# CHAPTER 8

# PASSWORD CRACKING / BRUTE-FORCE TOOLS

A smile, a house key, a password. Whether you're trying to get into a nightclub, your house, or your computer, you will need something that only you possess. On a computer network, users' passwords have to be strong enough so that Dwayne can't guess Norm's password and Norm can't steal Dwayne's password (since Dwayne might have written it on the bottom of his keyboard). Bottom line—one weak password can circumvent secure host configurations, up-to-date patches, and stringent firewall rules.

In general an attacker has two choices when trying to ascertain a password. He can obtain a copy of the password or hash if encrypted and then use brute-force tools to crack the encrypted hash. Or he can try to guess a password. Password cracking is an old technique that is most successful because humans are not very good random sequence generators.

It's important that you understand how (and where) most passwords are stored so you know what these tools are doing and the method behind their madness. Passwords on Unix and Windows systems are stored with "one-way" hashes, and these passwords cannot be decrypted. Instead, a user login goes through a simple process. For example, Neil's password *abc123* is stored on a Unix system as the hash *kUge2g0BqUb7k* (remember, we can't decrypt this hash). When Neil tries to log into the system, imagine he mistypes the password as *abc124*. The Unix system calls its `crypt()` function on the password *abc124* to generate a temporary hash. The hash for *abc124* will not match the stored hash for *abc123*, so the system tells Neil he has entered an incorrect password. Notice what has happened here. The candidate password (*abc124*) is hashed and matched to the stored hash (*kUge2g0BqUb7k*). The stored hash is not decrypted. Taking the hash of a known word and comparing it to the target hash of the password is the basis for password cracking attacks.

Other brute-force techniques take advantage of rising hardware performance combined with falling hardware cost. This time-memory tradeoff means that it is actually easier to pregenerate an entire password dictionary and execute lookups of password hashes. These pregenerated dictionaries, often referred to as Rainbow Tables, consist of the entire key space for a combination of length and content. For example, one dictionary might consist of all seven character combinations of lower- and uppercase alphanumerics, while another dictionary might consist of nine character combinations of only lower- and uppercase letters. These dictionaries are encrypted with DES, MD5, or whatever target algorithm the user desires. Of course, these dictionaries can quickly reach the size of hundreds of gigabytes of data; however, desktop systems with a terabyte of storage can be reasonably constructed in 2005.

With these great dictionaries in hand, an attacker need only wait for a single search through the dictionary. The benefits of this technique become readily apparent when you consider searches for hundreds of passwords no longer require hundreds of redundant iterations through the key space. The real time to crack a password comes only once at the beginning when the attacker must first construct the dictionary—a process that can take weeks or months (or longer!) to complete.

Note that precomputed dictionaries can be trivially defeated by the use of password salts. These dictionaries rely on the expectation that the word "ouroboros" will always be hashed to 0639bbc687a6a1be21576dc562a08fc4 in the MD5 scheme. Yet if any text is prepended or appended to the password, then the nine-character lowercase source of

the hash can become much longer. For example, it is less likely that an attacker will have a 13-character MD5 dictionary to crack 6b149393cf909a49576032be9d73de85 (wormouroboros). Salts, if properly implemented, greatly reduce the threat of precomputed dictionary attacks.

# PAM AND UNIX PASSWORD POLICIES

Some popular Unix systems such as FreeBSD, Linux, and Solaris contain a Pluggable Authentication Module (PAM)—differing from ISS's PAM feature. The PAM controls any user interaction that requires a password from the user. This may be telnet access, logging into the console, or changing a password. PAM implementations are also available for stronger authentication schemes such as Kerberos, S/Key, and RADIUS. The configuration of PAM remains the same regardless of the method or application that is performing the authentication. So, let's focus on how to enforce a password policy using the PAM.

## Linux Implementation

This cracklib (or libcrack) library is a password-checking library developed by Alec Muffet and is part of the default install for Debian, Mandrake, RedHat, and SuSE distributions. It enables system administrators to establish password composition rules that a user's password must meet before the system accepts a password change. This is a proactive step to prevent a user from ever choosing an insecure password, rather than continuously auditing password files to see if someone has used a poor password. To implement password checking, we need only modify a text file containing the PAM configuration. This will be one of two possible files:

```
/etc/pam.conf
```

or

```
/etc/pam.d/passwd
```

The entry in the /etc/pam.conf file that relates to password changes looks similar to this:

```
passwd  password required       /lib/security/pam_cracklib.so retry=3
passwd  password required       /lib/security/pam_unix.so nullok use_authtok
```

This file is logically divided into five columns. The first column contains the service name—the name of the program affected by the instructions defined in the remaining columns. The /etc/pam.d/passwd file has only four columns because its name determines the `passwd` service. This configuration style merely separates each service name into files, rather than using a monolithic file for multiple services. Regardless of the configuration style, a service may have multiple entries. This is referred to as *stacking modules* for a service. Here's an example of /etc/pam.d/passwd with stacked modules:

```
password required       /lib/security/pam_cracklib.so retry=3
password required       /lib/security/pam_unix.so nullok use_authtok
```

The first column indicates the module type to which the entry corresponds. It can contain one of four types (we are interested in modifying the module type that controls password changes):

- **account** Controls actions based on a user's (that is, an account's) attributes, such as checking user read-access permissions against a file. For example, you could use an *account* entry to allow access to a resource such as a file share. However, without an *auth* entry, the user would not be able to log into the system.
- **auth** Performs a challenge/response with the user, such as prompting for a password. This is used whenever the system or resource is going to permit the user to log in.
- **password** Updates authentication information, such as changing a user's password. This is not used for validating a user to the system. All it does is permit access to the security system that controls the user's credentials.
- **session** Handles actions that occur before or after a service, such as auditing failed logins. For example, this could be used to immediately display the time of day after a user logs into the system. The first entry would be for an *auth* to validate the user's password, then the next entry would be a *session* that calls a PAM module to display the current time. Another use of the *session* could be to perform a specific function when the user logs out of the system, such as writing a log entry or expiring a temporary identifier.

The next column determines the *control* for a service, or how its execution should be handled. Successful execution implies that the service performs a function, such as changing a user's password. Failed execution implies that the service did not receive the correct data, such as the user's password. The following are the control handles:

- **requisite** If the service fails, all subsequent actions (stacked services) automatically fail. This means that nothing else in the stack will succeed.
- **required** If the service fails, process subsequent actions, but ultimately fail. If there are other actions in the stack, they might succeed but that will not change the outcome.
- **optional** If the service succeeds or fails, process subsequent actions. This will not have a bearing on the overall success of the action or anything in its stack.
- **sufficient** If the service succeeds and no *requisite* or *required* steps have failed, stop processing actions and succeed.

The next column contains the *module path* of the authentication library to use. The module path should contain the full path name to the authentication library. We will be using cracklib, so make sure that `pam_cracklib.so` is in this column.

The final column contains arguments to be passed to the authentication library. Returning to the first example of /etc/pam.conf, we see that the `pam_cracklib.so` module must succeed with the `retry=3` argument in order for users to change their passwords with the `passwd` program:

```
passwd  password required       /lib/security/pam_cracklib.so retry=3
```

## Cracklib Arguments

Cracklib actually provides more arguments than the simple `retry=N`. The `retry` argument merely instructs `passwd` how many times to prompt the user for the new password. The success or failure of a service that requires `pam_cracklib.so` relies on the number of "credits" earned by the user. A user can earn credits based on password content. Module arguments determine the amount of credit earned for the particular composition of a new password.

- **minlen=N** Default = 9. The minimum length, synonymous with amount of credit, that must be earned. One credit per unit of length. The actual length of the new password can never be less than 6, even with credit earned for complexity.
- **dcredit=N** Default = 1. The maximum credit for including digits (0–9). One credit per digit.
- **lcredit=N** Default = 1. The maximum credit for including lowercase letters. One credit per letter.
- **ucredit=N** Default = 1. The maximum credit for including uppercase letters. One credit per letter.
- **ocredit=N** Default = 1. The maximum credit for including characters that are not letters or numbers. One credit per letter.

Five other arguments do not directly affect credit:

- **debug** Record debugging information based on the system's `syslog` setting.
- **difok=N** Default = 10. The number of new characters that must not be present in the previous password. If at least 50 percent of the characters do not match, this is ignored.
- **retry=N** Default = 1. The number of times to prompt the user for a new password if the previous password did not meet the `minlen`.
- **type=text** Text with which to replace the word *UNIX* in the prompts "New UNIX password" and "Retype UNIX password."
- **use_authtok** Used for stacking modules in a service. If this is present, the current module will use the input given to the module above it in the configuration file rather than prompting for the input again. This may be necessary if the cracklib module is not placed at the top of a stack.

Arguments are placed in the last column of the row and are separated by spaces. For example, our administrator wants her users to create 15-character passwords, but the passwords receive up to two extra credits for using digits and up to two extra credits for "other" characters. The /etc/pam.d/passwd file would contain the following (the \ character represents a line continuation in this code):

```
password  required /lib/security/pam_cracklib.so \
                                  minlen=15 dcredit=2 ocredit=2
password  required /lib/security/pam_unix.so nullok use_authtok md5
```

Notice that the administrator added the `md5` argument to the `pam_unix.so` library. This enables passwords to be encrypted with the MD5 algorithm. Passwords encrypted with the Data Encryption Standard (DES) algorithm, used by default, cannot be longer than eight characters. Even with generous credit limits, it would be difficult to create a 15-credit password using eight characters! Passwords encrypted with the MD5 algorithm are effectively unlimited in length.

Now let's take a look at some valid and invalid passwords checked by the new /etc/pam.d/passwd file and their corresponding credits. Remember, `lcredit` and `ucredit` have default values of 1:

| | |
|---|---|
| password | 9 credits (8 length + 1 lowercase letter) |
| passw0rd! | 12 credits (9 length + 1 lowercase letter + 1 digit + 1 other character) |
| Passw0rd! | 13 credits (9 length + 1 uppercase letter + 1 lowercase letter + 1 digit + 1 other character) |
| Pa$$w00rd | 15 credits (9 length + 1 uppercase letter + 1 lowercase letter + 2 digits + 2 other characters) |

As you can see, high `minlen` values can require some pretty complex passwords. Twelve credits is probably the lowest number you will want to allow on your system, with fifteen being the upper threshold. Otherwise, you'll have to write down the password next to your computer in order to remember it! (Hopefully not.)

# OPENBSD LOGIN.CONF

OpenBSD, in a well-placed paranoiac departure from the limitations of DES-based encryption, includes the algorithm used only for compatibility with other Unix systems. System administrators have the choice of multi-round DES, MD5 encryption, and Blowfish. We've already mentioned that one benefit of MD5 encryption is the ability to use passwords of arbitrary length. Blowfish, developed by Bruce Schneier and peers, also accepts passwords of arbitrary length. It also boasts the advantage of being relatively slow. This might sound counterintuitive, but we'll explain why in the "John the Ripper" section.

## Implementation

OpenBSD does not use a PAM architecture, but it still maintains robust password management. The /etc/login.conf file contains directives for the encryption algorithms and controls that users on the system must follow. The entries in the login.conf file contain more instructions about user requirements than just password policies. The options explained here should be appended to existing options. The first value of each entry corresponds to a type of login class specified for users. It has a special entry of "default" for users without a class.

To determine the login class of a user, or to specify a user's class, open the /etc/ master.passwd file with the `vipw` utility. The login class is the fifth field in a user's password entry. Here's an example, showing the login classes in boldface:

```
root:$2a$06$T22wQ2dH...:0:0:daemon:0:0:Fede:/root:/bin/csh
bisk:$2a$06$T22wQ2dH...:0:0:staff:0:0::/home/bisk:/bin/csh
```

Partial entries in the login.conf file might contain the following (the \ character represents a line continuation in this code):

```
default:\
        :path=/usr/bin:\
        :umask=027:\
        :localcipher=blowfish,6
staff:\
        :path=/usr/sbin:\
        :umask=077:\
        :localcipher=blowfish,8
daemon:\
        :path=/usr/sbin:\
        :umask=077:\
        :localcipher=blowfish,8
```

This instructs the system to use the Blowfish algorithm for every user. The `,6` and `,8` indicate the number of rounds through which the algorithm passes. This slows the algorithm because it must take more time to encrypt the password. If a password takes longer to encrypt, then it will also take more time to brute force. For example, it will take much longer to go through a dictionary of 100,000 words if you use 32 rounds (`localcipher=blowfish,32`) of the algorithm as opposed to six rounds.

The most important entries of the login.conf file are `default`, because it applies to all users, and `daemon`, because it applies to the root user.

Each entry can have multiple options:

- **localcipher=algorithm** Default = old. This defines the encryption algorithm to use. The best options are `md5` and `blowfish,N` where *N* is the number of rounds to use (N < 32). The "old" value represents DES and should be avoided because passwords cannot be longer than eight characters, and current password crackers work very efficiently against this algorithm.

- **ypcipher=algorithm** Same values as `localcipher`. This is used for compatibility with a Network Information System (NIS) distributed login.
- **minpasswordlen=N** Default = 6. The minimum acceptable password length.
- **passwordcheck=program** Specifies an external password-checking program. This should be used with care because the external program could be subject to Trojans, errors, or buffer overflows.
- **passwordtries=N** Default = 3. The number of times to prompt the user for a new password if the previous password did not meet OpenBSD standards. A user can still bypass the standards unless this value is set to 0.

An updated login.conf file would contain the following (the `ftpaccess` class is purposefully weak for this example):

```
default:\
        :path=/usr/bin:\
        :umask=027:\
        :localcipher=blowfish,8:\
        :minpasswordlen=8:\
        :passwordretries=0
ftpaccess:\
        :path=/ftp/bin:\
        :umask=777:\
        :localcipher=old:\
        :minpasswordlen=6:\
        :passwordretries=3
staff:\
        :path=/usr/sbin:\
        :umask=077:\
        :localcipher=blowfish,12:\
        :minpasswordlen=8:\
        :passwordretries=0
daemon:\
        :path=/usr/sbin:\
        :umask=077:\
        :localcipher=blowfish,31
```

The policy specified by this file requires the Blowfish algorithm for all users, except those in the `ftpaccess` class. The password policy for the `ftpaccess` class represents the requirements of old-school Unix systems as noted by the reference to "old:." The passwords for users in the `staff` class, a class commonly associated with administrative privileges, are encrypted with 12 rounds. The root password, by default a member of `daemon`, must be encrypted with the maximum number of Blowfish rounds. Although the Blowfish and MD5 algorithms support an arbitrary password length, OpenBSD currently limits this to 128 characters. That's enough for a short poem!

**NOTE** One of the best places to search for passwords is in the history files for users' shells. Take a look at .history and .bash_history files for strange commands. Sometimes an administrator will accidentally type the password on the command line. This usually occurs when the administrator logs into a remote system or uses the `su` command and mistypes the command or anticipates the password prompt. We once found a root user's 13-character password this way!

# JOHN THE RIPPER

John the Ripper (*www.openwall.com/John/*) is probably the fastest, most versatile, and definitely one of the most popular password crackers available. It supports six different password hashing schemes that cover various flavors of Unix and the Windows LANMan hashes also known as NTLM (used by NT, 2000, and XP). It can use specialized wordlists or password rules based on character type and placement. It runs on at least 13 different operating systems and supports several processors, including special speed improvements for Pentium and RISC chips.

## Implementation

First, we need to obtain and compile John. The latest version is John-1.6.38, but you will need to download both John-1.6.38.tar.gz and John-1.6.tar.gz (or the .zip equivalent for Windows). The 1.6.38 version does not contain all of the documentation and support files from the original 1.6 version. After untarring John-1.6.38 in your directory of choice, you will need to go to the /src subdirectory.

```
[Paris:~] mike% tar zxvf john-1.6.38.tar.gz
[Paris:~] mike% tar zxvf john-1.6.tar.gz
[Paris:~] mike% cd john-1.6.38
[Paris:~] mike% john-1.6.38]# cd src
```

The next command is simple: `make OS name`. For example, to build John in a Cygwin environment, you would type **make win32-cygwin-x86-mmx**. For you BSD folks, **make freebsd-x86-mmx-elf** should do nicely. Simply typing **make** with no arguments will display a list of all supported operating system and processor combinations.

```
[Paris:~] mike% make macosx-ppc32-altivec-cc
```

John will then configure and build itself on your platform. When it has finished, the binaries and configuration files will be placed in the John-1.6.38/run directory. The development download does not include some necessary support files. You will need to extract these from the John-1.6.tar.gz file and place them in the /run subdirectory:

```
[Paris:~] mike% cd john-1.6.38/run
[Paris:~] mike% cp ../../john-1.6/run/all.chr .
[Paris:~] mike% cp ../../john-1.6/run/alpha.chr .
[Paris:~] mike% cp ../../john-1.6/run/digits.chr .
```

```
[Paris:~] mike% cp ../../john-1.6/run/LANMan.chr .
[Paris:~] mike% cp ../../john-1.6/run/password.lst .
```

If all has gone well, you should be able to test John. For the rest of the commands, we will assume that you are in the John-1.6.38/run directory. First, verify that John works by generating a baseline cracking speed for your system:

```
[Paris:~] mike% ./john -test
Benchmarking: Traditional DES [128/128 BS AltiVec]... DONE
Many salts:     621260 c/s real, 635235 c/s virtual
Only one salt:  543974 c/s real, 567822 c/s virtual

Benchmarking: BSDI DES (x725) [128/128 BS AltiVec]... DONE
Many salts:     21324 c/s real, 21583 c/s virtual
Only one salt:  20249 c/s real, 20747 c/s virtual

Benchmarking: FreeBSD MD5 [32/32 X2]... DONE
Raw:    2904 c/s real, 2988 c/s virtual

Benchmarking: OpenBSD Blowfish (x32) [32/32]... DONE
Raw:    240 c/s real, 246 c/s virtual

Benchmarking: Kerberos AFS DES [24/32 4K]... DONE
Short:  86610 c/s real, 88918 c/s virtual
Long:   231782 c/s real, 235073 c/s virtual

Benchmarking: NT LM DES [128/128 BS AltiVec]... DONE
Raw:    4193K c/s real, 4395K c/s virtual
```

Two benchmarks deserve attention: `FreeBSD MD5` and `NT LM DES`. The cracks per second (c/s) difference between these two is a factor over 1400 (executed on a Mac OSX system). This means that a complete brute-force attack will take more than 1400 times longer against password hashes on a FreeBSD system than against a Windows NT system! OpenBSD Blowfish takes even longer to brute force. This is how an encryption algorithm can be more resistant to brute-force attacks than another type of algorithm. Instead of saying that one algorithm is more secure than the other, it would be fairer to say that Blowfish is more resistant to a brute attack.

## Cracking Passwords

Now let's crack a password. John will accept three different password file formats. In reality, John can crack any password encrypted in one of the formats listed by the `-test` option. All you have to do is place it into one of the formats the application will accept. If you are using a Unix passwd file or output from the pwdump tool, which is mentioned later in this chapter, then you should not have to modify the file format. Here are five different examples of password file formats that John knows how to interpret (the password hashes are in boldface):

1. root:**rf5V5.Ce31sOE**:0:0::
2. root:**KbmTXiy.OxC.s**:11668:0:99999:7:-1:-1:1075919134
3. root:**$1$M9/GbWfv$sktn.4pPetd8zAwvhiB6.1**:11668:0:99999:7:-1:-1:1075919134
4. root:**$2a$06$v3LIuqqw0pX2M4iUnCVZcuyCTLX14lyGNngtGSH4/dCqPHK8RyAie**:0:0::::::
5. Administrator:500:**66bf9d4b5a703a9baad3b435b51404ee**:17545362d694f996c37129225df11f4c:::

Following are the systems from which the previous five password hashes were obtained. Notice that even though there is a significant difference in the operating system, the file formats are similar. Also, realize that you can crack Solaris passwords using the Windows version of John—all you need is the actual password hash; the operating system is irrelevant.

1. Solaris DES from /etc/passwd
2. Mandrake Linux DES from /etc/shadow
3. FreeBSD MD5 from /etc/shadow
4. OpenBSD Blowfish from /etc/master.password
5. Windows 2000 LAN Manager from \WINNT\repair\SAM or \WINNT\system32\config

Passwords can be cracked from applications other than Unix and Windows systems. To crack one of these passwords, simply copy the hash (in bold in each example) into the second field of a Unix password file format:

- Cisco devices
  Original entry: `enable secret 5` **$1$M9/GbWfv$sktn.4pPetd8zAwvhiB6.1**
  John entry: `cisco:` **$1$M9/GbWfv$sktn.4pPetd8zAwvhiB6.1::::**
- Apache .htaccess files that use DES-formatted password hashes. Apache also supports passwords hashed with the SHA-1 and MD5 algorithms, but these are not compatible with John.
  Original .htaccess entry: `dragon:` **yJMVYngEA6t9c**
  John entry: `dragon:` **yJMVYngEA6t9c**::::
- Other DES-based passwords from applications such as WWWBoard.
  Original passwd.txt file: WebAdmin:**aepTOqxOi4i8U**
  John entry: WebAdmin:**aepTOqxOi4i8U**:0:3:www.victim.com::

To crack a password file using John's default options, you supply the filename as an argument. We'll use three different password files for the examples in this chapter: passwd.unix contains passwords hashed by the DES algorithm, passwd.md5 contains passwords hashed by the MD5 algorithm, and passwd.LANMan contains Windows NT–style passwords:

```
[Paris:~] mike% ./john passwd.unix
Loaded 189 passwords with 182 different salts (Traditional DES [64/64 BS MMX])
```

John automatically selects the correct encryption algorithm for the hashes and begins cracking. Press any key to display the current cracking statistics—CTRL-C will stop John. If a password is cracked, John will print it on the screen and save the cracked hash for future use. To view all the cracked passwords for a specific file use the `-show` option:

```
[Paris:~] mike% ./john -show passwd.unix
2buddha:smooth1:0:3:wwwboard:/:/sbin/sh
ecs:asdfg1:11262:0:40:5::11853:
informix:abc123:10864:0:40:5::12689:
kr:grant5:11569:0:35:5::11853:
mjs:rocky22:11569:0:35:5::11853:
np:ny0b0y:11572:0:35:5::11853:
```

All the cracked passwords are saved in the John.pot file, which is a text file that will grow as the number of passwords you collect grows.

Poor passwords, regardless of their encryption scheme, can be cracked in a few minutes to a day. Stronger passwords may take weeks or months to break; however, we can use some tricks to try and guess these stronger passwords more quickly. We can use complicated dictionary files (files with foreign words, first names, sports teams, science-fiction characters), use specific password combinations (always at least two numbers and a punctuation mark), or distribute the processing across multiple computers.

John's default dictionary is the password.lst file. This file contains common passwords that should show up most often among users. You can find several alternative dictionary files on the Internet using a simple Google search. One of the best (at 15MB) is bigdict.zip. Supply the `-wordfile` option to instruct John to use an alternative dictionary:

```
[Paris:~] mike% ./john -wordfile:password.lst passwd.unix
Loaded 188 passwords with 182 different salts (Traditional DES [64/64 BS MMX])
guesses: 0  time: 0:00:00:01 100%  c/s: 333074  trying: tacobell - zhongguo
```

We can even perform some permutations on the words in the dictionary using the `-rules` option:

```
[Paris:~] mike% ./john -wordfile:password.lst -rules passwd.unix
Loaded 188 passwords with 182 different salts (Traditional DES [64/64 BS MMX])
guesses: 0  time: 0:00:00:58 100%  c/s: 327702  trying: Wonderin - Zenithin
```

To understand what the `-rules` option did, let's take a look at the John.conf file (or the John.ini file for the 1.6 nondevelopment version). Here is a portion of the John.conf file that applies permutations to our wordlist (comments begin with the # symbol):

```
[List.Rules:Wordlist]
# Try words as they are
:
# Lowercase every pure alphanumeric word
-c >3!?XlQ
# Capitalize every pure alphanumeric word
```

```
-c >2(?a!?XcQ
# Lowercase and pluralize pure alphabetic words
<*>2!?Alp
# Lowercase pure alphabetic words and append '1'
<*>2!?Al$1
```

Although it looks like we'd need a Rosetta Stone to decipher these rules, they are not really that difficult to understand. The basic syntax for many of these rules is derived from the crack utility written by Alec Muffet (remember libcrack?). Imagine that the system's password policy requires every password to begin with a number. Obviously, we don't need to bother trying to guess "letmein" since it doesn't match the policy, but "7letmein" might be valid. Here's a rule to prepend digits to a word:

```
# Prepend digits (adds 10 more passes through the wordlist)
^[0123456789]
```

We can break this rule down into three parts. The ^ symbol indicates that the operation should occur at the beginning of the word. In other words, it should prepend the subsequent character. The square brackets `[` and `]` contain a set of characters, rather than using just the next character after the ^. The digits `0123456789` are the specific characters to prepend. So, if our rule operates on "letmein," it will make a total of 10 guesses from "0letmein" through "9letmein."

The placeholder rules that signify where to place a new character are as follows:

| Symbol | Description | Example |
|---|---|---|
| ^ | Prepends the character | ^[01]<br>0letmein<br>1letmein |
| $ | Appends the character | $[!.]<br>letmein!<br>letmein. |
| i[n] | Inserts a character at the *n* position | i[4][XZ]<br>letXmein<br>letZmein |

We can specify any range of characters to insert. The entire wordlist will be rerun for each additional character. For example, a wordlist of 1000 words will actually become an effective wordlist of 10,000 words if the 10 digits 0–9 are prepended to each word.

Here are some other useful characters to add to basic words:

- **[0123456789]** Digits
- **[!@#$%^&*()]** SHIFT-digits
- **[,.?!]** Punctuation

We can use conversion rules to change the case or type (lower, upper, *e* to *3*) of characters or remove certain types of characters:

- **?v** Vowel class (a, e, i, o, u)
- **s?v.** Substitute vowels with dot ( . )
- **@@?v** Remove all vowels
- **@@a** Remove all *a*'s
- **sa4** Substitute all *a*'s with *4*
- **se3** Substitute all *e*'s with *3*
- **l*** Where * is a letter to be lowercase
- **u*** Where * is a letter to be uppercase

Rules are an excellent method of improving the hit rate of password guesses, especially rules that append characters or *l33t* rules that swap characters and digits. Rules were more useful when computer processor speeds were not much faster than a monkey with an abacus. Nowadays, when a few hundred dollars buys chips in the 3+ GHz range, you don't lose much by skipping a complex rule phase and going straight to brute force.

Nor will complex rules and extensive dictionaries crack every password. This brings us to the brute-force attack. In other words, we'll try every combination of characters for a specific word length. John will switch to brute-force mode by default if no options are passed on the command line. To force John to use a specific brute-force method, use the `-incremental` option:

```
[Paris:~] mike% ./john -incremental:LANMan passwd.LANMan
Loaded 1152 passwords with no different salts (NT LM DES [64/64 BS MMX])
```

The default John.conf file has four different incremental options:

- **All** Lowercase, uppercase, digits, punctuation, SHIFT+
- **Alpha** Lowercase
- **Digits** 0 through 9
- **LANMan** Similar to `All` with lowercase removed

Each incremental option has five fields in the John.conf file. For example, the LANMan entry contains the following fields:

- **[Incremental:LANMan]** Description of the option
- **File = ./LANMan.chr** File to use as a character list
- **MinLen = 0** Minimum length guess to generate
- **MaxLen = 7** Maximum length guess to generate
- **CharCount = 69** Number of characters in list

Whereas the `All` entry contains these fields:

- **[Incremental:All]** Description of the option
- **File = ./all.chr** File to use as a character list
- **MinLen = 0** Minimum length guess to generate
- **MaxLen = 8** Maximum length guess to generate
- **CharCount = 95** Number of characters in list

The `MinLen` and `MaxLen` fields are the most important fields because we will modify them to target our attack. `MaxLen` for LANMan hashes will never be more than seven characters. Raise the `CharCount` to the `MaxLen` power to get an idea of how many combinations make up a complete brute-force attack. For example, the total number of LANMan combinations is about 7.6 trillion. The total number of combinations for `All` is about 6700 trillion! Note that it is counterproductive to use `incremental:All` mode against LANMan hashes as it will unnecessarily check lowercase and uppercase characters.

If we have a password list from a Unix system in which we know that all the passwords are exactly eight characters, we should modify the incremental option. In this case, it would be a waste of time to have John bother to guess words that contain seven or less characters:

```
[Incremental:All]
File = ./all.chr
MinLen = 8
MaxLen = 8
CharCount = 95
```

Then run John:

```
[Paris:~] mike% ./john -incremental:All passwd.unix
```

Only guesses with exactly eight characters will be generated. We can use the `-stdout` option to verify this. This will print each guess to the screen:

```
[Paris:~] mike% ./john -incremental:All -stdout
```

This can be useful if we want to redirect the output to a file to create a massive wordlist for later use with John or another tool that could use a wordlist file, such as Nessus or THC-Hydra.

```
[Paris:~] mike% ./john -makechars:guessed
Loaded 3820 plaintexts?Generating charsets... 1 2 3 4 5 6 7 8 DONE
Generating cracking order... DONE
Successfully written charset file: guessed (82 characters)
```

## Restore Files and Distributed Cracking

You should understand a few final points about John to be able to manage large sets of passwords at various stages of completion. John periodically saves its state by writing to a restore file. The period is set in the John.conf file:

```
# Crash recovery file saving delay in seconds
Save = 600
```

The default name for the restore file is *restore,* but this can be changed with the `-session` option.

```
[Paris:~] mike% ./John -incremental:LANMan -session:pdc \
> passwd.LANMan
Loaded 1152 passwords with no different salts (NT LM DES
[64/64 BS MMX])
```

The contents of the restore file will be similar to this:

```
REC2
5
-incremental:LANMan
-session:pdc
passwd.LANMan
-format:lm
6
0
47508000
00000000
0
-1
488
0
8
3
2
6
5
2
0
0
0
```

Lines nine and ten in this file (shown in bold) contain the hexadecimal value of the total number of guesses completed. The number of possible combinations is well over any number that a 32-bit value can represent, so John uses two 32-bit fields to create a 64-bit number. Knowledge of these values and how to manipulate them is useful for performing distributed cracking. Let's take our restore file and use it to launch two concurrent

brute-force cracks on two separate computers. The restore file for the first computer would contain this:

```
REC2
4
-incremental:LANMan
passwd.LANMan
-format:lm
4
0
00000000
00000000
0
-1
333
0
8
15
16
0
0
0
0
0
0
```

The restore file for the second computer would contain this:

```
REC2
4
-incremental:LANMan
passwd.LANMan
-format:lm
4
0
00000000
0000036f
0
-1
333
0
8
15
16
0
0
0
```

```
0
0
0
```

Thus, the first system will start the brute-force combination at count zero. The second computer will start further along the LANMan pool at a "crypt" value of `0000036f 00000000`. Now the work has been split between both computers and you don't have to worry about redundant combinations. A good technique for finding the right "crypt" values is to let a system run for a specific period.

For example, imagine you have a modest collection of 10 computers. On each of these systems, John runs about 400,000 c/s. It would take one of these systems about 30 weeks to go through all seven character combinations of a common LANMan hash (69^7 combinations). Run John on one of the systems for one week. At the end of the week, record the "crypt" value. Take this value and use it as the starting value in the restore file on the second system, and then multiply the value by two and use that as the starting value for the next system. Now, 10 systems will complete a brute-force attack in only three weeks. Here is the napkin arithmetic that determines the "crypt" multiplier, *X*, that would be necessary to write 10 session files—one for each system. The first system would start guessing from the zero mark, the next system would start guessing at the zero plus *X* mark, and so on:

Total time in weeks:

*Tw* = (69^7 / cracks per second) / (seconds per week)
*Tw* = (69^7 / 400,000) / (604800) = 30.8 weeks

"crypt" multiplier:

*X* = *Tw* / (10 systems)
*X* = 30.8 / 10 = 3
"crypt" value after one week (hexadecimal, extracted from restore file):
00030000 00000000

Here are the distributed "crypt" values (in hexadecimal notation). These are the values that are necessary to place in the session file on each system:

System 1 = 0
System 2 = "crypt" * X = 00090000 00000000
System 3 = "crypt" * X * 2 = 00120000 00000000
System N = "crypt" * X * (N - 1) = restore value
System 10 = "crypt" * X * 9 = 00510000 00000000

This method is far from elegant, but it's effective when used with several homogenous computers. Another method for distributing the work uses the `-external` option. Basically, this option allows you to write custom password-guessing routines and methods. The external routines are stored in the John.conf file under the List.External directives. Simply supply the `-external` option with the desired directive:

```
[Paris:~] mike% ./john -external:Parallel passwd.LANMan
```

**NOTE** If you're going to use this method, be sure to change the `node=1` line to `node=2` on the second computer's John.conf file. Also, the implementation of this node method is not effective for more than two nodes because the `if (number++ % total)` will create redundant words across some systems.

## Is It Running on My System?

The biggest indicator of John the Ripper running on your system will be constant CPU activity. You can watch process lists (`ps` command on Unix or through the process viewer for Windows) as well, but you will not likely see John listed. If you're trying to rename the executable binary to something else, like "inetd " (*note the extra space after the* d), it will not work without changing a few lines of the source code.

## Case Study: Attacking Password Policies

The rules that you can specify in the John.conf file go a long way toward customizing a dictionary. We've already mentioned a simple rule to add a number in front of each guess:

```
# Prepend digits (adds 10 more passes through the wordlist)
^[0123456789]
```

But what about other scenarios? What if we notice a trend in the root password scheme for a particular network's Unix systems? For example, what if we wanted to create a wordlist that used every combination of upper- and lowercase letters for the word *bank*? A corresponding rule in John.conf would look like this:

```
# Permutation of "ban" (total of 8 passes)
i[0][bB]i[1][aA]i[2][nN]
```

You'll notice that we've only put the first three letters in the rule. This is because John needs a wordlist to operate on. The wordlist, called password.lst, contains the final two letters:

```
k
K
```

Now, if you run John with the new rule against the shortened password.lst file, you will see the following:

```
$ ./John.exe -wordfile:password.lst -rules –stdout
bank
bank
bank
bank
```

## Attacking Password Policies *(continued)*

```
bAnk
bAnK
bANk
bANK
Bank
BanK
BaNk
BaNK
Bank
BAnK
BANk
BANK
words: 16  time: 0:00:00:00 100%  w/s: 47.05  current: BANK
```

Here's another rule that would attack a password policy that requires a special character in the third position and a number in the final position:

```
# Strict policy (adds 160 more passes through the word list)
i[2][`~!@#$%^&*()-_=+]$[0123456789]
```

Here's an abbreviated example of the output when operating on the word *password*:

```
$ ./John.exe -wordfile:password.lst -rules –stdout
pa`ssword0
pa`ssword1
pa`ssword2
...
pa~ssword7
pa~ssword8
pa~ssword9
pa!ssword0
pa!ssword1
pa!ssword2
...
```

As you can see, it is possible to create rules that quickly bear down on a network's password construction rules.

# L0PHTCRACK

At first, Windows systems seemed to offer improvements in password security over their Unix peers. Most Unix-heads could never create passwords longer than eight characters.

Windows NT boasted a maximum length of 14 characters, almost doubling the length! Then, Mudge and Weld Pond from L0pht Heavy Industries peeked under the hood of the LANMan hash. The company subsequently released a tool that took advantage of some inadequacies of the password encryption scheme.

We've already mentioned the LANMan hash quite a bit in this chapter. We know that it is the hashed representation of a user's password, much like a Unix /etc/passwd or /etc/shadow file. What we'll do now is take a closer look at how the LANMan hash is actually generated and stored. A Windows system stores two versions of a user's password. The first version is called the LANMan, or LM, hash. The second version is the NT hash, which is encrypted with MD4, a one-way function—that is, the password can be encrypted, but it can never be decrypted. The LANMan hash is also created by a one-way function, but in this case, the password is split into halves before being encrypted with the DES algorithm.

Let's take a quick look at the content of three LANMan hashes for three different passwords. They are represented in hexadecimal notation and consist of 16 bytes of data:

```
898f30164a203ca0 14cc8d7feb12c1db
898f30164a203ca0 aad3b435b51404ee
14cc8d7feb12c1db aad3b435b51404ee
```

It doesn't take a box of cereal and a secret decoder ring to notice some coincidences between these three examples. The last 8 bytes of the second and third examples are exactly the same: `aad3b435b51404ee`. This value will appear in the second half of any hash generated from a password that is less than eight characters long. This is a cryptography gaffe for two reasons: It implies that the content of the password is less than eight characters, and it reveals that the generation of the second half of the hash does not use any information from the first half. Notice that the second half of the first example (`14cc8d7feb12c1db`) matches the first half of the third example. This implies that the password is encrypted in independent sets of two (seven characters) rather than the second half depending on the content of the first half.

In effect, this turns everyone's potentially 14-character password into two smaller seven-character passwords. To top it off, the LANMan hashes ignored the case of letters, which reduces the amount of time to complete a brute-force attack by a factor of 10.

## Implementation

L0phtCrack brought password cracking to the GUI-rich environment of Windows NT and its descendants, Windows 2000 and XP. Trying to pilfer passwords from Unix systems usually requires nabbing the /etc/passwd or /etc/shadow file—both easily readable text files. Windows stores passwords in the Security Accounts Manager (SAM)—a

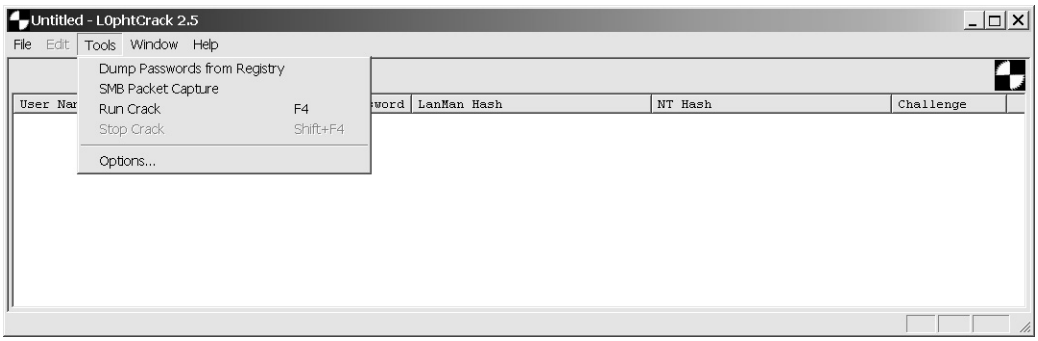
binary file that is difficult to read without special tools. Not only will L0phtCrack guess passwords, it will extract LANMan hashes from any SAM file, the local system, or a remote system, and it will even sniff hashes as they cross a network.

The SAM file resides in the \WINNT\system32\config\ directory. If you try to copy or open this file you will receive an error:

```
C:\WINNT\system32\config copy SAM c:\temp
The process cannot access the file because it is being used by
another process.
        0 file(s) copied.
```

Don't give up! Windows helpfully backs up a copy of the SAM file to the \WINNT\ repair\ or sometimes the \WINNT\repair\RegBack\ directory.

L0phtCrack will extract passwords from the local or remote computers with the Dump Passwords From Registry option.

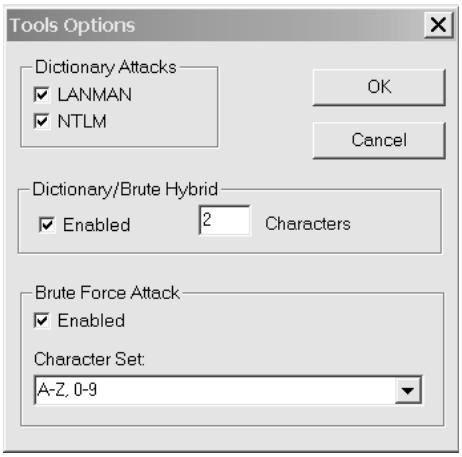



Remote extraction requires a valid session to the ADMIN$ share. This requires access to the NetBIOS TCP port 139. L0phtCrack can establish the session for you, or you can do so manually:

```
C:\>net use \\victim\admin$ * /u:Administrator
Type the password for \\localhost\admin$:
The command completed successfully.
```

It can also sniff LANMan hashes from the network. Each time a `net use` command passes the sniffing computer, the authentication hash will be extracted. You must be on the local network and be able to see the traffic, so its use tends to be limited.

The password cracking speed of L0phtCrack is respectable, but not on par with the latest versions of JJohn. Nor does it offer the versatility of modifying rules. It does allow for customizing the character list from the Options menu.




However, it's usually best to use L0phtCrack to extract the passwords, and then save the password file for JJohn to use—choose File | Save As.

You will need to massage the file for JJohn to accept it. This involves placing the password hashes in the appropriate fields.

Here's the L0phtCrack save file:

```
LastBruteIteration=0
CharacterSet=1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ
ElapsedTime=0 0
Administrator:"":"": A34E6990556D7BA3BA1F6705936BF461:
2B1437DBB1DC57DA3DA1B88BADAB13B2:::
```

And here's the file for John the Ripper. Note that the first three lines have been removed and there is only one field between the username (Administrator) and the password hash. The content of this field is unimportant to John, but we'll put the user's SID there as a reminder:

```
Administrator:500:A34E6990556D7BA3BA1F6705936BF461:
2B1437DBB1DC57DA3DA1B88BADAB13B2:::
```

Version 3.0 of L0phtCrack introduced improvements in the auditing ability of the application. Although it is easier for administrators to use and it's geared toward their needs (such as the option of reporting only that a password was cracked rather than displaying the result), we prefer to use L0phtCrack 2.52 to grab passwords and use John the Ripper to crack them.

Using L0phtCrack version 3.0 has its advantages. Pure Windows 2000 domains can have accounts with 15-character passwords. This effectively disables the LANMan storage. Consequently, version 2.5 will report "No Password" for both the LANMan and NTLM hashes for any account with a 15-character password. Version 3.0 will correctly load and identify accounts with 15 characters. If you ever find a hash such as this,

```
AAD3B435B51404EEAAD3B435B51404EE:FA95F45CC70B670BD865F3748CA3E9FC:::
```

then you have discovered one of these "super passwords." Note that the LANMan hash contains our friendly `AAD3B435B51404EE` null value repeated twice in the LANMan portion of the password (in bold).

The other advantage of L0phtCrack version 3.0 is its ability to perform distributed cracking. Its method breaks up the brute-force guesses into blocks. This is a significant advantage for running it on heterogeneous systems and tracking the current status.

## Protecting Your Passwords

Strong network and host security is the best method for protecting passwords and the password file. If a malicious user can grab the password file or the password hash for a Windows system, statistically speaking, it is only a short matter of time before the majority of the passwords are cracked. However, tools like John the Ripper and L0phtCrack cannot handle certain characters that Windows accepts as valid.

Several ALT-number pad combinations produce characters that will not be tested by current password crackers. To enter one of these combinations, remember to use numbers from the number pad. For example, the letters *p-a-s-s-w-ALT-242-r-d* (passw"rd) will remain safe until someone updates the password cracking tools. Plus, the additional characters made available by the ALT-*nnn* technique vastly expand the brute-force key space.

**TIP** The ALT combinations for special characters start at 160 (ALT-160) and end at 255.

## Removing the LANMan Hash

A benefit that Windows XP and Windows 2000 Service Pack 2 provided for security-conscious administrators is a registry key that removes the LANMan hash storage of a user's password. Remember, the LM hash is the weak version of the user's password that ignores the difference between upper- and lowercase characters. You could create a 15-character or longer password, as noted in the discussion of the L0phtCrack implementation. Or you could set the following registry key to instruct Windows not to store the LANMan hash for any later password change:

```
HKLM\SYSTEM\CurrentControlSet\Control\Lsa\NoLMHash
```

The *NoLMHash* value is a *REG_DWORD* that should be equal to 1. This will break compatibility with any Windows system in the 9*x* or Me series, but 2000 and XP will fare quite nicely. Once you've set this value, make sure to have all users change their passwords so the new setting will take effect. If setting this registry value doesn't sound like it adds much more security for your passwords, consider this: the difference in key space for an eight-character password (and the amount of time it would take to brute force a password) between the LANMan hash and the MD4 hash is well over a factor of 1000! In other words, there are roughly 69^7 combinations for the LANMan hash (remember, an eight-character password is really a seven-character password plus a one-character password) and 96^8 combinations for the MD4 hash.

## Case Study: Finding L0phtCrack on Your System

Anti-virus softwares may flag L0phtCrack as dangerous. This is because it is both a useful auditing tool for system administrators, but it's arguably an equally useful tool for malicious users who install it without permission. You may find files with .lc extensions, which is a good indicator that L0phtCrack has been there. If the tool has actually been installed on a system, as opposed to being run off of a floppy, you can perform registry searches for l0pht. Let's run through a few checks that a system administrator would make after discovering that the workstation of a temporary employee has been accessing the ADMIN$ share on the network's PDC.

We'll gloss over several steps, such as seizing data and finding out what commands have been run. Instead, we're worried about our network's passwords. There are over 600 employees. Already, we might want to consider every password as compromised, but if we're looking for direct evidence that the inside user has been cracking passwords, then we need to look for some key data. The most obvious entry that shows that L0phtCrack has been installed on the system is its own registry key:

```
HKLM\SYSTEM\Software\L0pht Heavy Industries\L0phtcrack 2.5
```

Unfortunately, this key is not present on the system. Now, there are other indicators that L0phtCrack was installed. One key is related to the packet capture driver it uses for sniffing LANMan hashes:

```
HKLM\SYSTEM\CurrentControlSet\Service\NDIS3Pkt
```

Other programs may set this key, but the correct value that these programs set will be the following (note the case):

```
HKLM\SYSTEM\CurrentControlSet\Service\Ndis3pkt
```

The *NDIS3Pkt* key exists, so we can start to suspect that L0phtCrack has been installed. The wily insider may have tried to erase most of the tool's presence, even going so far as to defragment the hard drive and write over the original space on the disk in order to prevent forensic tools from finding the deleted data on the hard drive. However, there is also another entry that Windows stores for the uninstall information for L0phtCrack. Even if L0phtCrack has been uninstalled, the following registry key remains:

```
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\L0phtcrack 2.5
```

If the system administrator finds this in the registry, she can be 100 percent sure that L0phtCrack had been installed on the system at some point in time. Next, the administrator could search the "most recently used" (MRU) values in the registry for files with a .lc extension. Even if the user deleted "sam_pdc.lc" from the file system, references to it could still exist in the registry! A diligent investigator will also search the MD5 fingerprints of all binaries on the system and look for anything that matches the fingerprint of the L0phtCrack binary.

# GRABBING WINDOWS PASSWORD HASHES

After reviewing the L0phtCrack section of this chapter, it's apparent that Windows password hashes can be viewed by the administrator just as easily as a Unix administrator can view the /etc/shadow file. On the other hand, the Unix /etc/shadow file is a text view that can be viewed in any text editor or simply output to the screen. The Windows SAM database is a binary format that does not lend itself to easy inspection. This is why we need tools such as pwdump or lsadump to grab a text version of the SAM database.

## Pwdump

Pwdump2, *http://www.openwall.com/passwords/nt.shtml*, by Todd Sabin, can be used to extract the hashed passwords from a Windows system. It is a command-line tool that must be run locally on the target system; however, we'll take a look at pwdump3, which can operate remotely, later in this section.

### Implementation

The program must be run locally on the system. This is version 2 of a tool first developed by Jeremy Allison of the Samba project. Unlike the first version, pwdump2 is not inhibited by SysKey encryption of the SAM database. SysKey was introduced in Windows NT in an attempt to add additional security to the SAM database, but its effectiveness is questionable, as we will see with pwdump2. The usage for pwdump2 is shown here:

```
C:\>pwdump2.exe /?

Pwdump2 - dump the SAM database.
Usage: pwdump2.exe <pid of lsass.exe>
```

It must be run with Administrator privileges in order to obtain the password hashes:

```
C:\>pwdump2.exe
Administrator:500:f1e5c5efbc8cfb7f18136fb05f77a0bf:55c77b761ffa46...
Orc:501:cbc501a4d2227783cbc501a4d2227783:f523558e22c95c62a6d6d00c...
skycladgirl:1013:aa5536a42ebe131baad3b235b51404ee:db31a1ee00bfbee...
```

You do not usually have to provide the process ID (PID) for the lsass.exe program. However, you can use some simple ways to find it with the `tlist` or `pulist` and the `find` command (the `/i` option instructs `find` to ignore case); or you could simply look in Task Manager if the pid column is selected for display.

```
C:\>tlist | find /i "lsass"
 244 LSASS.EXE

C:\>pulist | find /i "lsass"
LSASS.EXE         244  NT AUTHORITY\SYSTEM

C:\>pwdump2.exe 244
```

```
Administrator:500:f1e5c5efbc8cfb7f18136fb05f77a0bf:55c77b761ffa46...
Orc:501:cbc501a4d2227783cbc501a4d2227783:f523558e22c95c62a6d6d00c...
skycladgirl:1013:aa5536a42ebe131baad3b235b51404ee:db31a1ee00bfbee...
```

The only drawback with the output from pwdump2 is that L0phtCrack cannot read it. The sole reason for this is that the alphabet characters in the hashes are lowercase; L0phtCrack expects them to be uppercase. John the Ripper has no issue detecting case sensitivities, but we must massage the data into an acceptable format.

Fortunately, the tr utility (translate characters) will set this right for those of you who wish to use the GUI cracker. Tr is common on Unix systems and Cygwin, and it has been ported for Windows as part of the Resource Kit.

```
[user@hediwg ]$ cat pwdump.out | tr a-z A-Z
ADMINISTRATOR:500:F1E5C5EFBC8CFB7F18136FB05F77A0BF:55C77B761FFA46...
ORC:501:CBC501A4D2227783CBC501A4D2227783:F523558E22C95C62A6D6D00C...
SKYCLADGIRL:1013:AA5536A42EBE131BAAD3B235B51404EE:DB31A1EE00BFBEE...
```

## Pwdump3

Pwdump3, *http://www.openwall.com/passwords/nt.shtml*, by Phil Staubs, expanded the pwdump tool once more by adding remote access to a victim machine. There is even a version, pwdump3e, that encrypts remote connections to prevent malicious users from sniffing sensitive passwords. The usage for pwdump3e differs slightly:

```
Usage: PWDUMP3 machineName [outputFile] [userName]
C:\>PwDump3.exe victim pwdump.out root
C:\>type pwdump.out
guest:1001:NO PASSWORD*********************:2DEAC3223C70B24E90F02...
wwwadmin:500:NO PASSWORD*********************:9CBD10B05F8E69B62F2...
IUSR_WWW01:1003:6E72211CDC51C9F8EB9293C3135F3985:0E2A2DCE3B6ABFBA...
```

For pwdump3 to work correctly, you need to be able to establish a session to the ADMIN$ share. Pwdump3 will do this for you and prompt you for the administrator password. Otherwise, you could set up a manual session to the ADMIN$ share with the `net` command:

```
C:\>net use \\victim\admin$ * /u:Administrator
Type the password for \\localhost\admin$:
The command completed successfully.
```

## Pwdump4

Pwdump4 was written to address some shortcomings of pwdump3. You can grab a binary and source version from the OpenWall (John the Ripper) web site, *http://www.openwall.com/passwords/nt.shtml*. It uses the same technique as its nominal predecessor, pwdump3, but improves the usability when dealing with other character sets and when the ADMIN$ share is not available.

### Implementation

The pwdump4 command line closely resembles its peers.

```
C:\tools>PWDump4.exe
PWDUMP4.02 dump winnt/2000 user/password hash remote or local for crack.
   by bingle@email.com.cn
This program is free software based on pwpump3 by Phil Staubs
under the GNU General Public License Version 2.
Usage: PWDUMP4 [Target | /l] [/s:share] [/o:outputFile] [/u:userName]
```

Each option is described in Table 8-1.

Probably the most useful feature is the `/s` option. This enables you to target remote systems for which ADMIN$ is inaccessible, but some other share is accessible. Another by-product of this additional feature is that remote registry access is no longer a requirement. pwdump4 will try to communicate over named pipes (such as via the IPC$ share).

Here is a final tip for users trying to run pwdump4 inside a remote desktop connection. If you execute the command against *localhost* with the `/l` option, then you'll likely receive an error along the lines of *SRV>Status: CreateRemoteThread failed: 8*. In this case, simply try specifying *localhost* as the target and have pwdump4 access the ADMIN$ share (or whichever share you find available).

```
C:\tools>PWDump4.exe localhost /o:err.txt
PWDUMP4.02 dump winnt/2000 user/password hash remote or local for crack.
   by bingle@email.com.cn
This program is free software based on pwpump3 by Phil Staubs
under the GNU General Public License Version 2.
local path of \\localhost\ADMIN$ is: C:\WINDOWS
connect to localhost for result, plz wait...
SRV>Version: OS Ver 5.2, Service Pack 1, ServerTerminal
LSA>Samr Enumerate 6 Users In Domain WIN2K3-WEB.
 All Completed.
```

## Lsadump2

Lsadump2, *http://www.bindview.com/Services/RAZOR/Utilities/Windows/lsadump2_readme.cfm*, makes the password-harvesting process trivial. Another useful tool by Todd Sabin, it's an update to an original tool created by Paul Ashton. The difference between lsadump2 and the pwdump tool suite is that lsadump2 actually dumps the plaintext password instead of the encrypted hash. Obviously, this is preferable since you won't have to run any password-cracking utilities. Unfortunately, lsadump2 only retrieves a password if it is currently being stored in memory by the Local Security Authority (LSA). This could happen

| Option | Description |
|---|---|
| `Target` | Targets computer's IP address or hostname. For localhost use `/l`. |
| `/l` | Targets the local computer. This uses the pwdump2 method of dumping hashes, rather than pwdump3. |
| `/s:share` | By default pwdump4 will attempt to access the ADMIN$ (as does pwdump3). Specify an alternate share over which to attempt remote access. |
| `/o:outputFile` | Saves results to outputFile. |
| `/u:userName` | Connects to share as userName. You will be prompted for the password. |
| `/r:newname` | Rename the pwdump service and files copied to the remote computer to *newname*. This provides very basic stealth. |

**Table 8-1.** Pwdump4 Command-line Options

when web applications connect to SQL databases or when a backup utility connects to the system remotely in order to archive files.

## Implementation

Lsadump2 requires Administrator access to run. The usage for lsadump2 is shown here:

```
C:\>lsadump2.exe
Lsadump2 - dump an LSA secret.
Usage: lsadump2.exe <pid of lsass.exe> <secret>
```

You will have to determine the PID of the `lsass` (just as with pwdump2):

```
C:\>tlist | find /i "lsass"
 244 LSASS.EXE
```

**TIP** The PID for the LSA process is also stored in the registry under this key: HKLM\SYSTEM\CurrentControlSet\Control\Lsa\LsaPid.

This tool actually outputs the plaintext "secret" for security-related processes currently in memory. This secret might be the password used by a service account, phone number information for RAS services, or remote backup utility passwords. The output is formatted in two columns:

```
aspnet_WP_PASSWORD
 61 00 77 00 41 00 39 00 65 00 68 00 68 00 61 00  a.w.A.9.e.h.h.a.
 4B 00 38 00                                      K.8.
```

The left column represents the raw hexadecimal values related to the service. The right column contains the printable ASCII representation of the data. If you have recently installed the .NET services on your Windows 2000 system, then you most likely have an ASPNET user. Lsadump2 has kindly revealed the password for that user, shown in bold. Note that Windows stores passwords in Unicode format, which is why there is a null character (00) after each letter. Luckily, the default settings for this user do not permit it to log in remotely or execute commands.

# ACTIVE BRUTE-FORCE TOOLS

Active tools tend to be the last resort for password guessing. They generate a lot of noise on the network and against the victim (although they can go unnoticed for long periods of time). The toughest part of starting an active attack is obtaining a valid username on the victim system. Chapter 6 provides more information for techniques to gather usernames.

Another useful step is to try to discover the lockout threshold before launching an attack. If the lockout period on an account lasts for 30 minutes after it receives five invalid passwords, you don't want to waste 29 minutes and 30 seconds of guesses that can never succeed.

## THC-Hydra

Hydra easily surpasses the majority of brute-force tools available on the Internet for two reasons: It is fast and it can target authentication mechanisms for over a dozen protocols. The fact that it is open source (under the GPL) and part of the Nessus assessment tool also adds to Hydra's merits.

### Implementation

Hydra compiles on BSD and Linux systems without problem; the Cygwin and Mac OSX environments have been brought to equal par in the most current version. Follow the usual ./configure; make; make install method for compiling source code. Once you have successfully compiled it, check out the command-line arguments detailed in Table 8-2.

The target is defined by the *server* and *service* arguments. The type of service can be any one of the following applications. Note that for several of the services, a port for SSL

| Option | Description |
|---|---|
| `-R` | Restores a previous aborted/crashed session from the hydra.restore file (by default this file is created in the directory from which hydra was executed). |
| `-S` | Connects via SSL. |
| `-s` *n* | Connects to port *n* instead of the service's default port. |
| `-l` *name*<br>`-L` *file* | Uses *name* from the command line or from each line of *file* as the username portion of the credential. |
| `-p` *password*<br>`-P` *file* | Uses *password* from the command line or from each line of *file* as the password portion of the credential. |
| `-C` *file* | Loads user:password combinations from *file*. Each line contains one combination separated by a colon. |
| `-e [`*ns*`]` | Also tests the login prompt for null passwords (n) or passwords equal to the username (s). |
| `-M` *file* | Targets the hosts listed in each line of *file* instead of a single host. |
| `-o` *file* | Writes a successful username and password combination to *file* instead of stdout. |
| `-f` | Exits after the first successful username and password combination is discovered for the host. If multiple hosts are targeted (`-M`), then Hydra will continue to run against other hosts until the first successful credentials are found. |
| `-t` *n* | Executes *n* parallel connects to the target service. The default is 16. |
| `-w` *n* | Waits no more than *n* seconds for a response from the service before assuming no response will come. |
| `-v`<br>`-V` | Reports verbose status information. |
| server | The target's IP address or hostname. For multiple targets use the `-M` option. |
| service | The target's service to brute force. |

**Table 8-2.** Hydra Command-line Options

access has already been defined. The first number in the parentheses is the service's default port; the second number is the service's port over SSL. Make sure to use the `-s` option if the target service is listening on a different port. These are the current services that Hydra recognizes:

- **cisco (23)** Telnet prompt specific to Cisco devices when only a password is requested.
- **cisco-enable (23)** Entering the enable, or super-user, mode on a Cisco device. You must already know the initial login password and supply it with the `-m` option and without the `-l` or `-L` options (there is no prompt for the username).
  `hydra -m letmein -P password.lst 10.0.10.254 cisco-enable`
- **cvs (2401)** Source code versioning system.
- **ftp (21, 990)** File transfer.
- **http, http-head, http-get (80)** Brute-force HTTP Basic Authentication schemes on the web service. Note that this technique expects the server to send particular HTTP response codes; otherwise, the accuracy of this module may suffer.
- **https, https-head, https-get (n/a, 443)** Web services over SSL (see previous bullet).
- **http-proxy (3128)** Web proxies such as Squid.
- **icq (4000, n/a)** Chat software. ICQ is carried over UDP, which means it cannot be used over SSL.
- **imap (143, 993)** E-mail access.
- **ldap2, ldap3 (389, 636)** Lightweight Directory Access Protocol, often used for single-sign-on.
- **mssql (1433)** Microsoft SQL Server—remember that more recent installs of SQL Server may use integrated authentication. Try the default SQL accounts, such as 'sa', and Windows accounts.
- **mysql (3306, 3306)** MySQL database server.
- **nntp (119, 563)** USENET news access.
- **oracle-listener (1521)** Oracle database server.
- **pcnfs (0, n/a)** Used for printing files across a network. The default port varies among distributions and individual servers, so it must always be explicitly set with the `-s` option. This service also uses UDP, which means that SSL cannot be applied.
- **pop3 (110, 995)** E-mail access.
- **postgres (5432)** PostgreSQL database server.
- **rexec, rlogin, rsh (512)** Generic Unix service for remote execution; access to this service is not logged by default on some systems.

- **sapr3 (n/a)** SAP database.
- **sip (5060)** Voice-over IP protocol.
- **smb (139)** Windows SMB services such as file shares and IPC$ access.
- **smbnt (445)** As smb, but is also able to test LanMan hashes (such as those gathered by PwDump tools) for validity. This enables credential replay rather than actually brute forcing the content of the hash. Note that you must define a method (`-m`) when using this option. Valid methods are well-documented in the hydra-smbnt.c file. You'll most likely try 'LH' or 'DH' methods, which test LanMan password hashes against local or domain accounts. Use this for Windows XP and Windows 2003 servers.
- **smtp-auth (25, 465)** Login for mail servers.
- **snmp (161, 1993)** UDP-based network management protocol.
- **socks5 (1080)** Proxy.
- **svn (3690)** Source code versioning system.
- **teamspeak (8767)** Distributed voice chat system, often used by gamers.
- **telnet (23, 992)** Remote command shell.
- **vnc (5900, 5901)** Remote administration for GUI environments.

Running Hydra is simple. The biggest problem you may encounter is the choice of username/password combinations. Here is one example of targeting a Windows SMB service. If port 139 or 445 is open on the target server and an error occurs, then the Windows *Server* service might not be started—the brute-force attack will not work.

```
[Paris:~] mike% ./hydra -L user.lst -P password.lst 10.0.1.11 smbnt
[INFO] Reduced number of tasks to 1 (smb does not like parallel
 connections)
Hydra v5.0 [http://www.thc.org] (c) 2005 by van Hauser / THC <vh@thc.org>
[INFO] Reduced number of tasks to 1 (smb does not like parallel connections)
[DATA] 1 tasks, 1 servers, 4 login tries (l:2/p:2), ~4 tries per
 task
[DATA] attacking service smbnt on port 445
[STATUS] 1.00 tries/min, 1 tries in 00:01h, 3 todo in 00:04h
```

Hydra reports the total number of combinations that it will try (usually the number of unique usernames multiplied by the number of unique passwords) and how many parallel tasks are running.

**TIP**

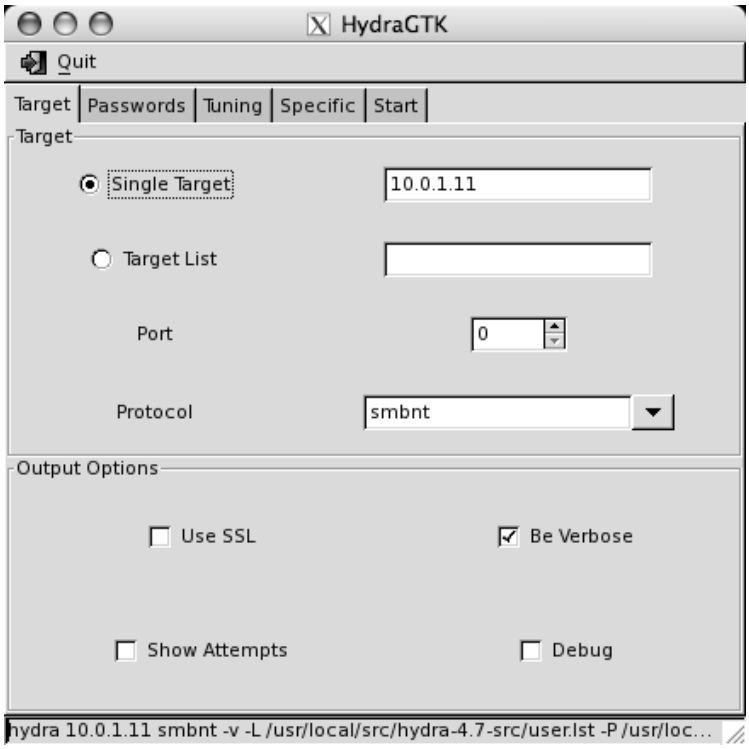
You will never be able to try more than one parallel task against an SMB service, even if you use the `-t` option to increase the number. For whatever reason, parallel logins against SMB produce too many false negatives. The default value for `-t` is 4, which is also recommended for Cisco devices and VNC servers. The maximum is 255, but that is not necessarily the optimum or most accurate setting to use.

If you really do wish to have an optimum test, as opposed to an exhaustive test, then you may wish to consider the `-c` option instead of supplying a file each for `-L` (users) and

`-P` (passwords). The `-C` option takes a single file as its argument. This file contains username and password combinations separated by a colon (:). This is often a more efficient method for testing accounts because you can populate the file with known username/password combinations, which reduces the number of unnecessary attempts when a username does not exist. This is more useful for situations where you only wish to test for default and the most common passwords.

Do not forget to use the `-e` option when auditing your network's services. The `-e` option turns on testing for the special case of no password (`-e n`) or a password equal to the username (`-e s`). Note that Hydra writes a state file (hydra.restore) to the current directory from which it is executed. You can use the `-R` option to restart an interrupted scan. This also means that if you wish to run concurrent scans against different servers or different services, then you should do so in different directories. From a forensic perspective, the hydra.restore file might be a good addition to the list of common "hacker" files to search for on suspect systems—just remember that a one-line change to the source code can change this filename.

Hydra now also includes a GUI based on the open-source GTK library. This version, called *xhydra*, provides all of the functionality of the command line. The following illustration shows the basic interface.



## Case Study: Checking Password Policy

There are two major reasons for using a tool like Hydra, either during a network penetration test or during a system audit. The two activities sound similar in execution but differ in their goals. Consider Iain, a system administrator in the Internal Audit department. The IA folks do not administer systems; they verify that systems have been built to corporate security policy. In other words, Iain's responsibilities include testing network accounts for passwords that do not meet the company's established policy.

The policy requires that all accounts be password protected (no NULL passwords allowed) and that the password must be nontrivial (open to interpretation, but at the very least that means the password should not equal the username), must contain at least one digit and one punctuation character (letters only are not permitted), and must be at least eight characters long. For some Windows and Unix systems, it is possible to enforce these rules when users go through the password-change process. On other systems, such as Cisco devices, it is not possible.

Iain faces the challenge of finding weak passwords in one of the following scenarios:

- A system does not have a method for enforcing good password choices. Users must be trusted to choose a strong password.
- A system has a method for enforcing good password choices but has been misconfigured. Users are still required by policy to choose a strong password, but it is not enforced.
- A system has a method for enforcing good password choices, but users can easily satisfy the requirements with a trivial password (password99!, pa$$w0rd, or adm1n1str@t0r).

Now, Iain has already identified some network services that could prove to be fruitful targets. However, it would not be a good idea to just obtain the list of users, grab a 200,000-word dictionary, and start Hydra (or several Hydras since there's a lot of work to do!). Instead, he crafts a dictionary with words that do not meet policy, plus some words that do meet policy but are passwords on number/vowel substitution or similar tricks. In fact, John the Ripper (mentioned previously in this chapter) provides the perfect method for creating password lists based on length and content. Then, just as a test, he creates an oldwords.txt file that contains the root and administrator passwords used before the last required password change. The oldwords.txt file follows the username:password syntax. For example,

```
root:web34addmin!
Administrator:thiS1&thaT1
oracle:2bdb|!2bdb
```

## Checking Password Policy *(continued)*

Let's recap for a moment. Iain has created three files (and will have a fourth and fifth option):

- **Users.txt** A list of every (known) username across systems.
- **Passwords.txt** A list of common 1–7 letter combinations, plus some selected 8+ combinations with number/vowel substitution. The majority of this file can be pulled from dictionaries available on the Internet, derived from the default password.lst that comes with John the Ripper, or created by John the Ripper. The list contains no more than 1000 combinations in order to limit the number of failed logins that will be logged by the servers.
- **Oldwords.txt** A list of account and password combinations that should have been changed in the last 90 days. Of course, this file *must* be kept secure.
- **NULL passwords** Use the `-e n` option for Hydra to check *all* accounts for a blank password.
- **"Same" passwords** Use the `-e s` option for Hydra to check *all* accounts for passwords that equal the username.

So far it sounds like quite a bit of work has been done without even worrying about whether or not Hydra will compile. Well, there's a good reason for this. Iain has set up a method for testing his company's password policy. At this point he is ready to launch Hydra against the selected services. (After he has once again verified that accounts will not be locked by failed login attempts.) Then, any positive matches can be brought to the attention of network and system administrators because the account has failed to meet policy requirements.

Just for a second, imagine that Iain had driven into the password audit without forethought; he grabs a random 10,000-word dictionary and launches Hydra over a three-day weekend against 200 accounts. If he's lucky it might even finish. If he's really lucky, no servers will have crashed because they ran out of disk space logging all of the failed attempts. Finally, what if a relatively strong password like "ou@te1tw2" or "-#*crAft0" shows up in the results simply because it was present in the dictionary? He would have a hard time convincing the user that they failed an audit when in reality they had chosen a strong password.

On the other hand, blind luck and a big dictionary are just the right ingredients for a successful penetration test. Thus, we come to the point where password auditing with Hydra ends and its use as a penetration-testing tool begins. In all cases, remember that locking accounts due to bad passwords is always a possible by-product of this type of testing.