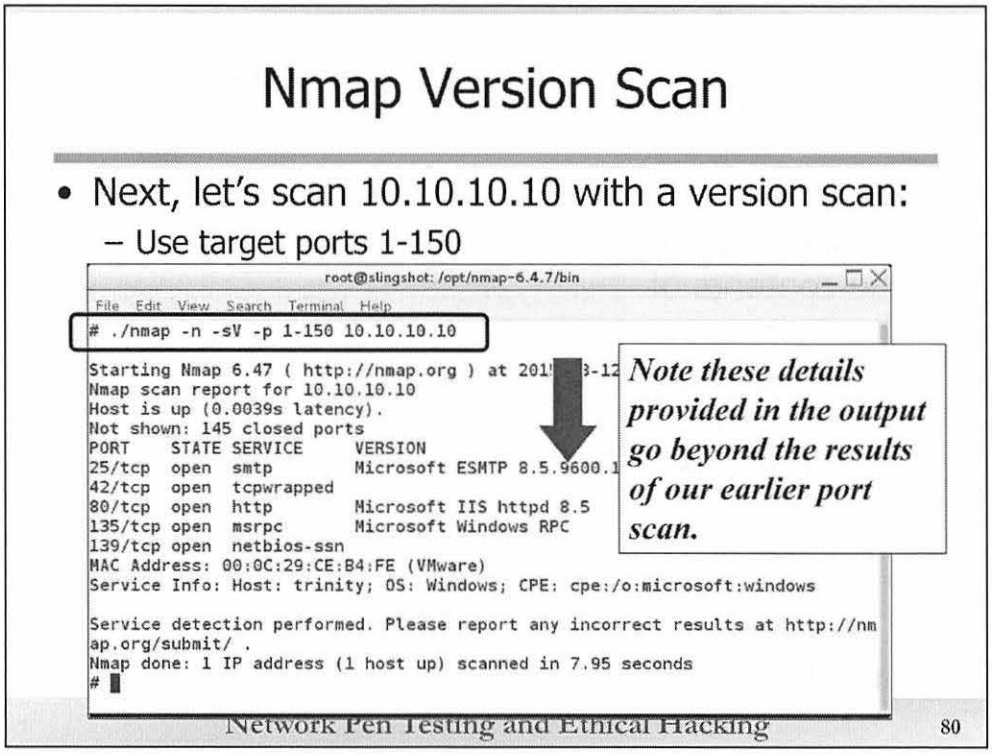


For version scanning, Nmap bases its analysis of services on the contents of a file called nmap-service-probes, located in the Nmap directory. In that file, lines that start with "Probe" indicate the messages to send to target services, whereas lines that start with "match" indicate the response text to look for when identifying the given service.

Open the file to look at its probe and match lines:

```
# gedit /opt/nmap-6.4.7/share/nmap/nmap-service-probes
```

Close the file, and we'll now use Nmap for version scanning.

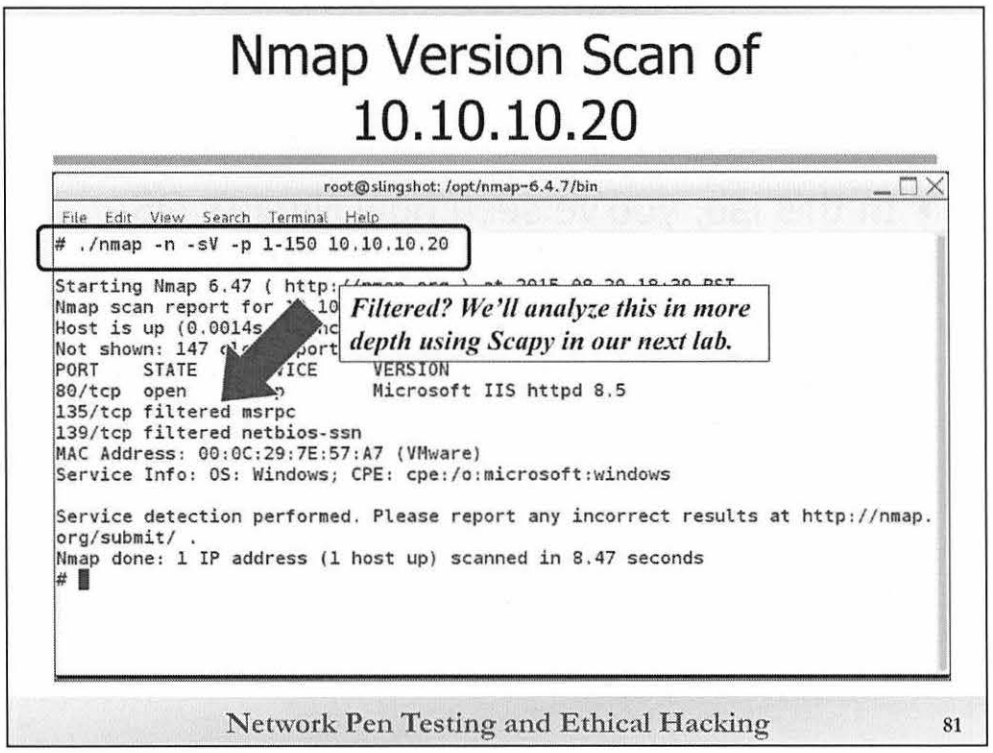


Next, we'll do an Nmap version scan, but only of ports between 1 and 150, and with one target host at a time. Start with 10.10.10.10.

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 two slides ago against 10.10.10.10. Are they different? How? You should now see a VERSION column with more detailed information in it than you did in the –sT port scan result.



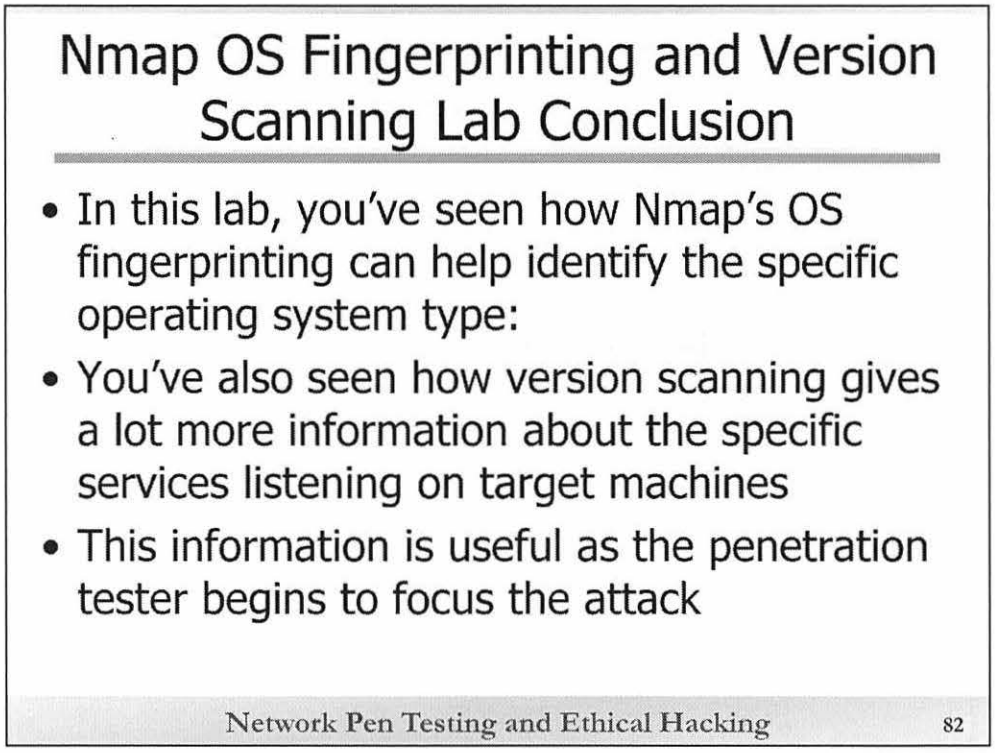
Next, proceed to do the same kind of scan against 10.10.10.20.

```
# ./nmap -n -sV -p 1-150 10.10.10.20
```

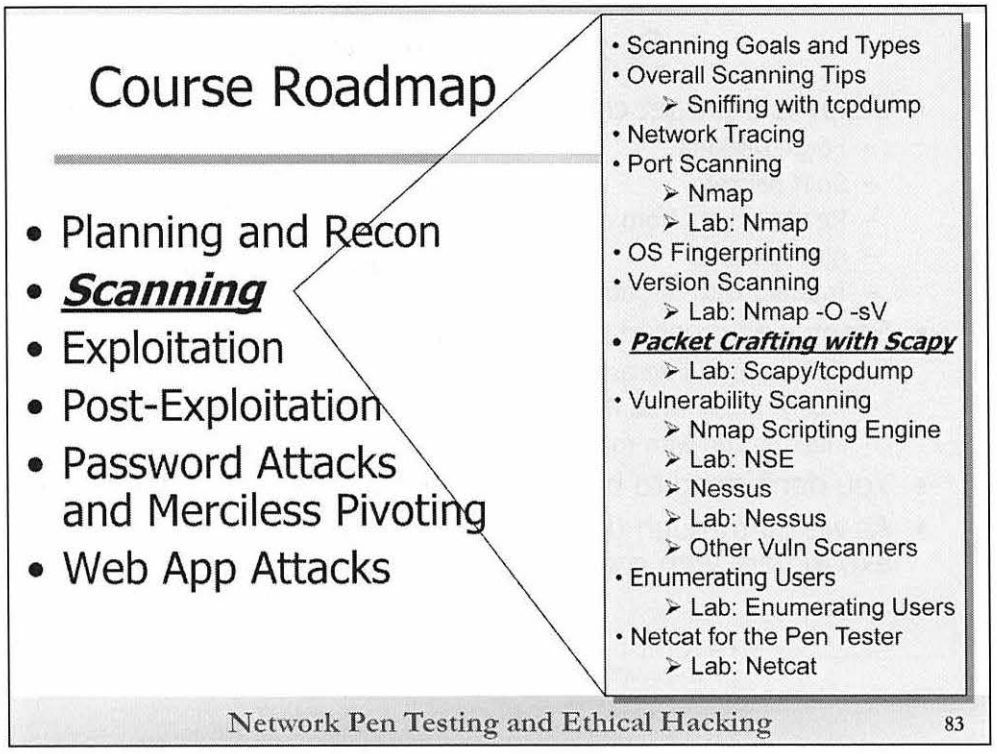
Both 10.10.10.10 and 10.10.10.20 are Windows machines, but they have significant differences in configuration, especially with respect to port filtering.

In your output for 10.10.10.20, note that TCP 135 and 139 are labeled as "filtered."

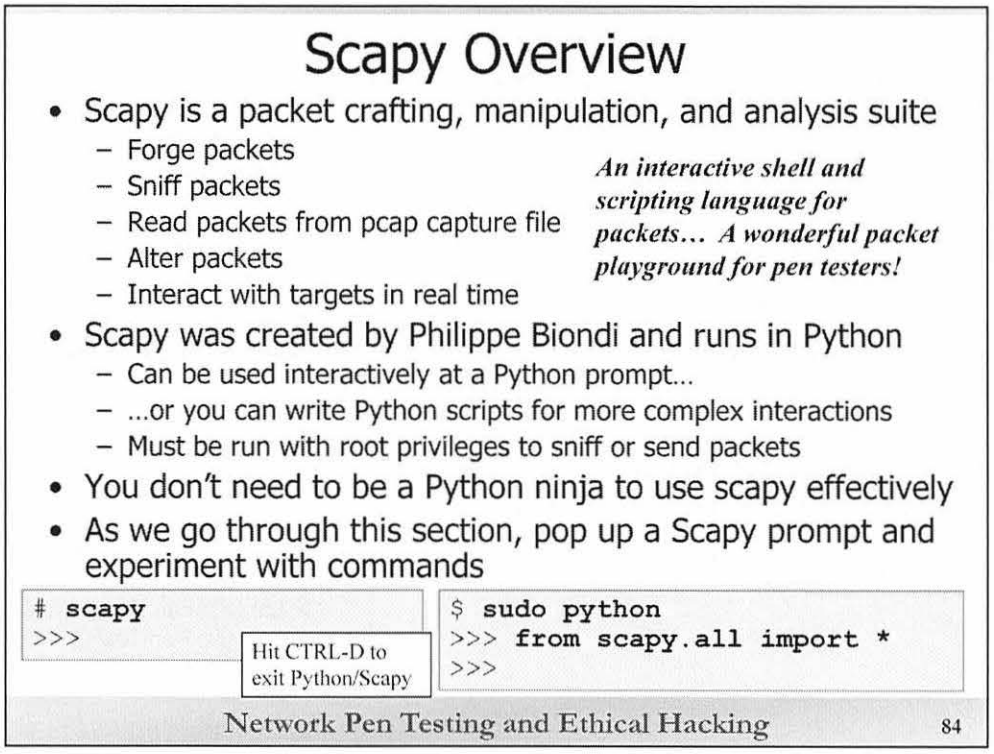
We'll investigate the reason these ports are labeled as filtered in a later lab on the Scapy tool for crafting packets in 560.2.



In this lab, we've seen how Nmap OS fingerprinting and version scanning can gather information about operating system types and the versions of software running on target machines. This information is tremendously useful to a penetration tester in focusing an attack and using specific tools and exploits to gain access to a target environment.



One of the best tools available for formulating packets used in packet sweeps, port scans, or many other forms of interaction with target systems is Scapy. As penetration testers, sometimes we need to formulate a series of specific packets to measure the behavior of a target machine with accuracy. Scapy is a perfect framework for doing so. Let's delve into this 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.

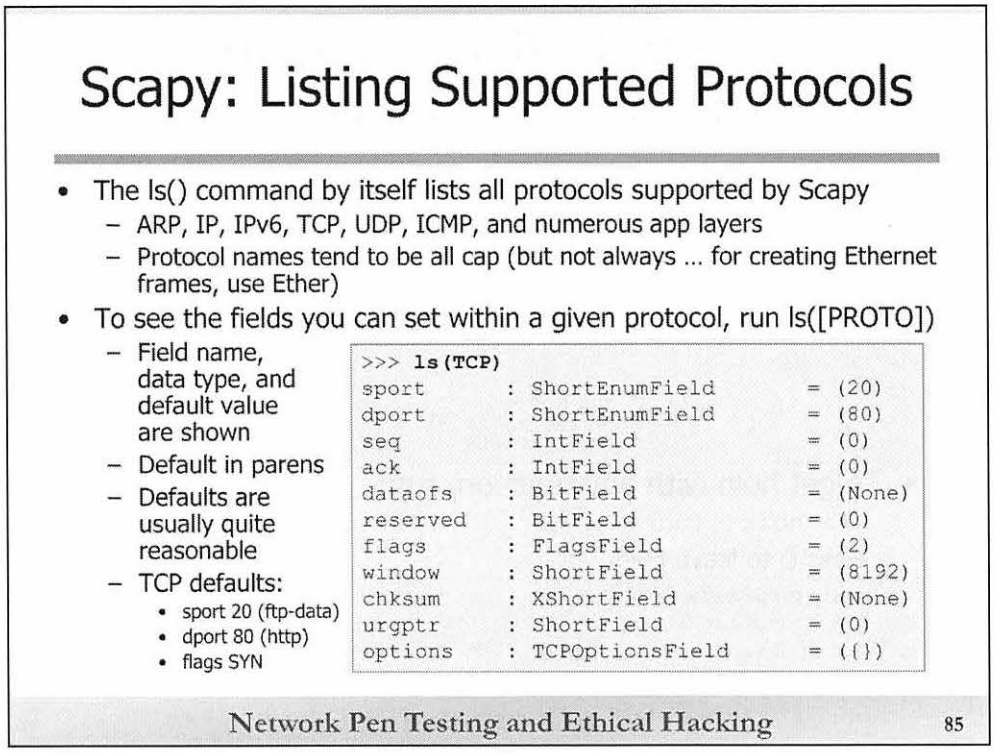
Created by Philippe Biondi, Scapy is an environment based on the Python programming language. Users invoke Scapy to get an interactive Python prompt (>>>). Or you can invoke a Python script (typically with a filename suffix of .py) which can call Scapy features from the script. To craft packets with Scapy, you 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 with UID 0 privileges and then import all the Scapy functionality as shown on this slide.

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, press CTRL-D.



Now jump right into Scapy by looking at the different protocols it supports. You can do this by running the ls() function:

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

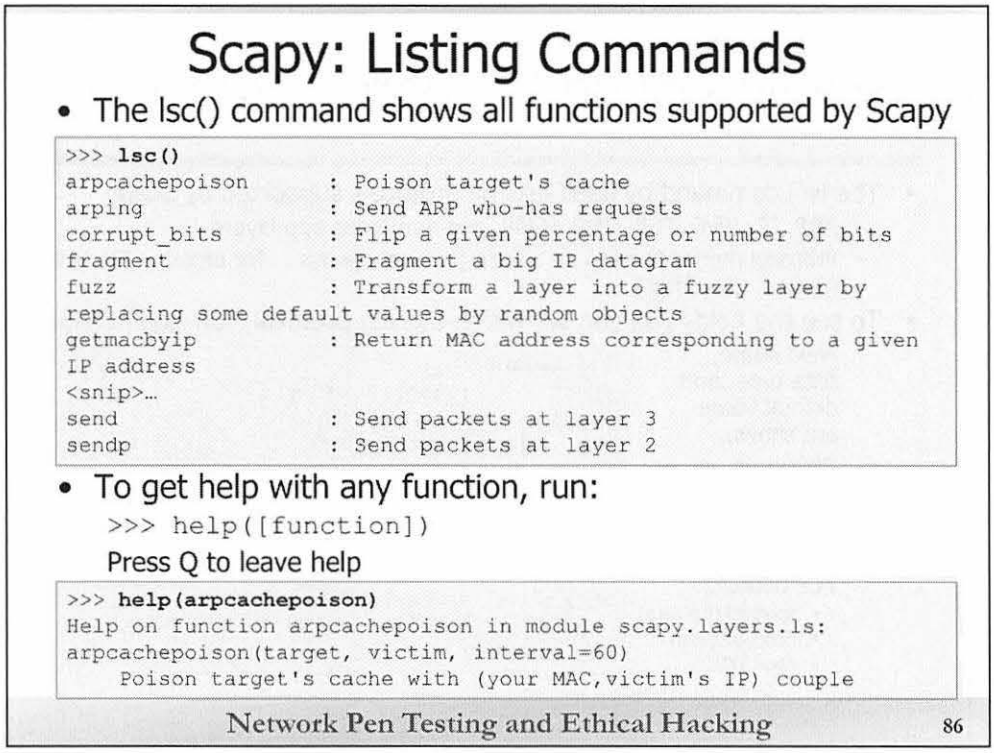
Here, you can see more than 300 different protocols Scapy supports, including application-layer protocols like HTTP, transport protocols such as TCP and UDP, Layer 3 protocols including IP (that's version 4) and IPv6, and data link layer protocols such as Ether (which generates Ethernet frames). Most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

To see the fields available for us to interact within 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 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.



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()
```

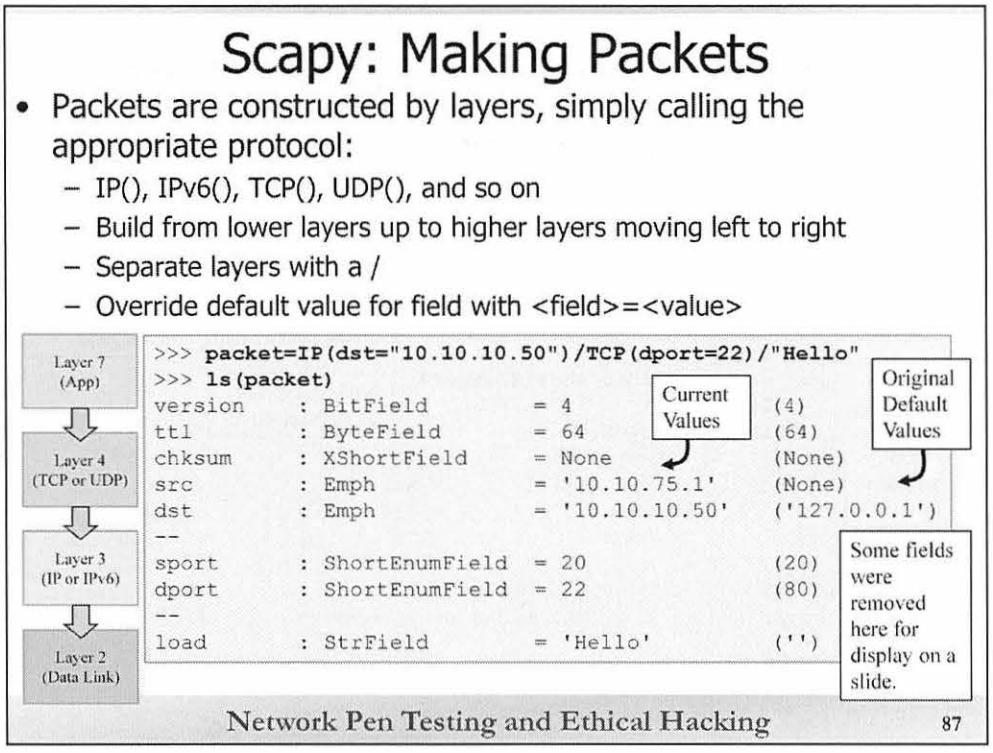
Here, we can see the numerous features of Scapy. Notice that there are numerous attack techniques embedded in Scapy, such as arpcachepoison, a topic we'll touch on in 560.5 when we cover the Cain tool. There are also techniques for sending packets. (send, sendp, sr, and sr1, among other things, fall 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, you 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 press the Q key.



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 want inside of parentheses, as in IP(dst="10.10.10.50"). We can build multilayer 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 to construct Layer 2.

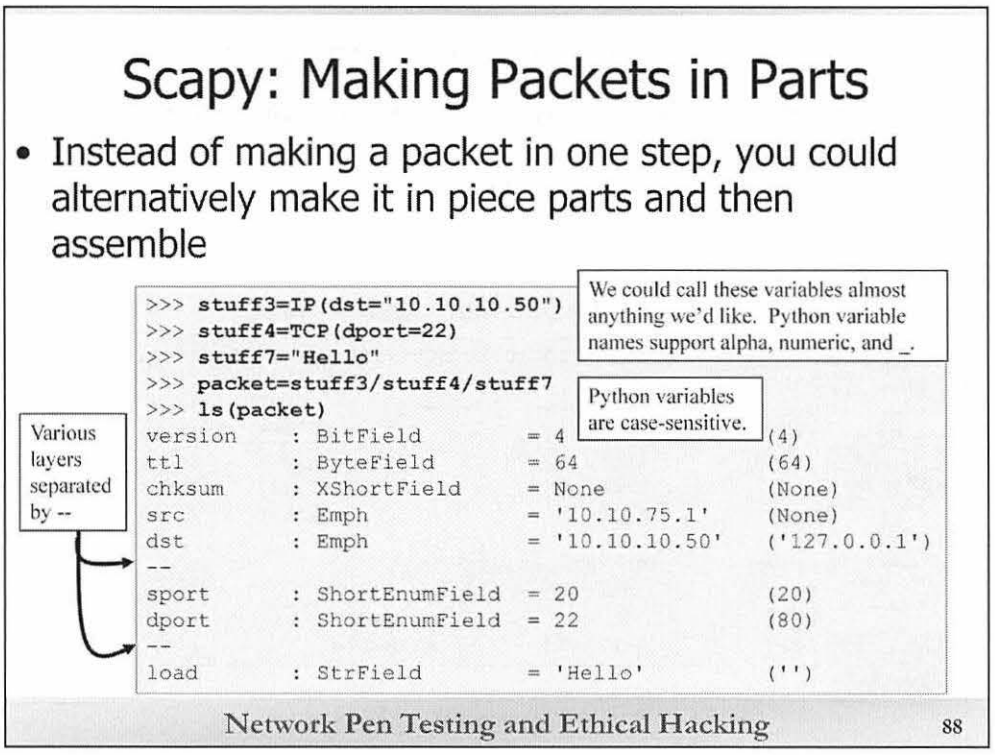
For example, we can create a packet by specifying Ether()/IPv6()/TCP()/"Application Data". If you don't need anything special for your Ethernet frame (such as spoofed MAC addresses), leave off the Ether() upfront, and it will be taken care of for you.

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 its details using the ls command on the packet. The output of ls shows 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.

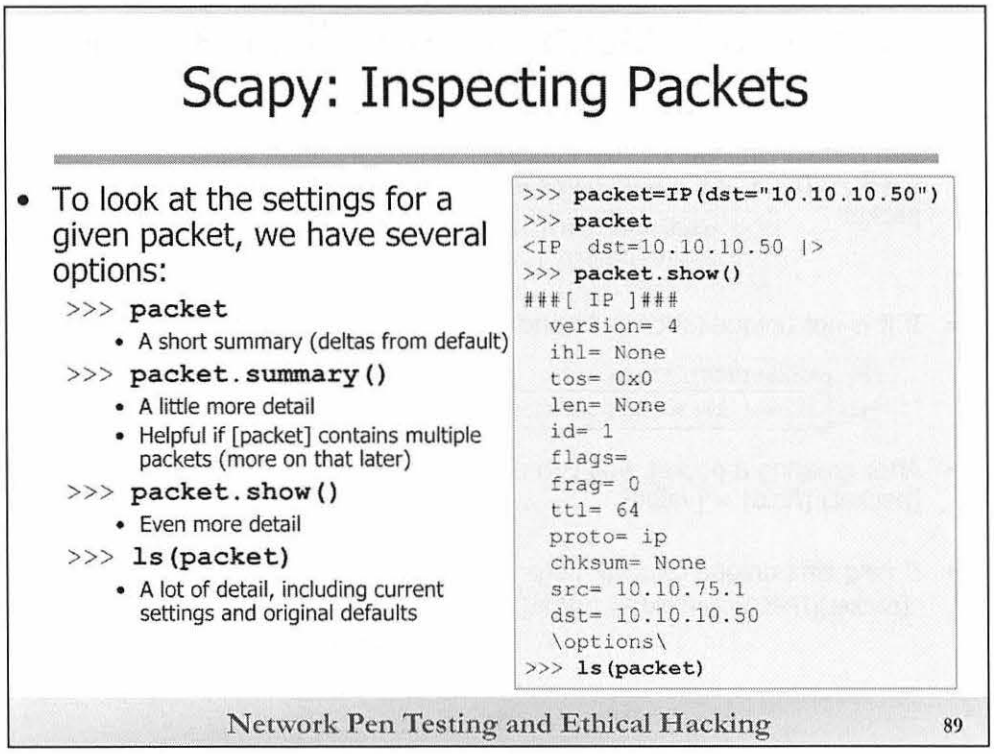
Now, let's take each of our layers and stuff them all together into a single packet:

```
>>> 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.



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 helpful because it displays some of the most interesting aspects of the packet information. As you'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.

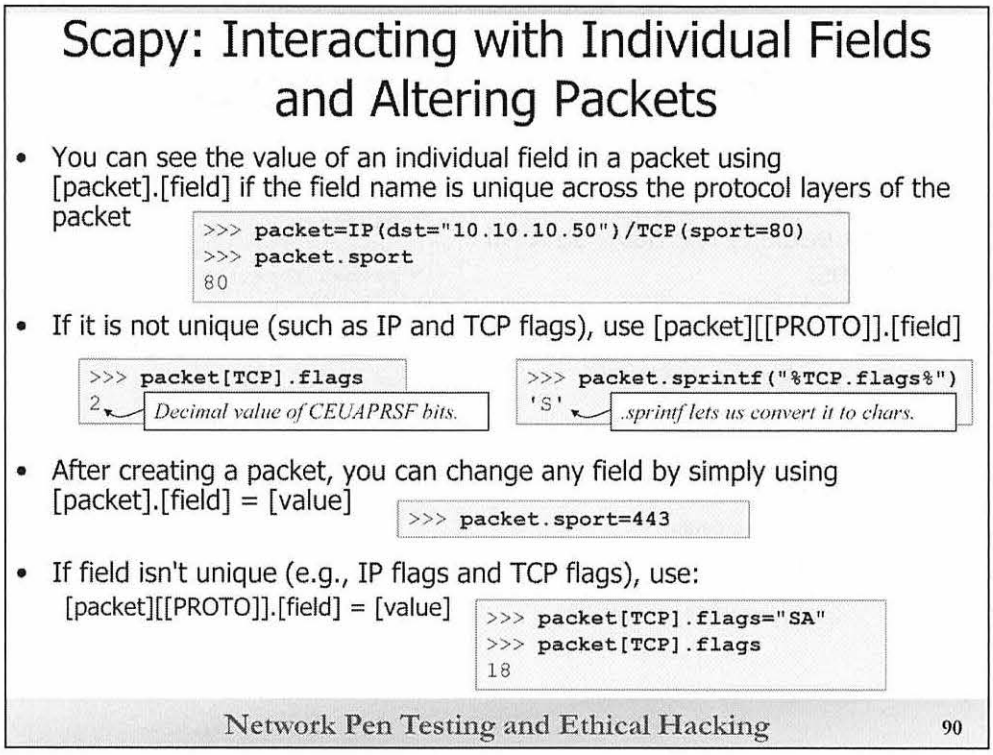
For even more detail, you can call the .show() method:

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

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

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

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



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, and such). 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 want the TCP flags? We have a couple 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 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. Or if we want to see a string indicating the Control Bits, we could use Scapy's capability to use .sprintf as follows:

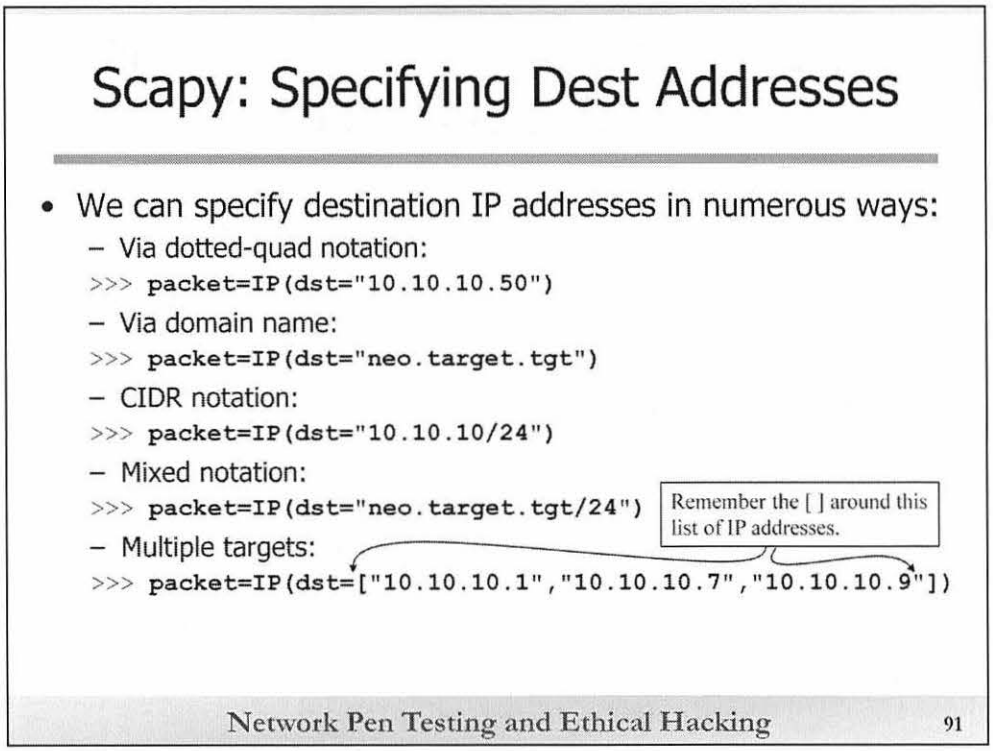
```
>>> packet.sprintf("%TCP.flags%")
'S'
```

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 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 causes 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")
```

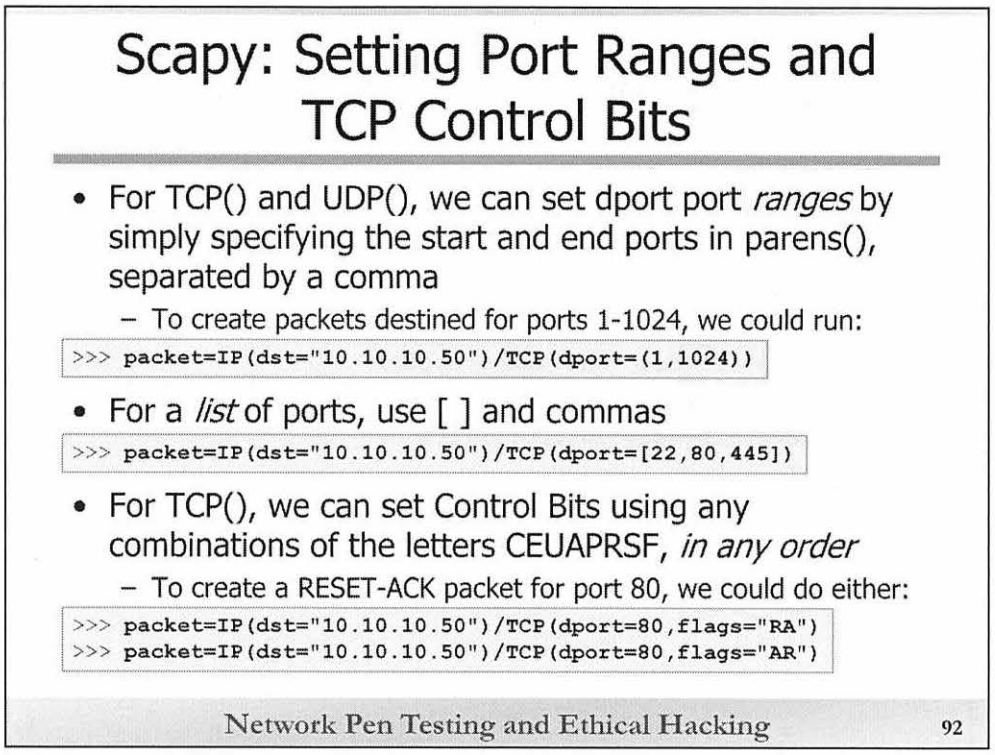
Here, we're starting to see how Scapy can take one packet structure we define and send it to multiple targets. This formulation would send the packet to every IP address on the 10.10.10 subnet.

Scapy also includes a nifty mixed notation, which uses domain names and CIDR formulations. The following causes 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"])
```



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:

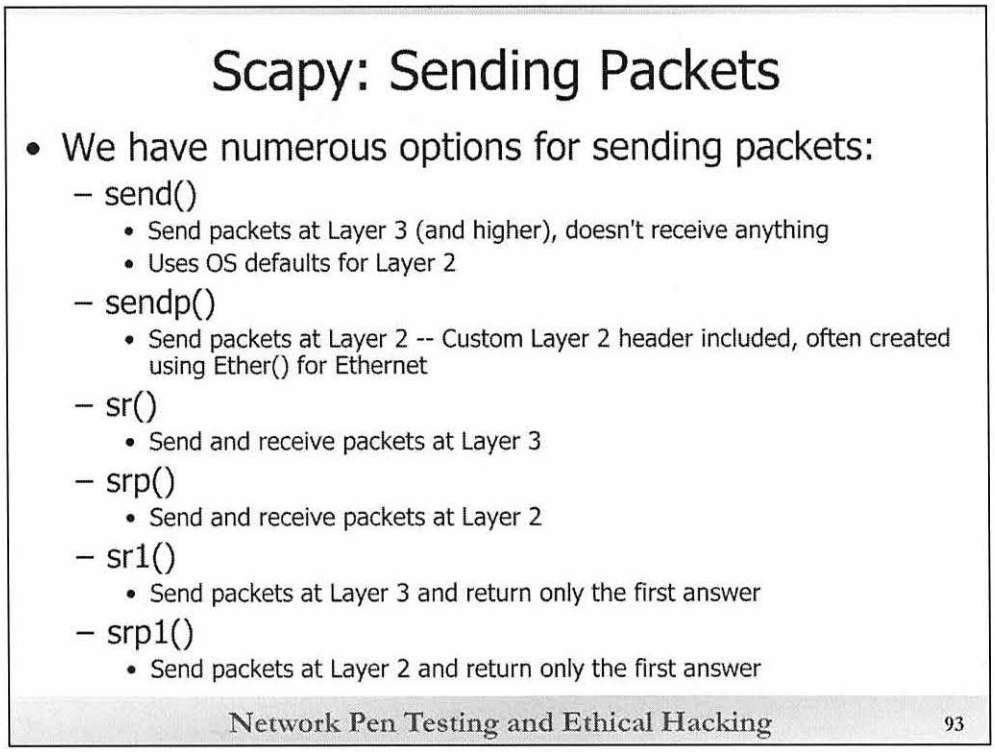
```
>>> packet=IP(dst="10.10.10.50")/TCP(dport=[22,80,445])
```

As you saw earlier, you can specify TCP Control Bits using the appropriate letters from CEUAPRSF depending on the Control Bit combinations you want to set. It is important to note that you can specify these Control Bits in any order that you choose so that you 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")
```



So, we've spent all this time creating packet data structures, but they are 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 for all the Layer 2 frame elements.

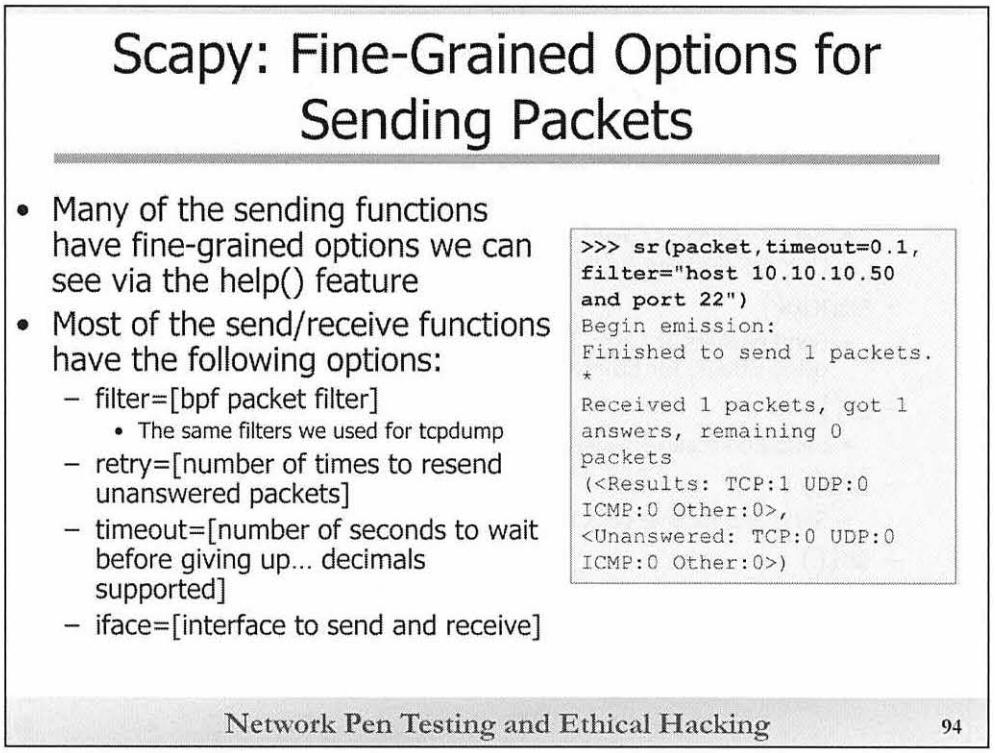
The sendp() function is used if you have crafted a Layer 2 header (as well as higher layer headers and payloads) for the packet you want to send. This function sends your packet without waiting for a response. The Layer 2 header is often constructed using Ether() for Ethernet.

The sr() function sends your packet and waits to receive responses from the target. Like send(), this function sends packets without custom Layer 2 frames, instead of 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.



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. To see these options and how they apply to each send function, remember to call help([function]).

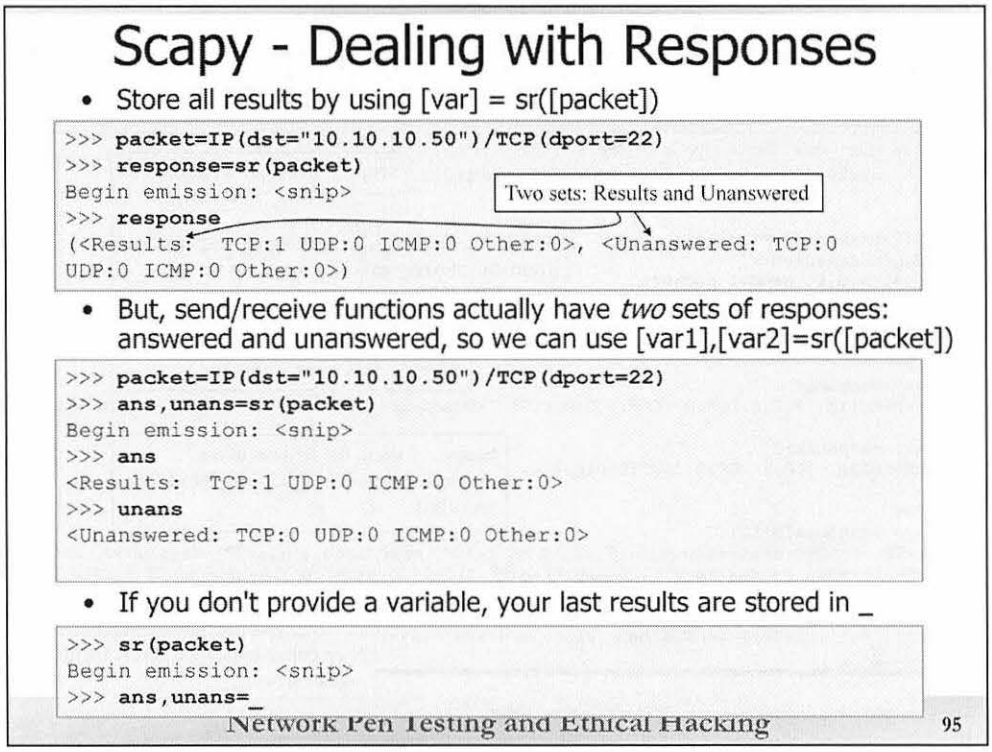
Some of the most useful of these options in sending packets include

- **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 re-send 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 one-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 wait only 0.1 seconds, and we want to receive only answers that match the filter of host 10.10.10.50 AND port 22. (That is, packets must involve 10.10.10.50 to or from port 22, or else we'll ignore them.)



When we use a send/receive function call, such as 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.

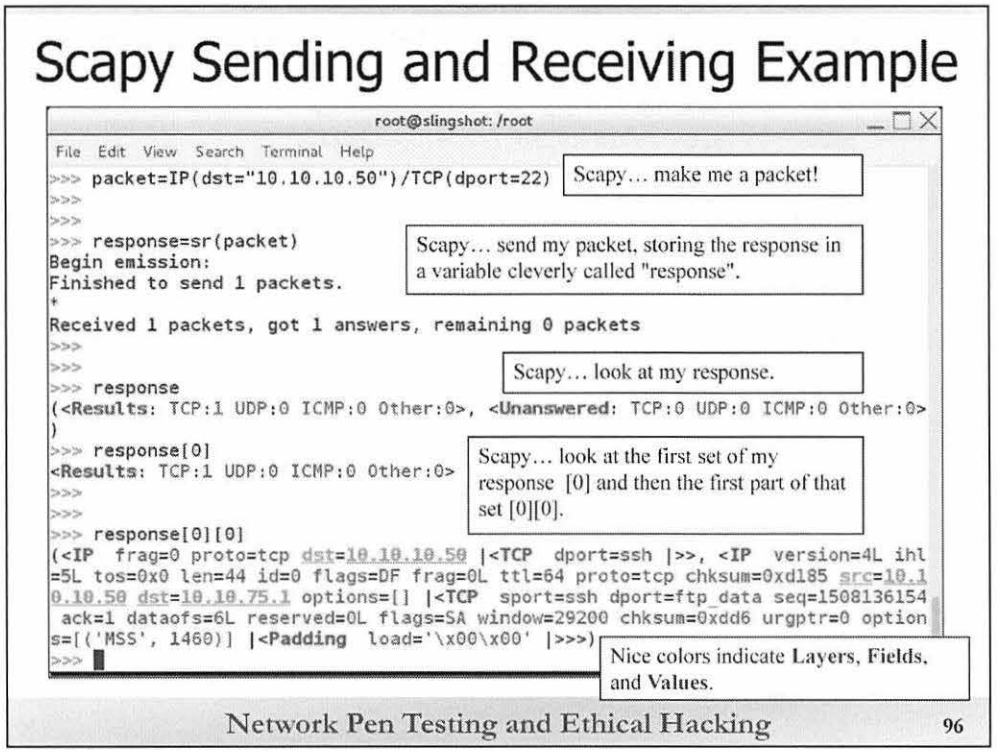
Often, it is useful to separate out the answered responses from the unanswered responses. We can do that using:

```
>>> 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 important to note that if you call a function to send a packet but don't provide a variable name to store the results (that is, 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 helpful because sometimes you get ahead of yourself, building packets and sending them quickly, without remembering to store your results in a variable. After you send 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=_
```



Now 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 one packet back. We can look at our response data structure using:

```
>>> response
```

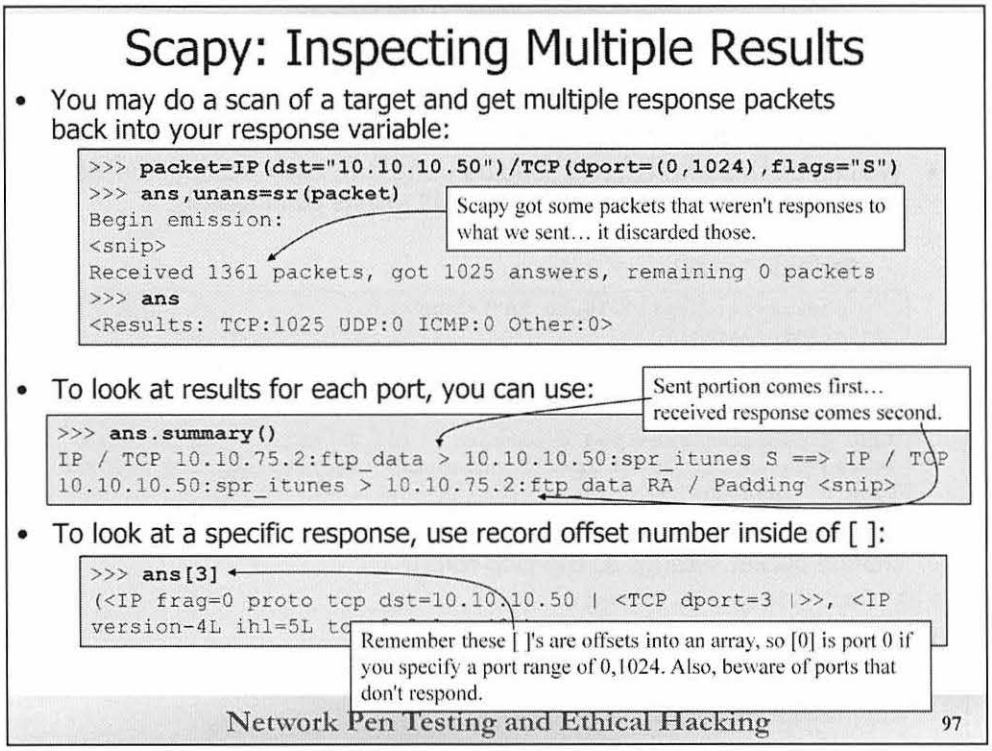
Here, we see that our response has two components: Results, which includes 1 TCP packet, and Unanswered, which doesn't have any packets in it. To look at the first component of our response data structure (that is, the "Results" piece), we can look at the first component (offset of 0) in that structure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



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 0 through 1024 on target 10.10.10.50:

```
>>> packet=IP(dst="10.10.10.50")/TCP(dport=(0,1024),flags="S")
>>> ans,unans=sr(packet)
```

First, note on the slide that Scapy says that it received 1361 packets and got 1025 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 1025 of them were responses to packets Scapy sent. The other packets beyond the 1025 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 1025 TCP packets in response. Great! But, what are they? All open ports? That is 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 [3]. (Remember our array range starts at 0 for port 0, so [3] is the result for port 3.)

```
>>> ans[3]
```

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 and Ethical Hacking 98

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)
```

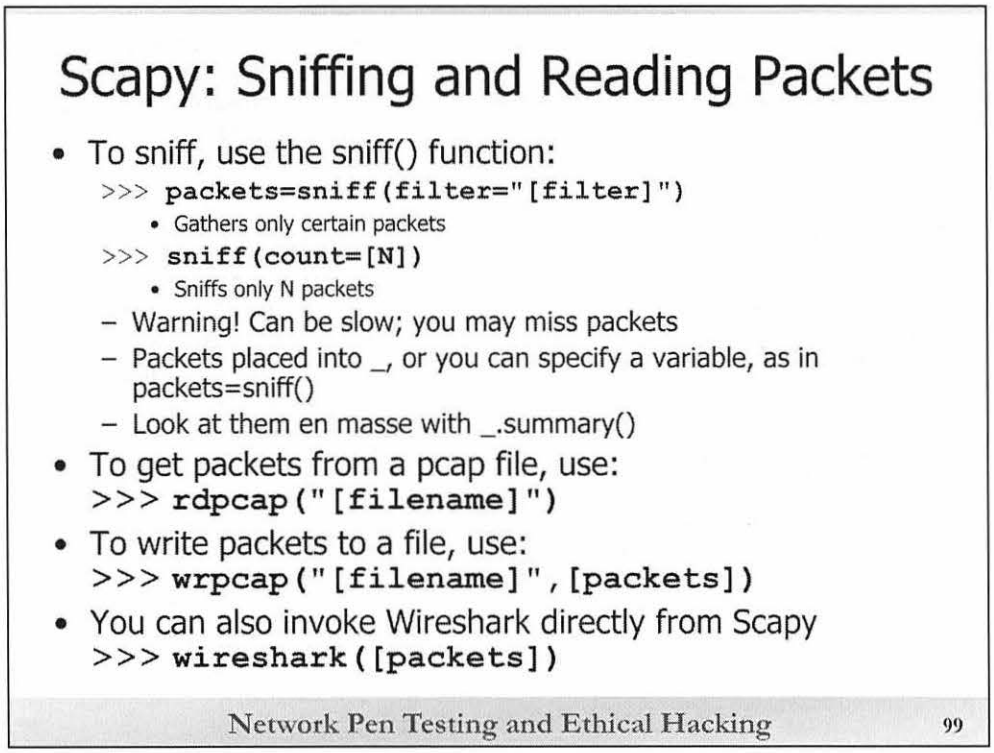
We can see the results here with RECV displayed directly on the screen, showing the result of our ICMP echo request packet going to target 10.10.10.50. This srloop feature is similar to the behavior of the hping command, a packet crafting tool for Linux, Windows, and Mac OS X that isn't nearly as flexible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



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

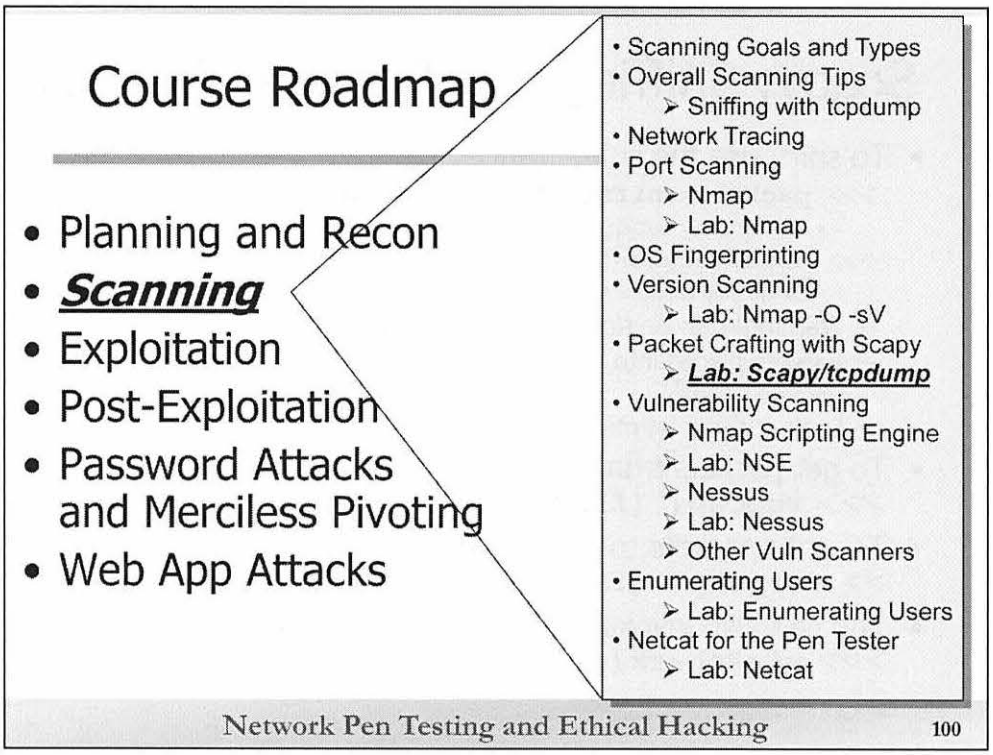
As you might expect, by default, packets grabbed by sniff() are placed into _, or you can put them into a given variable using [var]=sniff().

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 filename 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.



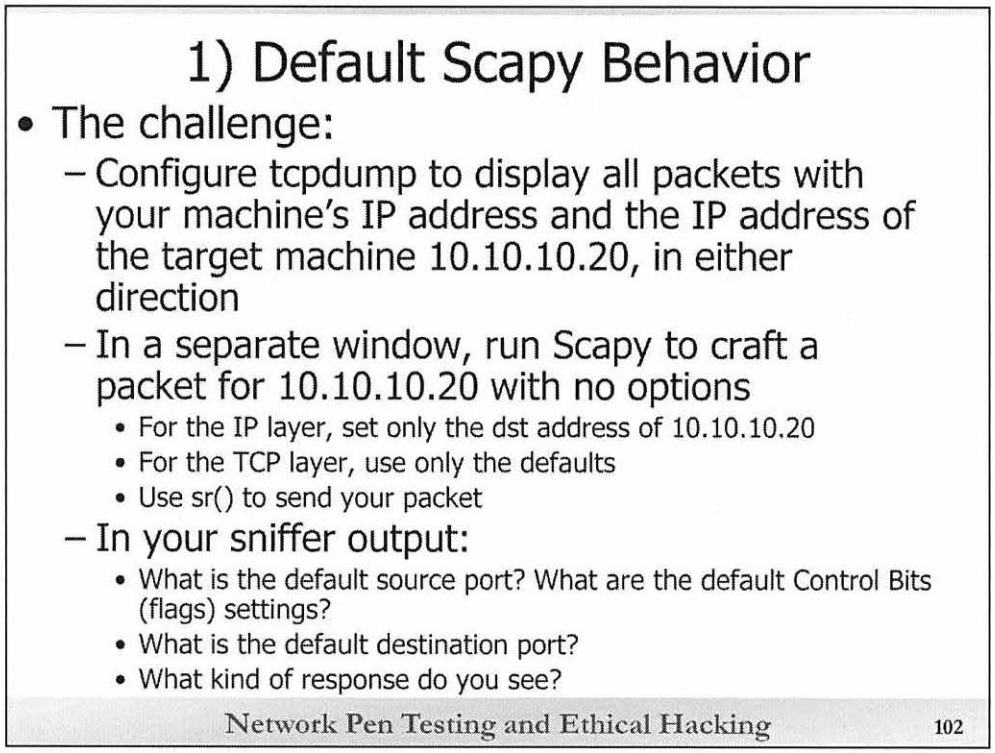
Next, let's do some hands-on labs with Scapy and tcpdump. These labs take the form of some challenges. We pose to you a scenario with packets you need to formulate with Scapy and tcpdump configurations you need to write to see these packets. All the answers to the challenges are included on the page 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, you can look at the next page for some hints.

Lab: Scapy and tcpdump

- We experiment with tcpdump and Scapy ...
- ... Honing our skills to formulate packets and get responses:
  1. Sniffing default behavior of Scapy
  2. Looking at ICMP payload behavior
  3. Crafting Land packets
  4. Analyzing unusual port behavior on a target machine

We are now going to perform a lab to hone our skills in using both tcpdump and Scapy. We specify certain tcpdump configurations that look for packets with specific settings. Then, we 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 lab components we 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.



We start 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 generate 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 the defaults for the TCP layer.

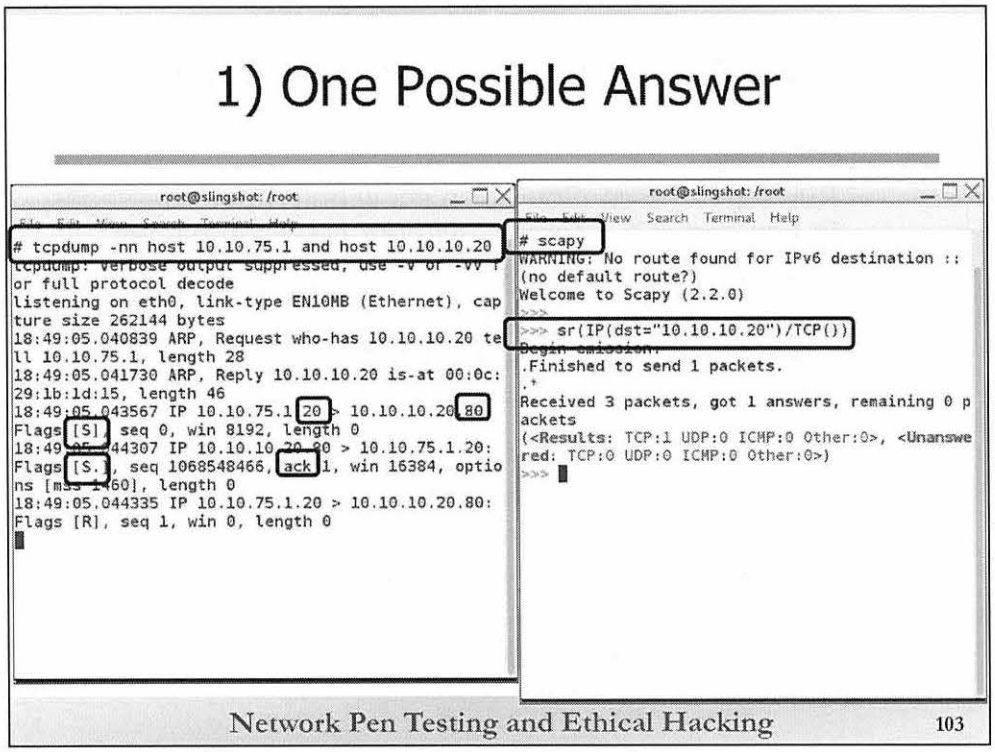
If your tcpdump has been configured appropriately, you should start seeing packets on its output. Based on those packets, discern the answer to the following questions:

What is the default source port? 20 What is the default Control Bits (that is, TCP flags) setting? S, R, A

What is the default destination port? 80

What kind of response do you see? RST

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



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

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

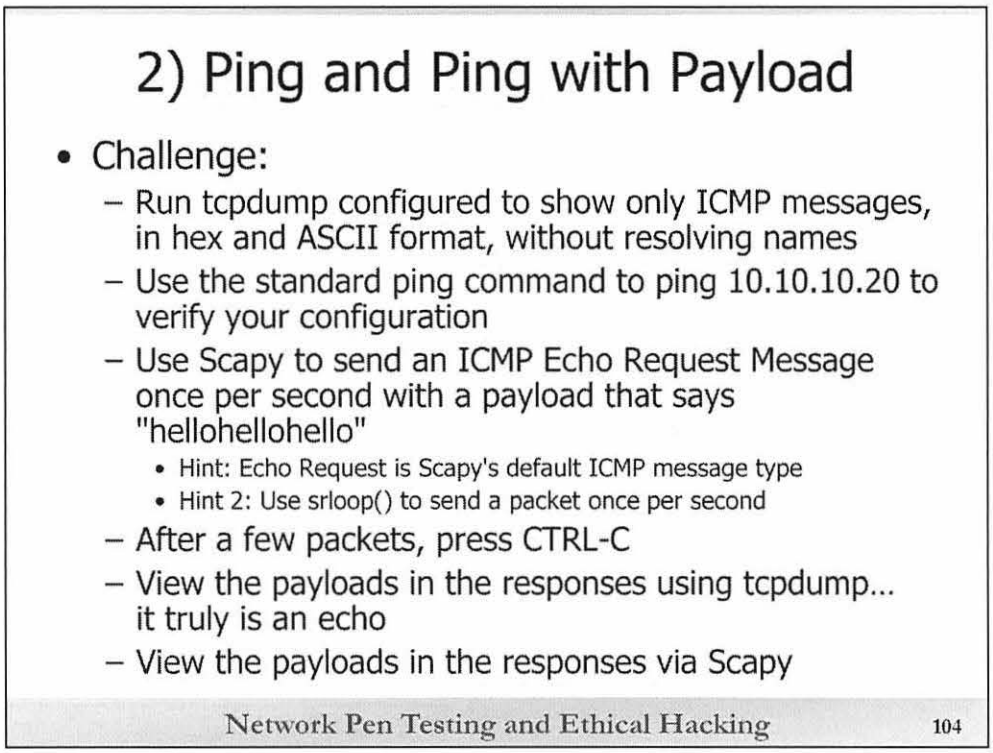
Note that you should replace [YourLinuxIPaddr] with the IP address you assigned to your Linux machine. This syntax makes tcpdump look for packets that are associated with both hosts [YourLinuxIPaddr] and 10.10.10.20, while not looking up their names or services (-nn).

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

In tcpdump, you 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.



For the next lab, you 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 captures 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:

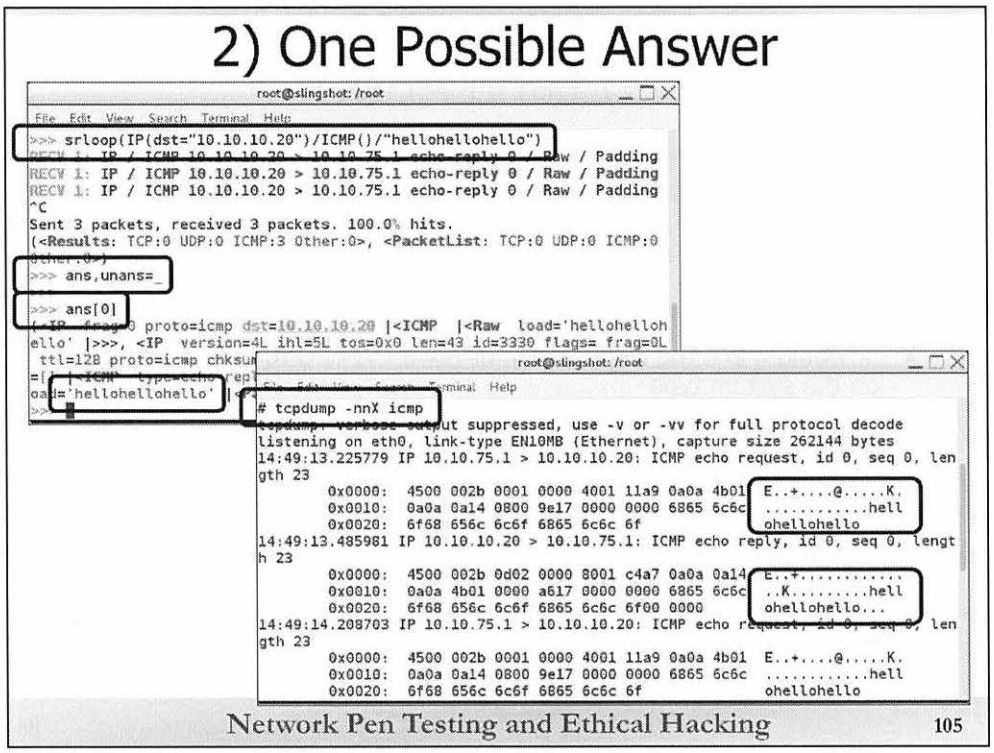
```
# ping 10.10.10.20
```

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 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 want only 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 makes 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.

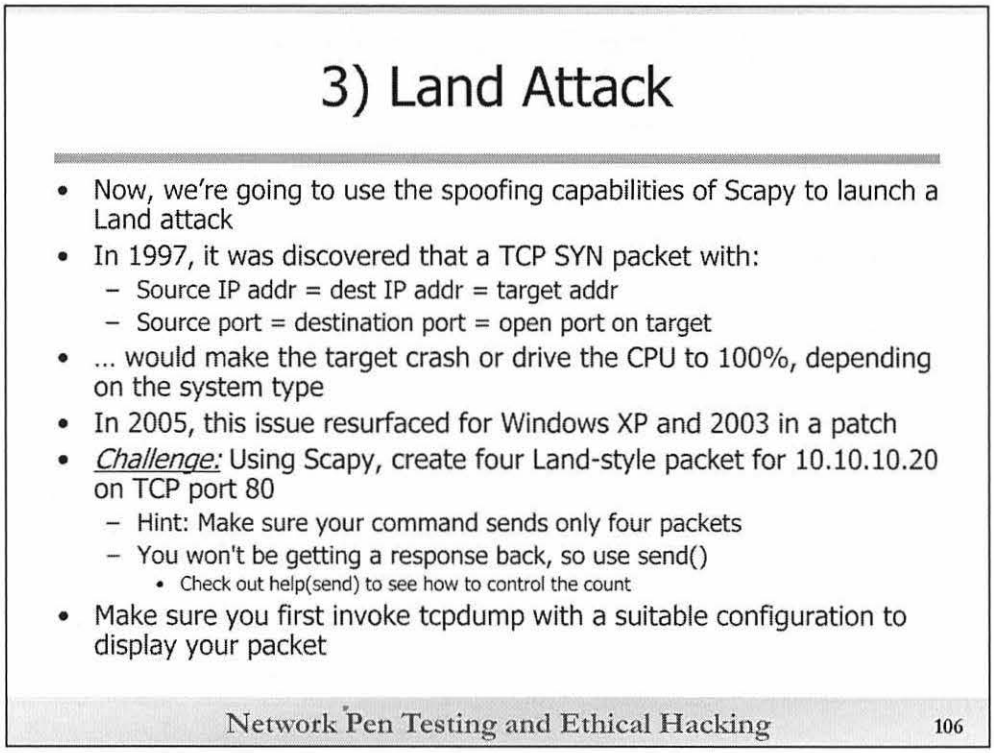
Press CTRL-C in both windows to get your command prompt back. Now, in the Scapy window, let's analyze our result. First, we'll store it into ans and unans:

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

Now, look at your first response:

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

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

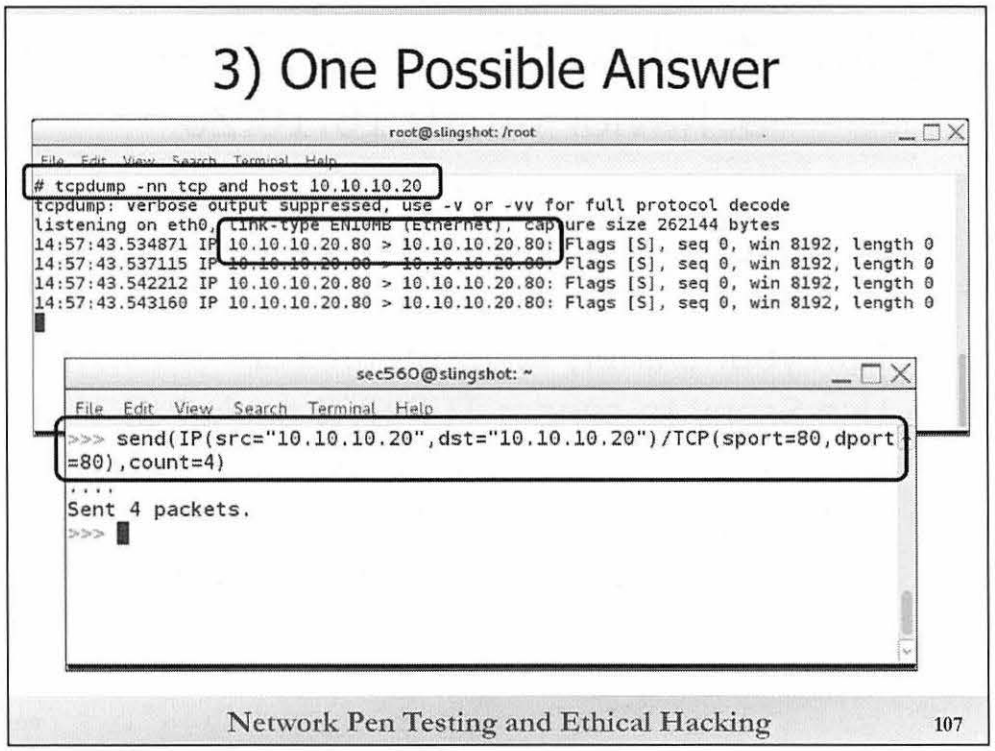


For our next lab component, we use Scapy to generate a Land attack. Way back in 1997, a security researcher discovered that if you send a machine a spoofed TCP SYN packet to an open port with the source IP address set to the same value as the destination IP address, and the source port the same value 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 in which it would look like it received a packet from itself, going in the same port that it is leaving, causing significant problems. 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 re-creating the Land attack.

Your challenge is to use Scapy to send 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(s) 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.

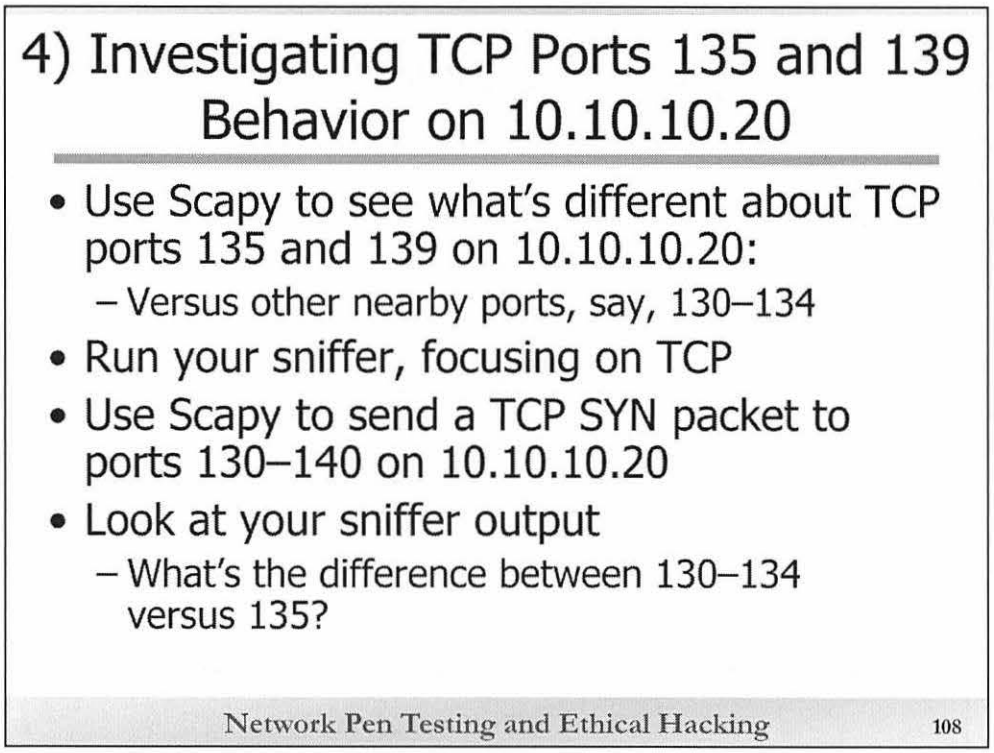
We can capture our packets with a tcpdump command line that doesn't lookup names (-nn) but does show all TCP packets (tcp) that are also (and) going to or from the target IP address (host 10.10.10.20). You could narrow this down further by specifying particular ports, but it is often useful to invoke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



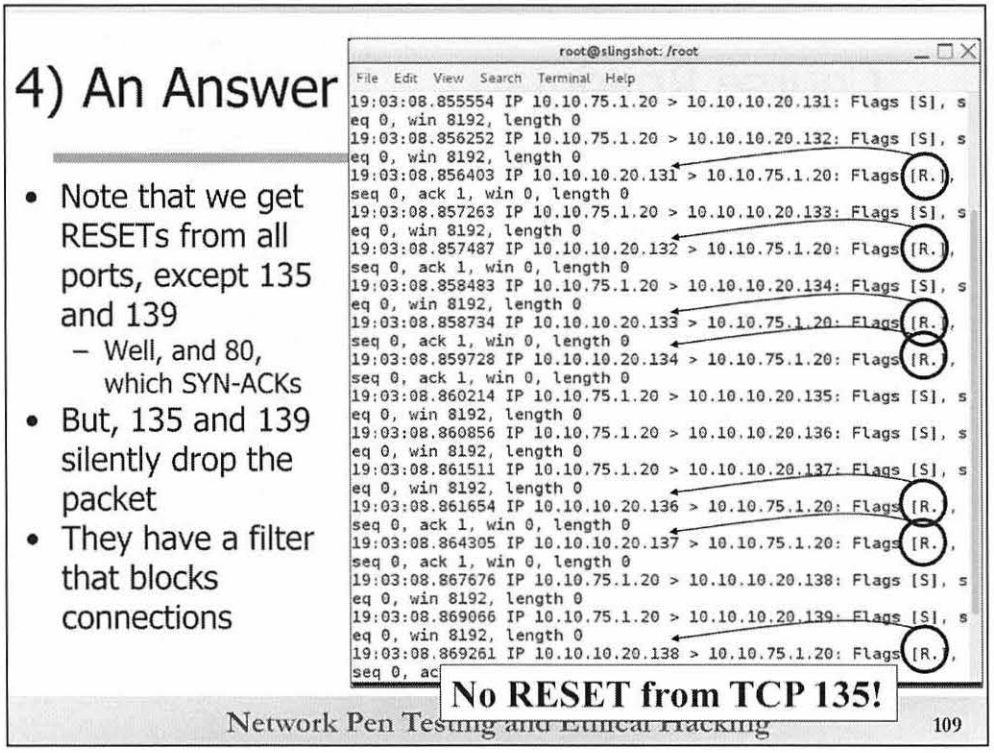
In our earlier Nmap fingerprinting lab, we saw some unusual results from TCP ports 135 and 139 on 10.10.10.20. Nmap didn't list this port as closed but instead indicated that it could not get any data from it. Why did Nmap sense that the port isn't closed, yet it cannot make a connection? Our trusty friends Scapy and tcpdump will help us find out.

First, run tcpdump so that it looks 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 parentheses ( ) around your port range for Scapy.

Try to compose these commands yourself, but you can flip to the next slide for one possible answer.



To achieve what we described on the previous slide, we can invoke tcpdump as follows:

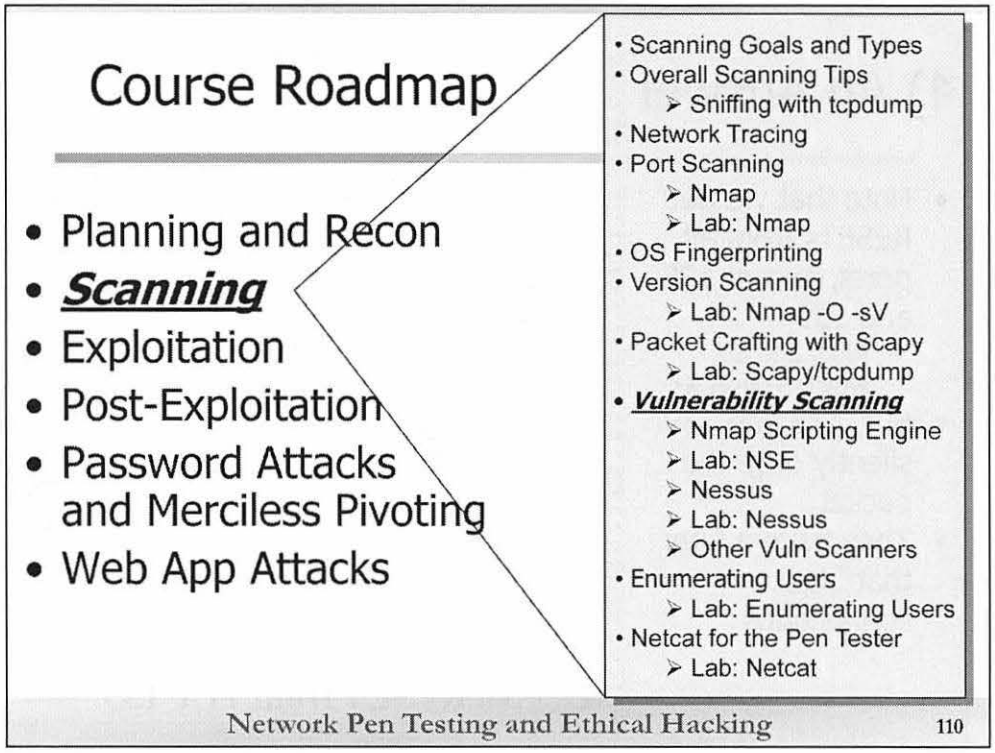
```
# tcpdump -nn tcp and host [YourLinuxIPaddr] and net 10.10.10
```

Then, we can run Scapy as follows:

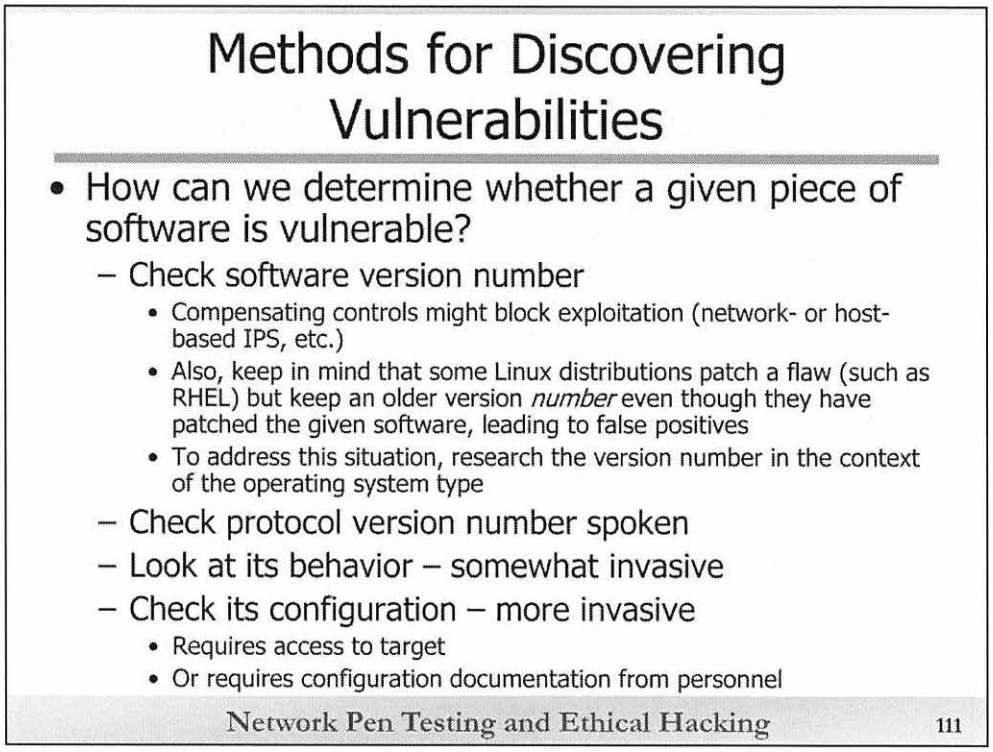
```
# scapy
>>> ans,unans=sr(IP(dst="10.10.10.20")/TCP(dport=(130,140)))
```

Press 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 ports 135 and 139, we don't get anything back. It silently rejects our packet because of a packet filter. 10.10.10.20 is configured to filter out all packets destined for TCP port 135 and 139, whereas all other ports simply send a reset.

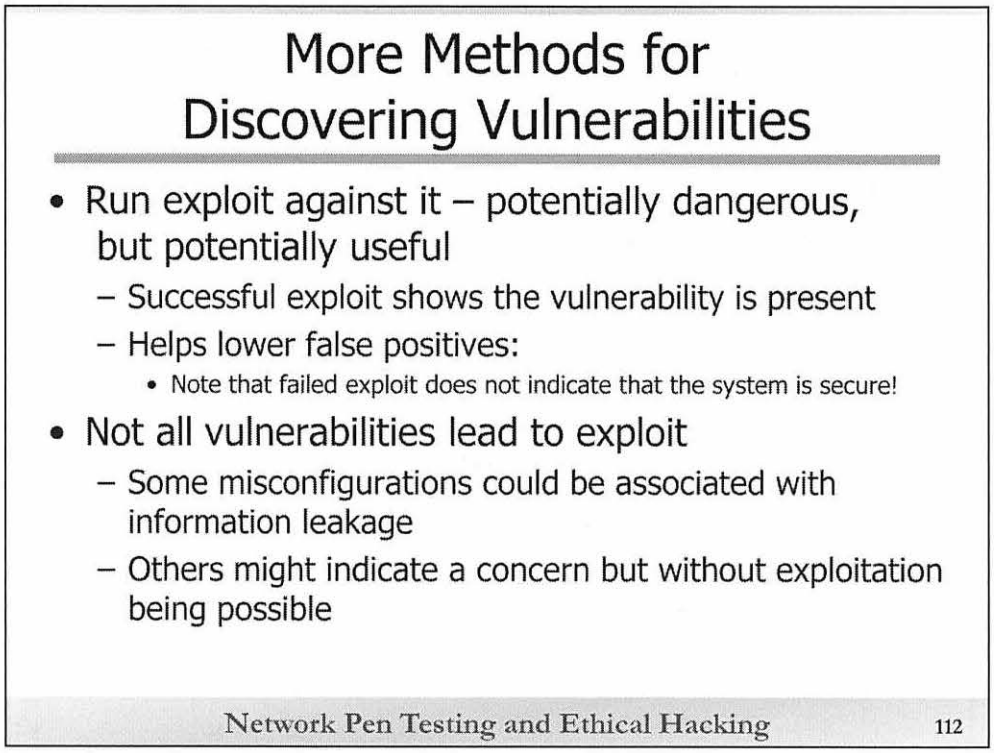


Now, we have reached the topic of Vulnerability Scanning. The goal of these kinds of scans is to find potential security flaws in the target environment. Discovering misconfigurations, unpatched services, architectural mistakes, and more are what this component of our test is all about.



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:

- **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. Also, keep in mind that some Linux distributions release a patch to fix a vulnerability for a given installed package, but such distributions sometimes still keep an older version *number* for that software, leading to false positives. To address this situation, penetration testers should research the version number in the context of the operating system type before reporting a flaw discovered solely based on version number. This situation is especially common with Red Hat Enterprise Linux (RHEL).
- **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 determine that it speaks an older version of a network protocol, possibly indicating that it hasn't been patched or hardened.
- **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 because the tester has to interact with the target in various ways.
- **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 preceding options, as they requires the tester to gain access to a target or get a copy of the system configuration from an administrator.

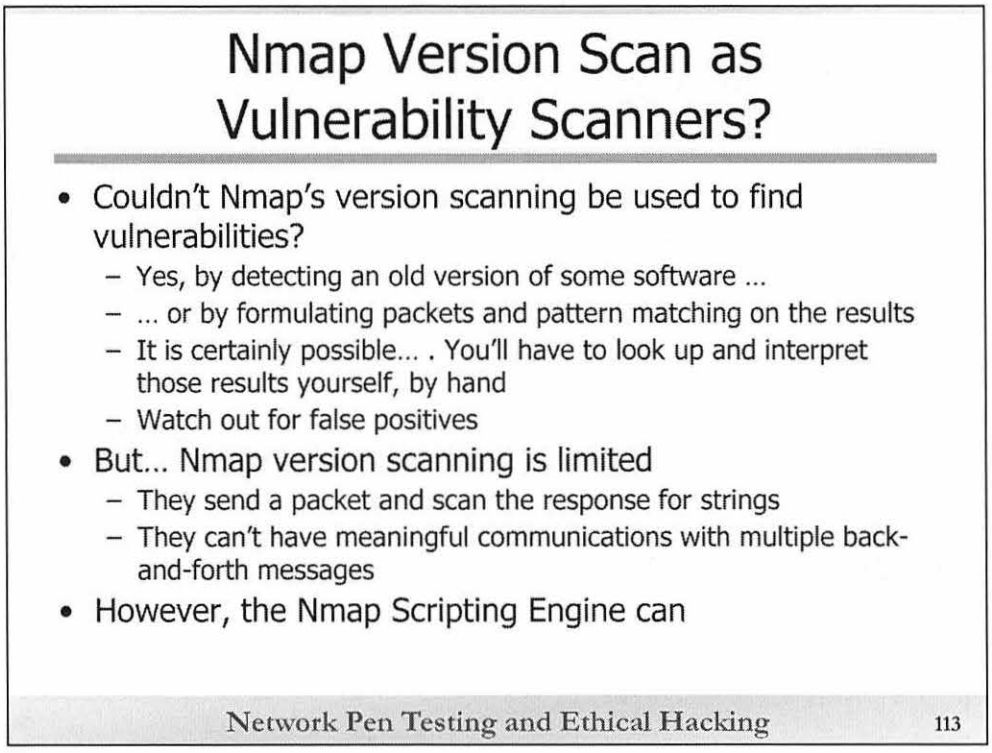


There is another method for finding vulnerabilities:

- **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 because it could bring down the target service or entire system. But running an exploit can be helpful for testers because 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 get it working. So, actual exploitation can lower the number of false positives, but it doesn't help us manage false negatives.

We analyze this issue of the safety of exploitation in further detail at the start of our 560.3 class.

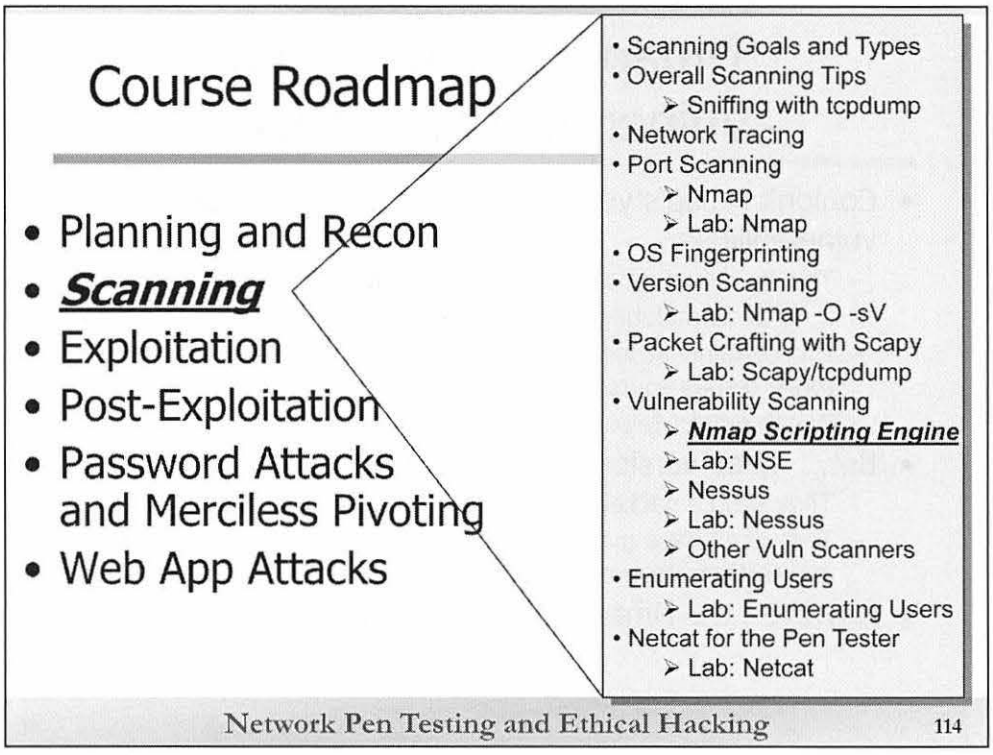
It's also important to note that not all vulnerabilities lead to exploitation. Many vulnerabilities don't let an attacker take over a machine at all. Instead, they could be associated with information leakage or other problems. As penetration testers and ethical hackers, we are interested in all kinds of vulnerabilities in a target environment. Our jobs involve reporting on the issues we discover, regardless whether they can be exploited. Obviously, exploitable vulnerabilities have higher importance than non-exploitable issues, but all discovered flaws should be reported.



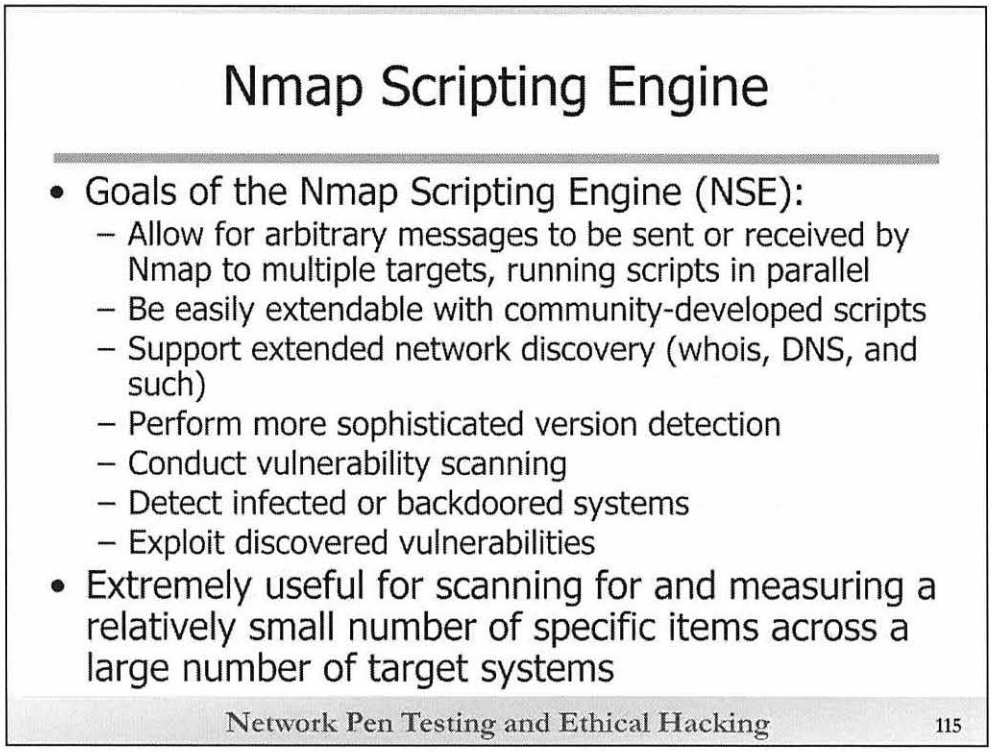
As we have seen, Nmap supports version checking, which sends probe packets to given ports and matches specific strings in the response to determine the version of a service. With that functionality, couldn't we use those tools to find vulnerable systems? We certainly could, by researching the versions 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 does 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 is limited. It sends a probe and scrapes through its response looking for certain text. It doesn'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 may not properly discover subtle vulnerabilities. It might look like a given service is vulnerable based on its version number. However, it's possible that other compensating controls 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).

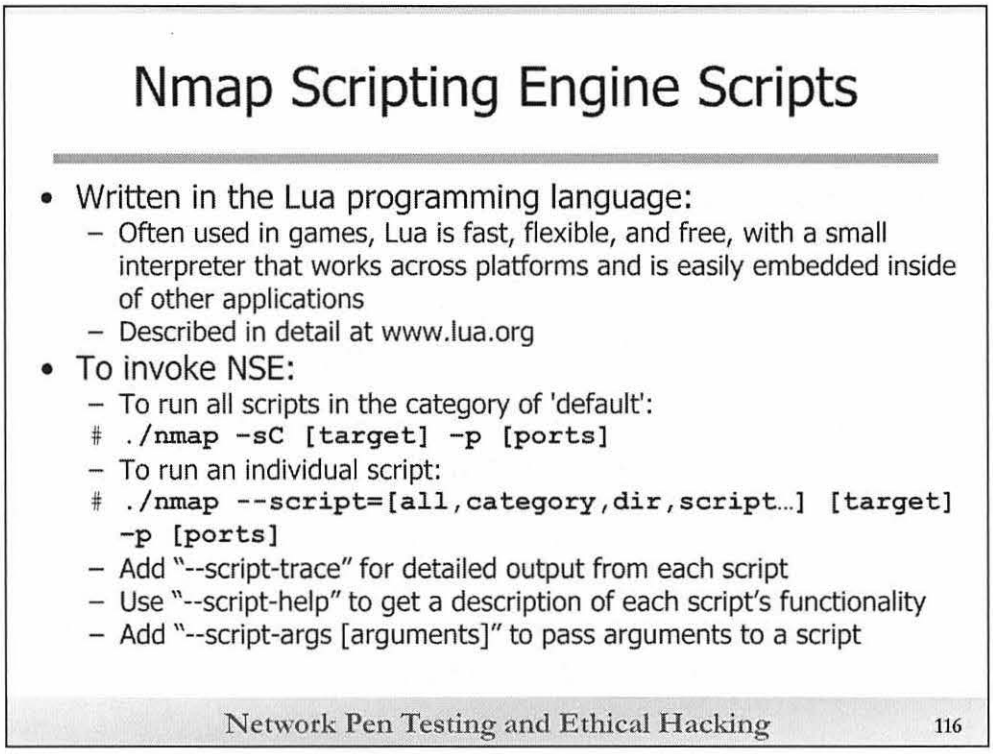


Because the Nmap Scripting Engine (NSE) can discover some vulnerabilities, let's zoom in on its functionality in more detail, running a lab on some of the NSE scripts to see their capabilities.



The Nmap Scripting Engine has numerous goals, which extend the capabilities of Nmap beyond mere port scanning and OS fingerprinting. These goals include:

- Utilize Nmap's efficient multithreaded 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, and so on.
- 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.



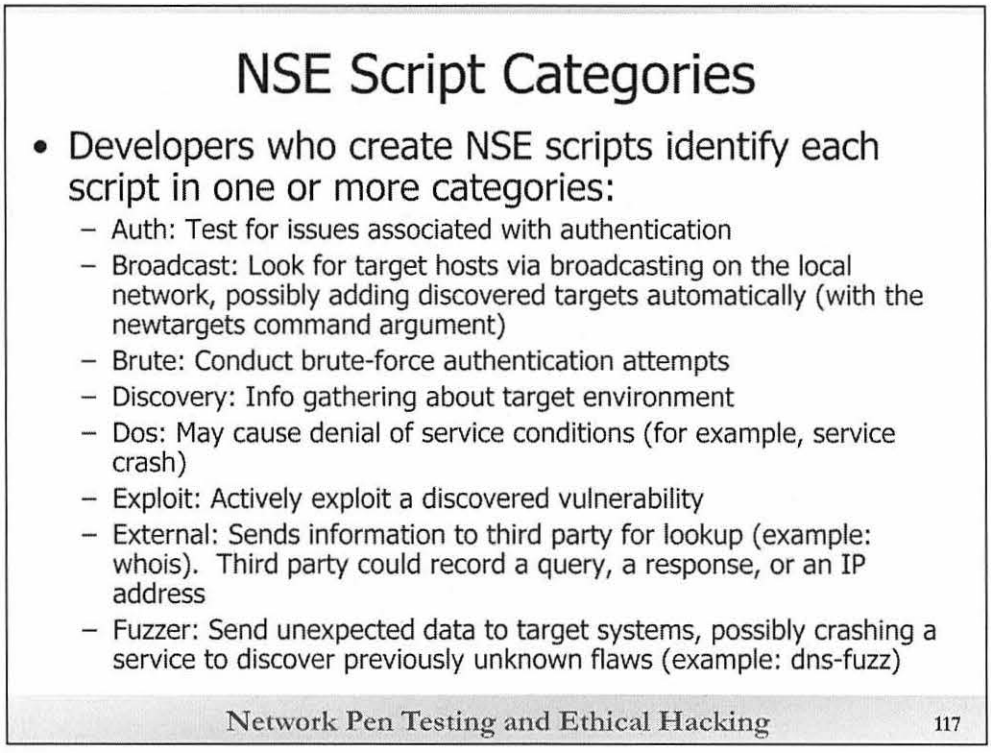
Nmap scripts are written in the Lua scripting language, which is commonly used in computer games. Lua is widely regarded as a flexible and extremely fast scripting environment. It's interpreter is free, cross-platform, and has a small footprint, making it ideal for incorporation into other applications, 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.

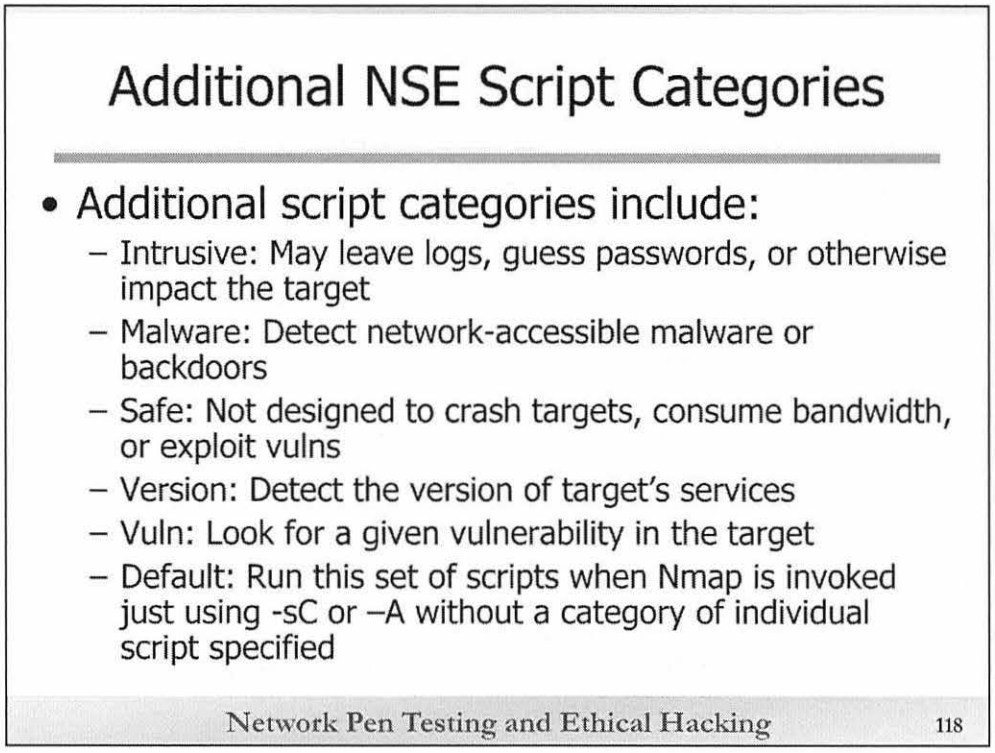
To get details on the function of each script, as well as potential arguments that can be passed to the script, you can add `--script-help` to your command line invocation.

For those scripts that support arguments, you can send the arguments by adding `--script-args [arguments]` to your command line invocation.



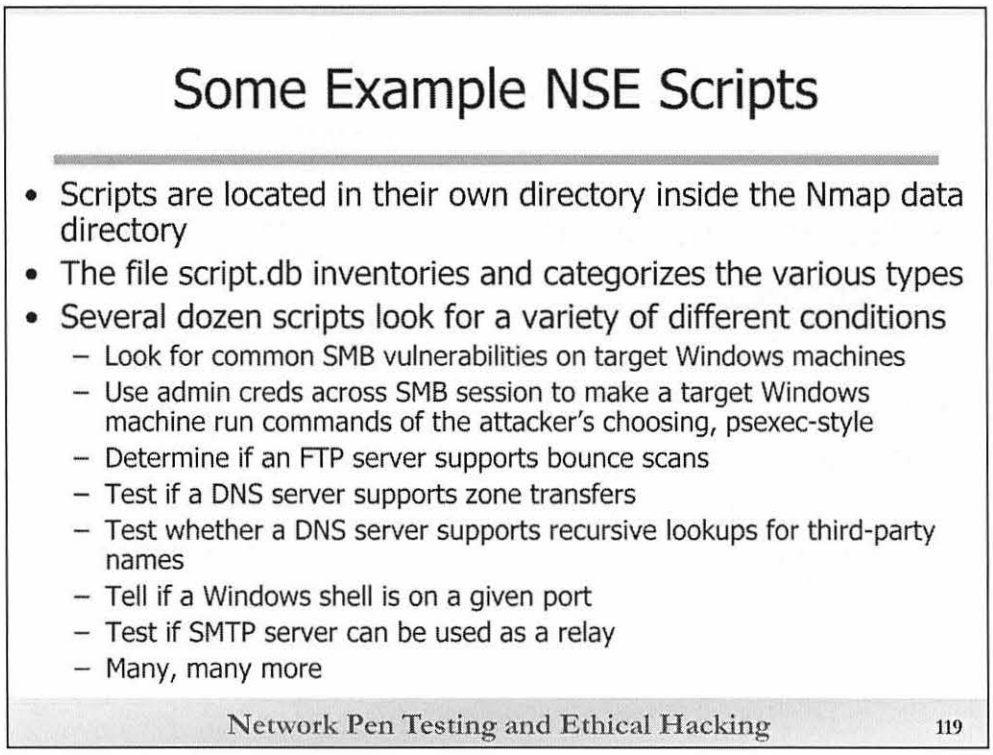
The Nmap Scripting Engine supports several different categories of tests, with each script fitting into one or more script categories.

- The Auth category are tests associated with authentication, including some password guessing and authentication bypass tests.
- The Broadcast scripts send packets on the local network destined for broadcast or multicast addresses to find new targets. If the script is invoked with the "newtargets" argument, these discovered targets automatically be included in the scan.
- The Brute scripts launch brute force authentication guessing attacks and are available for a variety of different protocols, including HTTP, database servers, FTP, and more.
- Discovery scripts determine more information about the network environment associated with the target and include some whois and DNS lookups, among other functions.
- Dos scripts may cause a denial of service condition on the target, possibly by crashing a service.
- Exploit scripts launch an exploit for some discovered vulnerability on a target, perhaps extracting information or even gaining shell on the target.
- 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.
- Fuzzer scripts send unexpected input to a target system to see if a crash condition or other anomaly can be induced. These scripts can be useful in discovering previously unknown vulnerabilities in targets. A classic example NSE fuzzer script is dns-fuzz, which sends malformed DNS requests to a target continuously.



Additional Nmap scripts include:

- The Intrusive scripts may leave logs, guess passwords (which could lock out accounts), consume excessive CPU or bandwidth, crash a target, or have other impacts on the target machines.
- 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 Safe scripts 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 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.
- The Vuln category includes scripts that determine whether a given target has a given security flaw, such as a misconfiguration or an unpatched service.
- 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.



The scripts associated with NSE are found in their own directory called, appropriately enough, scripts, which is located by default in the Nmap directory.

Inside this directory, there is a file called script.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 /opt/nmap-6.4.7/share/nmap/scripts/script.db
```

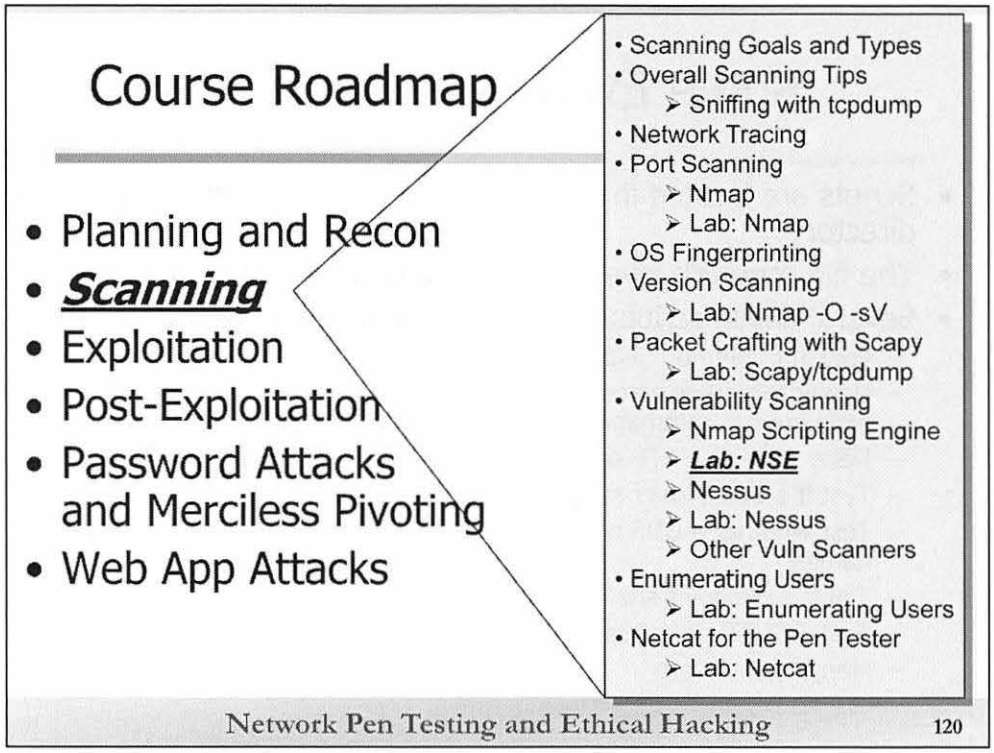
"Intrusive" scripts can be found by with

```
# grep intrusive /opt/nmap-6.4.7/share/nmap/scripts/script.db
```

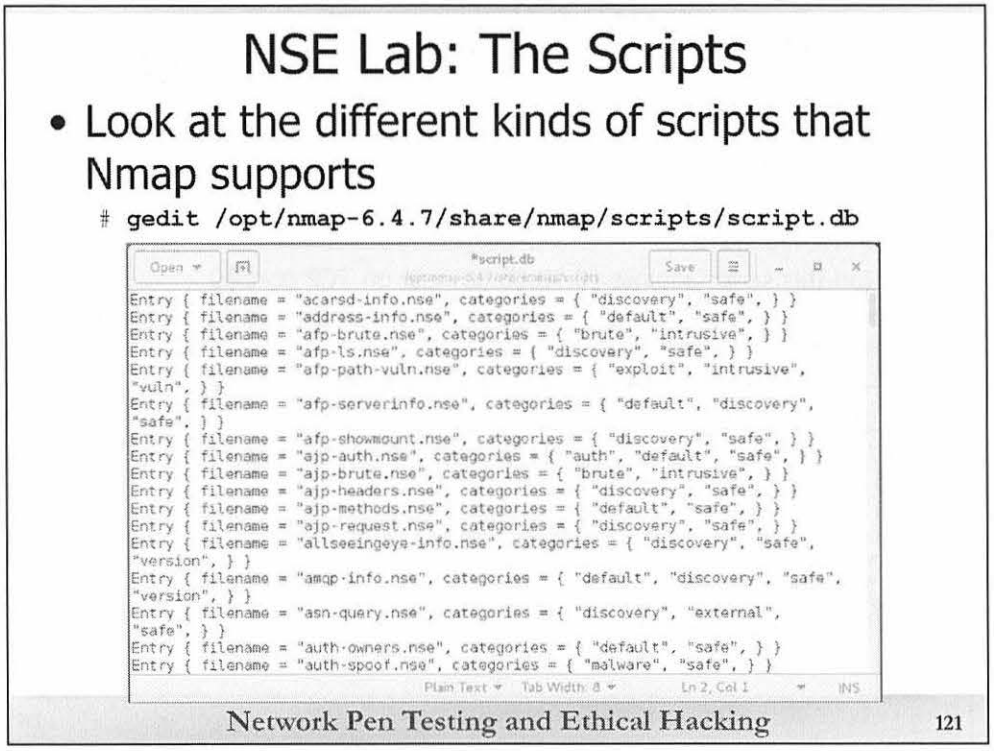
Note that the categories are in all lowercase within this file.

In the scripts directory, there are several dozen scripts. Some of the more interesting include:

- A script to find common SMB vulnerabilities on Windows targets
- A script that takes authentication credentials (such as an admin username and password or admin username and hash) and uses smb to cause a Windows target to run a command, similar in functionality to the Microsoft SysInternals psexec command
- A script to determine if an FTP server supports Nmap bounce scans
- A script that attempts to do a DNS zone transfer from a target
- A script that tries to determine if a target DNS server supports forwarding recursive queries for third-party names for which it isn't authoritative
- 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



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 labs, 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 the scripts that have been defined for NSE:

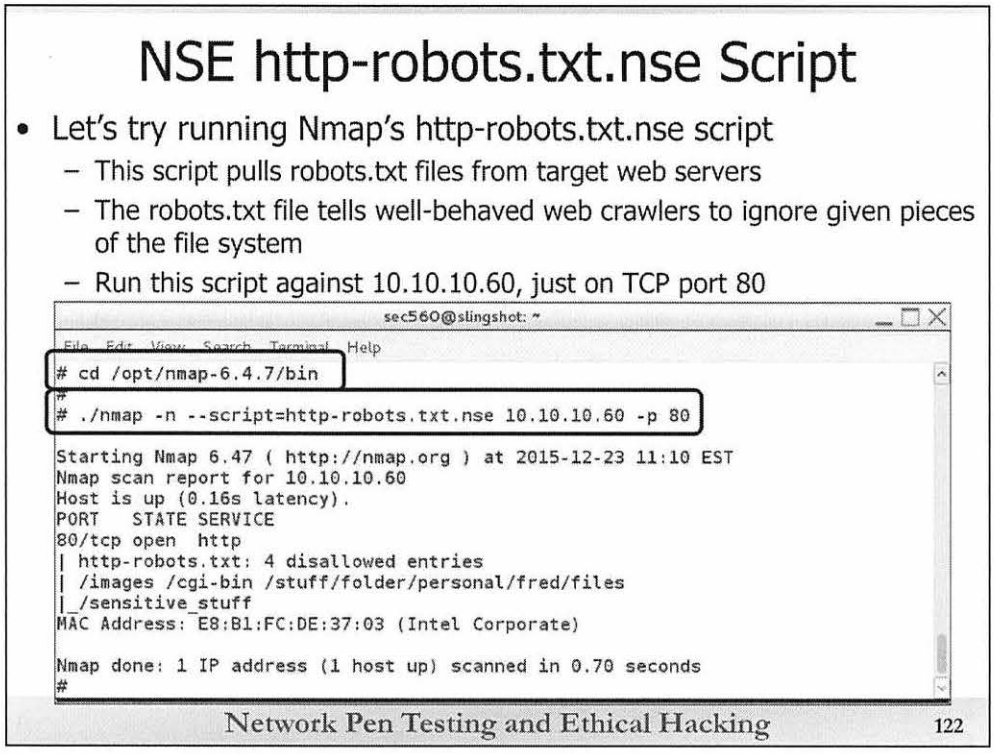
```
# gedit /opt/nmap-6.4.7/share/nmap/scripts/script.db
```

If you don't like gedit, a simple WYSIWIG editor, you can use another Linux/UNIX editor with which you are familiar, such as vi, emacs, nano, and so on.

This script.db file has a 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 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 lowercase L stands for linecount) option:

```
# cd /opt/nmap-6.4.7/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
```



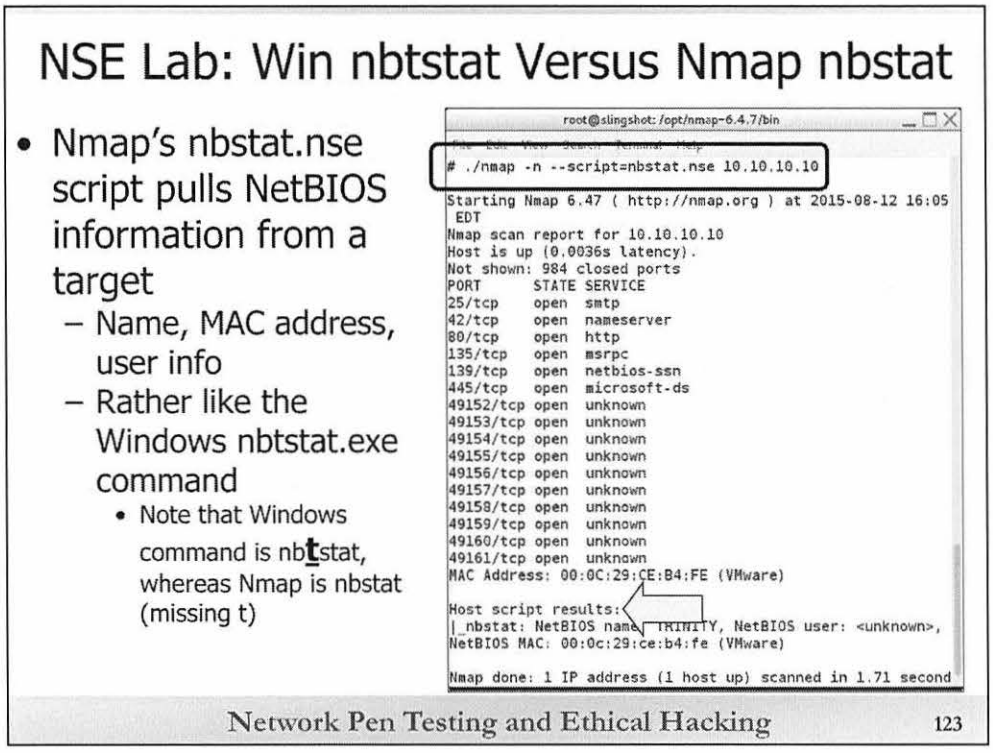
Let's experiment with the http-robots.txt.nse script. This script pulls the robots.txt file from target web servers. The robots.txt file tells well-behaved web crawlers (such as those from the major search engines that are attempting to find new pages on the world wide web) to ignore given directories 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 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 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 website might be located in the file system structure.

The Nmap script http-robots.txt.nse pulls robots.txt files from target machines. Let's test it by invoking it as follows:

```
# cd /opt/nmap-6.4.7/bin
# ./nmap -n --script=http-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 website.

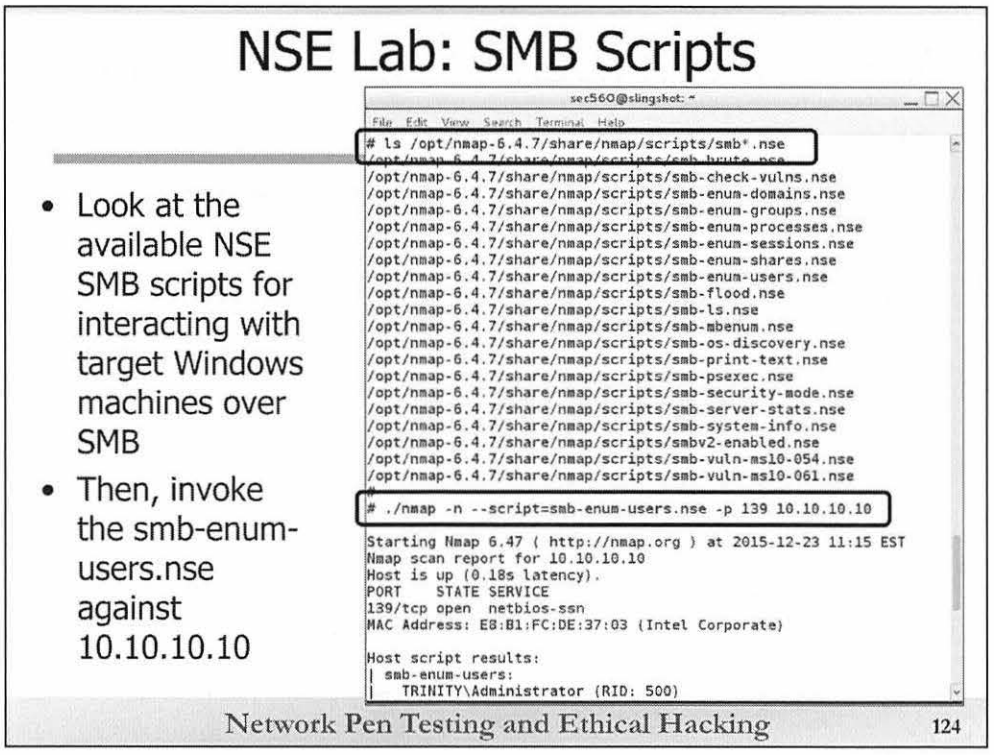


Next, let's explore the Nmap nbstat.nse script. This script works like the nbtstat.exe command included in some versions of Windows, which pulls NetBIOS over TCP statistics, including machine names, MAC addresses, and usernames. Note that although the Windows command (which is included on some but not all versions of Windows) is nbtstat.exe (with a T), the Nmap script is called nbstat.nse, without the t between the b and the s characters.

On a Linux terminal window, 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 its output. 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 SYN stealth scan (a "half-open" scan) has been run. Then, if the appropriate ports are open (TCP ports 135, 139, or 445), 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 the SMB NSE scripts included with this version of Nmap:

```
# ls /opt/nmap-6.4.7/share/nmap/scripts/smb*.nse
```

Here, you can see scripts that 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, and more).

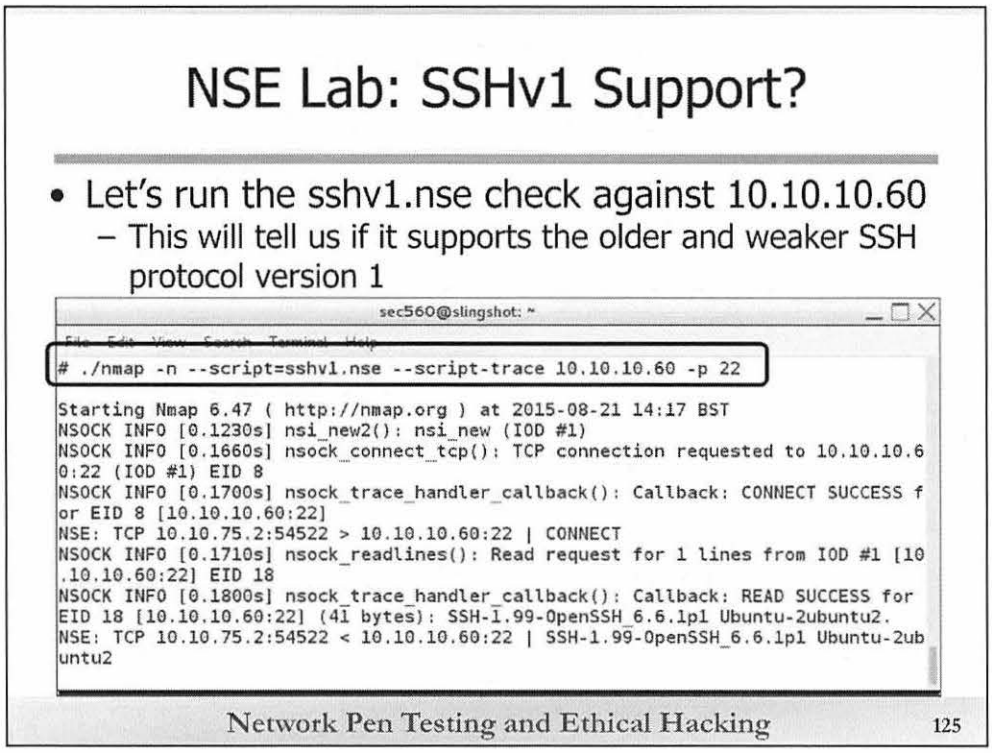
In addition, the smb-psexec command allows us to provide a username and password in the administrators group (with --script-args=smbuser=[AdminUser],smbpass=[AdminPass],config=[ConfigFileName], stored in [NmapDirectory]/nselib/data/psexec]), as well as one or more commands we want to run in a configuration 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 -p 139 10.10.10.10
```

In the output, you can see the results of the port scan, indicating that the given port is open. Then, you 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 usernames.

If you have extra time, you can try the other SMB.nse scripts in this directory.



Next, we 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:

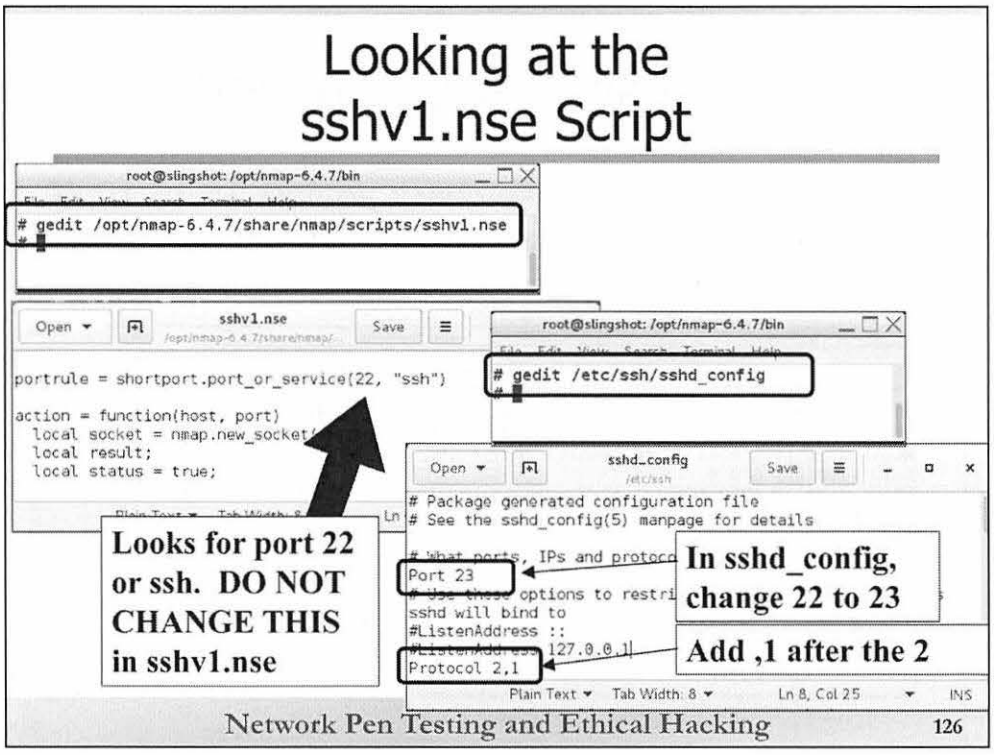
```
# ./nmap -n --script=sshv1.nse
--script-trace 10.10.10.60 -p 22
```

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 measuring only 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.

After you 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, based on the indications at the bottom of the output of your most recent Nmap command.



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 an important setting in each script:

```
# gedit /opt/nmap-6.4.7/share/nmap/scripts/sshv1.nse
```

Now, look for `portrule = shortport.port_or_service(22, "ssh")`

This line tells Nmap that it should run only 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. PLEASE DO NOT CHANGE THIS PORT VALUE OF 22 IN THE sshv1.nse FILE!

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
```

In the sshd_config file (NOT THE ssv1.nse file), find the line that says `Port 22`. Edit that line so that it says:

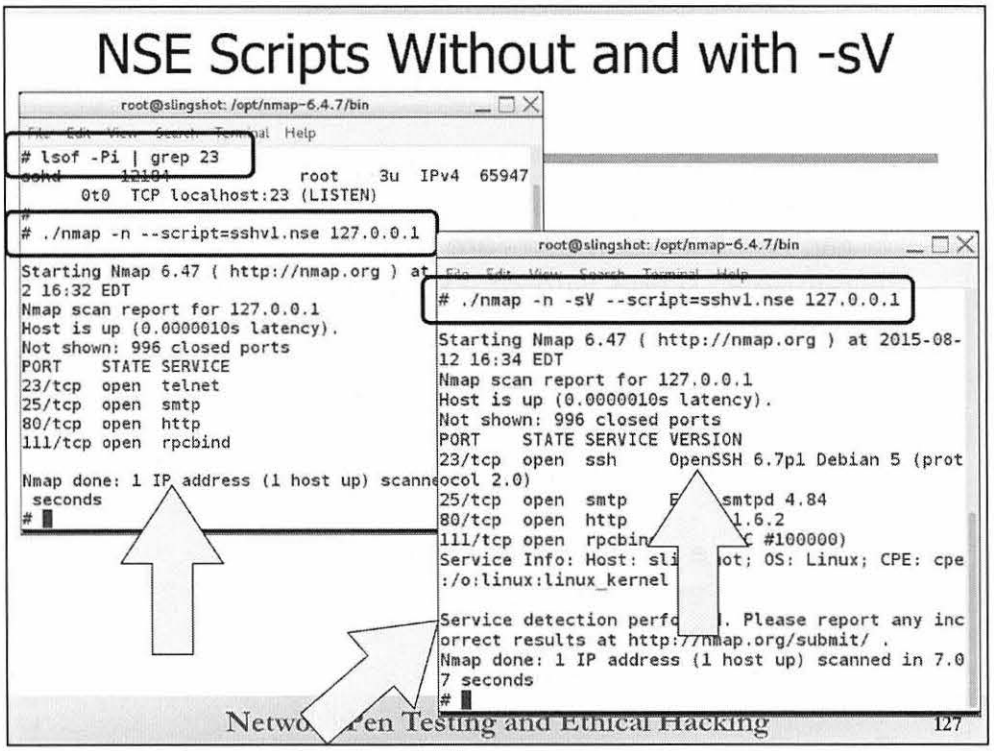
```
Port 23
```

Next, find the line that says Protocol 2. Append to that line a comma followed by a 1 so that it says:

```
Protocol 2,1
```

Save the file. Now, make your sshd reread 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, whereas 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:

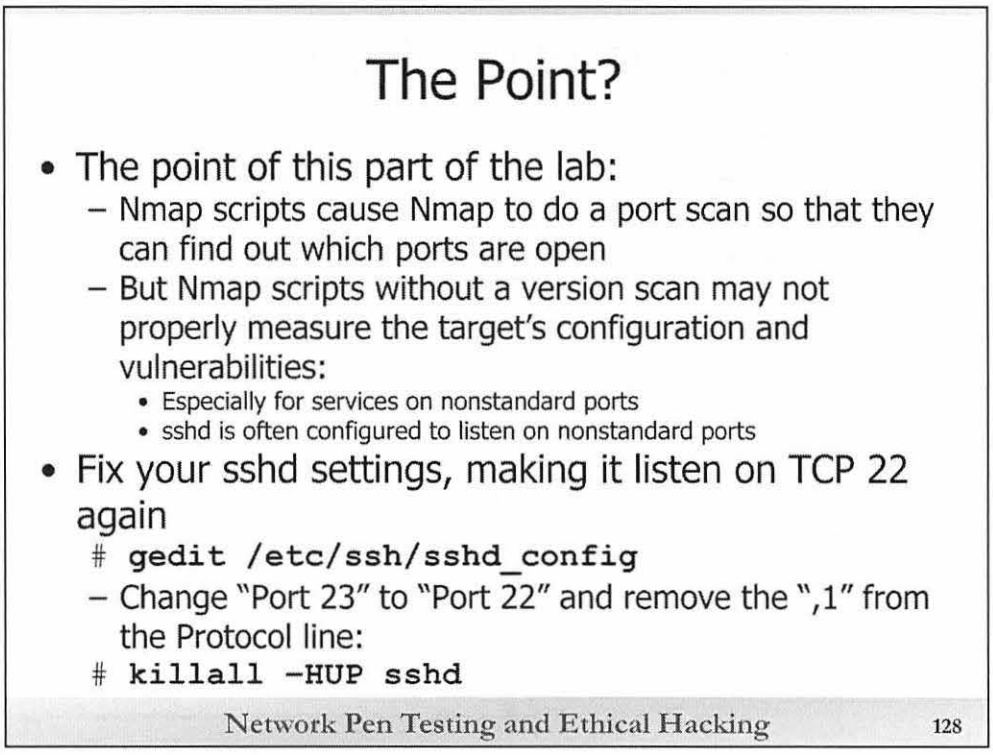
```
# ./nmap -n --script=sshv1.nse 127.0.0.1
```

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 performed only 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 detected that TCP 23 spoke the secure shell protocol.



So, what is the point of this component of this lab? It's actually twofold.

First, Nmap, when configured to run a script, performs a port scan of the target machine. It needs to do so, or the script cannot determine whether it should run against the target machine because it doesn't know whether the port(s) that the script is interested in are available. The –p option can help tailor this, making Nmap check a smaller number of ports, rather than all interesting ports.

Secondly, Nmap's scripts without a version scan may miss important services listening on nonstandard ports. Thus, when running a script from within Nmap, you are likely better served by making sure that you run a version scan (-sV) along with the script invocation (-sC or --script=).

These are important lessons for network penetration testers and ethical hackers who want to make the most out of NSE functionality.

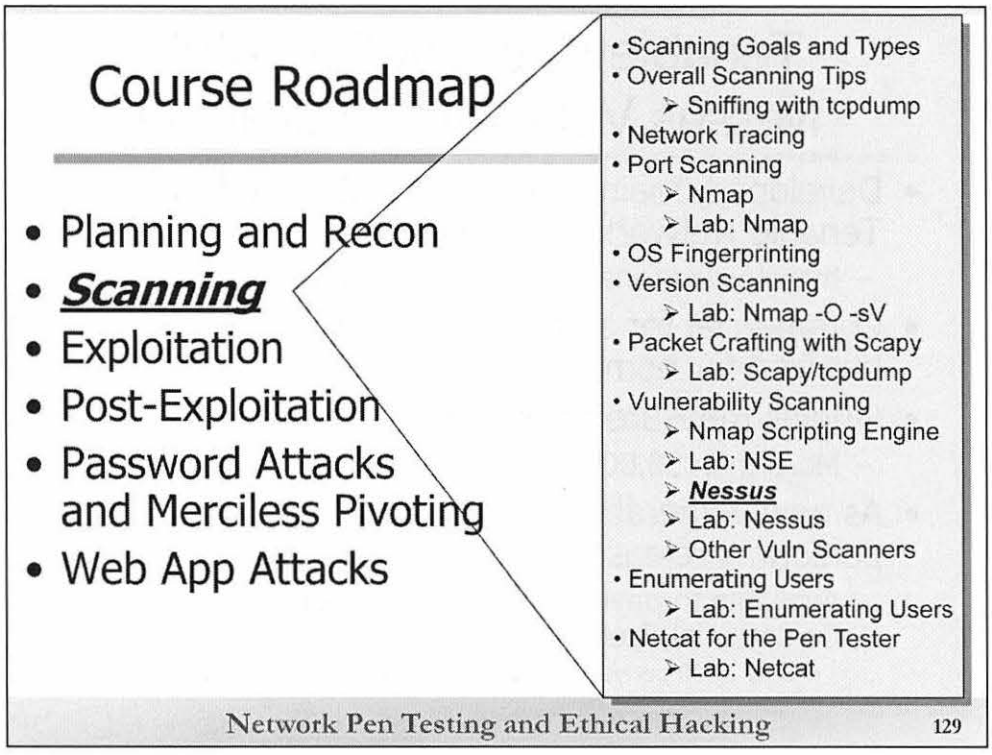
To finish with that piece of the lab, restore your sshd settings to their original value, making sshd listen on TCP port 22. Open the sshd configuration:

```
# gedit /etc/ssh/sshd_config
```

Change the line that says "`Port 23`" to "`Port 22.`" And, make sure you remove the ",1" from the Protocol so that your sshd will speak only ssh protocol version 2, the stronger form of ssh protocol.

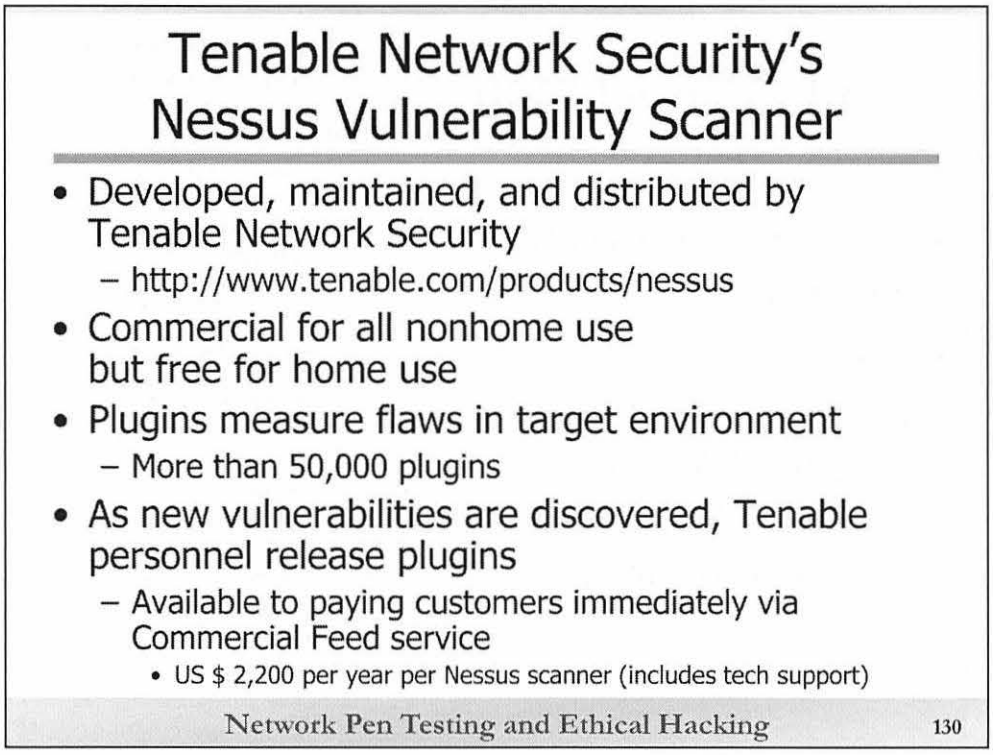
Save the file, and then send the HUP signal to sshd to make it reread its configuration:

```
# killall -HUP sshd
```



Although 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. Although 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.

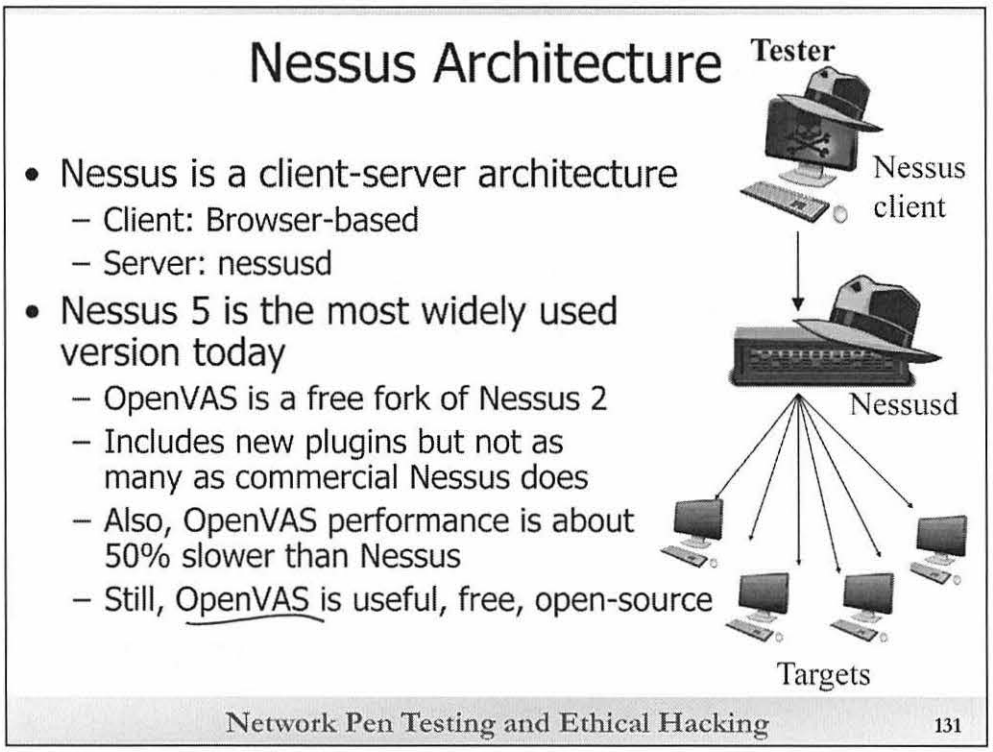
Most modern vulnerability scanners can measure for the presence of thousands of flaws in a target environment. One of the most full-featured vulnerability scanners available today is Nessus, our next major topic.



The Nessus Vulnerability Scanner is maintained and distributed by Tenable Network Security. Available at www.nessus.org, 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 older plugins are free and open source, whereas most (specifically the more recent ones) are commercial. There are more than 50,000 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.

All recent plugins require either a Commercial or Home feed subscription. Tenable's Commercial Feed service makes new plugins immediately available to paying customers for a US $ 1,500 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. According to Tenable license terms, all commercial use requires a Commercial subscription feed. Many professional penetration testers and ethical hackers who rely on Nessus subscribe to the commercial service.

nessusd bought by tenable.
↳ change to OpenVAS (free)

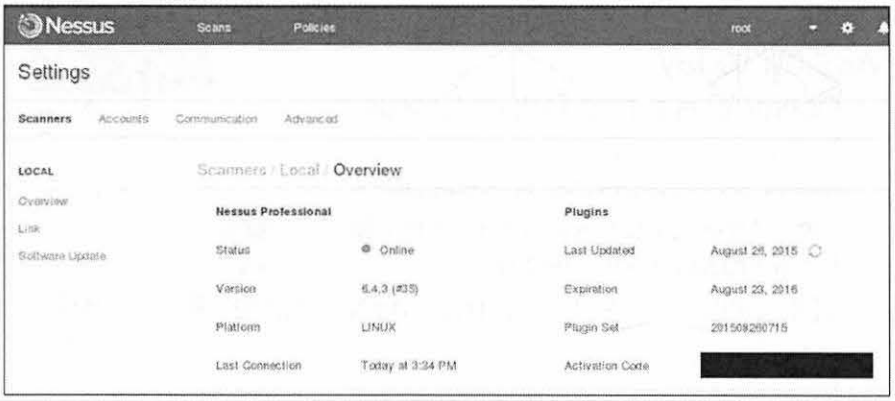


Nessus is a client-server architecture. The nessus client configures and manages things, whereas nessusd performs the scan. Although nessus and nessusd can run on separate systems, they are often run on the same machine. Nessus clients and servers have been released for Linux, Mac OS X, Windows, Solaris, and FreeBSD.

Today, Nessus 5 is the most widely used version of Nessus, with a web-based client interface. Many years ago, some open source developers created a fork of the last free, open source version of Nessus (version 2). The result was OpenVAS, a completely free vulnerability scanner. Although new plugins are distributed with and for OpenVAS, it is important to note that the commercial version of Nessus has more plugins and is more than 50% faster than OpenVAS. Still, OpenVAS is a useful, free, open-source alternative.

Update Plugins Regularly

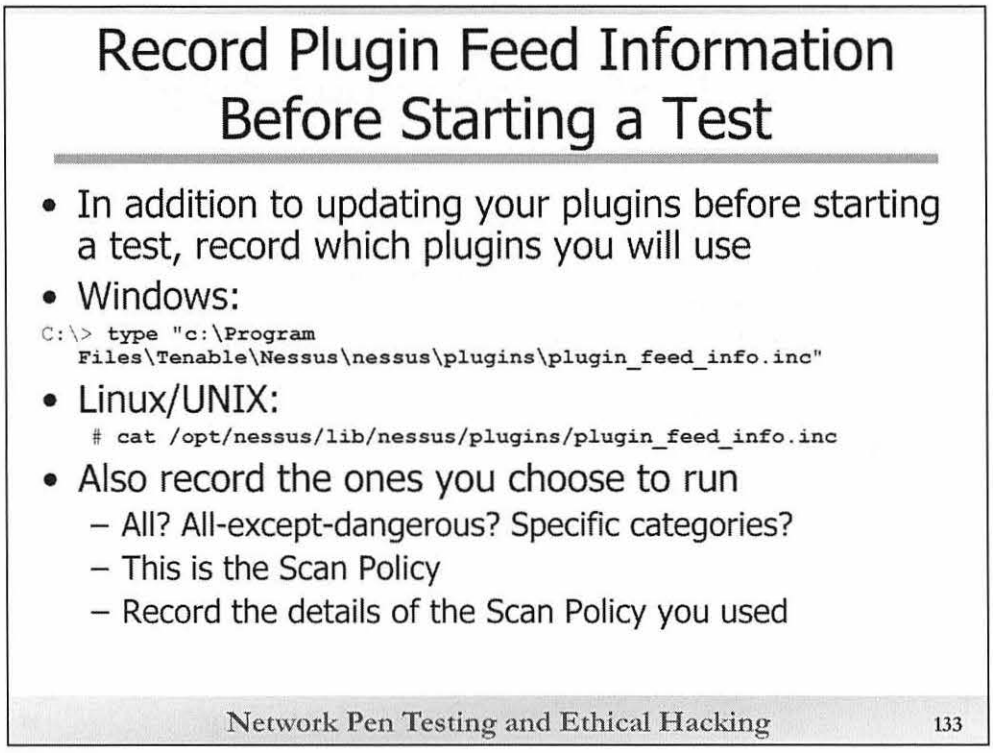
- Update Nessus plugins at the start of a test project
- To get latest plugins, you first need to register
  - Register and subscribe at www.nessus.org/plugins
  - You'll get a serial number
- Nessus auto-updates plugins every 24 hours by default



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 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.

After your serial number is registered, you can then use it to download plugins from within Nessus. Nessus 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.

When you have a serial number and register your version of Nessus to use it, you can update plugins manually immediately. In Linux, Solaris, and FreeBSD, this is accomplished by running the nessus-update-plugins script. On Windows and Mac OS X, you simply invoke a plugin update via the GUI.



In addition to updating your plugins before a test, you should also record the plugin feed information 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.

These plugin groups and their configuration are referred to as a "Scan Policy" within Nessus, as we'll see in the next lab. You need to record the details of the specific Scan Policy you used in a penetration test so that your results are repeatable.

Nessus and Dangerous Plugins

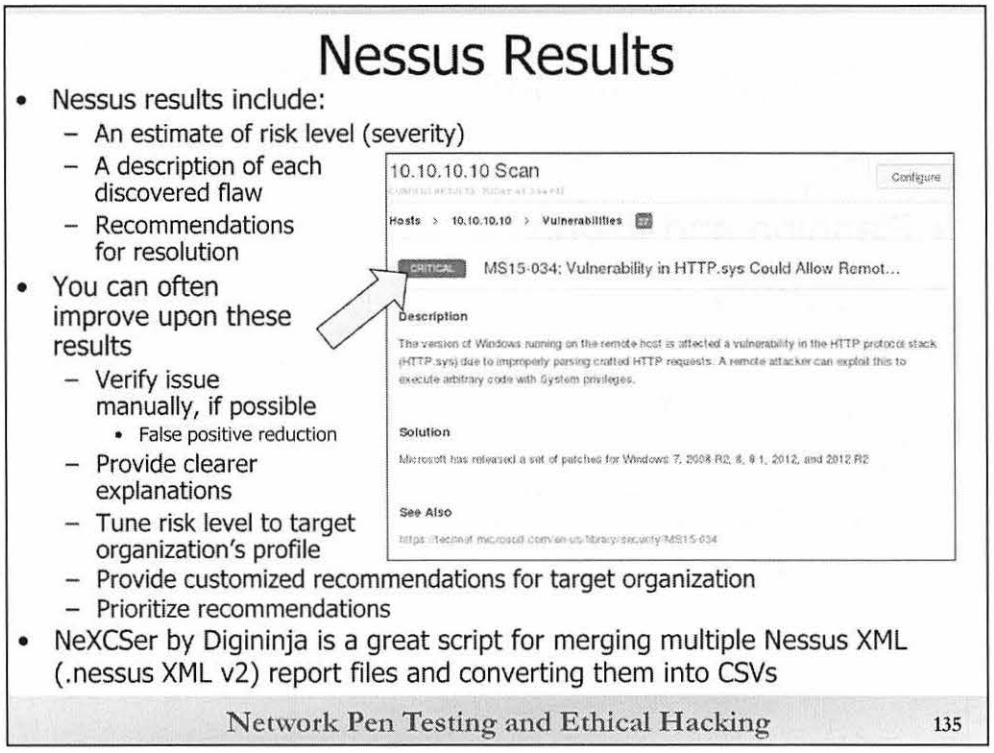
- Some Nessus plugins could crash a target system or otherwise impair it
  - Some denial of service plugins are dangerous, but not all
    - Some just measure version number
  - Password guessing plugins
  - Others
- By default Nessus does not run dangerous plugins
  - Default configuration is "Enable safe checks"
- You may choose to enable them but check the Rules of Engagement

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, a configuration known as "Enable safe checks" in the Nessus configuration GUI. You have to disable safe checks and then enable individual dangerous plugins in your Scan Policy if you want to run them. You'll note that the denial of service *family* of plugins is enabled by default (because it includes some plugins which merely check version numbers), but some of the plugins *within the family* are shut off (because some of them go beyond version checking and could impair a target).

So, should you run the dangerous plugins during a penetration test or ethical hacking project? Consult the Rules of Engagement. In most environments, you will not be allowed to run these plugins.

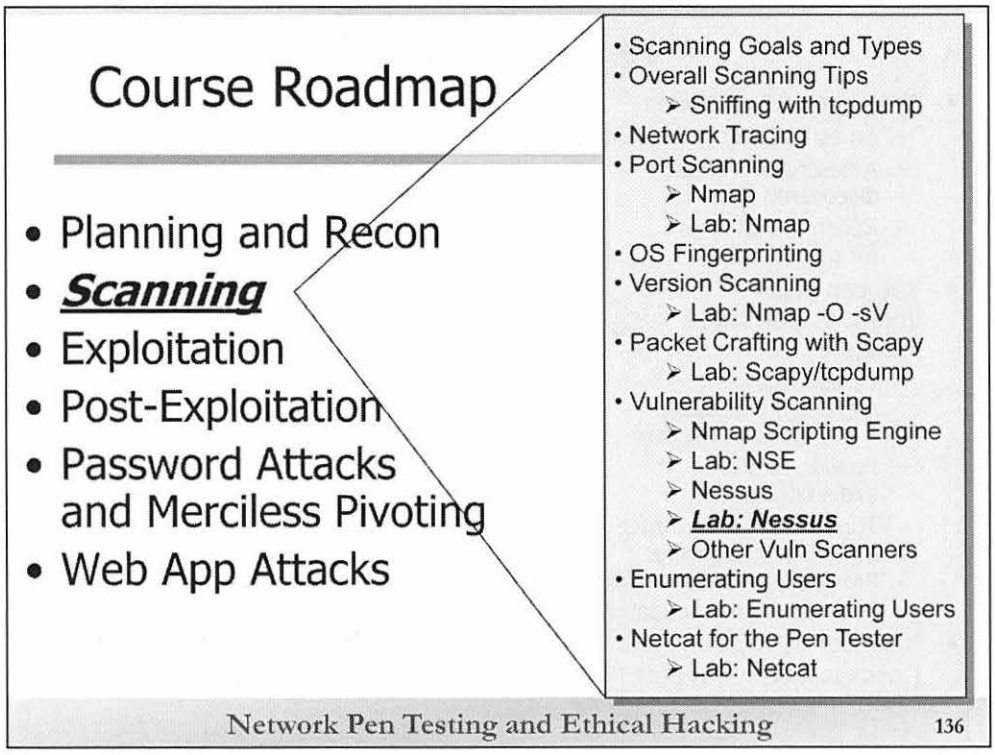


Nessus results include an estimate of the risk level associated with each finding (Critical, High, Medium, or Low severity), a brief description of each discovered flaw, and recommendations for resolution. Note that most professional penetration testers and ethical hackers use this Nessus output as a 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.

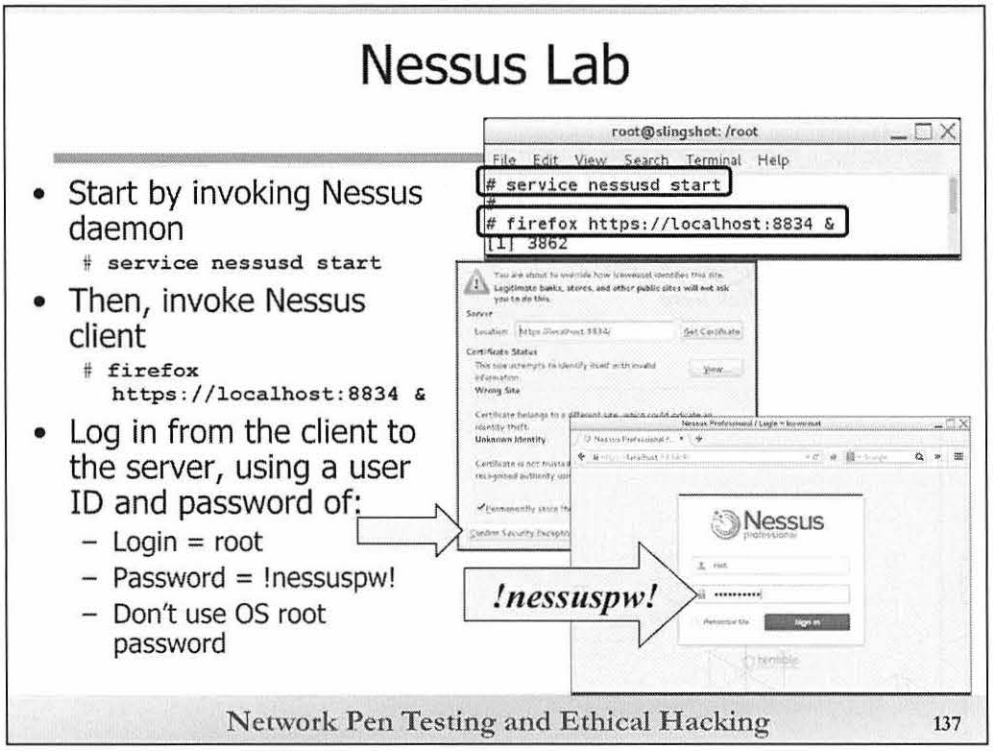
Instead, provide value-added services by verifying the Nessus results manually if possible, researching each discovered issue and trying to see if the given target machine actually exhibits that problem, or if we've got a false positive. You may need to review the configuration of the target with the system administrator, or research methods for using tools such as Scapy or Netcat (the latter of 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.

As a penetration tester, you may find yourself with multiple Nessus report files that you need to analyze and/or merge. The NeXCSer tool by Digininja (Robin Wood) is a great script for merging multiple Nessus XML report files (in the .nessus XML v2 format) and converting them into a CSV file. This tool is available at http://www.digininja.org/projects/nexcser.php.



Now that we've gotten an overview of Nessus functionality, let's tour its configuration during a hands-on lab. In your Linux machine, get ready to run Nessus against our target environment.



First, we need to invoke the nessusd service:

```
# service nessusd start
```

You will not see any indication of "OK" on the output. Instead, when 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.

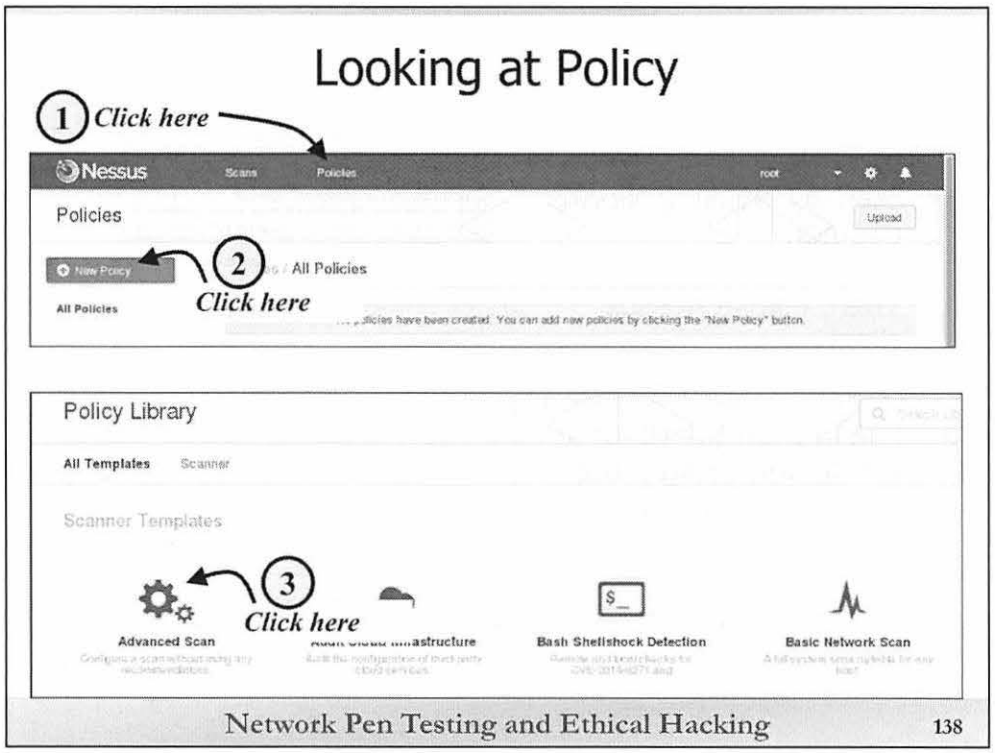
When the nessusd server is running, we can invoke the Nessus web-based interface, by launching our browser, using:

```
# firefox https://localhost:8834 &
```

You will now get an Alert message saying that the security certificate from the nessusd web server is not trusted. That's because the Firefox browser doesn't trust this Tenable certificate by default. Under I Understand the Risks, click Add Exception…. Then, click Get Certificate. Finally, click Confirm Security Exception.

The Nessus web-based GUI will ask for a Username and Password to access nessusd. Type in a login name of root and a password of !nessuspw! and click the Sign In button. 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.

You can click Remember password. When you see the Nessus splash screen showing My Scans, you are ready to proceed.



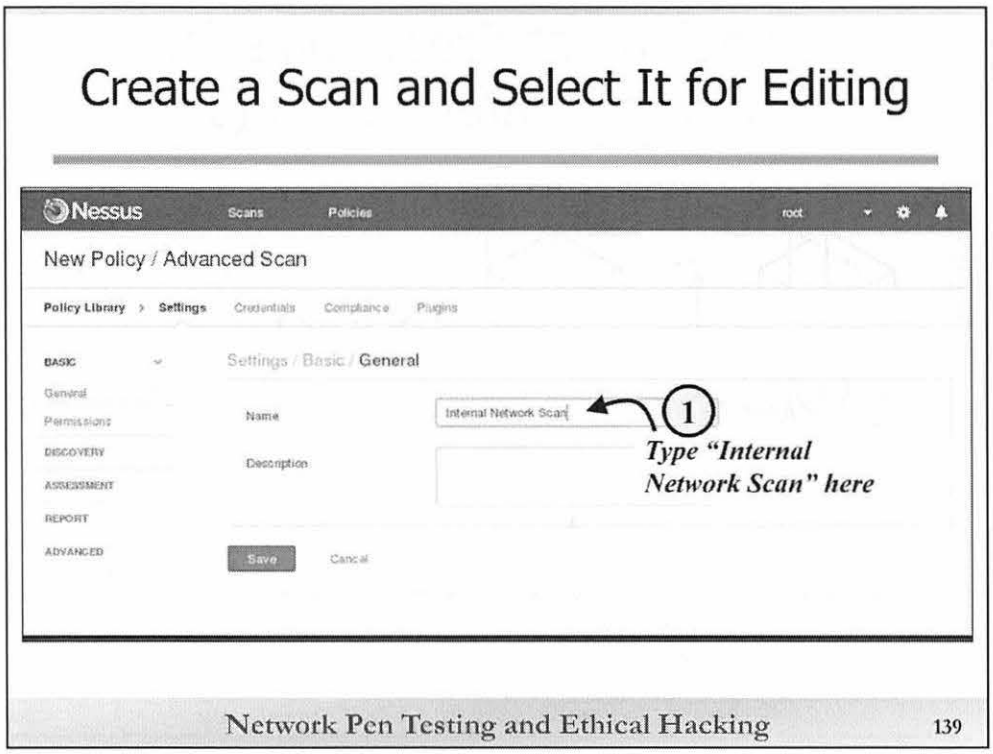
To explore Nessus, first look at how you can create a Scan Policy.

In Step 1, click the Policies component near the top of the GUI (next to Scans).

In Step 2, near the left side of the screen, click New Policy.

You now see the default policy templates distributed with Nessus, including Advanced Scan, Basic Network Scan, Credentials Patch Audit, Web Application Tests, and much more.

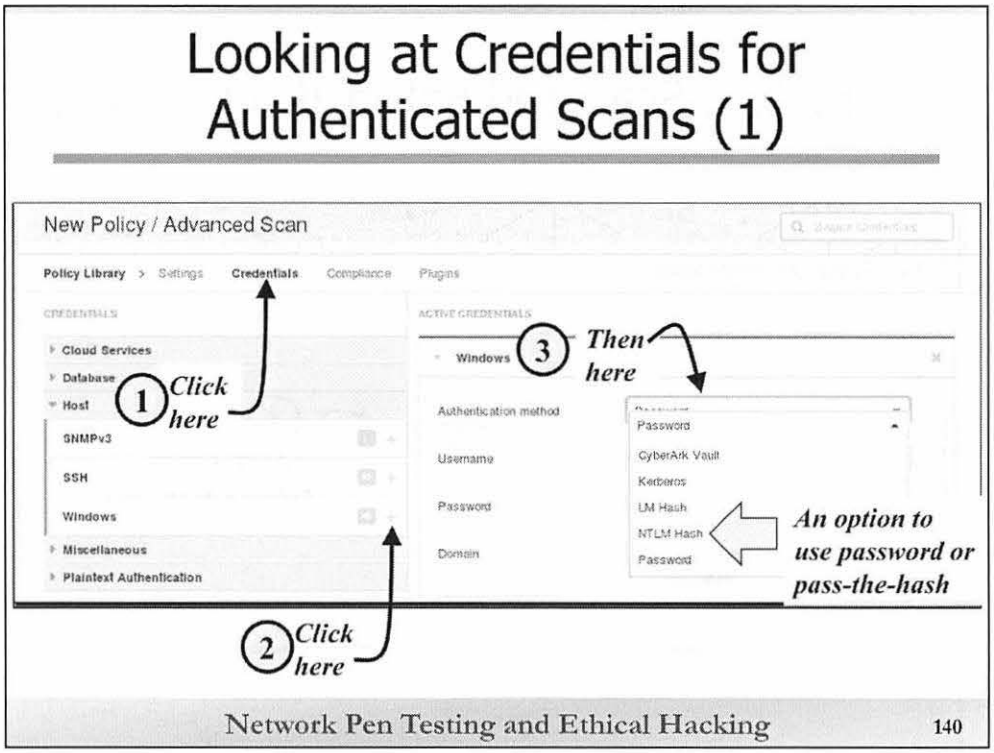
Click Advanced Scan because that is what we'll use as our template to scan the target environment for this lab.



You should now see a box for entering the name of the scan we want to create, based on the Advanced Scan template.

Type **Internal Network Scan** inside that box.

Next, we'll analyze and configure the details of that Internal Network Scan.



Let's look at the options Nessus provides for authenticated scans. Click the word "Credentials" near the top of the Nessus screen.

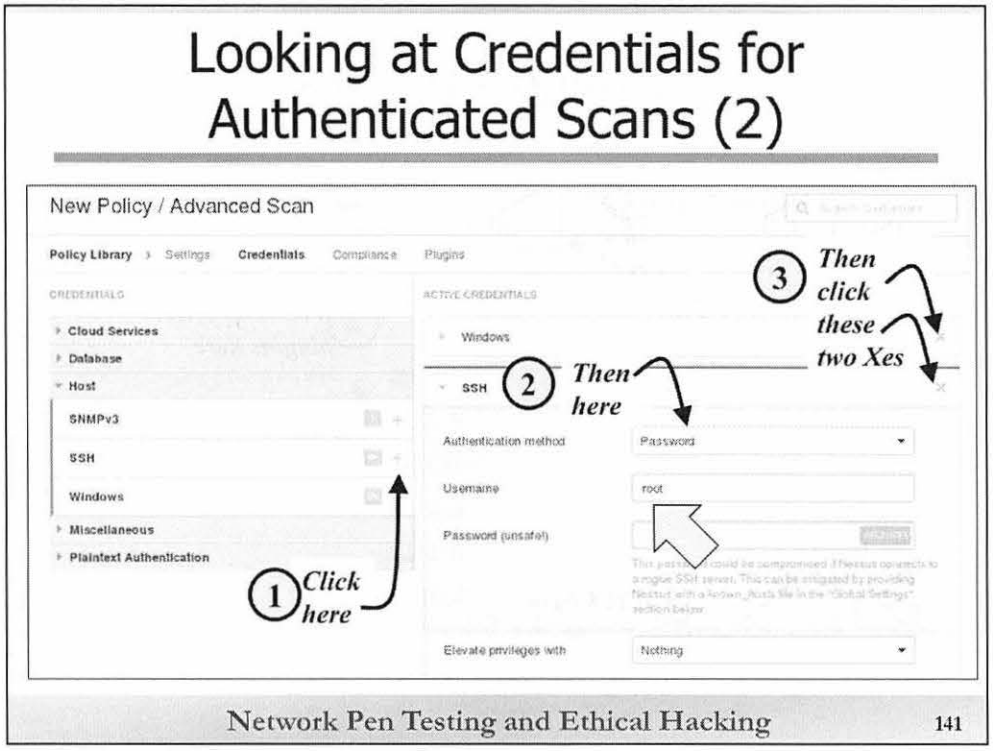
Nessus allows testers to enter user IDs and passwords for a target environment, which Nessus uses with various plugins that can supply user credentials to target machines. Some of these plugins actually try to log in to various target systems and measure them for vulnerabilities. Credentialed scans (also referred to as Authenticated Scans) can provide greater detail about configuration flaws and other issues in a target environment than unauthenticated scans.

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 be measured only using valid authentication credentials, especially for internal network scanning.

On the left side of the screen, we can see the Host credentials options, including the Simple Network Management Protocol v3 (SNMPv3), SSH, and Windows. Click the plus sign next to Windows. You should now see the screen on the right where we can enter Windows username, a password, and a domain name.

We can also use the pull-down menu (in Step 3) to indicate whether we have provided credentials in the form of a password, an LM hash, an NT hash (labeled NTLM), a Windows Kerberos ticket, or other. If we provide hashes, Nessus can perform authenticated scans by doing pass-the-hash to authenticate to targets using the hashes and not the passwords themselves, a topic we'll discuss in more detail in 560.4 and 560.5.

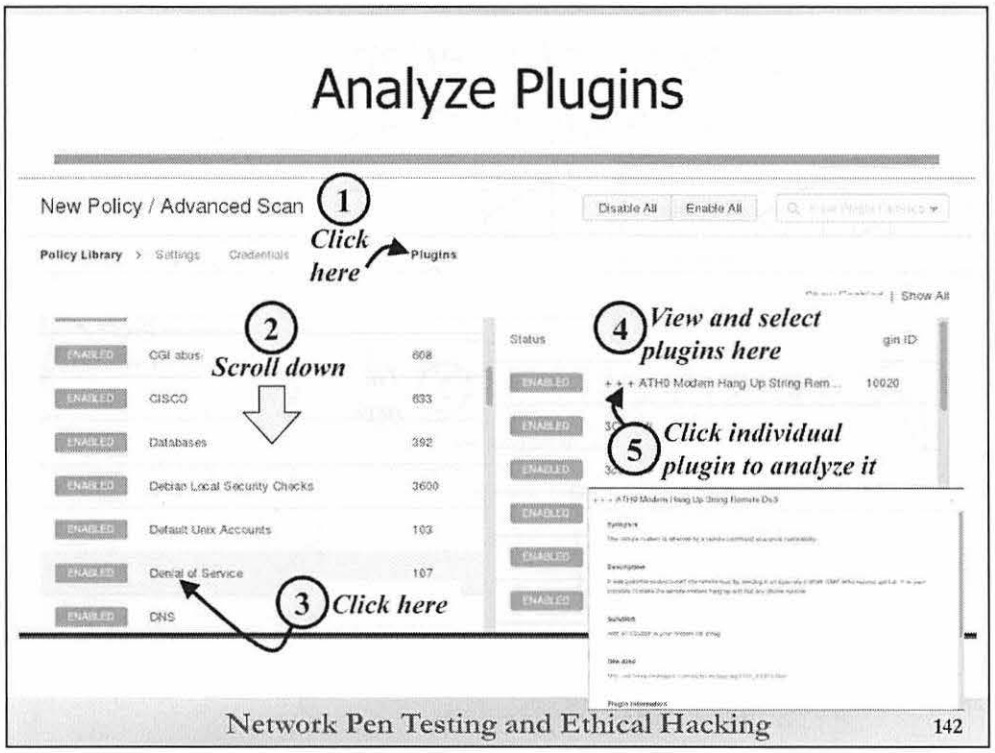
Because the scan we are conducting for this lab is unauthenticated, do not enter anything in these fields.



Next, on the left side of the screen, click the plus next to SSH, as indicated in Step 1. Here, we can configure Nessus with an SSH username and password, as well as public and private keys for authenticating to machines in the target environment. In Step 2, click the drop-down with the word Password to see the other authentication options Nessus supports for ssh. In addition to logging in via SSH, Nessus has an option to provide a method for elevating privileges on a target environment using su, sudo, or other options.

Again, because the scan we are conducting for this lab is unauthenticated, do not enter anything in these fields.

Then, in Step 3, click the two Xes on the right side of the screen, one on the Windows credentials tab and the other on the SSH credentials tab.



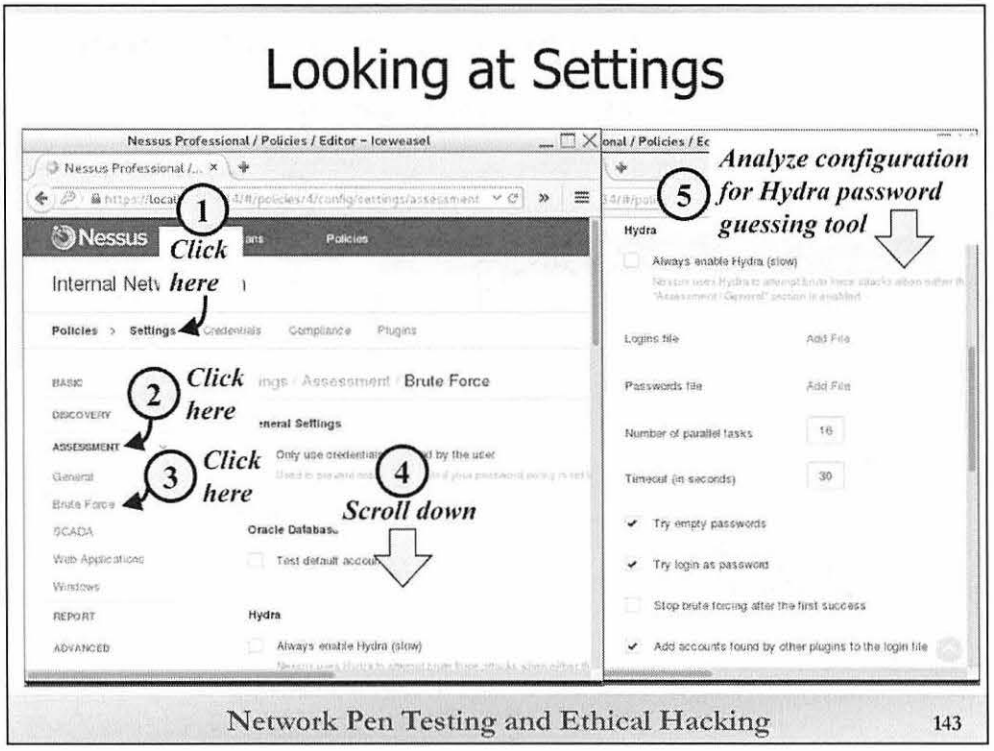
Next, look at the plugins. In Step 1, near the top of the Nessus screen, click Plugins. Then, on the left side of the screen, look at the Families of Plugins. You see categories such as Backdoors, Brute force attacks, and CGI Abuses. In Step 2, scroll down to see items such as Databases, Denial of Service, and DNS.

In Step 3, click the Denial of Service plugin family to reveal the individual plugins in this family.

Now, in Step 4, in the plugins pane on the right side of the screen, you can see the individual plugins inside the Denial of Service family. If you click one of the plugins here (in Step 5), you can see the details associated with it. Note that the detailed plugin screen includes a Synopsis, a Description, and a recommended solution, along with references and a Plugin ID number. You can close the detailed plugin screen.

Note that by default, various plugins are marked as active. (The green box indicates that.) If you click a plugin's green box on the left, the plugin gets turned off, making it gray and saying Disabled. Try deselecting one of the plugins inside of the denial of service family (such as plugin # 10020) by clicking its green box. When this plugin is turned off, its box turns gray. What's more, you can see that the family box on the left now turns a striped blue and says MIXED, indicating that some plugins within this family are disabled. You can also disable or enable entire Families by clicking the green box next to their name.

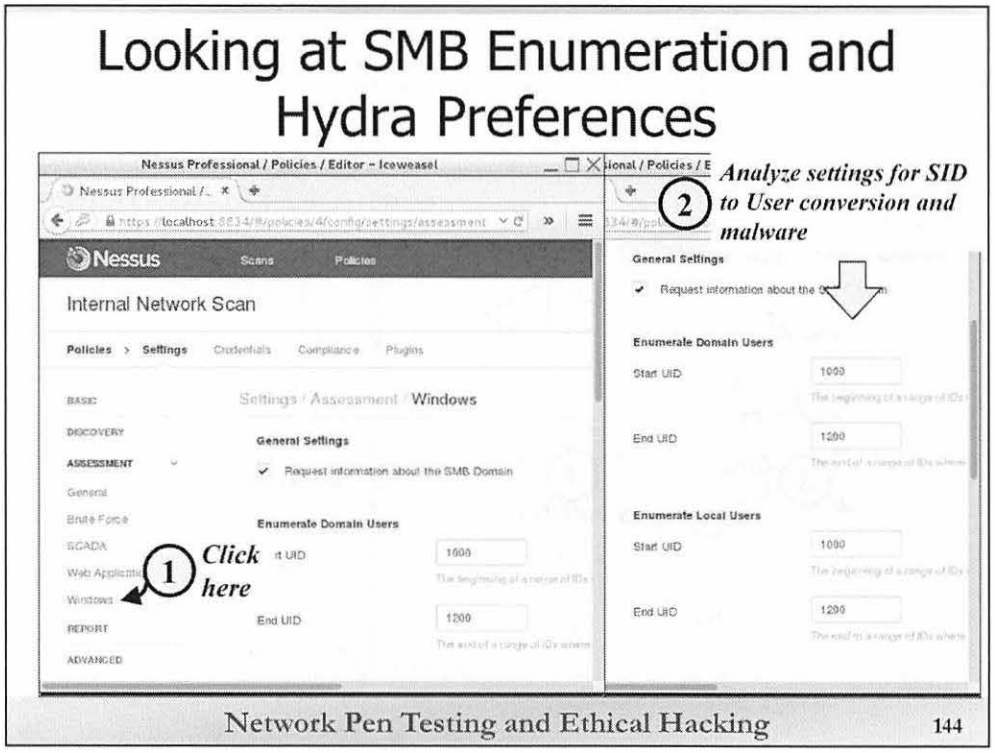
For this scan, make sure you leave the Denial of Service category enabled, as well as the various plugins in that category enabled. By default, Nessus turns off all dangerous plugins. These Denial of Service plugins are not dangerous, as most focus on checking version information without launching a denial of service attack.



Now look at some additional Nessus settings. In Step 1 of this slide, click Settings near the top of the Nessus screen. Here, you can configure some options that control various features of Nessus, not just individual plugins.

In Step 2, click Assessment. Then, in Step 3, click Brute Force. Here, we have options for password guessing against target machines.

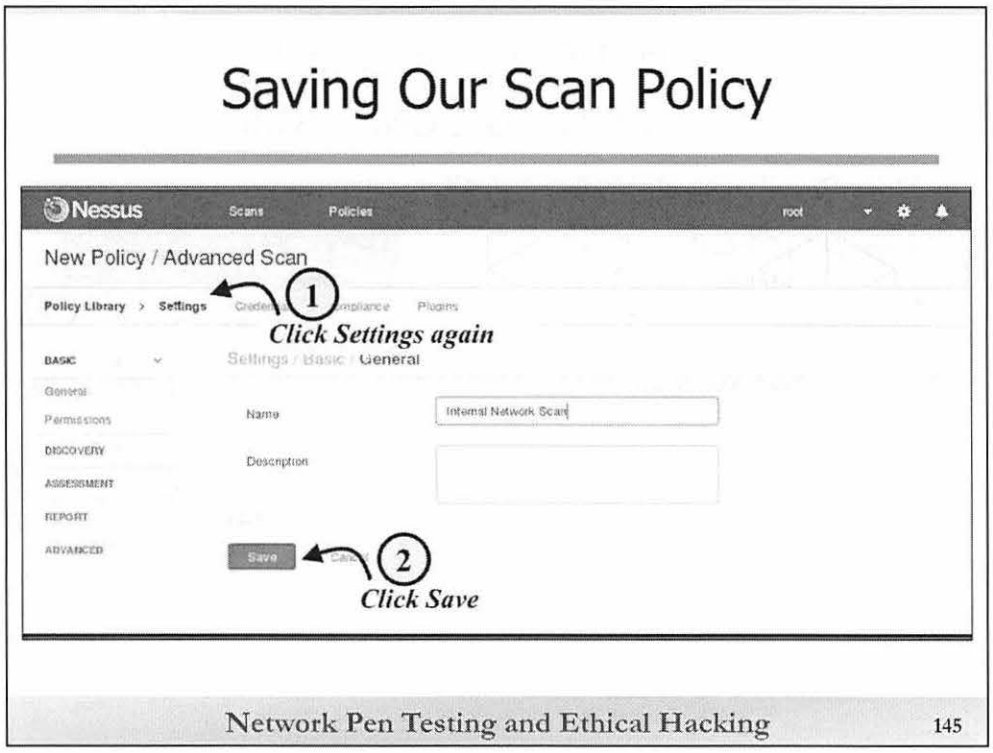
In Step 4, scroll down, and you can see that Nessus can call Hydra, a flexible password-guessing tool that we'll look at in depth in Section 560.4 when we cover password guessing. In Step 5, if you scroll down, you'll see the detailed Hydra configuration Nessus can use. We'll actually discuss many of these options in Hydra in 560.4 and use many of them in a password-guessing lab then. Note that Hydra scanning is disabled in this Nessus Advanced Scan policy by default. Also, note that password guessing can be a slow process, indicated in the Nessus Hydra configuration with the word "(slow)".



Next, in Step 1, still under the Assessment part of the screen on the left, click Windows.

Here, you see how Nessus can enumerate domain or local users by iterating through the Security IDentifiers (SIDs) of accounts, a technique we'll cover using different tools (SID2user and user2SID) in more detail in a lab later in this session, 560.2. We give Nessus a start Relative Identifier (RID), the suffix of a SID, and it iterates through the various SIDs sending requests to a target Windows machine looking for associated user accounts on the target.

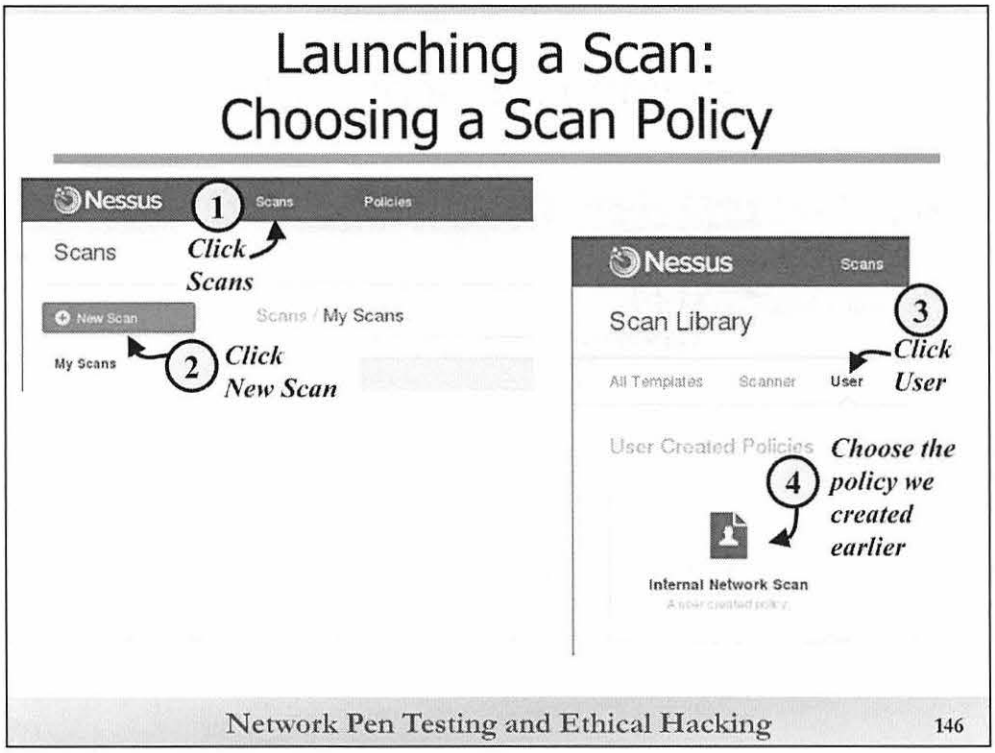
Briefly review these options. The default settings for them are quite reasonable for most tests, so do not change them.



Now that we've reviewed the Scan Policy, let's save it so we can run a scan using it.

Near the top of the Nessus screen, in Step 1, click Settings.

You should now see the Internal Network Scan name. In Step 2, click Save.



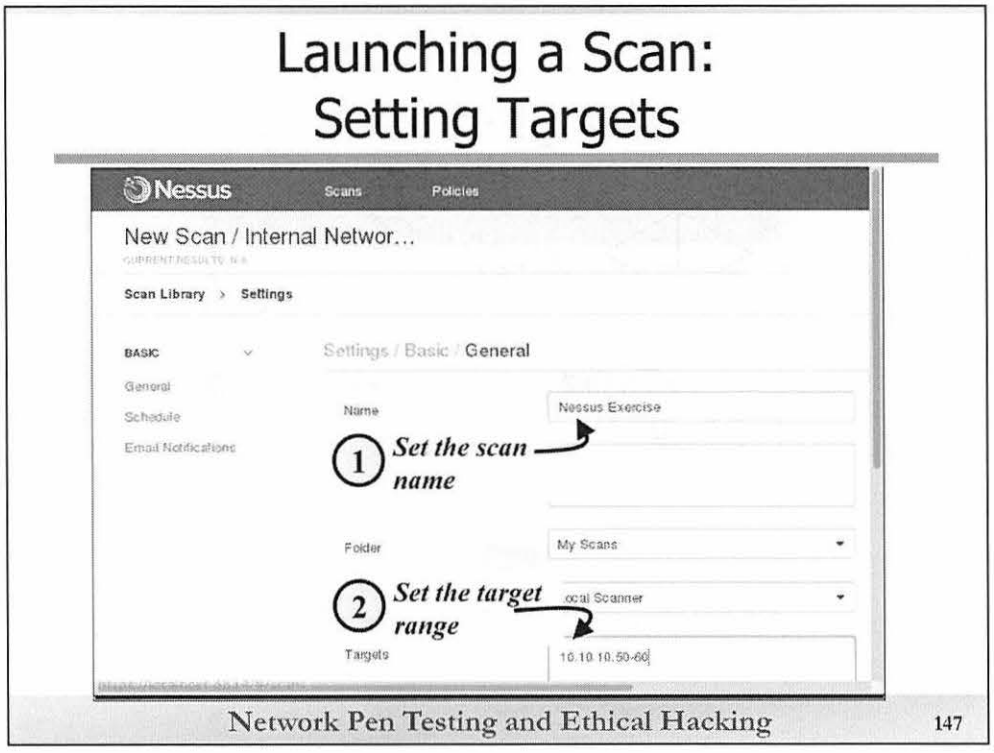
Now that we have reviewed the configuration options, let's actually launch a scan using Nessus and our Internal Network Scan policy.

In Step 1, we'll move from the Policies configuration component of Nessus to its Scans component. Click Scans on the bar near the top.

We're going to add a new scan, so in Step 2, click New Scan.

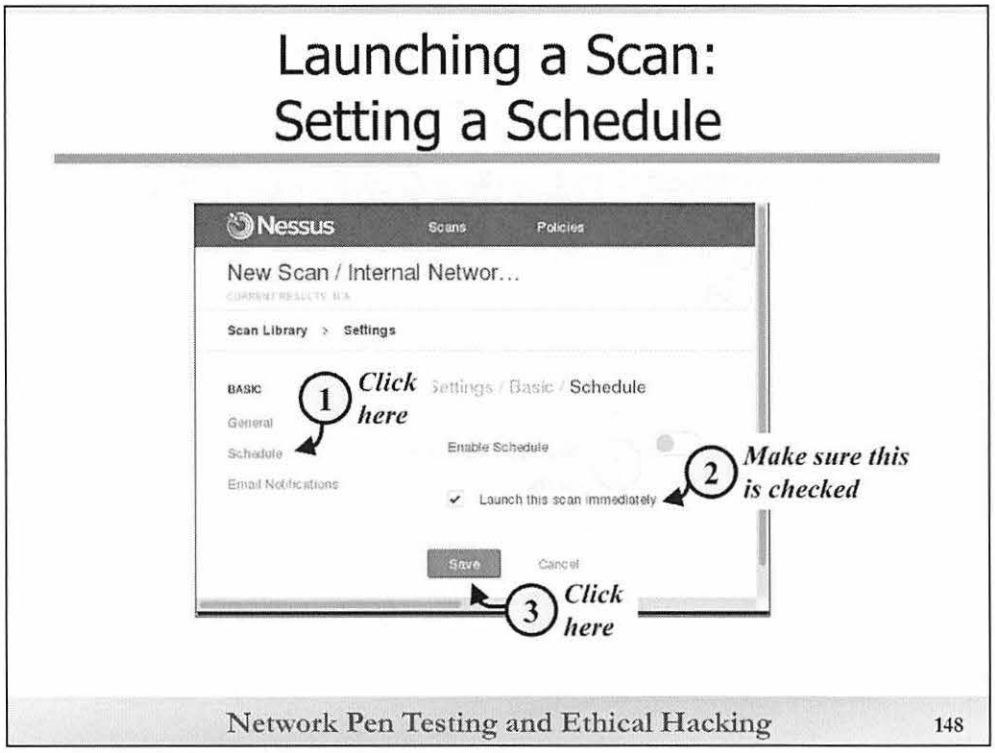
We now get to choose a Scan policy. For Step 3, click the word User near the top of the screen, so we can see the customized Scan Policy we created.

In Step 4, click the Internal Network Scan Policy we created.



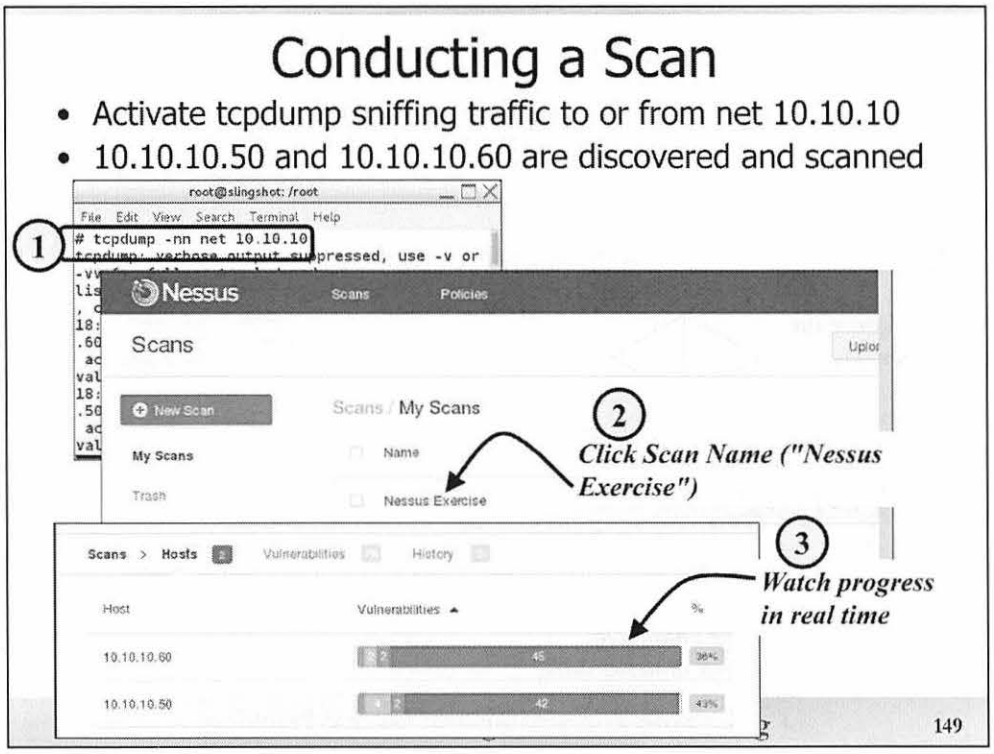
Now, we need to choose a name for the scan. In Step 1, call it Nessus Exercise.

We'll scan target IP address range 10.10.10.50-60, so enter that into the Targets field for Step 2.



Now, we need to schedule our scan, so click Schedule near the left side of the screen. Make sure Launch This Scan Immediately is checked. Most penetration testers use this option, but Scheduled Scans at Different Times can be useful for day-to-day vulnerability checks by auditors and a variety of different security personnel.

Click the Save button in Step 3 to save your scan. That should immediately launch your scan. You should see a screen that shows the Nessus Exercise scan, with rotating green arrows to indicate that the scan is under way.



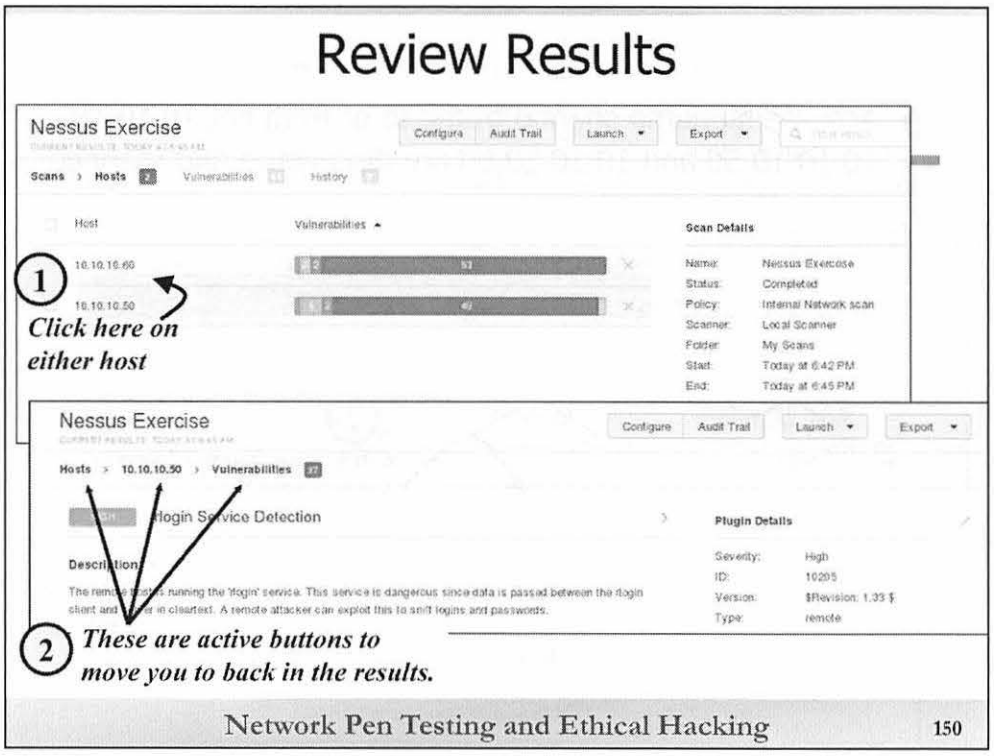
The scan will begin to run, moving you to a Nessus screen that shows the high-level scan status.

In Step 1, while it is running, activate tcpdump in a separate terminal to sniff traffic going to and from the 10.10.10 network so that you can watch the Nessus activity:

```
# tcpdump -nn net 10.10.10
```

Next, in Step 2, back in Nessus, click the scan name (Nessus Exercise) to see the real-time progress of your scan.

Initially, Nessus attempts to determine which IP addresses are actually in use by sending a series of probes to each address in the range we specified. After determining that 10.10.10.50 and 10.10.10.60 are live hosts, Nessus then focuses its scan on each of them.



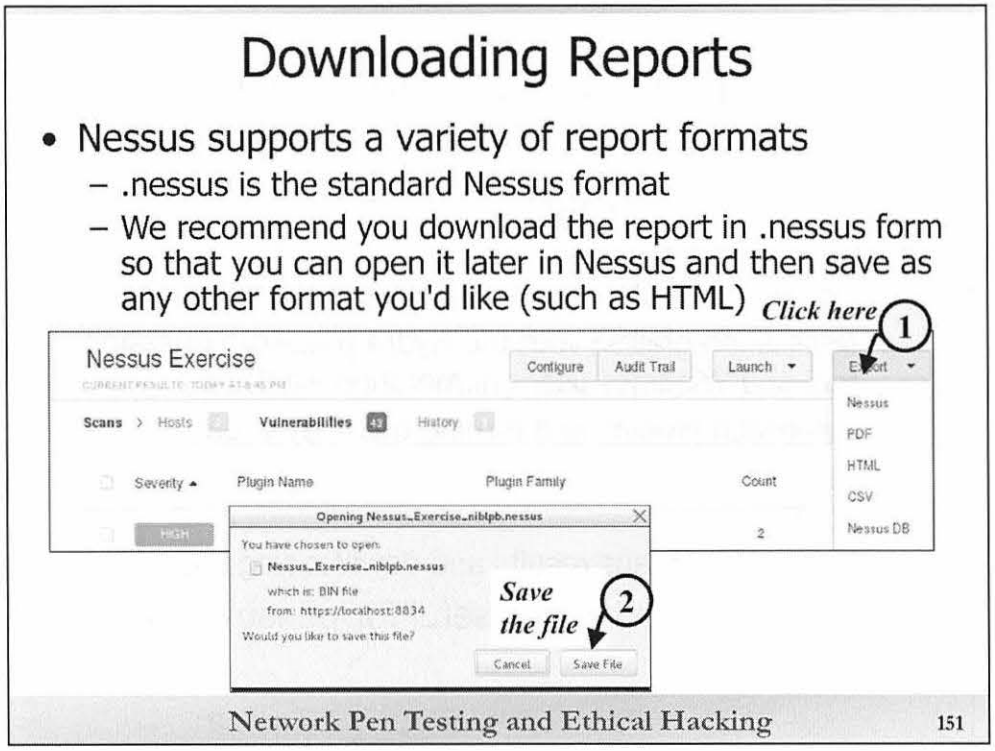
We can see interim counts of the number of discovered High, Medium, and Low issues identified so far in the scan next to the host IP addresses.

Before the scan finishes, as it is running, in Step 1, click the 10.10.10.50 or 10.10.10.60 host.

You now see the individual findings for that host discovered so far.

In addition, note that the section near the top of the screen with Hosts→10.10.10.60→Vulnerabilities is actually clickable, letting you explore the findings discovered so far.

Click a High risk finding and look at the details Nessus provides about the target machine. Nessus has determined that each host supports the rlogin service, which introduces security risks associated with cleartext authentication and other concerns.



Nessus allows us to download our findings in a variety of formats. Near the top of the screen, click the Export button near the right. It'll drop down to show you the formats Nessus supports.

The default format understood by Nessus is .nessus (the first one in the list). Other formats include HTML, PDF, CSV, and a Nessus DB format. The Nessus DB format is quite large and stores not only the findings, but all information about the Nessus scan.

We recommend that you save results immediately in .nessus format (the first in the list), as it is the most widely used form of Nessus results. Select that, and when prompted to save the file, click Save File.

When you finish the lab, you can simply click root near the upper-right side of the Nessus GUI. On the drop-down menu, select Sign Out. You can then close your browser.

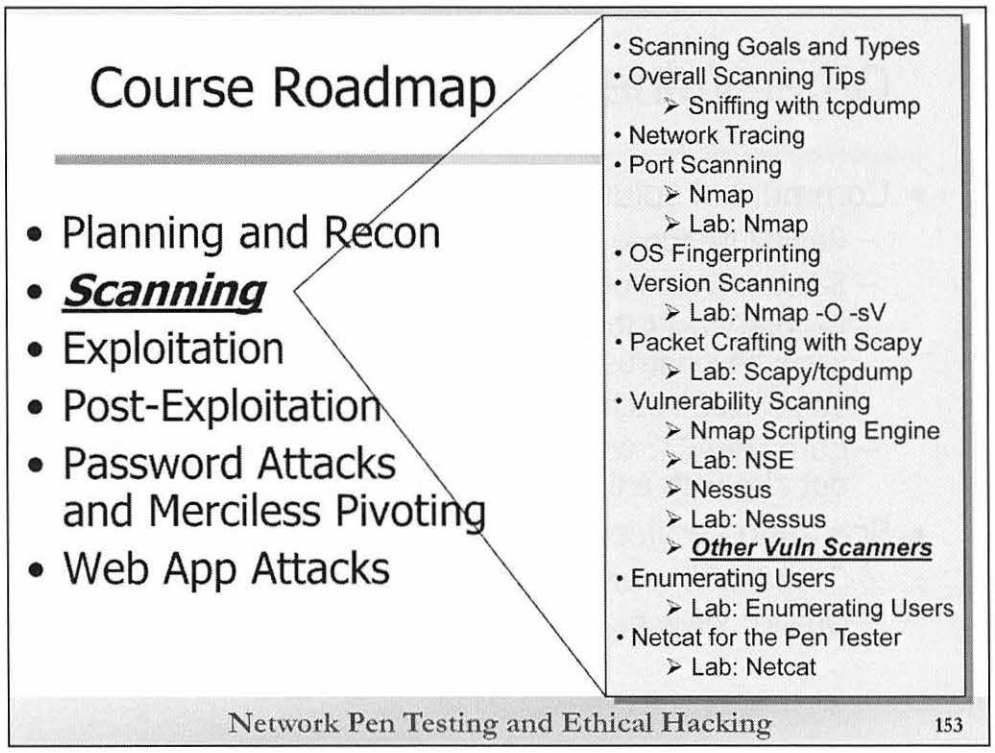
Then, you can shut down the Nessus daemon by running

```
# service nessusd stop
```

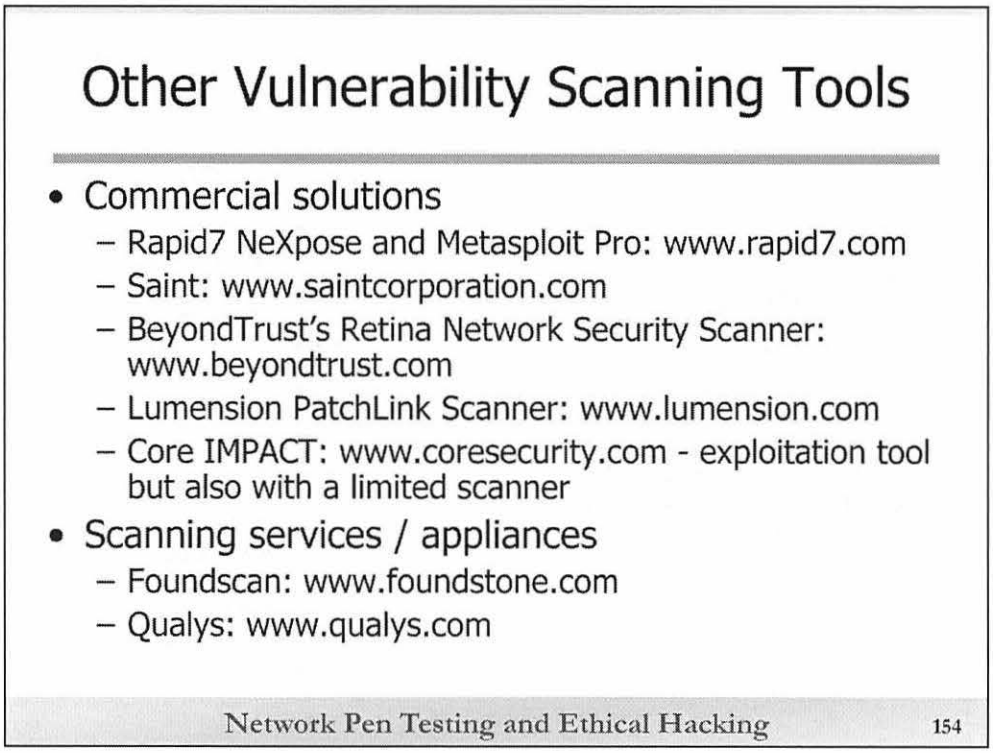
Nessus Lab Conclusion

- In this lab, you've analyzed:
  - How to create a custom Nessus Scan Policy
  - Options for authenticated scans for SMB and SSH access
  - Nessus integration with the Hydra password guessing tool and Windows user enumeration via SIDs
  - Reviewing plugins and turning them on or off
  - Interacting with Nessus while it is actively conducting a scan
  - Reviewing Nessus results and downloading them
- Each of these is highly useful for penetration testers

In conclusion, for this lab, we've looked at configuring Nessus and using it to launch a vulnerability scan. In particular, we created a custom Scan Policy and looked at several policy options for it, including authenticated versus unauthenticated scans, integration with the Hydra password guessing tool, and Windows user enumeration. We also reviewed plugin selection, as well as activating and deactivating plugins in a Scan Policy. We then launched a scan and interacted with the interim results while the scan was under way.



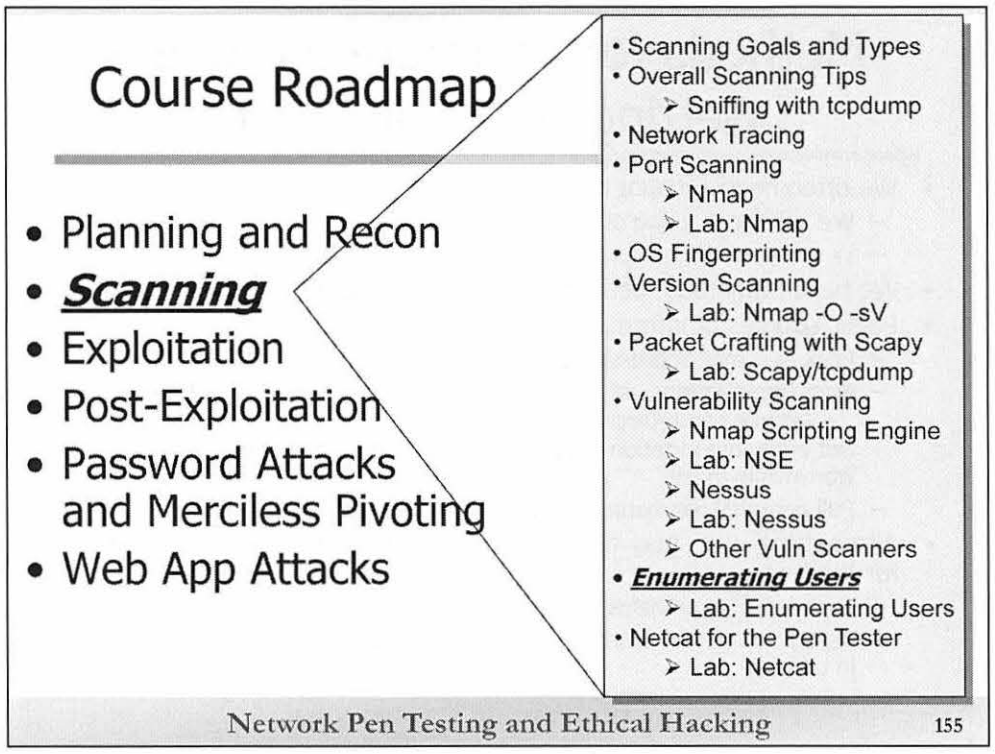
Although Nessus is popular, there are other vulnerability scanning tools on the market. Let's briefly survey some of them that you might want to consider using and then zoom into one of them to explore its capabilities in more detail.



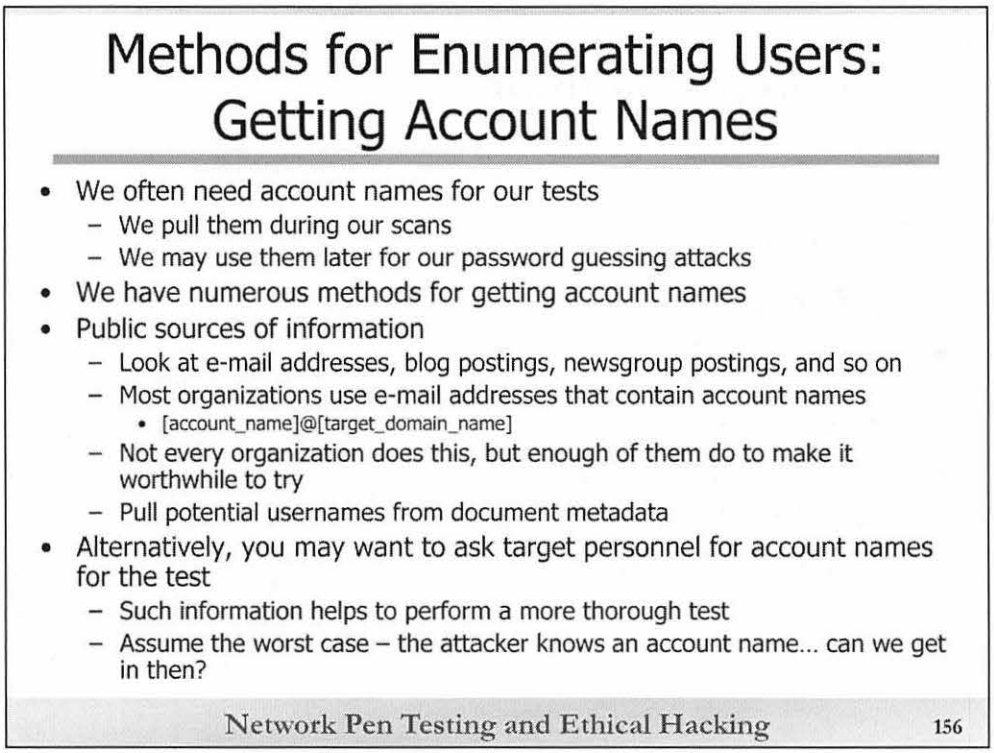
Besides Nessus, there are numerous other commercial and free vulnerability scanning solutions available today. From a commercial perspective, products include Rapid7's NeXpose, a comprehensive vulnerability scanning and management solution. Rapid7 also sells Metasploit Pro, a product that provides a 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. BeyondTrust's Retina Network Security Scanner has a comprehensive scanner called the Retina Network Security Scanner. The Lumension PatchLink Scanner is used by several U.S. government and military agencies as well as some commercial companies.

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. Although 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 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.



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.



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.

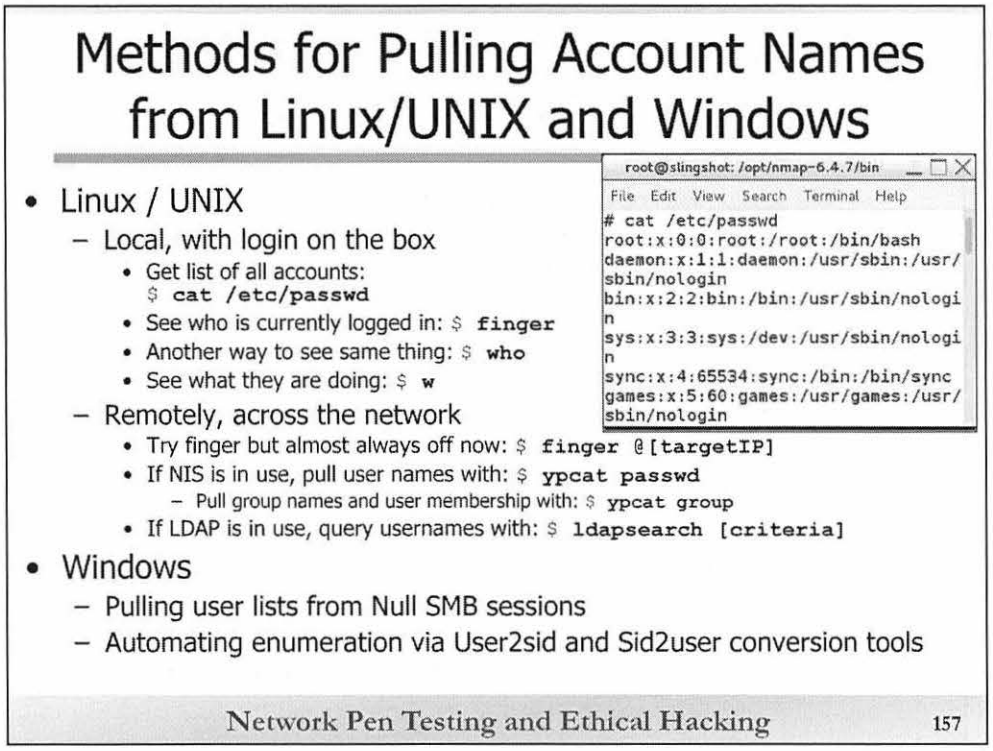
There are numerous methods for getting account names. One method involves doing research on the Internet in various public sources of information to pull potential account names. A tester can look at e-mail addresses in newsgroup postings, mailing list archives, blog posts, and social networking sites. 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.

In addition, as we saw in 560.1, we could try to pull usernames 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 she shoulder surfed them from a legit user at an airport or cyber café.



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 log in 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 usernames for the system:

```
$ cat /etc/passwd
```

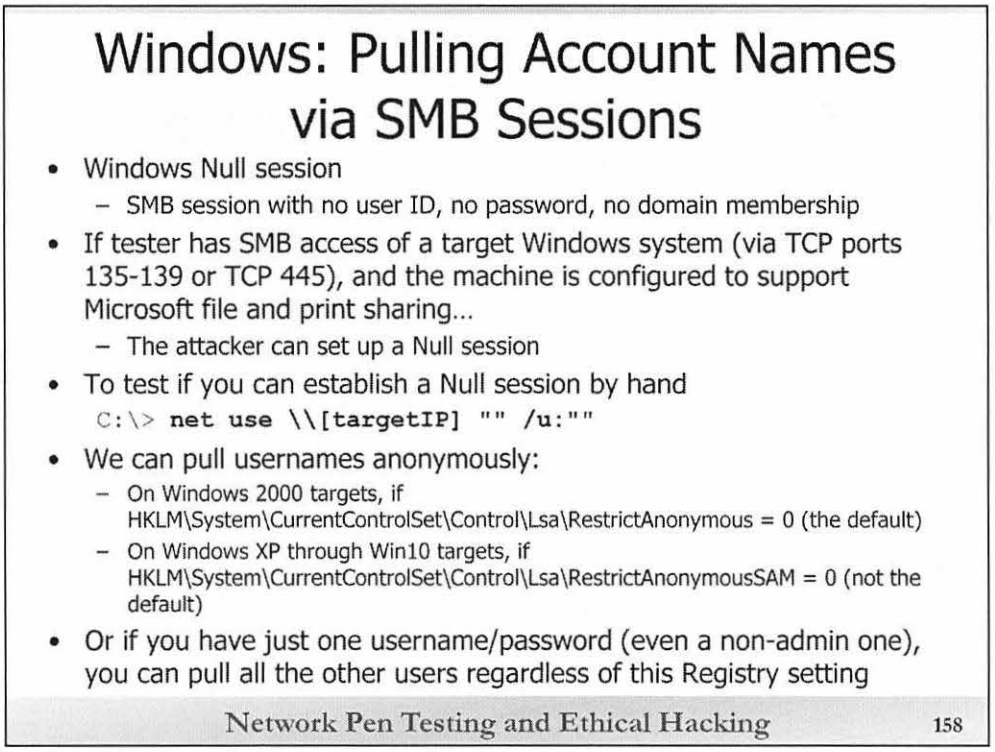
Alternatively, we could run the `finger` command locally, which shows us who is currently logged into the system. (Even if the finger service isn't active, the local finger command still works.) The finger command provides less information than we can get from looking at /etc/passwd (which shows us all users regardless of whether they are logged in our not), but it still might be interesting. In addition, we could run the `who` command to show us who is logged in, giving us pretty much the same information as finger, in a 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 in to some Linuxes can be used to formulate queries for usernames against it. For specific syntax of the ldapsearch command, consult the man pages.

From a Windows perspective, we have two 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.

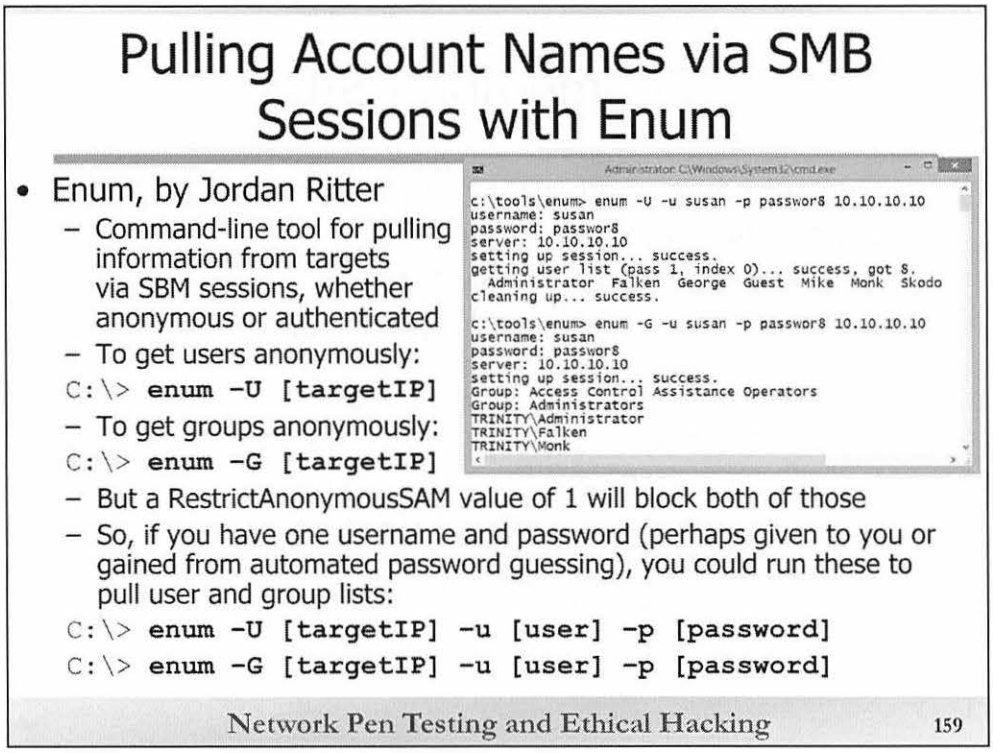


On Windows machines, a Null session is a Server Message Block (SMB) connection with a blank user ID, a blank password, and a blank domain. Literally, the information associated with who sets up such an SMB session is Null. If a tester can connect to the SMB over NetBIOS ports (TCP ports 135-139) or SMB 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 usernames on the target machine. We can pull usernames 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 get user information via this method. On Windows XP through Win8 machines, the ability to pull usernames 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.

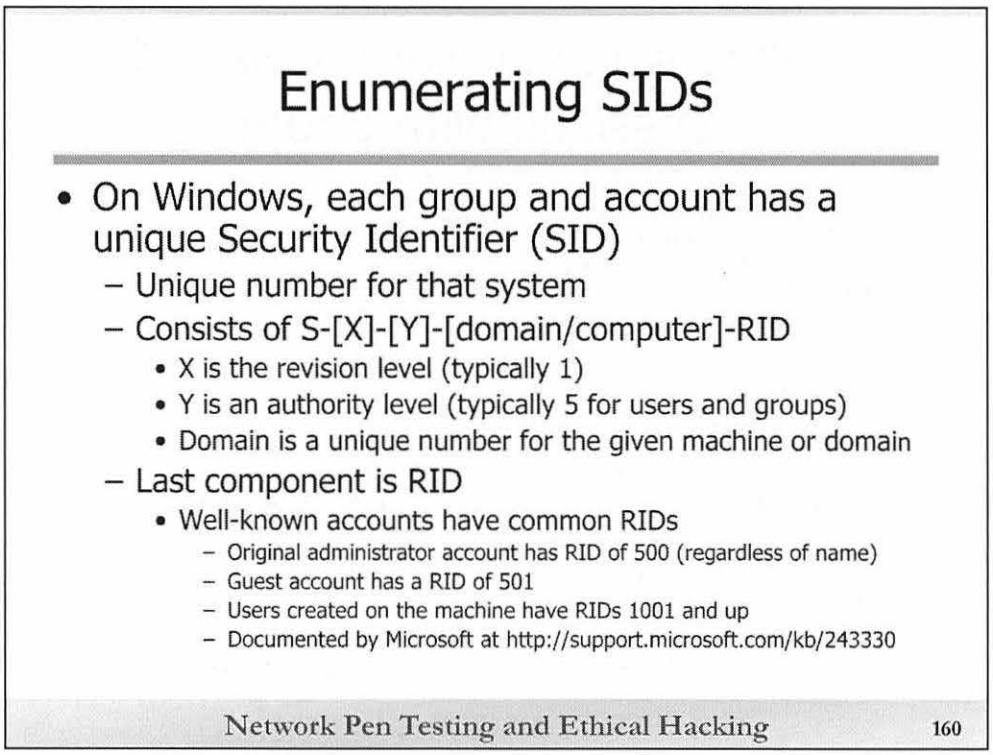
Alternatively, instead of going in anonymously with a Null session, if you have just one username with a password, you can pull all of the other users on the target machine or domain via SMB, even if the user account you have IS NOT an admin account. The Registry keys described above control only anonymous pulling of user names (that is, across a Null session). With just one non-admin account and password, you can pull all the other usernames in the environment regardless of the configuration of the RestrictAnonymousSAM Registry key value.



The enum tool included on the course USB can pull information across SMB sessions from a target Windows machine. 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. In addition, 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.

If you run enum with just a –U or –G (and without –u [user] –p [password] arguments), it sets up a Null session and tries to pull the information from the target anonymously. If the target is configured with the RestrictAnonymousSAM Registry key set to 1 (the default), enum will be unable to pull the data anonymously, giving you a "permission denied" error message.

If you get a username and password for the target Windows system (perhaps it was given to you as part of the test, or your automated password guessing tools determined it), you can run enum to set up an authenticated session, by giving it a –u [user] and –p [password] in its command line invocation. Then, regardless of the value of RestrictAnonymousSAM, you'll get the user and group information from the target. So, with a single username and password, you can determine all the other user accounts on the box using enum. We'll do this in a lab shortly.

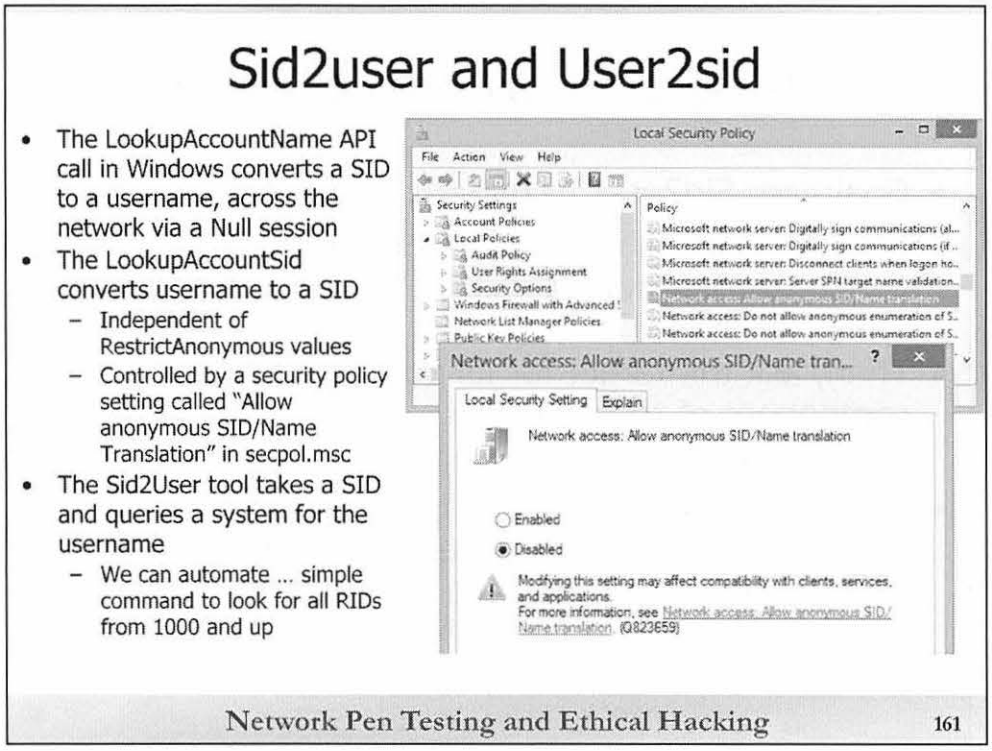


Besides pulling user and group names via Null sessions, we have another related method for pulling information based on the Security Identifier (SID) for each account. Windows assigns a unique SID for each user account and group defined on each system. The SID consists of several components, and is typically 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.

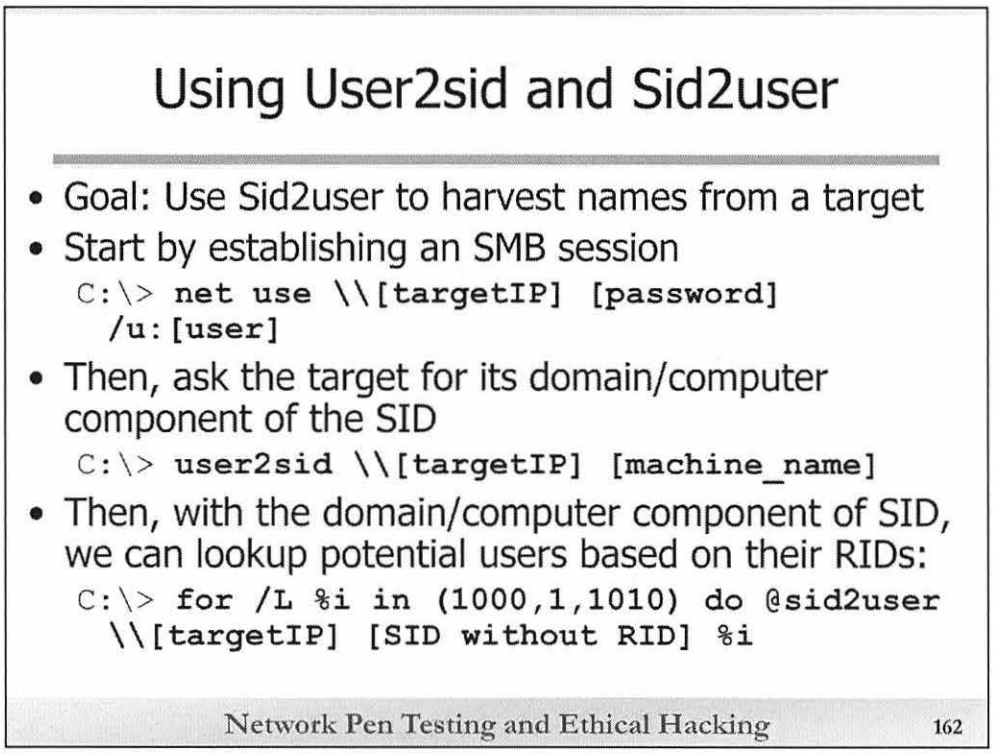
Various important accounts have known RIDs, with a comprehensive list of well-known SIDs and RIDs defined by Microsoft in an article at http://support.microsoft.com/kb/243330. The administrator account has an RID of 500, regardless of the name of the account. It is possible to rename the original administrator 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 usernames from a target machine by querying SID after SID, looking for those that successfully resolve into a username.



To harvest usernames from a target Windows machine with Sid2user, we could apply the following steps. First, we need to open an SMB session with the target. Unlike Enum and Winfingerprint, User2sid and Sid2user do not establish their own SMB sessions. We have to create one manually before running the tool.

```
C:\> net use \\[targetIP] [password] /u:[user]
```

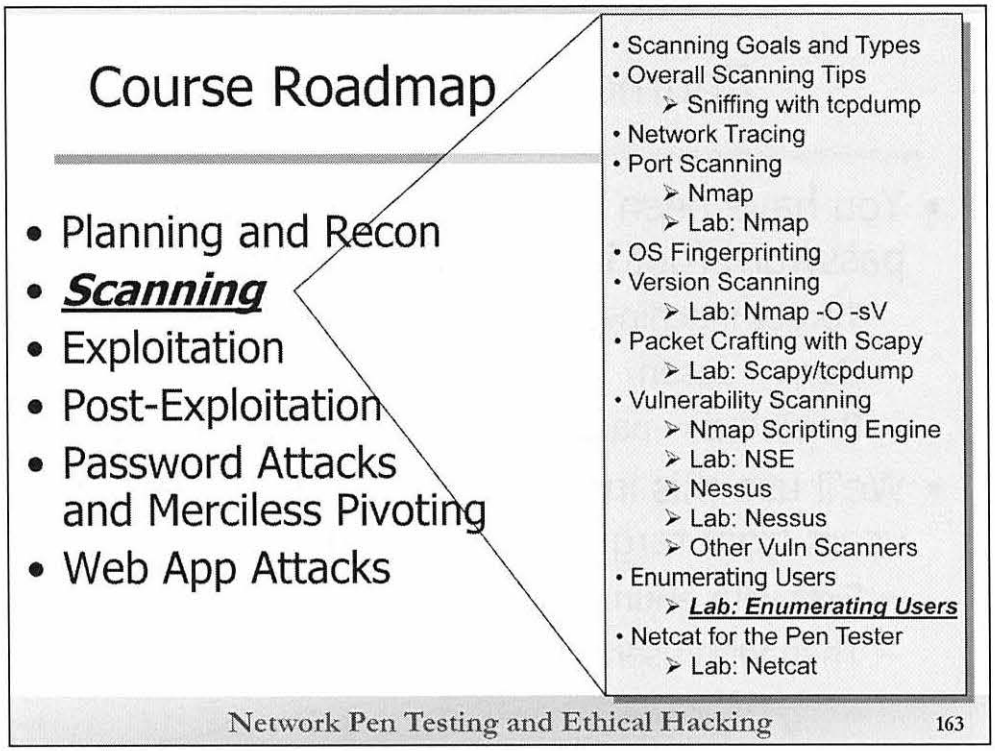
Now, we want to run Sid2user to ask the target machine about various SIDs. However, to do this, we need to know the [domain/computer] portion of the SID for the target machine. We can pull this by running User2sid against the target, with a username of the machine name :

```
C:\> user2sid \\[targetIP] [machine_name]
```

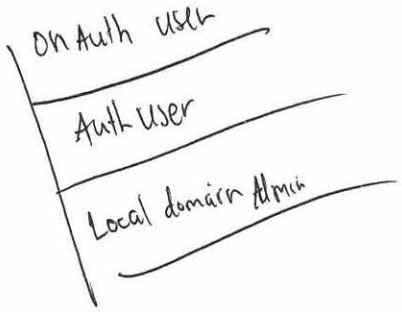
This command will tell us the overall SID for the target machine, a value of something like S-1-5-[some series of digits]. It's those series of digits we want because they are the unique numbers from which SIDs are built for that target machine. After we have those digits, we can then run an automated loop around them, asking for SID-to-username conversion for RIDs 1000 and up. We can accomplish this with a FOR loop as follows:

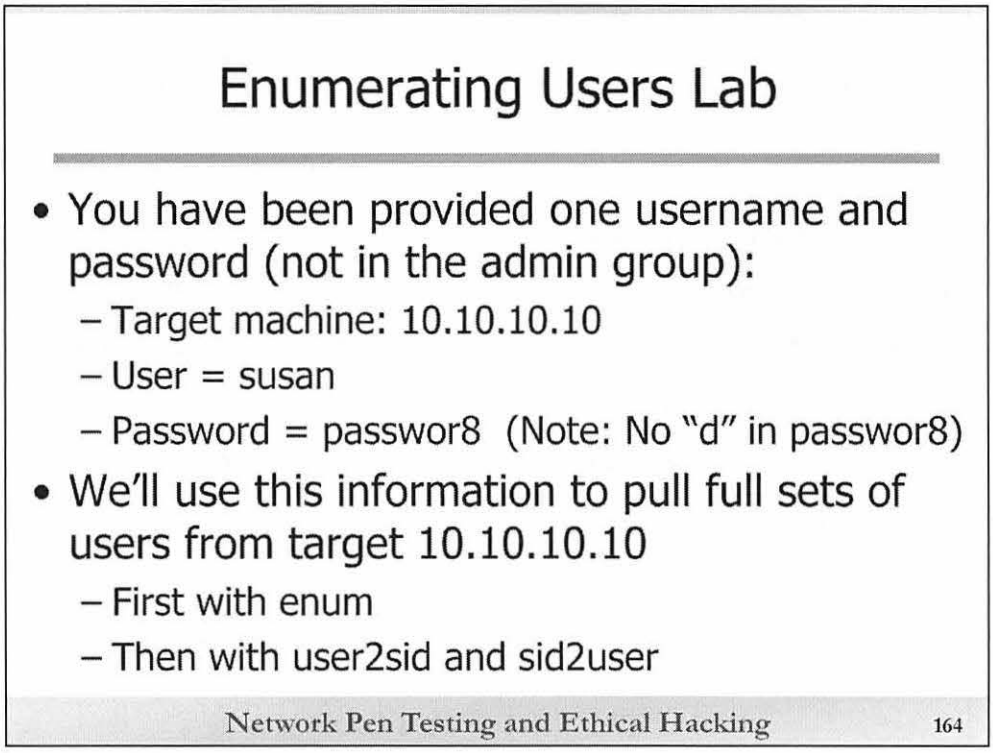
```
C:\> for /L %i in (1000,1,1010) do @sid2user \\[targetIP]
[domain/computer] %i
```

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 username. We'll do a lab on this later, and cover Windows FOR loops in more detail in session 560.4.



In our next lab, we look at methods for enumerating users on a target Windows machine. Specifically, we use the enum tool to get a list of users and groups via a Null session. Then, we explore useful techniques for applying User2sid and Sid2user to extract usernames from systems that have even enabled the RestrictAnonymous and RestrictAnonymousSam Registry keys.

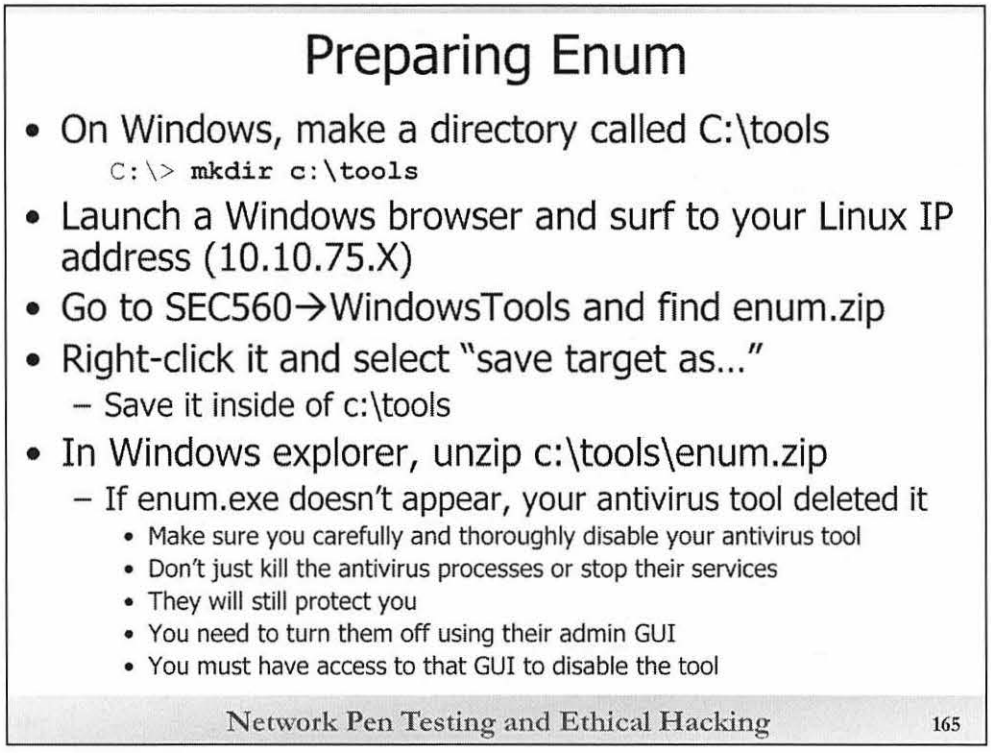




For this lab, we enumerate users on a target Windows machine, using both the enum and user2sid/sid2user methods.

Suppose you've been told that you are allowed to attack target 10.10.10.10 via SMB and that you are also given a non-admin username and password on the machine. The user is susan, and the password is passwor8. (Note that the password DOES NOT INCLUDE a "d." It is passwor8.)

We first attempt to enumerate users with enum, and then we rely on user2sid and sid2user.



For this component of the lab, you need to move the enum tool to your Windows machine. First, create a directory called C:\tools on Windows to store the Windows tools:

```
C:\> mkdir c:\tools
```

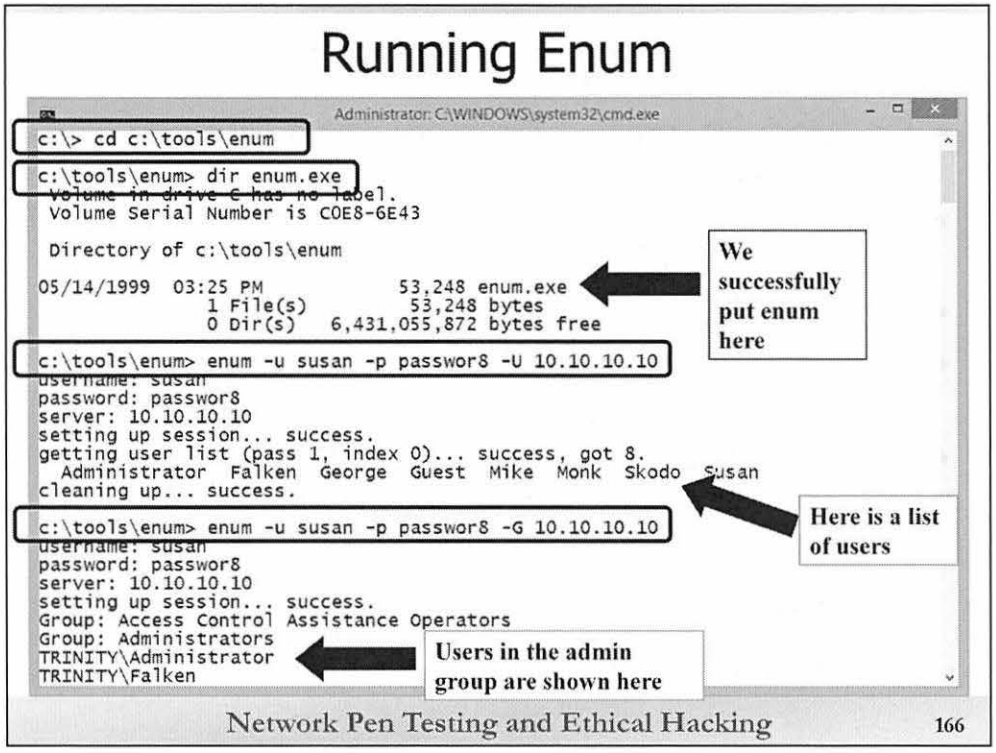
Then, launch a Windows browser and surf to http://10.10.75.X (where X is the last octet of your Linux IP address). Go to SEC560→WindowsTools. Right-click enum.zip to download it to your Windows machine in the c:\tools directory.

Then, in Windows File Explorer, navigate to c:\tools and unzip enum.

Your antivirus 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 antivirus vendors have written signatures for it. Thus, if you have such an antivirus tool, you must first disable it before you can unzip and run enum.

Disable your antivirus tool using the antivirus admin GUI. DO NOT DISABLE YOUR ANTIVIRUS TOOL BY KILLING ITS PROCESSES IN TASK MANAGER OR DISABLING ITS SERVICES IN THE SERVICES CONTROL PANEL! Most antivirus tools will still protect you even if you kill them using those methods. To disable antivirus protection, you must use the antivirus 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 lab.



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
```

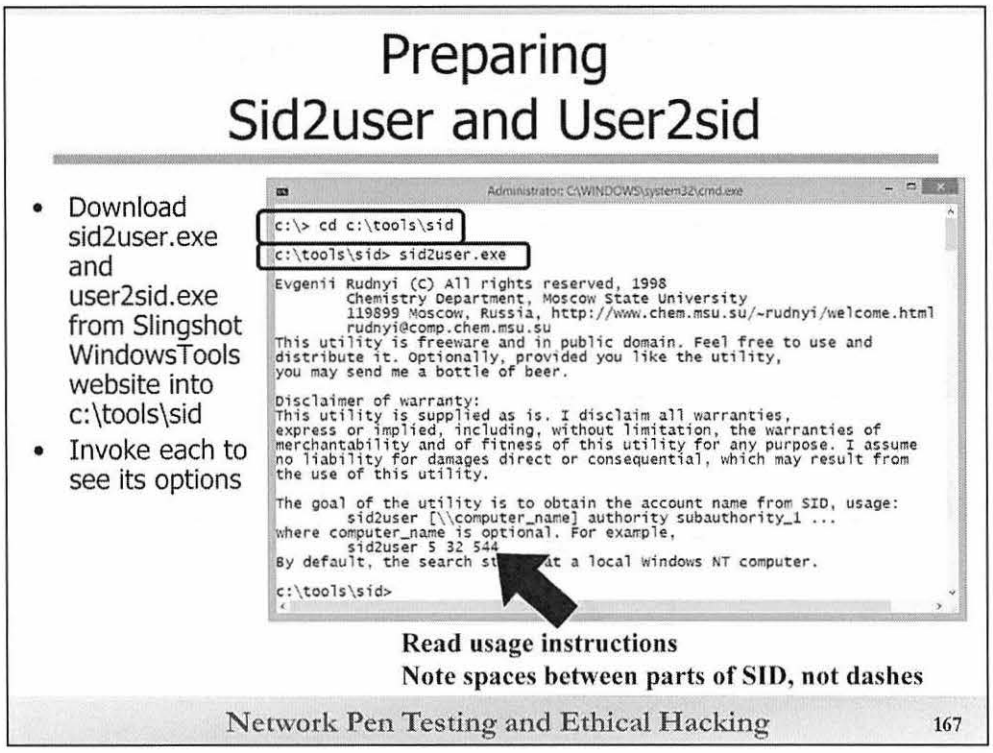
You should see enum.exe, with a size of 53,248 bytes.

Now, run enum against 10.10.10.10, configured to extract users, authenticating with the non-admin account we were given (susan with a password of passwor8):

```
C:\> enum -u susan -p passwor8 -U 10.10.10.10
```

Then, run it to extract groups:

```
C:\> enum -u susan -p passwor8 -G 10.10.10.10
```



Enum worked, but we have other options that are more widely applicable to machines that even have blocked extracting user and groups via the API calls used by enum. We could rely on Sid2user and User2sid instead. To start this part of the lab, create a directory called sid inside of your c:\tools directory:

```
C:\> mkdir c:\tools\sid
```

Then, use your Windows browser to access your Linux IP address (SEC560→WindowsTools) and right-click to download sid2user.exe and user2sid.exe to your c:\tools\sid directory.

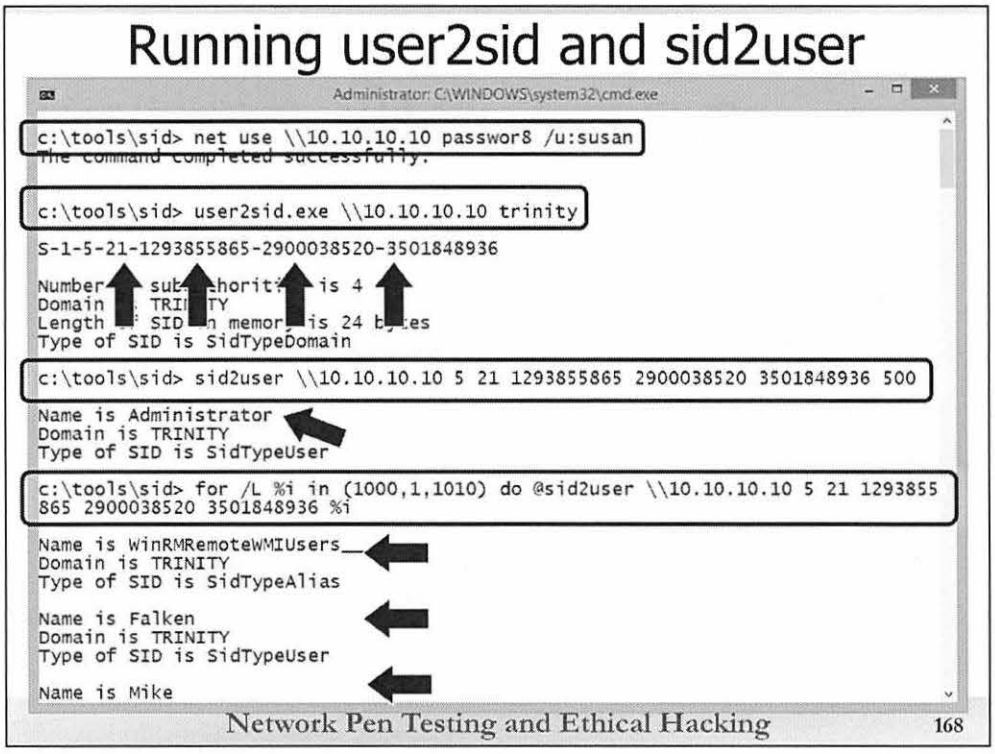
Then, change directories into c:\tools\sid:

```
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 an SMB session with the target with the non-admin credentials we've been given:

```
C:\> net use \\10.10.10.10 passwor8 /u:susan
```

Then, run the User2sid command to determine overall domain/computer component of the SID by providing it with the hostname of target. (We could get the hostname from an `nslookup` or `ping -a` command.)

```
C:\> user2sid \\10.10.10.10 trinity
```

Then, find out the administrator's name:

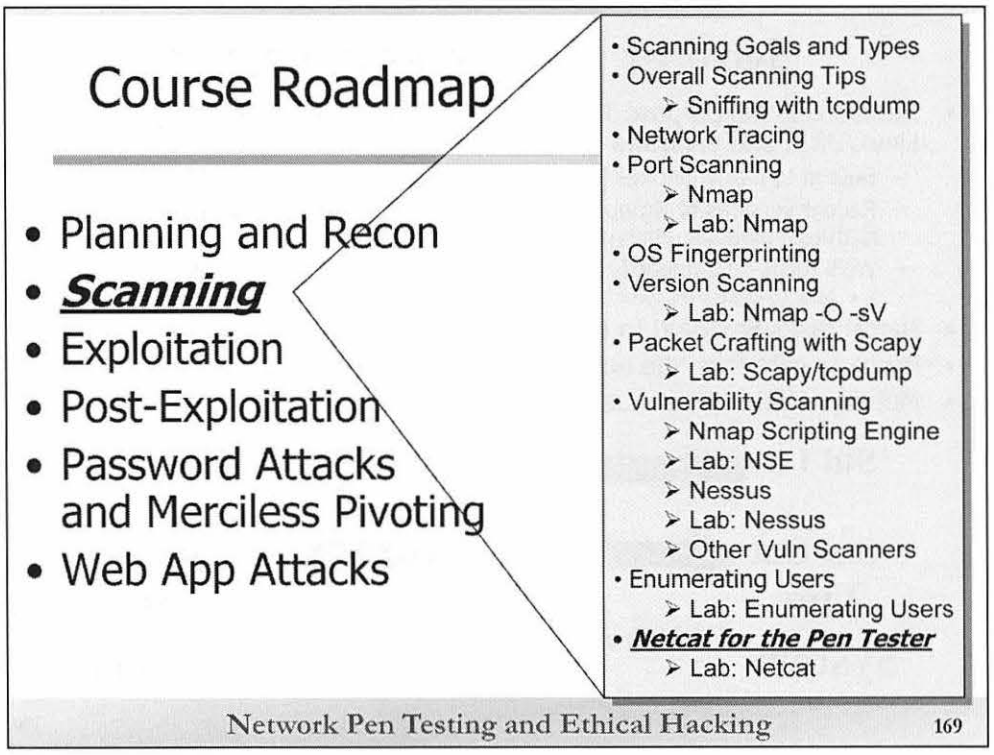
```
C:\> sid2user \\10.10.10.10 [domain number, starting with 5 followed by
space, followed by 21, followed by space, followed by 3 sets of digits] 500
```

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:

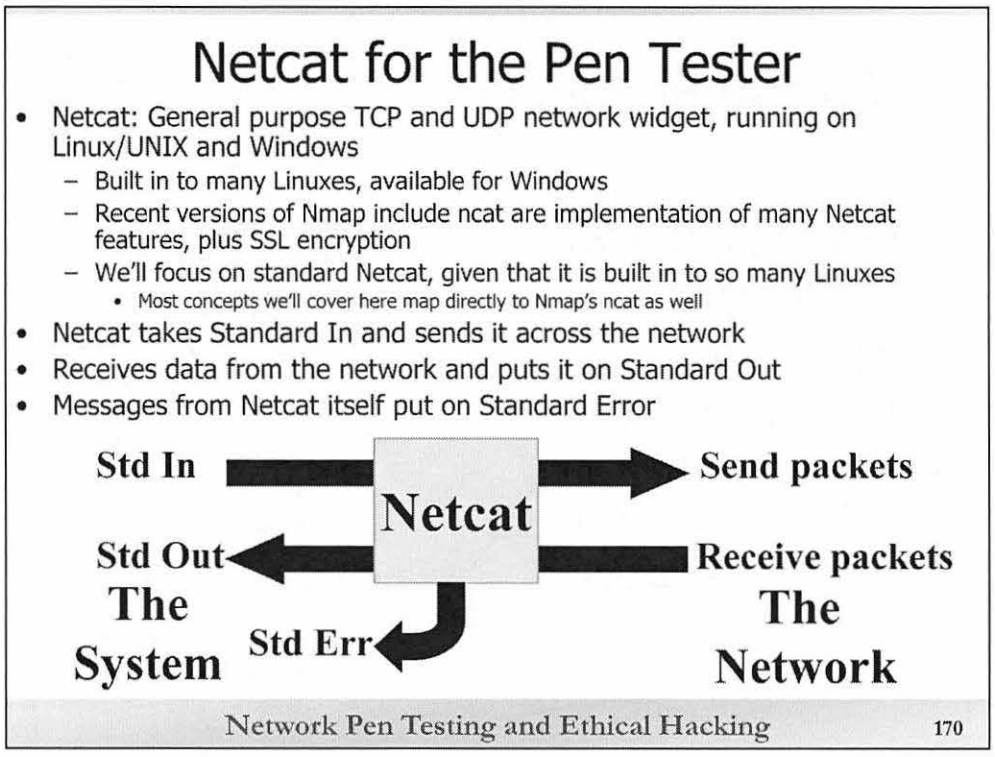
```
C:\> for /L %i in (1000,1,1010) do @sid2user \\10.10.10.10 [5 followed by
space, followed by 21, followed by space, followed by 3 sets of digits
separated by spaces] %i
```

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 560.4, we have a whole section on Windows command line capabilities, including FOR loops, for professional penetration testers and ethical hackers.



In our next section, we look at the incredibly flexible tool Netcat, specifically as applied to penetration testing and ethical hacking. Some of you may be Netcat fanatics, whereas 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 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 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 describe how the tool works, and then apply it directly to several important tasks.

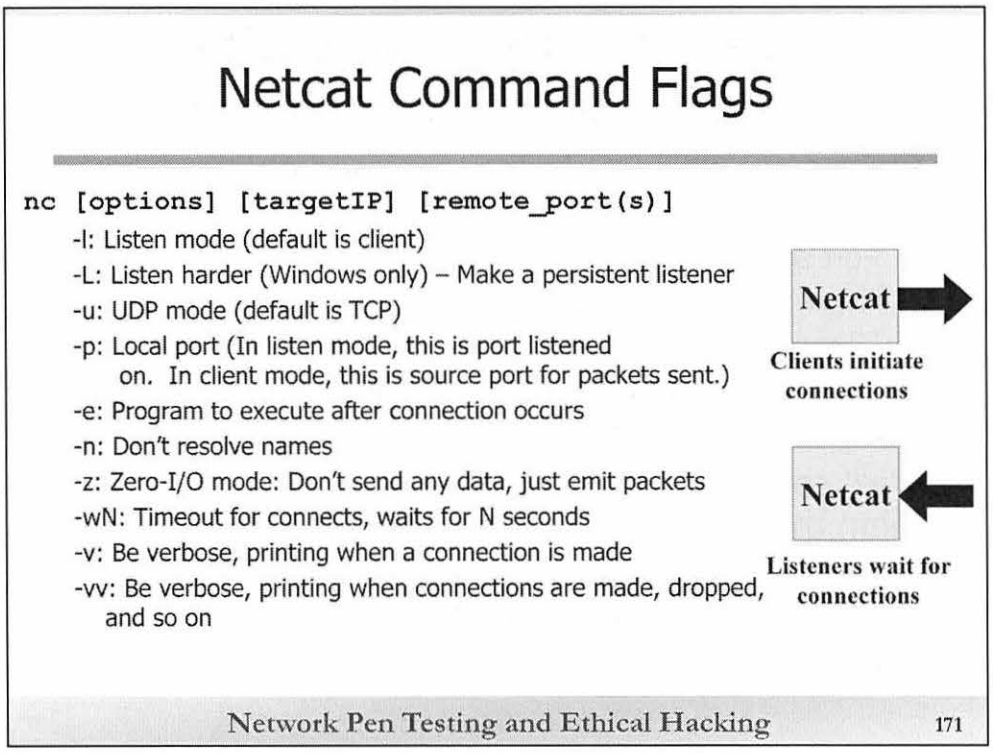


Netcat is a general-purpose TCP and UDP network widget for Linux/UNIX and Windows, sending data to or from a given TCP or UDP port, or listening for data to come in on a given TCP or UDP port. That's really it from a functionality-perspective. But, with those essential capabilities, we can use Netcat 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 cover in this class. There is also a great version of Netcat for Windows, which we also cover and use in this class. The Nmap development team re-implemented most of Netcat's features in its 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. An important property of Netcat involves its use of Standard Error. Any messages from Netcat associated with what it's doing on the network are sent to Standard Error. Reading and interacting with this form of Netcat commentary is useful, as you will see.

netcat builtin app, nmap not

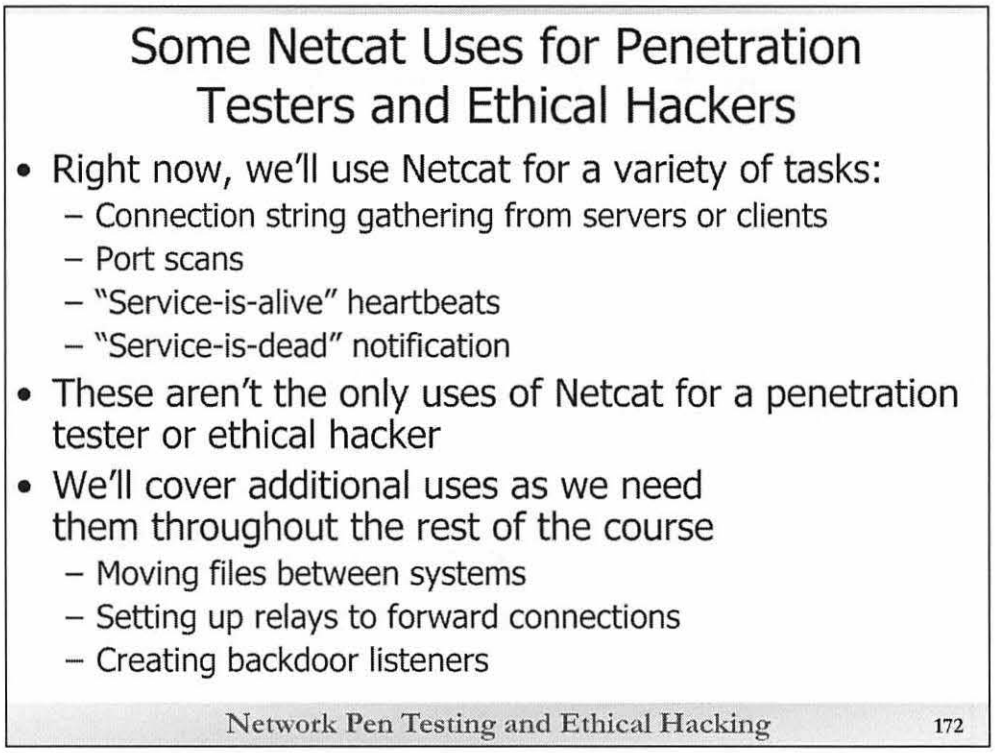


These are the most important command-line options for Netcat. Although there are (many) others, knowing these can help you diagnose Netcat's use in about 95 % of circumstances. The format is:

```
nc [options] [targetIP] [remote_port(s)]
```

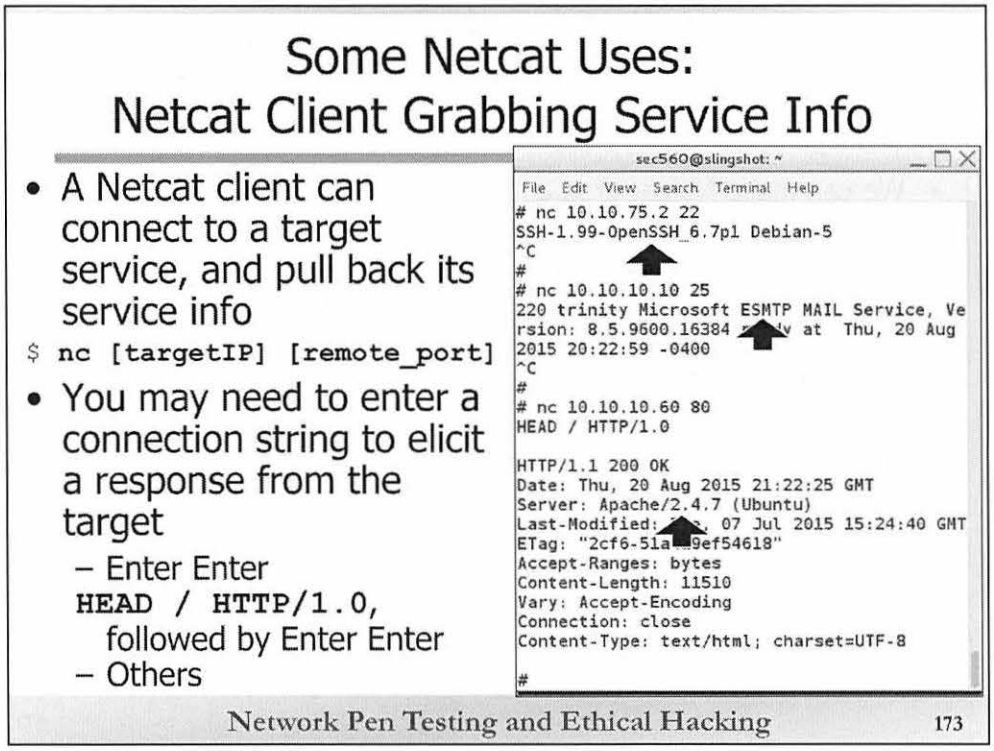
The target_system is simply the other side's IP address or domain name. It is required in client mode, of course (because we have to tell the client where to connect), and is optional in listen mode.

- **-l**: Listen mode. (The 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. (The default is TCP.)
- **-p**: Local port (In listen mode, this is the 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 waits 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.



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 lab 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.

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.



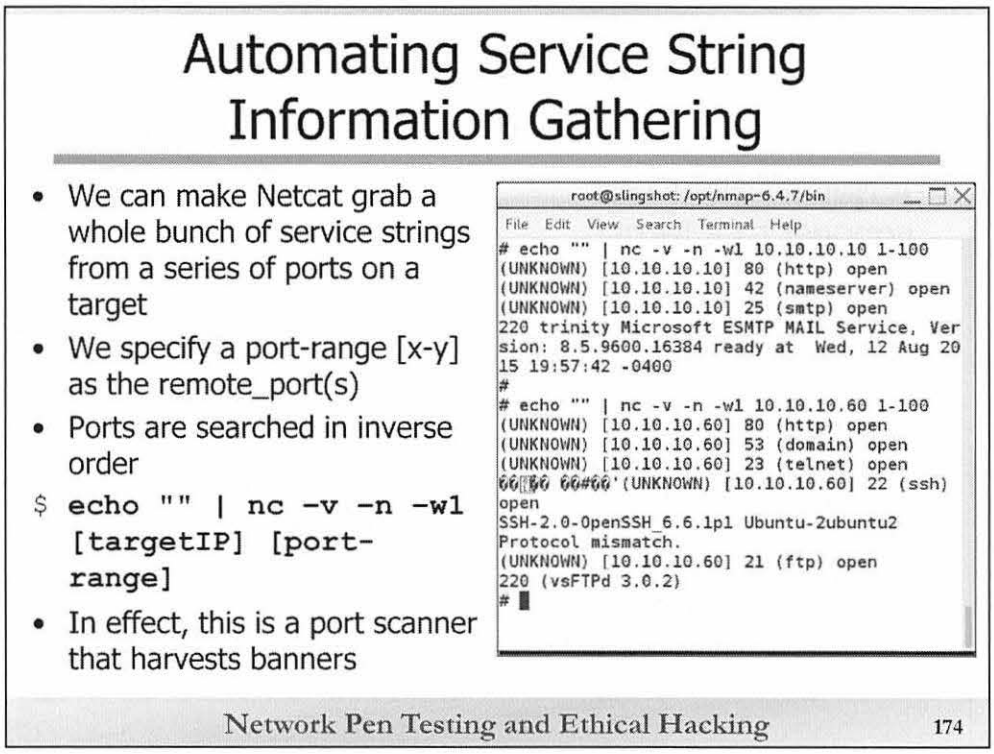
Now that we've had a brief discussion of those command flags, let's look at some practical uses of Netcat for penetration testers. You can harvest a connection string presented by services at connection by simply using a Netcat client to connect to the target service with the following syntax:

```
$ 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 pressing 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 screen shot in the slide, 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 press 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 . The system told us that it was running Apache, along with its version number and underlying operating system type. Although these connection strings can be altered to fool an attacker, they usually tell the truth.

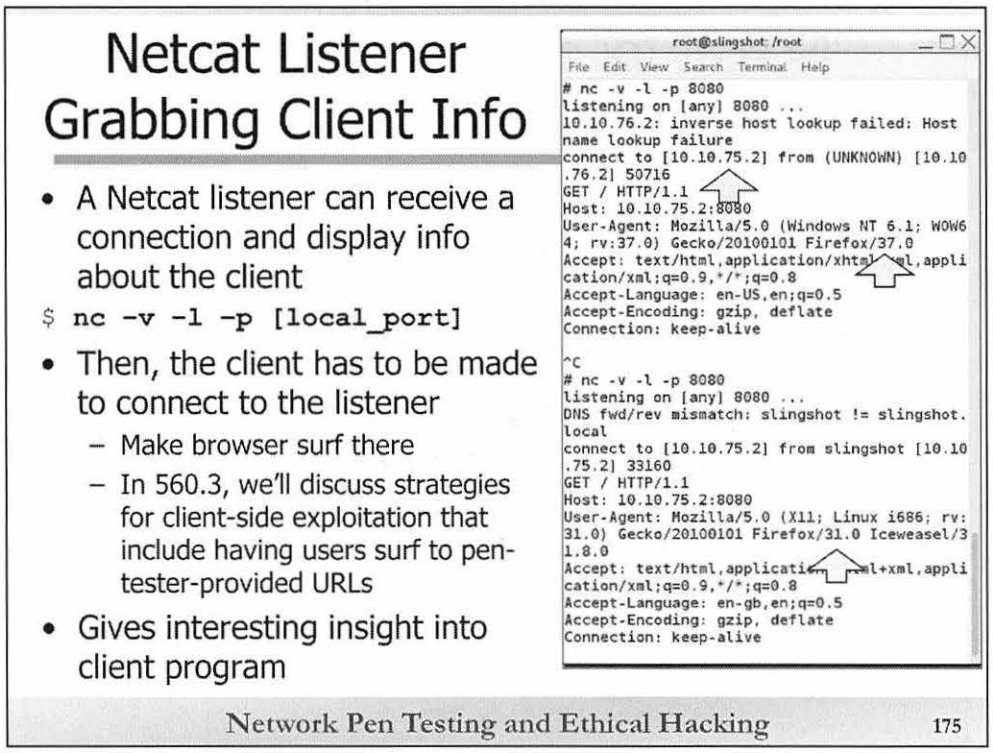


Of course, testing one target machine on one port is helpful, but we might want to automate this over a range of target ports. Netcat supports such functionality, with the [remote_port(s)] option taking a range of numbers, specified as [x-y]. This setting makes Netcat try to connect to the ports, starting at port y, and then decrementing by 1 going down until it tries to connect to port x. The –r flag makes Netcat work through ports in this range randomly but is used only 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 a lab 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 screen shot on the slide, 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-lowercase-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.

In the example in the screen shot on the slide, 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 press 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 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 and Ethical Hacking 176

As a professional penetration tester, while you are exploiting a network service on a target system, you 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 10 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
```

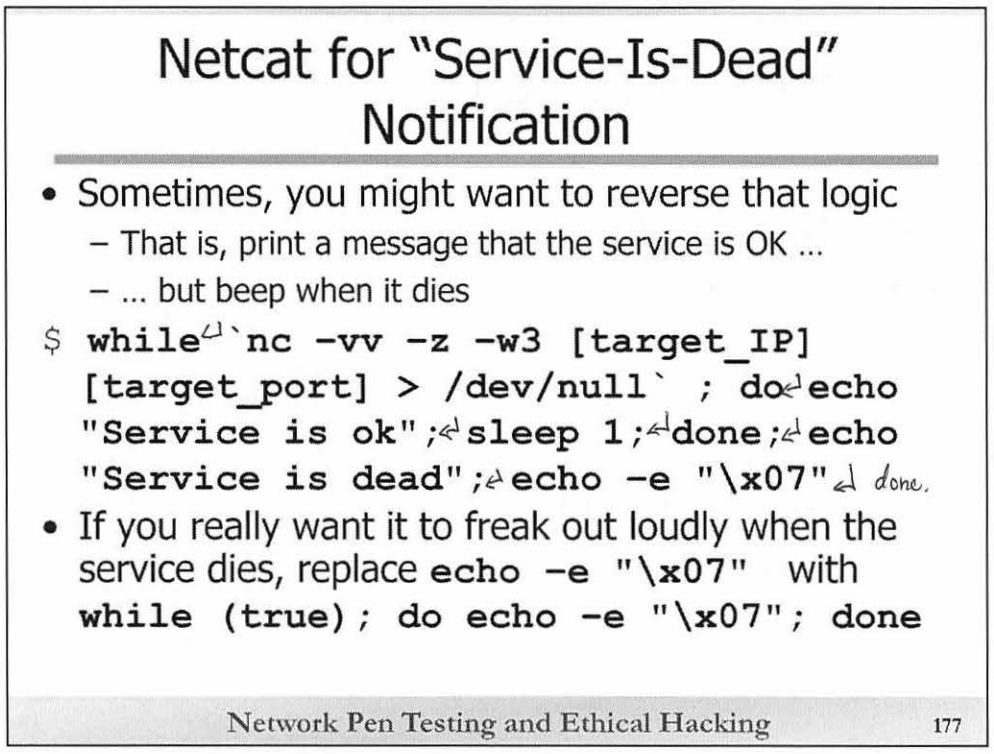
Little bell in Linux

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 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 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 useful.

Also, remember, even if you don't have sound support on your system, your terminal border still flashes visibly on the course Linux image when the system tries to beep.

```
while `nc -vv -z -w3 [target.IP] [port] > /dev/null`; do
    echo "service is ok";
    sleep 1;
done;
echo "service is dead"
while (true); do
    echo -e "\x07"
```

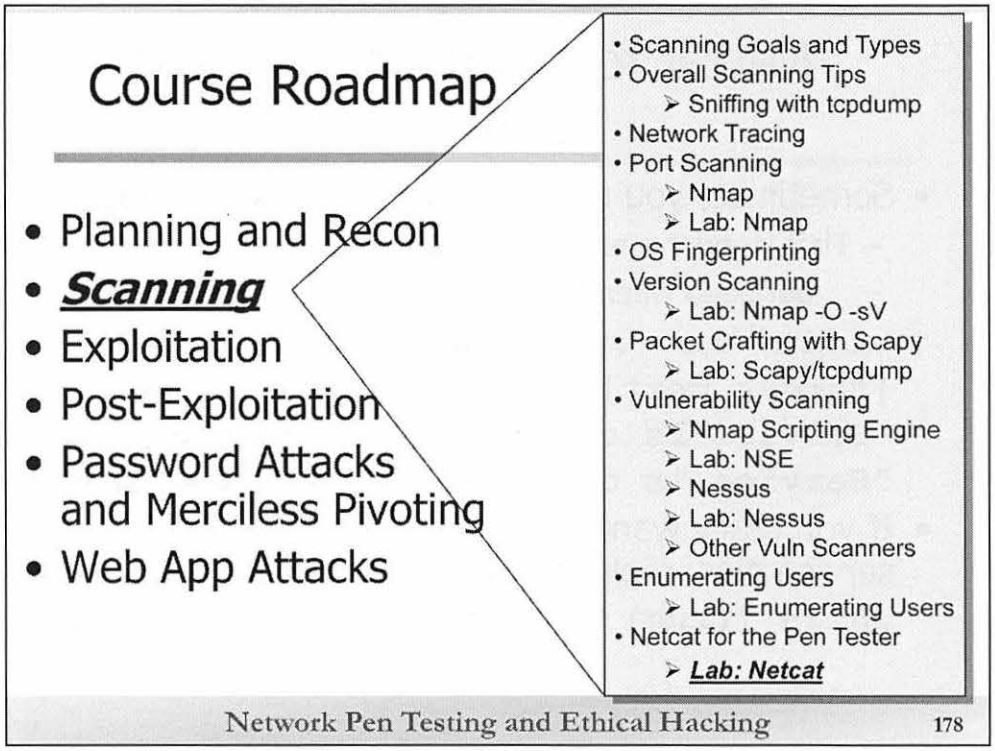


Sometimes, a tester may want different behavior from a service-monitoring command. Instead of a heartbeat showing that the service is alive, we might want a warning saying that the service is dead, beeping when it goes down. We can do that with the following shell command using Netcat:

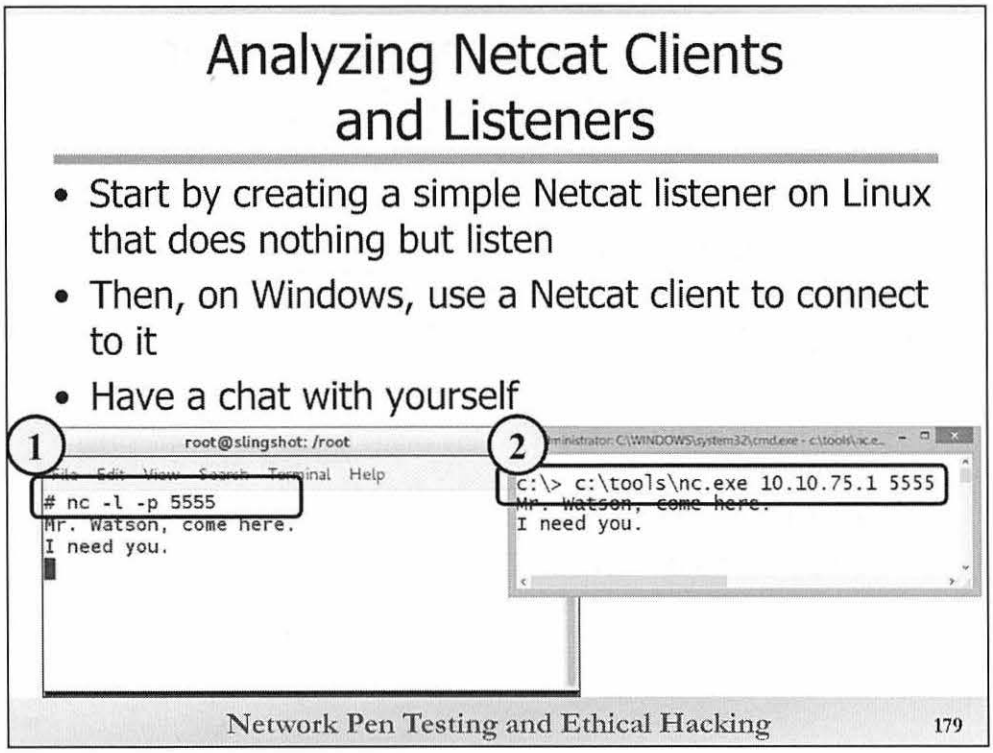
```
$ while `nc -vv -z -w3 [target_IP] [target_port] > /dev/null` ; do echo
"Service is ok"; sleep 1; done; echo "Service is dead"; echo -e "\x07"
```

PLEASE MAKE SURE THAT YOU USE A BACKTICK (AN UNSHIFTED TILDE AT THE UPPER-LEFT 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 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.



Now, let's apply some of these Netcat techniques in a hands-on lab.



We'll start this lab 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 downloading Netcat to your Windows machine, using your browser to surf to [YourLinuxIPaddress]/SEC560/WindowsTools/netcat.zip. Then, unzip Netcat from the course USB Windows directory (netcat.zip) into c:\tools on your Windows box. Then, run a listener on Linux:

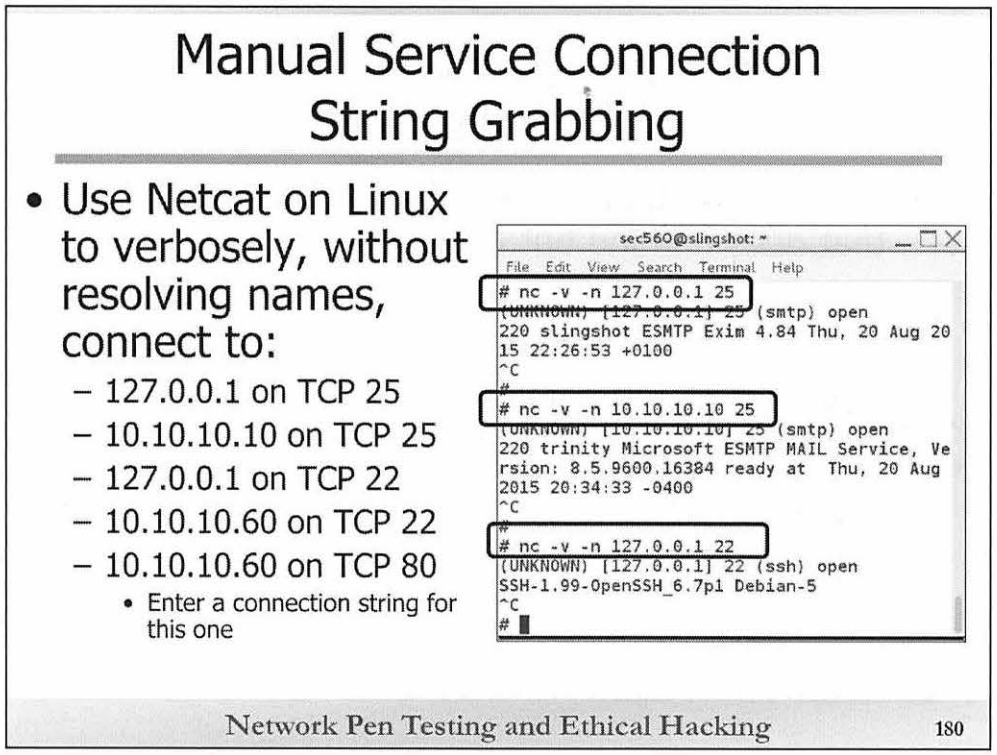
```
# nc -l -p 5555
```
← *Note: That is a dash-lowercase L, not a dash-one.*

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 press 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.



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.

Start by running an SMTP server (called exim4) on your localhost:

```
# service exim4 start
```

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? because of -n command that not resolve name

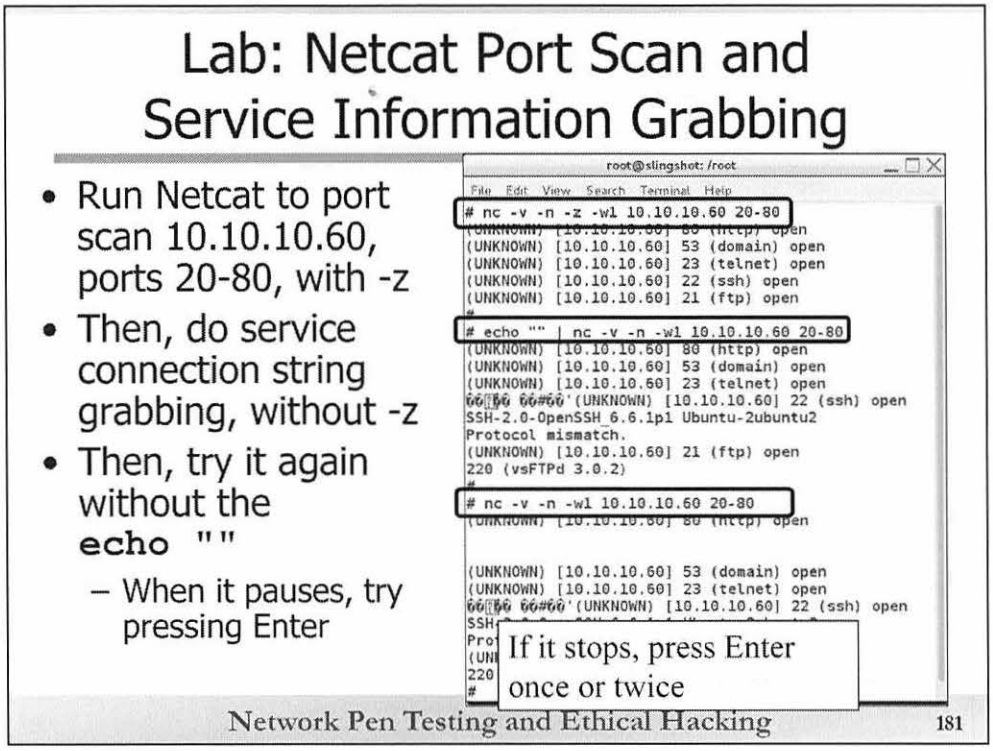
```
# nc -v -n 127.0.0.1 25
# nc -v -n localhost 25
```

Press 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. Use 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. *Note that you don't see any strings that come back from the services. You get an indication only of which ports are open, but not the connection string.*

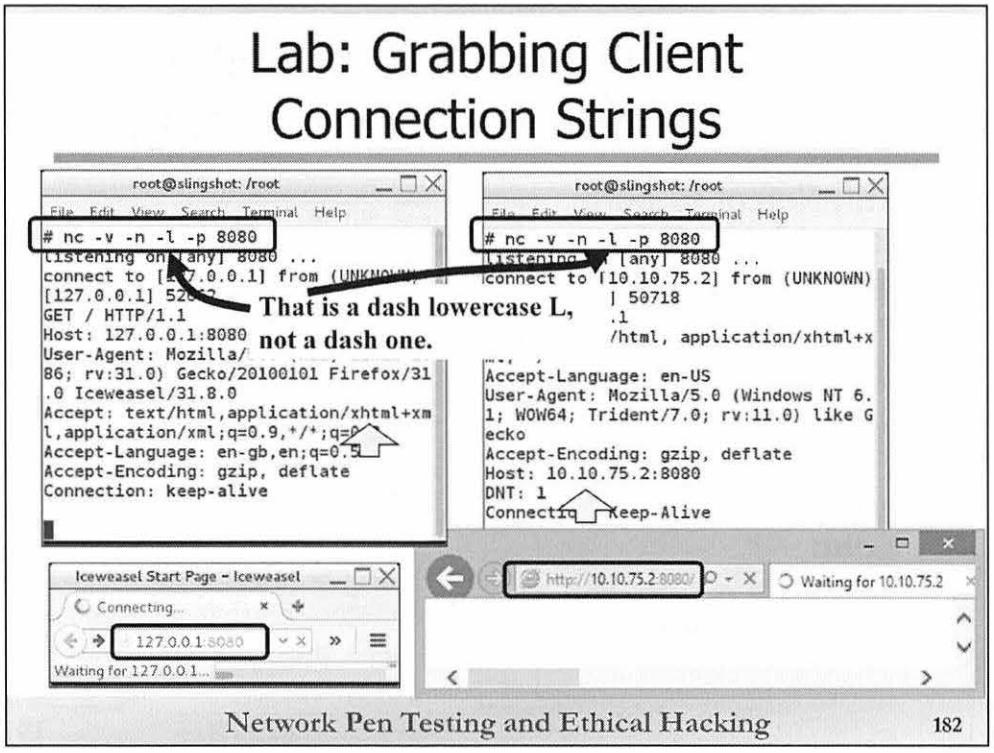
Then, let's do our connection string grabbing. Make sure you omit the –z from this command! If you include –z, you won't see the connection strings because Netcat will move on before it gets any data back. The –z and –w used together have that impact.

```
# 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. Press 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 8080, not resolving names of systems that connect there:

```
# nc -v -n -l -p 8080
```
← *Note: That is a dash-lowercase L, not a dash-one.*

Then, from another terminal on your Linux machine, run the Firefox browser, kicking it into the background with &:

```
# firefox &
```

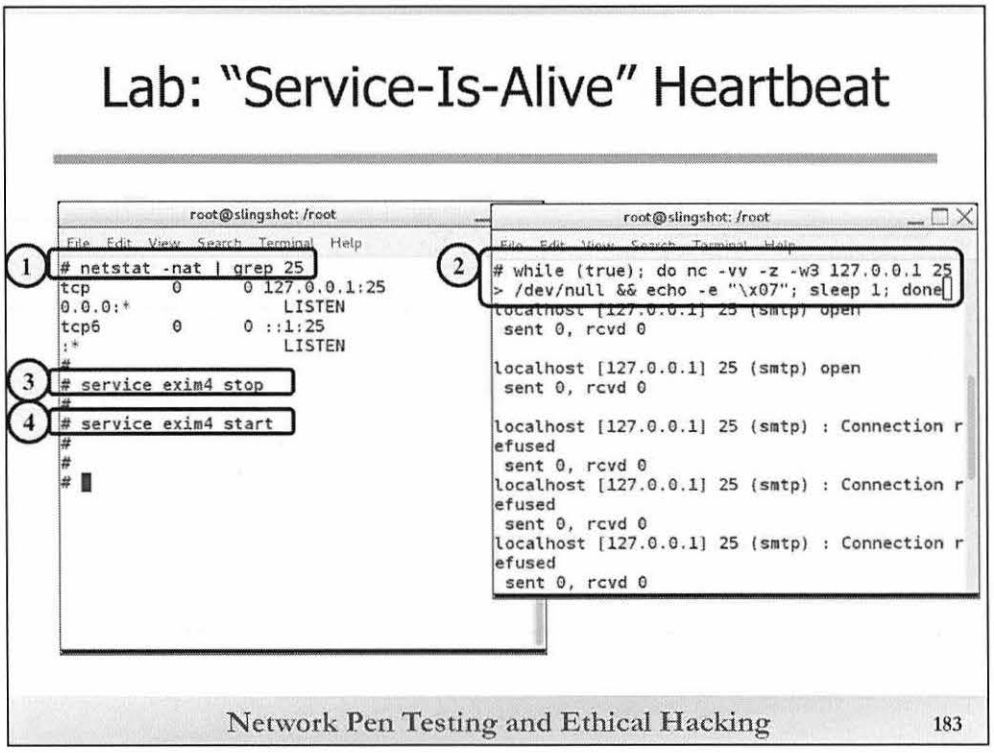
When the browser comes up, enter a URL for it to surf to http://127.0.0.1:8080

Look at your Netcat output, specifically the User-Agent string. It tells you the kind of browser that just accessed the Netcat listener.

Now, press CTRL-C in your Netcat window on Linux, and then restart your Netcat listener, again on TCP port 8080:

```
# nc -v -n -l -p 8080
```
← *Note: That is a dash-lowercase L, not a dash-one.*

Now, from Windows, run Internet Explorer, and have it surf to a URL of http://[LinuxIP]:8080. Note its User-Agent string. Try other Windows client programs that you might have, such as Firefox, Chrome, and others, having them surf to [YourLinuxIPaddr]:8080. 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, Chrome, and so on.



Let's try a "service-is-alive" heartbeat checker with Netcat. On our Linux machines, we have configured the system with a listening mail server on TCP port 25. The mail server is called "exim4" and is a Mail Transfer Agent (MTA) that is often used in place of sendmail.

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 exim4 service:

```
# service exim4 stop
```

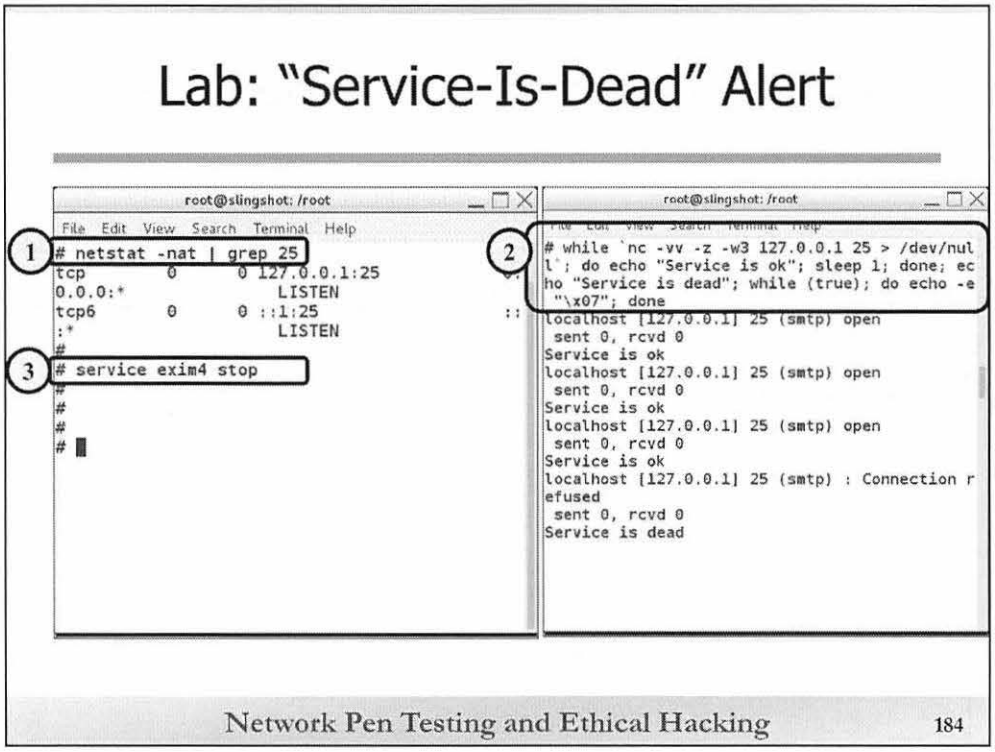
The Netcat service-checking heartbeat should go silent. The service is down!!!

In Step 4, when you bring your exim4 service back, the heartbeat starts again:

```
# service exim4 start
```

To end the heartbeat monitor, simply press CTRL-C in its window. If that doesn't stop it, press CTRL-Z, followed by:

```
# killall -9 nc
```



Next, let's create our "service-is-dead" red-alert message with Netcat, again monitoring our exim4 service.

In Step 1, verify that the port is listening:

```
# netstat -nat | grep 25
```

If the port isn't listening, start your exim4 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
```

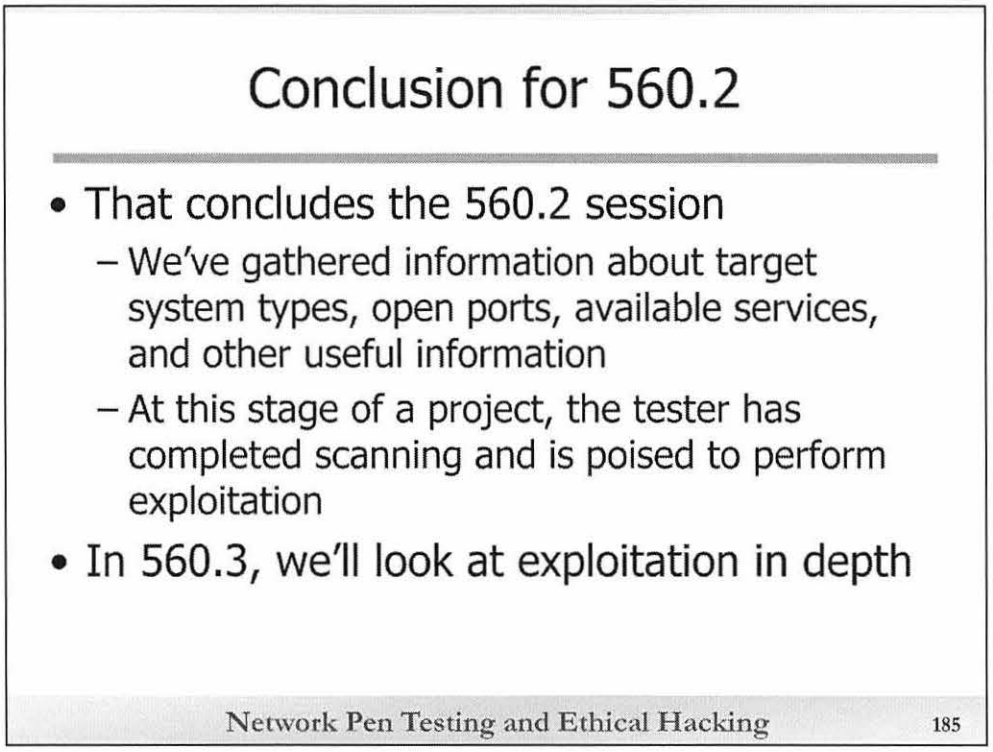
PLEASE MAKE SURE THAT YOU USE A BACKTICK (AN UNSHIFTED TILDE AT THE UPPER LEFT CORNER OF A U.S. ENGLIGH KEYBOARD) JUST BEFORE THE nc AND JUST AFTER THE /dev/null.

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 exim4 service:

```
# service exim4 stop
```

Your system should now make lots of noise. The service is down! Ouch. Stop it by pressing 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 exim4 service again.



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.