Global Information Assurance Certification Paper

Using Password Recovery Software Proactively

Blake Beller

March 26, 2004
Practical Assignment for GIAC GSEC Certification
Version 1.4b, Option 1

Abstract
The goal of this paper is to discuss the basics of password management and how password recovery software can be used proactively to increase the security in your environment. Additionally, demonstrations and discussion will reinforce the weakness of Lan Manager (LM) hashes on the Windows platforms. Testing the strength of passwords is an important component of internal audits and should be an expected component of any external audits performed. If you are unfamiliar with password recovery software, some of the examples and demonstrations in this paper may help you get started.

Introduction
Username and password combinations are a common means of user authentication. In some instances, a username and password combination is the only line of defense to prevent unauthenticated users from gaining access to certain data, such as an online website where you can purchase computer equipment. In other cases where defense-in-depth strategy is more prevalent, a username and password combination may only be one of several layers of defense to authorize access. Password attacks are still a common method for a malicious user to gain access to a system. Hopefully this paper will give examples and discussion of software tools and methods to help you defend against these types of attacks. The types of password hashes discussed will be a subset of those that exist in the Windows and UNIX operating systems, for which password recovery software already exists. Originally this paper was intended to be a case study, but was changed to combine theory with the demonstrations. If you are looking for examples of using password recovery tools, you will find summaries of the demonstrations near the end of the paper, with the full demonstrations in the appendices.

Best Practices

The System, Audit, Network, and Security (SANS) Institute provides, on its website, a sample password policy. According to the website, all of the SANS sample policies are sanitized versions of a large organization's policies. Focusing on the portion of the policy that relates to password rotation, section 4.1 states:

- All system-level passwords (e.g., root, enable, NT admin, application administration accounts, etc.) must be changed on at least a quarterly basis.
- All user-level passwords (e.g., email, web, desktop computer, etc.) must be changed at least every six months. The recommended change interval is every four months (Password Policy, p. 1)

A "strong" password is regarded as a password that is difficult to guess and is not easy to recover. Section 4.2 of the same SANS policy, relevant to password construction, states:

Strong passwords have the following characteristics:

- Contain both upper and lower case characters (e.g., a-z, A-Z)
- Have digits and punctuation characters as well as letters e.g., 0-9, !@#$%^&*()_+|~-=\`{}[]:";'<>?,./)
- Are at least eight alphanumeric characters long.
- Are not a word in any language, slang, dialect, jargon, etc.
- Are not based on personal information, names of family, etc.
- Passwords should never be written down or stored on-line. Try to create passwords that can be easily remembered. One way to do this is create a password based on a song title, affirmation, or other phrase. For example, the phrase might be: "This May Be One Way To Remember" and the password could be: "TmB1w2R!" or "Tmb1W>r~" or some other variation (Password Policy, p. 2)

To expand on the last bullet point, the password should have meaning to you, making it easy to remember, but appear to be gibberish to anyone else. While there are other important criteria, such as locking user accounts after multiple unsuccessful logins, the paper will focus on implications of password construction and types of password hashes. Here is a list of the password management features built into the following operating systems: Solaris 8, AIX 4.3.3, and Windows 2000 Professional. Some operating systems offer more management features than others.

The /etc/default/passwd file in Solaris 8 allows the root user to configure the following password settings:

- Maximum age – time before the password must be changed
- Minimum age – time a password must be kept before it can be changed
- Minimum Password Length

In AIX 4.3.3, the System Management Interface Tool (SMIT) allows the following password attributes to be configured by the root user.

- Dictionary checking
- Number of passwords remembered in password history
- Amount of time passwords are remembered in password history
- Weeks between password expiration and password lockout
- Maximum age
- Minimum age
- Minimum password length
- Minimum number of alpha characters
- Minimum number of other characters
- Maximum number of times a character can be repeated in the password
- Minimum number of characters that must differ from previous password

Microsoft Management Console (MMC) in Windows 2000 server has the following password settings:

- Number of passwords remembered in password history
- Maximum age
- Minimum age
- Whether or not password complexity rules are enforced
- Whether or not passwords are stored using reversible encryption

Additionally, third-party products can be used to strengthen password management capabilities. Examples include Npasswd, Password Policy Enforcer, and Control-SA. Npasswd is specific to various flavors of UNIX. Password Policy Enforcer runs on several Windows platforms and also as a client on Novell Netware. Control-SA, being a user provisioning tool for the enterprise, is a much broader cross-platform tool that, among other features, gives the ability to impose password standards across those platforms.

Understanding Password Hashes

In addition to using the capabilities of the operating system or third party software to adhere to password best practices, password recovery software can be used as an additional check. Much of this paper will focus on discussing and demonstrating the likelihood of obtaining your plain-text password if its corresponding encrypted hash is known. The various Windows and UNIX operating systems do not store user passwords in plain-text. Instead, the operating system stores the result of a one-way hash of the plain-text password. Webopedia.com defines a “one-way hash” as “an algorithm that turns messages or text into a fixed string of digits, usually for security or data management purposes. The ‘one way’ means that it's nearly impossible to derive the original text from the string” (One-way hash).” When a user enters his password to authenticate on a Windows or UNIX system, the one way hash algorithm is performed. If the result of the one-way hash is identical to the hash stored by the operating system for that user, it is assumed that the plain-text password entered is correct, and the user is authenticated. To better understand how hashing

algorithms work, FIPS publication 180-1 gives a very thorough and descriptive explanation of the popular hashing algorithm, SHA-1.

For discussion purposes, types of password hashes in the Windows and UNIX operating systems will be limited to Lan Manager, NTLM (NT Lan Manager), Traditional DES (Data Encryption Standard), and FreeBSD MD5 (Message Digest 5). While these hashes may have other names associated with them, I will use the terms in the previous sentence to identify the types of hashes. The following is an example of a plain-text password and what its corresponding hashes look like.

```
Plain-text            |n&Jnn37
LM hash               9432179b4cb24cf97c3113b4a1a5e3a0
NTLM hash             c6302ff6fcd6df7bda813fc442908571
Traditional DES hash  MRRfw7BvTPR.Y
FreeBSD MD5 hash      $1$QO72SDLf$PTGe.pbrKlEwR6tiAvxpY.
```

A "salt," which is used by some hashing algorithms, is text which can be combined with the original plain-text password before the hash is computed. The first two characters of a DES hash are the salt. In the DES hash example above, salting the plain-text password of "`|n&Jnn37`" with "`MR`" produces "`Rfw7BvTPR.Y`". Similarly with the FreeBSD MD5 hash above, the "1" is indicative of an MD5 hash and is always the first 3 characters of the salt. Neither LM nor NTLM hashes use a salt in their hashing algorithm. For example, the plain-text password of "`|n&Jnn37`" on any Windows server will always return the LM hash of "`9432179b4cb24cf97c3113b4a1a5e3a0`".

Adding a salt helps defend against dictionary attacks (explained later). It can also significantly slow down the rate at which password recovery software like John the Ripper (referred to as "John"), a freeware password recovery tool, searches through potential passwords (also explained later). Rainbow Crack (rcrack), another freeware software recovery tool, is an example of password recovery software that takes the approach of generating tables of plain-text passwords and their corresponding ciphertext. If you have the time, computing power, and disk space, rcrack allows you to provide the type of hash, minimum and maximum password length, and valid password characters to compute the corresponding "rainbow tables."

Why Use Password Recovery Software?

Theoretically, if given enough computing power and time, any password hash would be recovered. Think about what the goals of your testing should be, and make them realistic. If your systems can be configured to enforce best practices through technology, then it can be a realistic goal to show with your testing that 1% or less passwords can be recovered in the amount of time before the passwords would have to be rotated. If you are currently unable to enforce, with technology, best practices on your system, the previously mentioned goal is not necessarily realistic. Instead, your short-term goal may be to perform testing on

a more frequent basis to continue to educate users, whose passwords are recovered, how to choose stronger passwords. While your goals may change over time, do have a purpose to your testing. As you refine your testing and goals, hopefully you will be able to gain confidence that your systems will be relatively safe from malicious users or external auditors attacking the passwords.

Obtain Written Permission Before Testing

Determine which team or person(s) in your organization should be responsible for password recovery testing. Regardless of who it is, it is imperative to get written permission from an authorized party before testing. Being a system administrator or a member of the security team does not give you free reign to do whatever you want whenever you want. Define the scope of the testing that will be performed, who is authorized to perform the testing, when it will occur, and then obtain signature authority from the appropriate person or persons. While performing password recovery testing is simply one type of act that can fall under the larger umbrella of penetration testing, there are potential legal issues to consider. The book Hack I.T. helps summarize:

> A request from a company employee to perform a penetration test is not necessarily a valid request. If that person does not have the authority to request such actions and indemnify you if anything goes wrong, you may incur fees related to court costs in addition to loss of fees for services. Therefore, a legal agreement must be reached before the testing begins, and the tester needs to make sure he or she has a signed "Get Out of Jail Free Card" from a company officer authorized to enter the organization into a legally binding agreement. The "Get Out of Jail Free Card" generally entails a legal agreement signed by an authorized representative of the organization outlining the types of activities to be performed and indemnifying the test against any loss or damages that may result from the testing (Klevinsky, p. 20).

Ed Skoudis provides a sample memo on his "Counter Hack Web Site" to obtain such permission.

Obtaining Password Hashes

While the spirit of this paper is with regards to being proactive, a basic understanding of how malicious users fit into the picture is needed. The following list includes possibilities of how a person might obtain password hashes from one or more UNIX or Windows hosts:

1) Sniffing your network for username and password hashes traveling to or from a Network Information Services (NIS) master if running UNIX, or username and password hashes traveling to a Windows domain controller.
2) Using NIS in UNIX or not using a shadow password file allows regular users to use the "`ypcat passwd`" or "`cat /etc/passwd`" commands, respectively, to see the encrypted password hashes of all users in the NIS domain or on the local host.

3) Someone has remotely compromised your host with administrative privileges and now has access to the shadow password file in UNIX or can now run a utility like pwdump2 in Windows to dump the LM and NTLM hashes to a file.
4) Someone has physically compromised the host and could boot from media to modify the Administrator password or simply move the hard drive to another host and mount the root partition, etc.

Note that sniffing a network for unencrypted protocols like ftp and telnet is also a possibility, but in these cases, there is no decryption to be performed as the plain-text password has already been obtained.

Protect the Password Hashes During Testing

When obtaining encrypted hashes for proactive testing, be diligent about protecting the data in transit and at rest. Otherwise, you will carelessly defeat the inherent protection of files like /etc/shadow in UNIX or the SAM database in Windows. If moving the encrypted hashes from one host to another is needed, do not use unencrypted methods of transfer like ftp or email. If using a utility like pwdump2 to dump the encrypted LM and NTLM hashes, make sure that permissions are appropriate in the target directory and on the target file to prevent non-privileged users from accessing the data. If using the pwdump3 tool to pull the encrypted hashes from a remote machine, be aware that the hashes are not inherently encrypted as they travel across the network to your local host. If it is necessary to remotely pull the encrypted hashes for testing, consider using pwdump3e (e for encryption) or tunnel the session through an encrypted protocol like ssh. Further steps to protect the encrypted hashes can include running your password recovery software on a non-networked machine, so that it may not be remotely compromised. Also consider encrypting the area of storage containing the input and output of your testing. It would be bad enough to let the encrypted hashes get into the hands of an unauthorized party, and even worse to let the decrypted hashes be viewed by an unauthorized party.

Using the Results of Password Recovery Testing

You should follow up with each user whose password was recovered in what you deem too short an amount of time and require each user to change his/her password. Help educate the users so they better understand how to choose a strong password. Once all the users with weak passwords from the first round of testing have updated their passwords, consider rerunning your testing on those password hashes only. If your hosts, for any reason, cannot be configured to impose complex password criteria, diligent testing with password recovery software may be your only way to ensure that users are choosing strong passwords. Once you are able to impose minimum password length and password composition requirements consistent with best practices, it should be unlikely that you will be able to recovery any passwords at all in a reasonable amount of time.

Password Search Space

John defaults to 95 valid choices for each character in a password: 26 upper-case letters, 26 lower-case letters, 10 numerals, and 33 printable special characters (including space). Consequently, there exists 95^1 total one-character passwords, 95^2 total two-character passwords, etc. to 95^8 eight-character passwords. The total number of 1-8 character passwords is the sum of the previous numbers. In analyzing the feasibility of brute-force attacks (explained later), password length is important. While the worst-case scenario may take 6 days to brute-force a certain type of password hash that is x characters long, the same type of password hash that is (x+1) characters long could take up to 6*95=570 days. With regards to search space only, reference chart 1.

Number of characters in password	Number of possible passwords with printable characters
1	95
2	9,025
3	857,375
4	81,450,625
5	7,737,809,375
6	735,091,890,625
7	69,833,729,609,375
8	6,634,204,312,890,625

Chart 1

It is important to understand that the numbers listed in the second column are actually upper bounds for the total possible number of passwords given a particular password length. If password complexity rules are imposed, the number of possible passwords is actually *reduced*. To better illustrate this point but keep the computations simple, we will use the example of a password with the following criteria: it must be exactly 3 characters long and must contain exactly one special character. Out of 95 possible characters, 33 are special characters and 62 account for the other characters. Because the password requires exactly one special character, it must go in the first, second, or third position. Once the special character is chosen, there can be no other special characters. The resulting computation is (33)(62)(62) + (62)(33)(62) + (62)(62)(33) = 380,556. From Chart 1, out of 857,375 possible passwords with no restrictions, we reduced the total number of valid passwords by over 50% in this case. Though reducing the total number of possible passwords may sound counterproductive, the tradeoff is that the passwords eliminated from the search space will tend to be those simple passwords found quickly by an attacker and/or password recovery software like John. Quantifying how much the search space of a given password length will be reduced depends on the password complexity rules imposed. Attempting to elegantly count, for instance, the number of valid 8-character passwords that require at least one letter, one number, and one special character with no character occurring more than three times in the password is,

unfortunately, a complex combinatorics problem beyond the scope of this paper. Consequently, the numbers in Chart 1 above will be used in Appendix C and theoretical computations in this paper.

Password Recovery Tools

John is the tool used in most examples in this paper. Related tools include, Proactive Windows Security Explorer (PWSE), LC4 (formerly L0phtcrack), Cisilia, Rainbow Crack, and Distributed John the Ripper (djohn). Several of these tools are contained in the Knoppix Security Tools Distribution (STD). See Appendix A for URLs to learn more about these tools.

John is freeware available on the Windows platform and several flavors of UNIX and is capable of recovering passwords from several different types of hashes. John is a fairly popular password recovery tool. References to John include the United States General Accounting Office's (GAO) Federal Information Systems Controls Audit Manual (FISCAM) and a reference from the Jacksonville chapter of Information Systems Audit and Controls Association (ISACA). John's current version of stable binaries is 1.6 and the latest development version is 1.6.36. In addition to the significant performance improvements in the recent development versions of John, as shown by Chart 2, the verbose logging contained in john.log contains a lot of useful information, like a log entry for each password recovered. Additionally, a timestamp is at the beginning of every log message. To use John, an input file containing a list of usernames and the users' corresponding password hashes is needed. John comes with a utility to combine /etc/passwd and /etc/shadow in UNIX into a single file. To extract the LM and NTLM hashes from Windows for use with John, a utility like pwdump2 must be used. Similar to the process that occurs when logging into a UNIX or Windows host explained earlier, John hashes a string and compares it against all hashes in the text file of hashes that you are trying to recover. If the hashes match, it is assumed that the string used to create the hash is identical and therefore the password has been recovered.

Using John can be very effective without making any configuration changes at all. Reference the documentation that comes with version 1.6 of the software to get started. Later examples will demonstrate certain configuration changes. Simpler configuration changes include using different wordlists or tightening the constraints on password length. A more complex example includes writing some custom rules to add to John's rule set.

The documentation that comes with John explains that John has four modes of operation; the first three are executed sequentially by default if no options are given. Summarizing the four modes:

1. Single-Crack – Rules relating to the text from the username and GECOS fields are used. (The GECOS field is the third field in a UNIX password file and commonly has a text description, such as the user's full name.)

2. Wordlist – This starts with a simple dictionary attack. A list of words (strings) is read in, creating a hash from each word. Additionally, several simple rules like appending a character or forcing the word to all lowercase are performed.
3. Incremental – This is far slower than the first two modes. It is not the same as a simple brute-force attack, as it uses something called character frequency tables to recover simpler passwords more quickly. Similar to a brute-force attack, it should eventually find the password.
4. External – This mode allows the user to completely define and configure an additional mode to recover passwords. For instance, you can configure a mode to make John behave exactly like the brute-force method in PWSE.

To compare and contrast, PWSE has three types of attacks for LM or NTLM hashes only. Additionally, you must choose whether to attack the LM or NTLM hashes. One type of attack is chosen at a time:

1. Dictionary – This is similar to the wordlist mode in John, but only tries the words in the dictionary with no variations.
2. Password-Masking – This allows you to fix certain characters and brute-force the other characters. If, for instance, you are trying to recover your own password and remember that it was 7 characters, started with a capital "E" and ended with an exclamation point, you could enter a mask of "E?????!" where "?" corresponds to a character to brute-force. In this example, such an attack reduces the search space from 7 characters to only 5 characters.
3. Brute-force – You choose the minimum and maximum password length and the possible character sets to use, and then all possible permutations are methodically attempted.

When using software like PWSE, the time needed to recover a password with a brute-force attack is primarily dependent on two factors: the length of the unencrypted password and the number of passwords per second that the software can hash. The length of the password will determine the total possible search space, as seen in Chart 1. The number of passwords per second processed, of which Chart 2 is an example, is largely dependent upon processor speed of the computer and how computationally intensive it is for the computer to generate the encrypted hash from the plain-text password. Referencing Chart 2, of the four types of password hashes discussed in this paper, the same computer running the same version of John in the same operating system can process LM hashes roughly 1,000 times as fast as FreeBSD MD5 hashes. A mostly insignificant factor is the number of passwords that you are trying to recover. Meaning, the bulk of the processor's work is to create the encrypted hash from the plain-text password, not the string comparison of the encrypted hashes. For instance, if John can compute roughly 100,000 NTLM hashes per second, it will still achieve roughly that same rate whether you are trying to recover only one password or even 1,000 passwords at the same time.

Some of the pros and cons of the different types of attacks are as follows. A positive aspect of a dictionary attack is that it is performed very quickly. A negative aspect of a dictionary attack is that if the original password came from a system where any decent password complexity is imposed, you will likely not find any passwords with this approach. The other extreme is a brute-force attack. The good news is that assuming that the scope of password length and valid character sets are accurate, a brute-force attack will eventually find any password given enough time (eg. a brute-force attack can still fail, for instance, if performing the attack of passwords 1-6 characters in length on a password that is actually 8 characters). The bad news is that the time needed could be prohibitively long, such as many years. Somewhere between the dictionary and brute-force attacks are the built-in rule sets in John that intelligently attack passwords or the password-masking attack in PWSE.

To help better understand how the brute-force attack in PWSE works, consider attacking a 6-character password that could potentially consist of a mix of upper-case letters, lower-case letters, numbers, and special characters. A visual analogy is to picture the odometer of a car, but instead of each dial going from 0-9 only, it goes from A-Z, then a-z, then 0-9, then through the remaining 33 special characters. In some sense, it is like counting in base 95 where A-Z=1-26, a-z=27-52, 0-9=53-62, and the 33 special characters correspond to 63-95.

Benchmarks

John the Ripper comes with a built-in benchmark (syntax is “john –test”). Screen Shot 1 is an example of the output from Config-3 (see Chart 2):

```
root@darkstar:/usr/local/src/john-1.6.36/ntlm# ./john -test
Benchmarking: Traditional DES [64/64 BS MMX]... DONE
Many salts:     543014 c/s real, 543014 c/s virtual
Only one salt:  482841 c/s real, 482841 c/s virtual

Benchmarking: BSDI DES (x725) [64/64 BS MMX]... DONE
Many salts:     18995 c/s real, 18995 c/s virtual
Only one salt:  18688 c/s real, 18688 c/s virtual

Benchmarking: FreeBSD MD5 [32/32]... DONE
Raw:    4072 c/s real, 4072 c/s virtual

Benchmarking: OpenBSD Blowfish (x32) [32/32]... DONE
Raw:    319 c/s real, 319 c/s virtual

Benchmarking: Kerberos AFS DES [48/64 4K MMX]... DONE
Short:  146790 c/s real, 146790 c/s virtual
Long:   378828 c/s real, 378828 c/s virtual

Benchmarking: NT LM DES [64/64 BS MMX]... DONE
Raw:    3856473 c/s real, 3856473 c/s virtual

Benchmarking: NT MD4 [TridgeMD4]... DONE
Raw:    616675 c/s real, 616675 c/s virtual
```

Screen Shot 1

The “c/s” in each of the benchmarks is the “crypts per second”, corresponding to the number of hashes that John is able to generate with that type of encryption in one second. However, when running John against password hashes with no different salts, the “c/s” rate can be far greater than that of the benchmark. The number in this case is actually the product of multiplying the hashing rate by the number of passwords that you are trying to recover. As discussed earlier in the paper, most of the processor’s time is spent performing the password hash, compared to the string comparisons of the hash just generated against the one or more hashes that we are trying to recover. Looking at Screen Shot 2, the value is not too much less than the benchmark from Screen Shot 1 (616,675) multiplied by the number of passwords (50) in Screen Shot 2.

```
root@darkstar:/usr/local/src/john-1.6.36/ntlm# ./john -format:nt lm.txt
Loaded 50 password hashes with no different salts (NT MD4 [TridgeMD4])
guesses: 0  time: 0:00:01:15 (3)  c/s: 28490485  trying: anC40
```

Screen Shot 2

The following chart was created by running John in three configurations on the same physical laptop followed by a fourth configuration on a slower desktop PC. Note that running the same version of John under Windows and Slackware (Linux) had mostly comparable results. However, running the latest version of John on the same hardware had a significant performance increase; even the slower desktop with the newer version of John outperformed the faster desktop with the earlier version of John.

	Config-1	**Config-2**	**Config-3**	**Config-4**
Processor speed	P-4 2.2 GHz	P-4 2.2 GHz	P-4 2.2 GHz	P-3 700MHz
Operating System Version	Windows 2000	Slackware 9.1	Slackware 9.1	Slackware 9.1
Version of John the Ripper	1.6	1.6	1.6.36	1.6.36
Traditional DES [64/64 BS MMX]				
Many salts	134,900	161,638	543,014	230,131
One salt	124,341	154,905	482,841	194,598
FreeBSD MD5 [32/32]				
Raw	2,207	2,205	4,072	1,720
NT LM DES [64/64 BS MMX]				
Raw	636,455	837,209	3,856,473	1,337,011
NT MD4 [TridgeMD4]				
Raw	N/A	N/A	620,534	310,049

Chart 2

Config-1, Config-3, and Config-4 were used in examples in this paper. Theoretical examples will use the numbers in Config-3, which in turn are used in Appendix C, for calculations. For reference, Appendix C shows the theoretical time needed to completely brute-force the search space of passwords 6-8 characters in length using John, depending on the type of password hash. Keep in mind that these are purely theoretical numbers and that John's incremental mode is not the same as a simple brute-force attack. Nonetheless, the numbers can be used as estimations.

Weaknesses of LM hashes

In the SANS Top 20 list for 2002, one of the top 10 Windows-specific issues is to disable the use of "LAN Manager Authentication – Weak LM Hashing" (Kamerling, p. 4). Appendix C helps show mathematically why LM hashes are considered to be weak by today's standards. From the perspective of using a password recovery software, there are four key issues which result in LM hashes being weak:

1. High rate of passwords/second tested (from the benchmarks listed in Chart 2, John the Ripper can process LM hashes approximately 1,000 times as quickly as FreeBSD MD5 hashes, for instance).
2. Windows passwords 8-14 characters in length are actually split into two 7-byte passwords and are hashed separately. The LM hash is actually two 16-character hashes concatenated together. Any passwords shorter than 14 characters are padded with binary zeroes before hashing.
3. Windows passwords are converted to all uppercase before the LM hash is performed, reducing the number from 95 down to 69 (no lowercase letters) valid possibilities per character.
4. No salt. (Note that NTLM hashes do not have a salt either, but NTLM hashes do not suffer from weaknesses 2 or 3, and are roughly one order of magnitude slower to hash than LM).

The concept of a salt was introduced in the latter portion of "Understanding Password Hashes" earlier in the paper. To better illustrate the effect that a salt has on the speed of password recovery software like John, we will use the following example. We will pretend that we are trying to recover 3 DES password hashes with John, each of the 3 password hashes having salts "aY", "DH", and "/X", respectively. If the current string that John is testing is "hello5" then John must generate the DES hashes of "hello5" with each of the three salts and compare each hash against the 3 hashes that we are trying to recover. Effectively, we are slowing the rate of passwords per second to something close to 1/x where x is the number of salts, in this case 1/3. Bear this in mind when doing any proactive testing. If the goal of your particular test, for instance, is to try to recover the root password from a Solaris host, then create a file with the root entries from /etc/passwd and /etc/shadow only. Do not include any other user entries, unless they happen to have the same salt.

Because of items 2 and 3 above, the maximum search space, as seen in Appendix C, is actually only 69^7 and does not increase as the password length increases from 7 to 8 characters (or higher).
To better illustrate the third point from above, see Chart 3:

Plain-Text	LM hash	NTLM hash
Wubba	Ccdb46f1dde44902aad3b435b51404ee	81fd742d50b5e3e1e6e7c973e18685ae
WUBBA	Ccdb46f1dde44902aad3b435b51404ee	fdd691abd000d71d278c48beef82dece
WuBbA	Ccdb46f1dde44902aad3b435b51404ee	98df8ba04308a24f7423f03f36c8ad72
wUBBa	Ccdb46f1dde44902aad3b435b51404ee	1e9e62d1b3d1b4ca45d392a3874bbecc

Chart 3

Though the capitalization is varied, notice the first half of the LM hash remains constant. Because of the padding with binary zeroes, any passwords 7 characters or shorter will all have the same second half of the LM hash. Well explained on one of Lepton Crack's web pages, "you can immediately know a Windows® password is shorter than 8 characters by watching to the second half of the hash: in this case it will be always **AAD3B435B51404EE** (the LM hash of 7 binary zeroes)" (Brunati). But also understand that trying to recover a password from the LM hash does not necessarily give you the original password as it will be in all uppercase. For every letter in the recovered LM hash, there are two possibilities. If John recovers the password of "A3D6[RY" from the LM hash, because there are four letters contained in the password, there are actually 2^4=16 possible passwords. From trial and error, when PWSE performs an "LM+NTLM attack" it appears to first find the upper-case plain-text password from the LM hash. Almost immediately after, PWSE then shows the case-sensitive password. My guess on how this is accomplished is that PWSE probably uses the case-insensitive password recovered from the LM hash and then brute-forces all upper/lower case combinations against the NTLM hash. After all, even if the plain-text password was 14 alpha characters, 2^14=16,384 possible passwords and can be brute-forced against the NTLM hash very quickly.

Of the four types of hashes discussed in this paper, another weakness unique to LM hashes is a consequence of issue 2 listed above. Certain passwords which are composed of an entire dictionary word with other characters appended can become trivial to recover from the LM hash. Even though basing a password on a dictionary word goes against best practice, programs like John will have a much more difficult time recovering those passwords if another hashing algorithm is used. Demonstrations 1-3 will reinforce the weakness of LM hashes.

See Appendix E for Demonstration 1. The basic lesson learned from Demonstration 1 is that in certain cases, basing a password on a dictionary word can make the password trivial to recover from the LM hash. This is largely due to the unique property mentioned earlier that the LM hash is really just two concatenated hashes from 7-byte strings.

See Appendix F, which references Appendix B, for Demonstration 2. This demonstration contrasts recovering LM hashes versus recovering DES hashes. 50 plain-text passwords were used to create their corresponding LM and DES hashes. All plain-text passwords were randomly generated, exactly 8 characters in length, consisting of a mix of alpha, numeric, and special characters. The main point of this exercise is to show that given a set of passwords with composition that follows best practices, the DES hashes is relatively strong while the LM hashes are comparatively weak.

See Appendix G for Demonstration 3. The purpose of this demonstration is to intentionally go a bit overboard, trying to recover *all* passwords, not just one or some. In retrospect, it would have been a bit more useful to do the same experiment that had stricter password complexity rules enforced. Nonetheless, it should give you cause for concern having demonstrated that all LM hashes on a server were recovered in under 39 days on a single PC, and under 9 days on that same PC running a newer version of John. The risk of using LM hashes on this server, for instance, even if following the practice of rotating passwords every four months, a malicious user that was able to obtain a password hash from this host would have a (120 days – 9 days)/(120 days) = 92.5% chance of recovering the password before it was rotated. Note that this requires no specialized knowledge. All that is needed is a single PC with a fast processor, the most recent version of John with its default configuration, and any LM hash from the particular host. If you are still not convinced to move away from using LM authentication, I encourage you to perform Demonstration 3 in your own environment and to closely look at the demonstrations presented in the "Project Rainbow Crack" web page or the recent announcement on the "Whitehat Project" web page.

Hopefully the results of demonstrations 1-3 and the numbers presented in Appendix C strongly reinforce the "SANS Top 20" recommendation to move away from using LM authentication. Look into using NTLM or even NTLMv2 for authentication, if possible. Additionally, Microsoft Knowledge Base Article 299656 provides an explanation of how to configure Windows not to store LM hashes.

Shifting the focus from demonstrating the weaknesses of LM hashes, we will move on to a couple of practical demonstrations that may be useful for internal auditing in general. See Appendix H for Demonstration 4. Demonstration 4 shows the usefulness of using a larger dictionary with John to recover, more quickly, a larger subset of weaker passwords. As mentioned in the demonstration, should you be conducting your internal auditing in an environment that historically has used a small number of simple passwords any time that a new user account was created or when a password was reset, go ahead and add those simple passwords directly to the dictionary that you are using (if they are not already in your wordlist) to recover them more quickly.

See Appendix I for Demonstration 5. Demonstration 5 gives an example of taking password complexity rules imposed in your environment and then writing some corresponding custom rules to perform a more focused password attack. If you have password complexity requires posted as a policy on your company's Intranet, then it is not unreasonable to assume that a malicious attacker that is sniffing your network also is able to see this policy and also create custom rules in John to perform such a focused attack.

Hopefully Demonstrations 4 and 5 give you some ideas of how you might vary, and consequently improve your own testing. If you can significantly enhance your testing methods and still not recover any passwords, you are improving the security of your environment.

Conclusion

Hopefully this paper has helped educate you regarding the basics and importance of password management, demonstrated the use of password recovery software to be used when internally auditing your environments, and used math to reinforce key concepts. The use of the password management capabilities of the operating system in addition to the proactive use of password recovery software will make it much more difficult for a malicious user performing a password attack in addition to better preparing your organization for a third-party audit.

List of References

Brunati, Piero, “An experiment with Lepton’s Crack”, June 22, 2003, URL: http://www.nestonline.com/lcrack/lcexp1.html, (February 5, 2004).

“Computer and Network Hacker Exploits – Part 3”, URL: http://www.sans.org/sans2004/description.php?course=t4&day=4, (February 9, 2004).

“Federal Information Processing Standards Publication 180-1, April 17, 1995, URL: http://www.itl.nist.gov/fipspubs/fip180-1.htm, (February 7, 2004).

“Federal Information Systems Controls Audit Manual (FISCAM)”, June 13, 2000, URL: http://csrc.nist.gov/ispab/june13-15/Heim.pdf, February 7, 2004.

“The Hack FAQ: 13.0 NT Passwords”, URL: http://www.nmrc.org/pub/faq/hackfaq/hackfaq-13.html, (March 24, 2004).

Kamerling, Erik, “Top 20 Overview,” October 8, 2003, URL: http://www.sans.org/top20/overview03.pdf, (February 3, 2004)

Klevinsky, T.J., Laliberte, Scott, and Gupta, Ajay, Hack I.T., Pearson Education, Inc., 2002, p. 20-21.

“Microsoft Knowledge Base Article -299656”, October 13, 2003, URL: http://support.microsoft.com/default.aspx?scid=KB;EN-US;q299656&, (February 7, 2004).

“One-way hash function”, URL: http://www.webopedia.com/TERM/O/one-way_hash_function.html, (February 7, 2004).

“Password Policy”, URL: http://www.sans.org/resources/policies/Password_Policy.pdf, (February 3, 2004).

“Password Policy Enforcer: FAQ”, URL: http://www.anixis.com/products/ppe/faq.htm, (March 16, 2004).

“Project Rainbow Crack,” January 25, 2004, URL: http://www.antsight.com/zsl/rainbowcrack/, (February 5, 2004).

“Security Resource Document”, URL: www.isaca-jax.org/Security%20Resources.doc, (February 9, 2004).

Skoudis, Ed, “Permission Memo”, URL: http://www.counterhack.net/permission_memo.html, (February 8, 2004).

"Whitehat Project", URL: http://www.whitehat.co.il/forum_viewtopic.php?14.149, (March 25, 2004).

Appendix A:
Links to some of the tools referenced in this paper:

Name: Cisilia
URL: http://www.cisiar.org/proyectos/cisilia/home_en.php

Name: Control-SA
URL: http://www.bmc.com/products/proddocview/0,,0_0_0_1587,00.html

Name: djohn
URL: http://mobile.securiteam.com/tools/6R00E2K8UA.html

Name: John the Ripper
URL: http://www.openwall.com/john/

Name: Knoppix Security Tools Distribution (STD)
URL: http://www.knoppix-std.org/

Name: LC 4 (formerly L0phtcrack)
URL: http://www.atstake.com/products/lc

Name: Lepton's Crack
URL: http://usuarios.lycos.es/reinob/

Name: Npasswd
URL: http://www.utexas.edu/cc/unix/software/npasswd/doc/
Dictionaries: http://www.utexas.edu/cc/unix/software/npasswd/dist/npasswd-words.tar.gz

Name: Password Policy Enforcer
URL: http://www.anixis.com/products/ppe/default.htm

Name: Proactive Windows Security Explorer
URL: http://www.elcomsoft.com/pwsex.html

Name: pwdump2
URL: http://razor.bindview.com/tools/desc/pwdump2_readme.html

Name: Rainbow Crack
URL: http://www.antsight.com/zsl/rainbowcrack/

Appendix B:

Plain-text	LM hash	DES hash	Time to find LM hash
|n&Jnn37	9432179b4cb24cf97c3113b4a1a5e3a0	MRRfw7BvTPR.Y	Not found
.cUJNQ$3	30ec11f5a8fc7c1f1aa818381e4e281b	ZRXdGwGI8M43k	1:21:59:43
6KYK4,9g	b8a9b7179731e499df128b2dd32bad07	jRyJgjd7M17BQ	6:22:41:33
6022sZh_	e763b954973d121912cf283a437f8859	sRik7Mde/u3Qo	0:12:07:43
r4Dl>:^M	55d22c7ab07810101486235a2333e4d2	0SoP9lCBmunEI	Not found
8`$b;>8Z	fafc84d05bb4f91d1d91a081d4b37861	8SYPJT4IljTBQ	Not found
uQu`V522	7e0734a05919857c1d71060d896b7a46	KSB17uHqZaXjI	Not found
0o4yR^.d	67174945192e338b4a3b108f3fa6cb6d	aS.AGuW/uXLRQ	1:16:48:32
IYF93j1`	666af25efe19fff35f0f0b90039414f3	nSL6xVNKSNF.A	0:06:29:55
H,-;Y1q1	ce5d50fb1b5cf0afc2265b23734e0dac	wSkclsExdui8M	Not found
'#0H8qL`	78895c2d6f506d175f0f0b90039414f3	9TC17C1qIFz42	Not found
P/G`c$?0	28ba04fba55c2e3525ad3b83fa6627c7	NT63d6McmDzzE	Not found
@@w3E&f}	c3909886fb2c2e89487f15d729d904ea	WTYIKqcUbiVKA	2:05:10:54
nA05H]85	cf2af2217453eb8c9c5014ae4718a7ee	fTaxON.tR3XHw	Not found
5&p9RDi,	f1fa1da6e47bcac0873dadf71449719f	nTUIQ4lSZAziA	2:01:23:24
#E5,r9?u	dab277700c8942af613e9293942509f0	zT53Pd7CK9nrI	Not found
u<7G0P5!	bc20b9050d5d215f695109ab020e401c	6UOq1ORshdzMM	Not found
V0@q!b9(	45a9aed8c91ae10a57c147c060d0438a	GU4fmQhW3MHC.	2:11:20:45
3t1EQ]7~	6c0b82e6e1ec838c86bb3c2b4237a797	QU7EwGzYfDqD6	Not found
R~Xu1vCa	2e42b9907c508d527584248b8d2c9f9e	bUGStfOkBefjk	Not found
G3{@C+;n	838cb8e99ccbe9d5e72c57ef50f76a05	kUtT3WrXDSsnY	Not found
Mh19$B8L	d7b55cdc3109b0edf500944b53168930	rUZ1DnxYfF1Dk	1:11:12:22
e&P{1C3w	ce4091d20a9913663832c92fc614b7d1	/VN.EO2p/l6C2	Not found
A_OKck;9	9af23ab9f98c8d0e09752a3293831d17	AVZT6hJecm47M	3:07:27:41
V6{m,wwS	1e702de9ac79759093e28745b8bf4ba6	MV8.Jh11THx/6	Not found
cFjW1*9M	67fcec4a0f567bd61486235a2333e4d2	YVbQmipWgo9BQ	4:18:57:47
15gvJm*"	aaf67ed65018bc6da02baaa4a6ae5fea	.Wf8UF9rXMTI2	0:03:50:50
R6<Z5g0?	74fad0f9e662efc28b4ddca42d5815ff	7W6IFZNM09cBk	Not found
x&x)QC84	278a744902bd0d80ff17365faf1ffe89	GWRrFaHOu850A	Not found
f6&4v<yY	350ce694cb95941eb79ae2610dd89d4c	SWDMNjKfMXyDY	Not found
k74EP-Hq	8f40e5e84d10b361d8f7f5860820ed3f	bWCgNU.kuVffE	1:09:13:50
9&p\/igY	3320534257f86395b79ae2610dd89d4c	nW7an5LS9ADks	Not found
TjU^#1D9	ad243f31805ac38209752a3293831d17	/XCNiiqmUy8uw	4:20:54:25
}rWxHG74	47673e30c9fa096cff17365faf1ffe89	9XP8zLVf/1Tio	Not found
Lt5e,_NJ	d99db062b4da04ff7ca65f36030673dd	GXe54LnpnjEYc	Not found
2[0CR7<o	7f68e6b47b441e6ee68aa26a841a86fa	NXudVV8fxNu4g	Not found
57Aho=}9	31cef9be876b7f2909752a3293831d17	YXI5mz.YO1ZzE	Not found
Q:O!5;`o	3d8553c34562900ce68aa26a841a86fa	fXmX6NLeUepOc	Not found
40S1Z\z$	365d2155b6c8a661db2294261f598b4c	oXNAposEQ.cHw	Not found
Fv6@32?e	8093af02f1b7996717306d272a9441bb	vXC18ytS0DyR.	1:03:05:45
kkYJe^N1	1370250dd9e0ee79c2265b23734e0dac	6YEhZ2bdZDJD2	1:04:03:13
sm\3xX|6	4c342e2e921cfbb3c81667e9d738c5d9	EYb5NXhZYlQPg	Not found

yC!2v7-3	964b1be7bb9c81c61aa818381e4e281b	LY15jO6R8wyJc	5:19:55:17
@/oLH7-P	b47b32c603adf29c8b0ea5a7df135b03	SYaq6CfYymius	Not found
05jOdQ(3	4d20fbc3c14491441aa818381e4e281b	bY9XIY9WnYYzQ	5:05:49:29
94V2ah'Y	8f099644600b6d40b79ae2610dd89d4c	mYKsXtnuWJmnA	2:14:11:59
^UI.y5W\	1134dfec82c9664ed994ad8c40370504	yYQ6cSOesCrso	Not found
t)FX1)'P	cd1e88ba76e1224e8b0ea5a7df135b03	3ZhYcmtDa8ktg	6:15:10:20
tB$+P6F]	f5f0d824efa93c765e314a31e03c844b	BZJ6BdC.zF8og	Not found
>7i30S*"	cbcf87f7fcc6e212a02baaa4a6ae5fea	IZoJjzDUiFhl.	Not found

The above hashes were created by manually creating user accounts and manually setting the user passwords so that the operating system would create the password hashes. Since there was significant opportunity for human error due to the manual copying and pasting, I had to verify the validity of all of the hashes. To accomplish this, I created a text file that contained the 50 plain-text passwords only and copied this file on top of the password.lst file. I then ran the LM hashes against John, followed by the DES hashes and ensured that all passwords were found very quickly. In case you are wondering how John quickly recovered the passwords in wordlist mode considering all of the dictionary words were 8 characters but LM hashes are only generated from 7 characters at a time, John has a rule to truncate the words at 7 characters and to try those.

Appendix C:

Total number of 6-character passwords	107,918,163,081	735,091,890,625	735,091,890,625	735,091,890,625
Type of hash	LM	NTLM	DES	FreeBSD MD5
Passwords/second tested	3,856,473	620,534	543,014	4,072
Total seconds needed to brute-force password	27,984	1,184,612	1,353,725	180,523,549
Total minutes	466.39	19,743.53	22,562.09	3,008,725.81
Total hours	7.77	329.06	376.03	50,145.43
Total days	0.32	13.71	15.67	2,089.39
Total years	0.00	0.04	0.04	5.72

Total number of 7-character passwords	7,446,353,252,589	69,833,729,609,375	69,833,729,609,375	69,833,729,609,375
Type of hash	LM	NTLM	DES	FreeBSD MD5
Passwords/second tested	3,856,473	620,534	543,014	4,072
Total seconds needed to brute-force password	1,930,871	112,538,120	128,603,921	17,149,737,134
Total minutes	32,181.19	1,875,635.33	2,143,398.68	285,828,952.23
Total hours	536.35	31,260.59	35,723.31	4,763,815.87
Total days	22.35	1,302.52	1,488.47	198,492.33
Total years	0.06	3.57	4.08	543.44

Total number of 8-character passwords	7,446,353,252,589	6,634,204,312,890,625	6,634,204,312,890,625	6,634,204,312,890,625
Type of hash	LM	NTLM	DES	FreeBSD MD5
Passwords/second tested	3,856,473	620,534	543,014	4,072
Total seconds needed to brute-force password	1930871.356	10691121378	12217372504	1.62923E+12
Total minutes	32,181.19	178,185,356.29	203,622,875.07	27,153,750,462.06
Total hours	536.35	2,969,755.94	3,393,714.58	452,562,507.70
Total days	22.35	123,739.83	141,404.77	18,856,771.15
Total years	0.06	338.78	387.15	51,627.03

Appendix D: Acronyms and abbreviations used in this paper

Acronym or Abbreviation	Explanation
DES	Data Encryption Standard
FAQ	Frequently Asked Questions
FISCAM	Federal Information Systems Controls Audit Manual
GAO	General Accounting Office
GHz	Gigahertz
ISACA	Information Systems Audit and Controls Association
IT	Information Technology
John	John the Ripper
LC4	L0phtcrack 4
LM	Lan Manager
MD5	Message Digest 5 – as defined in RFC 1321
MHz	Megahertz
MMC	Microsoft Management Console
NIS	Network Information Service
NTLM	NT Lan Manager
P-4	Pentium 4
PWSE	Proactive Windows Security Explorer
RFC	Request For Comments
SAM	System Account Manager
SANS	System, Audit, Network, and Security
SMIT	System Management Interface Tool

Appendix E: Demonstration 1

Experiment: 6 Windows 2000 user accounts are set to 6 different passwords. The first password is composed of a dictionary word plus one number. Each successive password contains the exact previous password plus one more character.
The pwdump2 utility is used to extract the LM and NTLM hashes. The LM hashes are run against John. What sort of results might we expect?
Results:
Config-4 was used. The passwords get progressively more complex due to the length of the password incrementing by one character each time and by there being a mix of alpha, numeric, and special characters. To the naked eye, the main weakness appears to be that all contain a complete dictionary word.
It took only 2 hours, 43 minutes, and 55 seconds to recover all of them. See Chart 4:

Plain-text password	First 7 characters	Time to recover first half of hash	Second 7 characters	Time to recover second half of hash
harley7	harley7	0:00:00:00		N/A
harley7$	harley7	0:00:00:00	$	0:00:00:02
harley7$-	harley7	0:00:00:00	$-	0:00:00:10
harley7$-E	harley7	0:00:00:00	$-E	0:00:00:46
harley7$-E4	harley7	0:00:00:00	$-E4	0:00:12:42
harley7$-E4!	harley7	0:00:00:00	$-E4!	0:02:43:55
Note that time is expressed in DAYS:HOURS:MINUTES:SECONDS logged in john.log.				

Chart 4

John has a built-in rule in wordlist mode that converts a pure alphabetic word to lowercase and then appends a digit or simple punctuation. Because “harley” happens to be in John’s default dictionary, the first half of all of the passwords was recovered very quickly (in less than one second). Now all that is left is to recover the second halves of passwords, the longest of which is 5 characters. When the corresponding NTLM hashes were run against John, the first password was still recovered very quickly (due to the same rule described in the first sentence of the paragraph), but none of the other passwords were recovered, not even “harley7$” after 72 hours of testing. By default, when John enters incremental mode, it looks for passwords between 1 and 8 characters long, inclusive. Unless any of the last 4 passwords were recovered in single-crack or wordlist mode, John would never find them without the maximum length of the password being increased. Appendix C shows that it could theoretically take the single PC used in Config-4 approximately 338 years to recover an 8-character password from its NTLM hash. As explained earlier, expanding the search space to recover 9-character passwords increases the worst-case to 338 * 95 = 32,110 years.

For a variation on demonstration 1, the “An Experiment with Lepton’s Crack” web page gives a helpful walk-through of how to use that particular toolset with regular expressions to attack LM hashes where a dictionary word is split across the two halves of the LM hash and one half has been recovered.

Appendix F: Demonstration 2

Experiment: A random password generator was used to generate 50 passwords, all 8 characters in length. The "space" character was omitted as a valid character, so there were 94 instead of 95 possibilities per character. Dummy accounts were created under Windows and UNIX so the LM and DES hashes could be captured. The LM hashes were run against John with no configuration changes for one week. The DES hashes were also run for one week against John. Because the 1-7 character password search space for DES hashes is still significant, I tried to help level the playing field by making one modification to john.conf. In the "[Incremental:All]" stanza, I modified the "MinLen = 1" to read "MinLen = 8" to restrict the length of the password guesses to exactly 8 characters. What were the results?

Results: Config-3 was used for both searches. None of the DES hashes were recovered in one week. As for the LM hashes, John will quickly find any single-character passwords; some from its word mangling rules in wordlist mode and some from incremental mode. Consequently, the 8th character of all 50 LM hashes was found very quickly. Additionally, 20 out of 50 (40%) complete passwords were found in one week. See Appendix B for the list of plain-text passwords, their corresponding LM and DES hashes, and also the timestamps for the complete LM hashes that were recovered. Also note that if we extrapolate these findings, they are fairly consistent with Appendix C. If 40% of the passwords were recovered in 7 days, one would estimate that 7 * 2.5 = 17.5 days would be needed to recover all passwords. This is within the boundaries of the estimated 22 days. However, even though the 50 passwords in this exercise are strong passwords, it is not a very large sampling of passwords.

Appendix G: Demonstration 3:

Experiment: Just how long will it take to recover ALL (not just some) LM hashes from a Windows server? For this demonstration, a Windows 2000 domain controller was used. The only password restriction imposed at the time on the server was a minimum length of 6 characters. How long did it take to recover all of them?

Results: Config-1 was used. No configuration changes to John were made.

- There were 5,866 user accounts split into 11,178 7-byte chunks which will be referred to as "fragments."
- 5,424 fragments were found in the first minute. These were mostly the single-character second-halves of passwords plus some dictionary words found quickly in wordlist mode.
- Another 61 fragments were found in the first hour, making a total of 5,485 fragments that included 180 complete passwords in the first hour.
- All passwords were recovered in 38 days, 9 hours, 54 minutes, and 37 seconds. An interesting note here is that the last 7 of the 38 days were spent finding a solitary fragment. To the naked eye, it did not appear very complex, being three letters, followed by a special character, followed by three more letters.

This experiment was rerun with Config-3. From Chart 2 we see that Config-3 should theoretically run approximately 6 times as fast as Config-1 for NTLM hashes. In reality, the ratio was closer to 4.5 as fast, taking 8 days, 8 hours, 40 minutes, and 0 seconds to recover all NTLM hashes.

Appendix H: Demonstration 4:

Experiment: John's FAQ recommends finding a better wordlist and gives a link to an ftp site. Because the link appears to be dead, wordlists from Npasswd's website (see Appendix A) were downloaded. The dump of LM hashes from demonstration 3 will be used and run against each of these dictionaries in John's wordlist mode and again in wordlist mode with the word mangling rules enabled. Does the use of different wordlists make much of a difference?

Results:
Config-1 was used. See Chart 5:

Name of Wordlist	Number of words in wordlist	Fragments recovered in wordlist mode	With word mangling rules enabled
Antworth	89523	109	2557
CIS	8714	0	2676
Congress	740	889	1887
CRL-Words	44880	109	2557
Domains	514	0	1705
Dosref	535	0	1938
Ethnologue	42441	2	2676
Family-Names	13484	2	2567
Ftpsites	831	1	1674
Given-Names	8605	0	2368
Jargon	9460	2	2202
Koran	2678	1354	1889
Lcarrol	2158	0	1381
Movies	38137	3	2566
Muffet-Words	1450251	5260	5266
Paradise-Lost	6036	0	2432
Python	3444	1	2098
Roget-words	17474	107	2716
Trek	530	0	1670
Unabr-Dict	213557	111	2670
UNIX-Dict	25104	2781	2787
World-Factbook	12441	1	2676
Zipcodes	15489	0	2478
<combined> *	1471706	5260	5266

Chart 5

Additionally, I ran all wordlists through the "cat * | sort | uniq" commands (which concatenates all of the lists, sorts the resulting list alphabetically, and then removes duplicate entries) from a UNIX shell and used the resulting file as a wordlist. See the bottom entry in Chart 5. As you can see, it did not find any more words than the "Muffet-Words" dictionary. Reminder that each "word" in the wordlist is not limited to letters only, it is any string of characters. If you are

proactively testing in an environment where, historically, you are aware that new user passwords tend to be simple passwords like “newuser” or “hello1”, simply add these simple passwords to John’s default dictionary (password.lst). Even though John would likely recover such passwords quickly anyway, this will ensure that such passwords are recovered while John is still in wordlist mode.

Appendix I: Demonstration 5:

Experiment: Because John enables you to add your own rules, in your testing, consider writing some custom rules tailored to the password complexity rules of your environments. If your passwords can still not be recovered even after adding custom rules like in this example, you can have even more confidence that a malicious user or auditor will be unable to recover the passwords either. For instance, we will pretend that passwords must have at least one letter, one number, and one special character. Given this criteria, it is not unreasonable to expect lots of people to choose a dictionary word with a single digit and single special character prefixed or appended or some combination thereof. Use the RULES document that comes with John's documentation for an explanation of how to interpret existing rules or to write your own. John already has several rules that prefix and append certain characters. In this example, we will cover all possibilities of a dictionary word (with no capitalization changes) combined with a single digit and single special character.

Results: Config-3 was used. See Chart 6 for the test bed of passwords.

Username	Password	Rule that will recover password
user0	tigger	Default wordlist mode
user1	Tigger7-	First custom rule
user2	Tigger!6	Second custom rule
user3	2tigger:	Third custom rule
user4	?tigger9	Fourth custom rule
user5	8<tigger	Fifth custom rule
user6	[3tigger	Sixth custom rule
user7	1tigger	Default wordlist mode with file mangling rules enabled

Chart 6

To show the progression, John will be run three different times.

Run 1: "tigger" is a word in John's default dictionary. John is run in wordlist mode only. The syntax of

```
./john –wordlist:password.lst tigger.txt
```

will run the hashes contained in tigger.txt against John in wordlist mode only using the default dictionary. See Screen Shot 3:

```
root@darkstar:/tmp/demo5# ./john -wordlist:password.lst tigger.txt
Loaded 8 password hashes with 8 different salts (FreeBSD MD5 [32/32])
tigger           (user0)
guesses: 1  time: 0:00:00:00 100%  c/s: 400  trying: tigger
```

Screen Shot 3

Run 2: Now we will add the "-rules" option to enable the word mangling rules in wordlist mode, one built-in rule being to prefix pure alphabetic words with "1" . See Screen Shot 4:

```
root@darkstar:/tmp/demo5# ./john -rules -wordlist:password.lst
tigger.txt
Loaded 8 password hashes with 8 different salts (FreeBSD MD5 [32/32])
tigger           (user0)
1tigger          (user7)
guesses: 2  time: 0:00:00:00 100%  c/s: 1783  trying: Tiggering
```
Screen Shot 4

Run 3: Now we will add the following six rules to the john.conf file in the [List.Rules:Wordlist] section to run *before* the other word mangling rules. The same syntax as Run 2 will be used. See Screen Shot 5:

```
# Append a single digit followed by a special character
$[0-9]$[`~!@#$%^&*()\-_=+\[{\]}\\|;:'",<.>/?]
# Append a special character followed by a single digit
$[`~!@#$%^&*()\-_=+\[{\]}\\|;:'",<.>/?]$[0-9]
# Prefix a single digit, then append a special character
^[0-9]$[`~!@#$%^&*()\-_=+\[{\]}\\|;:'",<.>/?]
# Prefix a special character, then append a single digit
^[`~!@#$%^&*()\-_=+\[{\]}\\|;:'",<.>/?]$[0-9]
# Prefix a single digit followed by a special character
^[0-9]^[`~!@#$%^&*()\-_=+\[{\]}\\|;:'",<.>/?]
# Prefix a special character followed by a single digit
^[`~!@#$%^&*()\-_=+\[{\]}\\|;:'",<.>/?]^[0-9]

root@darkstar:/tmp/demo5# ./john -rules -wordlist:password.lst
tigger.txt
Loaded 8 password hashes with 8 different salts (FreeBSD MD5 [32/32])
tigger7-         (user1)
tigger[6         (user2)
2tigger:         (user3)
?tigger9         (user4)
8<tigger         (user5)
\3tigger         (user6)
tigger           (user0)
1tigger          (user7)
guesses: 8  time: 0:00:00:04 100%  c/s: 2175  trying: 1tigger
```
Screen Shot 5

To explain the first rule, the "$" appends a single character (versus the "^" which prefixes a character) and the "[0-9]" is a regular expression that tries the range of single digits 0,1,...,8,9. The rest of the rule appends a single special character, all 32 of which (no space) are enumerated within the square brackets. Worth noting is that the four characters "`[]-\`"have special meaning with the regular expressions and must be escaped by preceding the character with a "\". That is why "\" is listed multiple times in square brackets. Once the first custom rule is understood, the other five rules simply vary the order and placement of the single digit and special character with respect to the dictionary word. John actually has a preprocessor that creates individual rules from the regular expressions. Consequently, the first custom rule above generates 10 * 32 = 320 (number of

digits * number of special characters) rules. Now multiply 320 by 6 (total custom rules with same criteria reordered), which gives 1,920 rules. Now multiply 1,920 by the number of words in the dictionary (default password.lst has 2,290), resulting in 4,396,800 new hashes to try. Using the benchmarks from Config-3 in Chart 2, this will take a couple of seconds when run against LM hashes, but almost 17 minutes when run against FreeBSD MD5 hashes. Exercise caution in writing rules with regular expressions or you might create a rule that would take long enough to execute to make it impractical. For instance, modifying the custom rules in this exercise to include any two characters (not just a digit and special character), along with the merged dictionary from Demonstration 4, the calculation would be (94 characters)(94 characters)(6 rules)(1,471,706 words) = 78,023,965,296 new hashes. From Chart 2, this might only take under 6 hours against LM hashes, but would take roughly 226.5 days to complete against the FreeBSD hashes.