Global Information Assurance Certification Paper

Password Cracking using Focused Dictionaries
By Paul Bobby

1 Problem

Can the chances of cracking an individual's password be improved by using a more focused dictionary? This dictionary would contain words and information that have a specific relationship to the owner of the targeted password. Such information could comprise, names of spouse and children, birth dates, pet names, schools etc.

2 Background

When attempting to crack passwords, there are two often-used approaches: dictionary and brute-force attacks. A dictionary attack involves taking a list of words, then feeding them through the password routine to see if they are the correct password. More specifically, in the case of Unix passwords, for example, the dictionary word is fed through the password hashing mechanism and compared to the stored password. If they match, then we have cracked the password.

Furthermore, when using a dictionary attack, the words are often hybridized to account for subtle password changes. For example, under a password policy where the password must change every 30 days, the password could start off as 'puddles21' and then next time it's changed become 'puddles22'. This is a common practice among password users as it makes it easier for the individual to remember a new password. But 'puddles21' does not exist in any dictionary that I know of. Yet a simple hybrid of the dictionary word 'puddles' could put numbers ranging from 1-100 at the end of the word, and we would then crack the password.

Many dictionaries exist that have been crafted based on language, movies, music, location, schools, books, politics and so forth. By using one dictionary after another, and then feeding them through some hybridization routines, we can increase the chance of cracking an individual's password.

The brute-force method of cracking passwords is a sure way to find any password, providing that the character set used is large enough. This method can take a lengthy amount of time to complete. In a sense, the brute-force method is essentially using the largest dictionary that can be created from any given character set.

However, can we improve the dictionary that we initially start off with?

3 Data Collection

Do people tend to use passwords that have a close relationship to some aspect of their life? This question is hard to answer without qualitative study. But as an administrator I have seen people use names of pets, children, race car drivers, rock stars, planets and even swear words as passwords. My particular password policy also requires that an individual use at least one (1) digit in their password.

There are also those individuals who use passwords that have no relation to them, such as perkins, c1t1bank, puzz[{s and so forth.

And then there are those who use phrases as their password, for example "hmniwbmvimpvm" (From that great movie Sneakers – can you guess?)

If we are looking for a specific password of a specific individual, we could improve our chances of cracking their password by using a dictionary that contains words with a close relationship to that individual.

What follows is a spreadsheet that could be used to collate personal information about your target. Remember the movie Wargames? Turns out that the password was the name of the target's son: Joshua. (Of course, joshua is in a standard dictionary, but hey you know what I mean)

	Target	Spouse	Child 1	Child 2	Child 3	Father	Mother
First name							
Middle name(s)							
Last name							
Maiden name							
SSN							
Major part of street address							
City							
County							
State							
Zip							
Telephone (1231231234)							
Birthdate (mmddyy)							
Employer							
High school name (major part)							
University name (major part)							
Job title							
Major (eg Computer Science)							
Pet breed							
Pet names							
Car model							
Hobby names and keywords							
TV show names and keywords							
Movie titles and keywords							
Already known passwords							

4 Hybridization

Now that we have our raw data, we need some rules for creating hybrids of this information. The hybrids of these passwords aren't actually stored in the dictionary themselves because the dictionary would grow too big. Rather, the hybrids are generated on the fly by the password-cracking program.

I have chosen to use three basic rules:

- Grammar
- Letter changing
- Number spread

4.1 Grammar

Words can be changed through simple appends. Specifically, append the following:

- -ing (e.g. pauling)
- -er (e.g. pauler)
- -s (e.g. pauls)
- -'s (e.g. paul's)
- -4u (e.g. paul4u)
- -4you
- -2b (e.g. paul2b)
- -2be
- reverse the word

It should be noted that many of the words become 'nonsense' after the grammar change, but our process is not designed to care about nonsense words. Rather it generates quick hybrids of the words, and feeds them through the password program.

4.2 Letter changing

Words can be changed by substituting characters that 'look' like their letter equivalent. The list I suggest follows:

• O or o	becomes	0 (zero)	(e.g. p00l)
• 0	becomes	O or o	
• G or g	becomes	@	(e.g. scra@@y)
• L or l or I or i	becomes	1 or ! or |	(e.g. pub!|c)
• 3	becomes	] or }	
• E or e	becomes	[or {	(e.g. jo{)
• S or s	becomes	$ or 5	(e.g. $tuff)
• C or c	becomes	(or [	(e.g. ar(h)

A common technique of hardening passwords is to substitute non-standard characters for the regular letters of the alphabet, or other numbers. These letter-changing rules can help attack this practice.

4.3 Number spread

Each newly generated hybrid should be used with a number spread. The range I suggest is 0 through 99. For example,

Puddles21
Paul23bobby
Cbs12news

5 Attack Process

Now that we have defined our dictionary and hybridization rules, in what order should we apply them? The following table suggests a procedure. Some definitions that apply are:

Grammar change = gc
Letter change = lc
Number spread = ns
Two sample dictionary words are 'joe' and 'public'

Please note that these calculations were based on all words being used as permutations together. In other words, when analyzing wordword pairs, the word pairs themselves are not duplicates, but rather mathematical permutations. For example, the words joe and public would form 4 permutations unlike most dictionary cracking programs that would only form duplicates of the word; i.e. joejoe and publicpublic.

word	joe
word(gc)	joe's
lc(word)	J0{
word(ns)	Joe21
(ns)word	14joe
word(gc)(ns)	Joeer34
(ns)word(gc)	87joe's
lc(word(ns))	J0[34
lc((ns)word)	45j0e
wordword	Joepublic
word(gc)word	Joe'spublic
wordword(gc)	Joepublics
word(gc)word(gc)	Joe'spublics
word(ns)word)	Joe31public
word(gc)(ns)word	Joe's31public
word(ns)word(gc)	Joe31publics

word(gc)(ns)word(gc)	Joe’s31publics
lc(wordword)	J0epubli{
lc(word(gc)word)	Jo[‘spublic
lc(wordword(gc))	J0epublics
lc(word(gc)word(gc))	J0e’spub!!{s
lc(word(ns)word))	J0e21pub!!c
lc(word(gc)(ns)word)	J0e’s34publi{
lc(word(ns)word(gc))	Jo[23publi{s
lc(word(gc)(ns)word(gc))	J0e’s45pub!!cs

This list of hybridization combinations is not exhaustive. It is a good start. For example, wordword(gc)(ns) is missing, along with others.

The reason I chose the format of the last series of letter changes is because if the individual makes a letter change in one word, chances are they’ll do it to the second word also. Of course someone could concatenate two dictionary words together as their password, and only make letter changes to the first password, but as I said, this is not an exhaustive hybrid list. For example, word(gc)lc(word(ns)) is missing, along with many others.

6 Analysis

How efficient? What numbers are involved? Here’s the hybrid rules list again with new definitions:

Number of words = w
Number of grammar changes = g
Number of letter changes = l
Number spread = n

Just how many combinations will we generate? I hope my math is correct.

word	w
word(gc)	gw
lc(word)	lw
word(ns)	nw
(ns)word	nw
word(gc)(ns)	gnw
(ns)word(gc)	gnw
lc(word(ns))	lnw
lc((ns)word)	lnw
wordword	w^2
word(gc)word	gw^2
wordword(gc)	gw^2
word(gc)word(gc)	g^2w^2
word(ns)word)	nw^2
word(gc)(ns)word	ngw^2

word(ns)word(gc)	ngw^2
word(gc)(ns)word(gc)	ng^2w^2
lc(wordword)	l^2w^2
lc(word(gc)word)	gl^2w^2
lc(wordword(gc))	gl^2w^2
lc(word(gc)word(gc))	$l^2g^2w^2$
lc(word(ns)word))	nl^2w^2
lc(word(gc)(ns)word)	n((lgw)+lw)
lc(word(ns)word(gc))	n((lgw)+lw)
lc(word(gc)(ns)word(gc))	$nl^2g^2w^2$

Total number of combinations is the sum of column 2

Example

w = 4
g = 6
l = 2 (average of 2 letter changes per word)
n = 100

w	4
gw	24
lw	8
nw	400
nw	400
gnw	2400
gnw	2400
lnw	800
lnw	800
w^2	16
gw^2	96
gw^2	96
g^2w^2	576
nw^2	1600
ngw^2	9600
ngw^2	9600
ng^2w^2	57600
l^2w^2	64
gl^2w^2	384
gl^2w^2	384
$L^2g^2w^2$	2304
nl^2w^2	6400
n((lgw)+lw)	5600
n((lgw)+lw)	5600
$nl^2g^2w^2$	230400

Sum = 337556 combinations

Average of 7 bytes per word, would produce a dictionary size of

= 2.25Mbytes

Using a password-cracking program on my P2-400Mhz computer, I can achieve an average crack rate of 4800 login/password combinations per second.

Therefore, time to complete the above example:

= 1.2 minutes (Not bad at all)

Here is a spreadsheet that can be used to compute variations on the 4 parameters:

Number of words (w)	4	w	4
Number of Grammar changes (g)	6	gw	24
Number of Letter changes (l)	2	lw	8
Number spread (n)	100	nw	400
		nw	400
Total Combinations	337556	gnw	2400
		gnw	2400
Average Word Size (bytes)	7	lnw	800
		lnw	800
Dictionary Size (Mbytes)	2.25	w^2	16
		gw^2	96
Cracking speed (tries/second)	4800	gw^2	96
		g^2w^2	576
Time to crack (seconds)	70.32	nw^2	1600
Time to crack (minutes)	1.17	ngw^2	9600
Time to crack (hours)	0.02	ngw^2	9600
		ng^2w^2	57600
		l^2w^2	64
		gl^2w^2	384
		gl^2w^2	384
		$l^2g^2w^2$	2304
		nl^2w^2	6400
		n((lgw)+lw	5600
		n((lgw)+lw	5600
		$nl^2g^2w^2$	230400

You can double click on this object to edit any of the initial values.

7 Implementation

To implement the strategy in this paper we need a password-cracking program. This program should be fast in its attempts to crack passwords, and it should have a flexible enough language to generate the required hybrids of the input dictionary. The best program I could find is one called John the Ripper. Unfortunately, it is not complete, and its shortcomings I will mention later.

7.1 John the Ripper

This excellent program, whose primary purpose is to test weak UNIX passwords, comes with a very large number of rules that change the dictionary words. An exhaustive list of all these rules is beyond this paper, but the reader is referred to http://www.openwall.com/john/ where one can find downloads and documentation.

The standard option of John is to crack passwords using information from the password file itself. The words in the password file can be used in a true 'permutable' fashion; i.e. the generated passwords come from combinations of these words. However, when operating in dictionary mode, John will only hybridize one word at a time. This is a shortcoming of the program; one that the author assures me will be dealt with at some future time.

The workaround to this problem is that we can include our focused dictionary inside of the password information. More specifically, with respect to cracking the UNIX password file, we can place all of these words in the GECOS information areas. The number of words inserted here could be up to 200 or so, which would require a compilation change for John. Specifically:

params.h: SINGLE_WORDS_PAIR_MAX 200
params.h: LINE_BUFFER_SIZE

Unfortunately, John the Ripper is capable of cracking other password types, LDAP for example, and I guess we'll have to wait for the change to the dictionary mode until that time.

The initialization file that is used by John includes a language to construct the hybridization rules. The number of components that comprise a rule has a maximum number of eight (8). This makes some of my rules unusable unless a compilation change is made. The rules are commented out in the supplied initialization file. The compilation change:

params.h: RULE_RANGES_MAX 16

7.2 Data Collection

Convert the collected data from the spreadsheet into a text file with each word delimited by a space. For example:

"paul bobby benji buick high street hacking passwords giac"

Insert this complete line into the following field of a UNIX password file:

root:DDDDDvXK7RDAM:0:0:INSERT HERE:/root:/bin/bash

root:DDDDDvXK7RDAM:0:0: paul bobby benji buick high street hacking passwords giac:/root:/bin/bash

7.3 Initialization File

Appendix A contains the Single Mode cracking rules. For ease of use I also included all the other options within the .INI file so simply cut and paste Appendix A into notepad or emacs, and save it as john.ini.

7.4 Program Execution

Command line:

John.exe –single password.txt

8 Conclusions

Is it possible that compiling the information about the intended target takes much longer than a brute-force attack on the same password? The actual time taken to hybridize and password-test the 200 or so words is the fastest part of this whole process, but it might take a week or so to compile the target information.

Here is a quick calculation, of a brute-force attempt:

Character set: [A-Z][a-z][0-9][~!@#$%^%&*()-=_+`[]{};':"\|,.<>/?]
Total: 95 characters

PC capable of 48000 tries/second at a maximum length of 8 character passwords

Therefore total possible passwords to try:
95^8 (i.e. each character position could be 1 of 95 choices)

$= 6.6e10^{15}$ passwords (then divide by 48000)
$= 1.38e10^{11}$ seconds or about 4382 years!

Here is a small spreadsheet that you can play with to analyze the brute-force possibilities:

Character Set	Number of characters	Tries/Second	Number of characters in password	Total number of passwords	Number of seconds to complete	Number of days	Number of years
[a-z]	26	48000	8	2.088E+11	4350563.8	50.35375	0.137955
[a-z][A-Z]	52	48000	8	5.346E+13	1.114E+09	12890.56	35.3166
[a-z][A-Z][0-9]	62	48000	8	2.183E+14	4.549E+09	52647.59	144.24
95 characters	95	48000	8	6.634E+15	1.382E+11	1599683	4382.692
127 characters	127	48000	8	6.768E+16	1.41E+12	16318295	44707.66

A dictionary attack will always take less time than the corresponding brute-force attack using the same dictionary's character set.

The administrator's password is often the intended target, and they *may* sometimes pick a good password. In that case, brute-force is the only choice. But for the most part they pick poor passwords just like the rest of us. My focused dictionary approach, along with the hybrid rule set, should improve the chances of cracking a password.

9 Authors comments

I have had this idea for some time now. What I been unable to do is perform some qualitative analysis on this. At some point, with the blessing of Sans perhaps, I would like to create a website for a form and a CGI that collects the information and the encrypted/unencrypted password, and then performs a series of tests.

I would collect the unencrypted password in case the end-user is unable to supply it.

The tests to try:

1. Default rules, default dictionary
2. Default rules, focused dictionary
3. My rules, default dictionary
4. My rules, focused dictionary

Such an exercise would have to be run for a period of time.

If an administrator is willing to supply me with password files for testing purposes, I would be more than grateful. I do not need to know where they came from; I am just interested in the results from a cracking attempt.

Appendix A

```
[List.Rules:Single]
# word
:
# word(gc)
# append -ing
!?A$i$n$g
# append -er
!?A$e$r
# append -s
/?a$s
# append -'s
/?a$'$s
# append -4u, 4you
/?a$4$u
/?a$4$y$o$u
)s]$4$u
)s]$4$y$o$u
# append -2b, -2be
/?a$2$b
/?a$2$b$e
#
#
# Letter changing (lc)
#
# O|o
!?A/[oO]s[oO]0
!?A[ul]/[oO]s[oO]0
# L|l|I|i
!?A/[lLIi]s[lLIi][!1|]/[!1|]
!?A[lu]/[lLIi]s[lLIi][!1|]/[!1|]
# 0
!?X/0s0[oO]
# G|g
!?A/[Gg]s[Gg]@/@
!?A[ul]/[Gg]s[Gg]@/@
# 3
!?X/3s3[\]}]
# E|e
!?A/[Ee]s[Ee][\[{]/[\[{]
!?A[lu]/[Ee]s[Ee][\[{]/[\[{]
# S|s
!?A/[Ss]s[Ss][\$5]/[\$5]
!?A[ul]/[Ss]s[Ss][\$5]/[\$5]
# C|c
!?A/[Cc]s[Cc][\(\[]/[\[\(]
!?A[lu]/[Cc]s[Cc][\(\[]/[\[\(]
#
#
#Combo's (O,G)(O,E)(O,S)(O,C)(O,L)
#
# (O,G)
!?A/[oO]s[oO]0/[Gg]s[Gg]@/[@0]
!?A[ul]/[oO]s[oO]0/[Gg]s[Gg]@/[@0]
#
```

```
# (O,E)
!?A/[oO]s[oO]0/[Ee]s[Ee][\[{]/[\[{0]
!?A[ul]/[oO]s[oO]0/[Ee]s[Ee][\[{]/[\[{0]
# (O,S)
!?A/[oO]s[oO]0/[Ss]s[Ss][\$5]/[\$50]
!?A[ul]/[oO]s[oO]0/[Ss]s[Ss][\$5]/[\$50]
# (O,C)
!?A/[oO]s[oO]0/[Cc]s[Cc][\(\[]/[\[\(0]
!?A[ul]/[oO]s[oO]0/[Cc]s[Cc][\(\[]/[\[\(0]
# (O,L)
!?A/[oO]s[oO]0/[lLIi]s[lLIi][!1|]/[!1|0]
!?A[ul]/[oO]s[oO]0/[lLIi]s[lLIi][!1|]/[!1|0]
#
# Combo's (G,L)(G,E)(G,S)(G,C)
#
# (G,L)
!?A/[Gg]s[Gg]@/[lLIi]s[lLIi][!1|]/[!1|@]
!?A[ul]/[Gg]s[Gg]@/[lLIi]s[lLIi][!1|]/[!1|0@]
# (G,E)
!?A/[Gg]s[Gg]@/[Ee]s[Ee][\[{]/[\[{@]
!?A[ul]/[Gg]s[Gg]@/[Ee]s[Ee][\[{]/[\[{@]
# (G,S)
!?A/[Gg]s[Gg]@/[Ss]s[Ss][\$5]/[\$5@]
!?A[ul]/[Gg]s[Gg]@/[Ss]s[Ss][\$5]/[\$5@]
# (G,C)
!?A/[Gg]s[Gg]@/[Cc]s[Cc][\(\[]/[\[\(@]
!?A[ul]/[Gg]s[Gg]@/[Cc]s[Cc][\(\[]/[\[\(@]
#
# Combo's (L,E)(L,S)(L,C)
#
# (L,E)
!?A/[lLIi]s[lLIi][!1|]/[Ee]s[Ee][\[{]/[\[{!1|]
!?A[lu]/[lLIi]s[lLIi][!1|]/[Ee]s[Ee][\[{]/[\[{!1|]
# (L,S)
!?A/[lLIi]s[lLIi][!1|]/[Ss]s[Ss][\$5]/[\$5!1|]
!?A[lu]/[lLIi]s[lLIi][!1|]/[Ss]s[Ss][\$5]/[\$5!1|]
# (L,C)
!?A/[lLIi]s[lLIi][!1|]/[Cc]s[Cc][\(\[]/[\[\(!1|]
!?A[lu]/[lLIi]s[lLIi][!1|]/[Cc]s[Cc][\(\[]/[\[\(!1|]
#
# Combo's (E,S)(E,C)
#
# (E,S)
!?A/[Ee]s[Ee][\[{]/[Ss]s[Ss][\$5]/[\$5\[{]
!?A[lu]/[Ee]s[Ee][\[{]/[Ss]s[Ss][\$5]/[\$5\[{]
# (E,C)
!?A/[Ee]s[Ee][\[{]/[Cc]s[Cc][\(\[]/[\[\({]
!?A[lu]/[Ee]s[Ee][\[{]/[Cc]s[Cc][\(\[]/[\[\({]
#
# Combo (S,C)
#
# (S,C)
!?A/[Ss]s[Ss][\$5]/[Cc]s[Cc][\(\[]/[\[\(\$5]
!?A[ul]/[Ss]s[Ss][\$5]/[Cc]s[Cc][\(\[]/[\[\(\$5]
#
# THAT"S AS FAR AS I GO.... Wanna go further?
#
```

```
# word(ns)
$[0-9]
$[0-9]$[0-9]
#
# (ns)word
#
^[0-9]
^[0-9]^[0-9]
#
# word(gc)(ns)
#
!?A$i$n$g$[0-9]
!?A$e$r$[0-9]
/?a$s$[0-9]
/?a$'$s$[0-9]
/?a$4$u$[0-9]
/?a$4$y$o$u$[0-9]
)s]$4$u$[0-9]
)s]$4$y$o$u$[0-9]
/?a$2$b$[0-9]
/?a$2$b$e$[0-9]
!?A$i$n$g$[0-9]$[0-9]
!?A$e$r$[0-9]$[0-9]
/?a$s$[0-9]$[0-9]
/?a$'$s$[0-9]$[0-9]
/?a$4$u$[0-9]$[0-9]
/?a$4$y$o$u$[0-9]$[0-9]
)s]$4$u$[0-9]$[0-9]
)s]$4$y$o$u$[0-9]$[0-9]
/?a$2$b$[0-9]$[0-9]
/?a$2$b$e$[0-9]$[0-9]
#
# (ns)word(gc)
#
!?A^[0-9]$i$n$g
!?A^[0-9]$e$r
/?a^[0-9]$s
/?a^[0-9]$'$s
/?a^[0-9]$4$u
/?a^[0-9]$4$y$o$u
^[0-9])s]$4$u
^[0-9])s]$4$y$o$u
/?a^[0-9]$2$b
/?a^[0-9]$2$b$e
!?A^[0-9]^[0-9]$i$n$g
!?A^[0-9]^[0-9]$e$r
/?a^[0-9]^[0-9]$s
/?a^[0-9]^[0-9]$'$s
/?a^[0-9]^[0-9]$4$u
/?a^[0-9]^[0-9]$4$y$o$u
^[0-9]^[0-9])s]$4$u
^[0-9]^[0-9])s]$4$y$o$u
/?a^[0-9]^[0-9]$2$b
/?a^[0-9]^[0-9]$2$b$e
#
# lc(word(ns))
#
```

```
# THESE RULES WONT WORK. BUG BUG BUG?
# NEED TO REDEFINE rule_ranges_max
;$[0-9]$[0-9]/[lLIi]s[lLIi][!1|]/[Ee]s[Ee][\[{]/[\[{!1|]
;$[0-9]$[0-9][lu]/[lLIi]s[lLIi][!1|]/[Ee]s[Ee][\[{]/[\[{!1|]
;$[0-9]$[0-9]/[lLIi]s[lLIi][!1|]/[Ss]s[Ss][\$5]/[\$5!1|]
;$[0-9]$[0-9][lu]/[lLIi]s[lLIi][!1|]/[Ss]s[Ss][\$5]/[\$5!1|]
;$[0-9]$[0-9]/[lLIi]s[lLIi][!1|]/[Cc]s[Cc][\(\[]/[\[\(!1|]
;$[0-9]$[0-9][lu]/[lLIi]s[lLIi][!1|]/[Cc]s[Cc][\(\[]/[\[\(!1|]
;$[0-9]$[0-9][ul]/[lLIi]s[lLIi][!1|]/[Ee]s[Ee][\[{]/[\[{!1|]
;$[0-9]$[0-9]/[lLIi]s[lLIi][!1|]/[Ss]s[Ss][\$5]/[\$5!1|]
;$[0-9]$[0-9][lu]/[lLIi]s[lLIi][!1|]/[Ss]s[Ss][\$5]/[\$5!1|]
;$[0-9]$[0-9]/[lLIi]s[lLIi][!1|]/[Cc]s[Cc][\(\[]/[\[\(!1|]
;$[0-9]$[0-9][lu]/[lLIi]s[lLIi][!1|]/[Cc]s[Cc][\(\[]/[\[\(!1|]
;$[0-9]$[0-9]/[Ss]s[Ss][\$5]/[Cc]s[Cc][\(\[]/[\[\(\$5]
;$[0-9]$[0-9][ul]/[Ss]s[Ss][\$5]/[Cc]s[Cc][\(\[]/[\[\(\$5]
;$[0-9]$[0-9]/[lLIi]s[lLIi][!1|]/[Ee]s[Ee][\[{]/[\[{!1|]
# BUG BUG BUG
$[0-9]/0s0[oO]
$[0-9]/[oO]s[oO]0
$[0-9][ul]/[oO]s[oO]0
$[0-9]/[lLIi]s[lLIi][!1|]/[!1|]
$[0-9][lu]/[lLIi]s[lLIi][!1|]/[!1|]
$[0-9]/[Gg]s[Gg]@/@
$[0-9][ul]/[Gg]s[Gg]@/@
$[0-9]/3s3[\]}]
$[0-9]/[Ee]s[Ee][\[{]/[\[{]
$[0-9][lu]/[Ee]s[Ee][\[{]/[\[{]
$[0-9]/[Ss]s[Ss][\$5]/[\$5]
$[0-9][ul]/[Ss]s[Ss][\$5]/[\$5]
$[0-9]/[Cc]s[Cc][\(\[]/[\[\(]
$[0-9][lu]/[Cc]s[Cc][\(\[]/[\[\(]
$[0-9]/[oO]s[oO]0/[Gg]s[Gg]@/[@0]
$[0-9][ul]/[oO]s[oO]0/[Gg]s[Gg]@/[@0]
$[0-9]/[oO]s[oO]0/[Ee]s[Ee][\[{]/[\[{0]
$[0-9][ul]/[oO]s[oO]0/[Ee]s[Ee][\[{]/[\[{0]
$[0-9]/[oO]s[oO]0/[Ss]s[Ss][\$5]/[\$50]
$[0-9][ul]/[oO]s[oO]0/[Ss]s[Ss][\$5]/[\$50]
$[0-9]/[oO]s[oO]0/[Cc]s[Cc][\(\[]/[\[\(0]
$[0-9][ul]/[oO]s[oO]0/[Cc]s[Cc][\(\[]/[\[\(0]
$[0-9]/[oO]s[oO]0/[lLIi]s[lLIi][!1|]/[!1|0]
$[0-9][ul]/[oO]s[oO]0/[lLIi]s[lLIi][!1|]/[!1|0]
$[0-9]/[Gg]s[Gg]@/[lLIi]s[lLIi][!1|]/[!1|@]
$[0-9]l/[Gg]s[Gg]@/[lLIi]s[lLIi][!1|]/[!1|0@]
$[0-9]u/[Gg]s[Gg]@/[lLIi]s[lLIi][!1|]/[!1|0@]
$[0-9]/[Gg]s[Gg]@/[Ee]s[Ee][\[{]/[\[{@]
$[0-9][ul]/[Gg]s[Gg]@/[Ee]s[Ee][\[{]/[\[{@]
$[0-9]/[Gg]s[Gg]@/[Ss]s[Ss][\$5]/[\$5@]
$[0-9][ul]/[Gg]s[Gg]@/[Ss]s[Ss][\$5]/[\$5@]
$[0-9]/[Gg]s[Gg]@/[Cc]s[Cc][\(\[]/[\[\(@]
$[0-9][ul]/[Gg]s[Gg]@/[Cc]s[Cc][\(\[]/[\[\(@]
$[0-9]/[lLIi]s[lLIi][!1|]/[Ee]s[Ee][\[{]/[\[{!1|]
$[0-9]/[lLIi]s[lLIi][!1|]/[Ss]s[Ss][\$5]/[\$5!1|]
$[0-9]/[lLIi]s[lLIi][!1|]/[Cc]s[Cc][\(\[]/[\[\(!1|]
$[0-9]l/[lLIi]s[lLIi][!1|]/[Ee]s[Ee][\[{]/[\[{!1|]
$[0-9]u/[lLIi]s[lLIi][!1|]/[Ee]s[Ee][\[{]/[\[{!1|]
$[0-9]u/[lLIi]s[lLIi][!1|]/[Ss]s[Ss][\$5]/[\$5!1|]
$[0-9]l/[lLIi]s[lLIi][!1|]/[Ss]s[Ss][\$5]/[\$5!1|]
```

```
$[0-9]u/[lLIi]s[lLIi][!1|]/[Cc]s[Cc][\(\[]/[\[\(!1|]
$[0-9]l/[lLIi]s[lLIi][!1|]/[Cc]s[Cc][\(\[]/[\[\(!1|]
$[0-9]/[Ee]s[Ee][\[{]/[Ss]s[Ss][\$5]/[\$5\[{]
$[0-9][lu]/[Ee]s[Ee][\[{]/[Ss]s[Ss][\$5]/[\$5\[{]
$[0-9]/[Ee]s[Ee][\[{]/[Cc]s[Cc][\(\[]/[\[\({]
$[0-9]l/[Ee]s[Ee][\[{]/[Cc]s[Cc][\(\[]/[\[\({]
$[0-9]u/[Ee]s[Ee][\[{]/[Cc]s[Cc][\(\[]/[\[\({]
$[0-9]/[Ss]s[Ss][\$5]/[Cc]s[Cc][\(\[]/[\[\(\$5]
$[0-9]l/[Ss]s[Ss][\$5]/[Cc]s[Cc][\(\[]/[\[\(\$5]
$[0-9]u/[Ss]s[Ss][\$5]/[Cc]s[Cc][\(\[]/[\[\(\$5]
$[0-9]$[0-9]/0s0[oO]
$[0-9]$[0-9]/[oO]s[oO]0
$[0-9]$[0-9][ul]/[oO]s[oO]0
$[0-9]$[0-9]/[lLIi]s[lLIi][!1|]/[!1|]
$[0-9]$[0-9][lu]/[lLIi]s[lLIi][!1|]/[!1|]
$[0-9]$[0-9]/[Gg]s[Gg]@/@
$[0-9]$[0-9][ul]/[Gg]s[Gg]@/@
$[0-9]$[0-9]/3s3[\]}]
$[0-9]$[0-9]/[Ee]s[Ee][\[{]/[\[{]
$[0-9]$[0-9][lu]/[Ee]s[Ee][\[{]/[\[{]
$[0-9]$[0-9]/[Ss]s[Ss][\$5]/[\$5]
$[0-9]$[0-9][ul]/[Ss]s[Ss][\$5]/[\$5]
$[0-9]$[0-9]/[Cc]s[Cc][\(\[]/[\[\(]
$[0-9]$[0-9][lu]/[Cc]s[Cc][\(\[]/[\[\(]
$[0-9]$[0-9]/[oO]s[oO]0/[Gg]s[Gg]@/[@0]
$[0-9]$[0-9][ul]/[oO]s[oO]0/[Gg]s[Gg]@/[@0]
$[0-9]$[0-9]/[oO]s[oO]0/[Ee]s[Ee][\[{]/[\[{0]
$[0-9]$[0-9]u/[oO]s[oO]0/[Ee]s[Ee][\[{]/[\[{0]
$[0-9]$[0-9]l/[oO]s[oO]0/[Ee]s[Ee][\[{]/[\[{0]
$[0-9]$[0-9]/[oO]s[oO]0/[Ss]s[Ss][\$5]/[\$50]
$[0-9]$[0-9]u/[oO]s[oO]0/[Ss]s[Ss][\$5]/[\$50]
$[0-9]$[0-9]l/[oO]s[oO]0/[Ss]s[Ss][\$5]/[\$50]
$[0-9]$[0-9]/[oO]s[oO]0/[Cc]s[Cc][\(\[]/[\[\(0]
$[0-9]$[0-9][ul]/[oO]s[oO]0/[Cc]s[Cc][\(\[]/[\[\(0]
$[0-9]$[0-9]/[oO]s[oO]0/[lLIi]s[lLIi][!1|]/[!1|0]
$[0-9]$[0-9]u/[oO]s[oO]0/[lLIi]s[lLIi][!1|]/[!1|0]
$[0-9]$[0-9]l/[oO]s[oO]0/[lLIi]s[lLIi][!1|]/[!1|0]
$[0-9]$[0-9]/[Gg]s[Gg]@/[lLIi]s[lLIi][!1|]/[!1|@]
$[0-9]$[0-9]u/[Gg]s[Gg]@/[lLIi]s[lLIi][!1|]/[!1|0@]
$[0-9]$[0-9]l/[Gg]s[Gg]@/[lLIi]s[lLIi][!1|]/[!1|0@]
$[0-9]$[0-9]/[Gg]s[Gg]@/[Ee]s[Ee][\[{]/[\[{@]
$[0-9]$[0-9][ul]/[Gg]s[Gg]@/[Ee]s[Ee][\[{]/[\[{@]
$[0-9]$[0-9]/[Gg]s[Gg]@/[Ss]s[Ss][\$5]/[\$5@]
$[0-9]$[0-9][ul]/[Gg]s[Gg]@/[Ss]s[Ss][\$5]/[\$5@]
$[0-9]$[0-9]/[Gg]s[Gg]@/[Cc]s[Cc][\(\[]/[\[\(@]
$[0-9]$[0-9][ul]/[Gg]s[Gg]@/[Cc]s[Cc][\(\[]/[\[\(@]
$[0-9]$[0-9]/[Ee]s[Ee][\[{]/[Ss]s[Ss][\$5]/[\$5\[{]
$[0-9]$[0-9][lu]/[Ee]s[Ee][\[{]/[Ss]s[Ss][\$5]/[\$5\[{]
$[0-9]$[0-9]/[Ee]s[Ee][\[{]/[Cc]s[Cc][\(\[]/[\[\({]
$[0-9]$[0-9][lu]/[Ee]s[Ee][\[{]/[Cc]s[Cc][\(\[]/[\[\({]
#
#
#
# lc((ns)word)
# Invalid rules (too long)
;^[0-9]^[0-9]/[lLIi]s[lLIi][!1|]/[Ee]s[Ee][\[{]/[\[{!1|]
#
```

```
^[0-9]/0s0[oO]
^[0-9]/[oO]s[oO]0
^[0-9][ul]/[oO]s[oO]0
^[0-9]/[lLIi]s[lLIi][!1|]/[!1|]
^[0-9][lu]/[lLIi]s[lLIi][!1|]/[!1|]
^[0-9]/[Gg]s[Gg]@/@
^[0-9][ul]/[Gg]s[Gg]@/@
^[0-9]/3s3[\]}]
^[0-9]/[Ee]s[Ee][\[{]/[\[{]
^[0-9][lu]/[Ee]s[Ee][\[{]/[\[{]
^[0-9]/[Ss]s[Ss][\$5]/[\$5]
^[0-9][ul]/[Ss]s[Ss][\$5]/[\$5]
^[0-9]/[Cc]s[Cc][\(\[]/[\[\(]
^[0-9][lu]/[Cc]s[Cc][\(\[]/[\[\(]
^[0-9]/[oO]s[oO]0/[Gg]s[Gg]@/[@0]
^[0-9][ul]/[oO]s[oO]0/[Gg]s[Gg]@/[@0]
^[0-9]/[oO]s[oO]0/[Ee]s[Ee][\[{]/[\[{0]
^[0-9][ul]/[oO]s[oO]0/[Ee]s[Ee][\[{]/[\[{0]
^[0-9]/[oO]s[oO]0/[Ss]s[Ss][\$5]/[\$50]
^[0-9][ul]/[oO]s[oO]0/[Ss]s[Ss][\$5]/[\$50]
^[0-9]/[oO]s[oO]0/[Cc]s[Cc][\(\[]/[\[\(0]
^[0-9][ul]/[oO]s[oO]0/[Cc]s[Cc][\(\[]/[\[\(0]
^[0-9]/[oO]s[oO]0/[lLIi]s[lLIi][!1|]/[!1|0]
^[0-9][ul]/[oO]s[oO]0/[lLIi]s[lLIi][!1|]/[!1|0]
^[0-9]/[Gg]s[Gg]@/[lLIi]s[lLIi][!1|]/[!1|@]
^[0-9]l/[Gg]s[Gg]@/[lLIi]s[lLIi][!1|]/[!1|0@]
^[0-9]u/[Gg]s[Gg]@/[lLIi]s[lLIi][!1|]/[!1|0@]
^[0-9]/[Gg]s[Gg]@/[Ee]s[Ee][\[{]/[\[{@]
^[0-9][ul]/[Gg]s[Gg]@/[Ee]s[Ee][\[{]/[\[{@]
^[0-9]/[Gg]s[Gg]@/[Ss]s[Ss][\$5]/[\$5@]
^[0-9][ul]/[Gg]s[Gg]@/[Ss]s[Ss][\$5]/[\$5@]
^[0-9]/[Gg]s[Gg]@/[Cc]s[Cc][\(\[]/[\[\(@]
^[0-9][ul]/[Gg]s[Gg]@/[Cc]s[Cc][\(\[]/[\[\(@]
^[0-9]/[lLIi]s[lLIi][!1|]/[Ee]s[Ee][\[{]/[\[{!1|]
^[0-9]/[lLIi]s[lLIi][!1|]/[Ss]s[Ss][\$5]/[\$5!1|]
^[0-9]/[lLIi]s[lLIi][!1|]/[Cc]s[Cc][\(\[]/[\[\(!1|]
^[0-9]l/[lLIi]s[lLIi][!1|]/[Ee]s[Ee][\[{]/[\[{!1|]
^[0-9]u/[lLIi]s[lLIi][!1|]/[Ee]s[Ee][\[{]/[\[{!1|]
^[0-9]u/[lLIi]s[lLIi][!1|]/[Ss]s[Ss][\$5]/[\$5!1|]
^[0-9]l/[lLIi]s[lLIi][!1|]/[Ss]s[Ss][\$5]/[\$5!1|]
^[0-9]u/[lLIi]s[lLIi][!1|]/[Cc]s[Cc][\(\[]/[\[\(!1|]
^[0-9]l/[lLIi]s[lLIi][!1|]/[Cc]s[Cc][\(\[]/[\[\(!1|]
^[0-9]/[Ee]s[Ee][\[{]/[Ss]s[Ss][\$5]/[\$5\[{]
^[0-9][lu]/[Ee]s[Ee][\[{]/[Ss]s[Ss][\$5]/[\$5\[{]
^[0-9]/[Ee]s[Ee][\[{]/[Cc]s[Cc][\(\[]/[\[\({]
^[0-9]l/[Ee]s[Ee][\[{]/[Cc]s[Cc][\(\[]/[\[\({]
^[0-9]u/[Ee]s[Ee][\[{]/[Cc]s[Cc][\(\[]/[\[\({]
^[0-9]/[Ss]s[Ss][\$5]/[Cc]s[Cc][\(\[]/[\[\(\$5]
^[0-9]l/[Ss]s[Ss][\$5]/[Cc]s[Cc][\(\[]/[\[\(\$5]
^[0-9]u/[Ss]s[Ss][\$5]/[Cc]s[Cc][\(\[]/[\[\(\$5]
^[0-9]^[0-9]/0s0[oO]
^[0-9]^[0-9]/[oO]s[oO]0
^[0-9]^[0-9][ul]/[oO]s[oO]0
^[0-9]^[0-9]/[lLIi]s[lLIi][!1|]/[!1|]
^[0-9]^[0-9][lu]/[lLIi]s[lLIi][!1|]/[!1|]
^[0-9]^[0-9]/[Gg]s[Gg]@/@
^[0-9]^[0-9][ul]/[Gg]s[Gg]@/@
```

```
^[0-9]^[0-9]/3s3[\]}]
^[0-9]^[0-9]/[Ee]s[Ee][\[{]/[\[{]
^[0-9]^[0-9][lu]/[Ee]s[Ee][\[{]/[\[{]
^[0-9]^[0-9]/[Ss]s[Ss][\$5]/[\$5]
^[0-9]^[0-9][ul]/[Ss]s[Ss][\$5]/[\$5]
^[0-9]^[0-9]/[Cc]s[Cc][\(\[]/[\[\(]
^[0-9]^[0-9][lu]/[Cc]s[Cc][\(\[]/[\[\(]
^[0-9]^[0-9]/[oO]s[oO]0/[Gg]s[Gg]@/[@0]
^[0-9]^[0-9][ul]/[oO]s[oO]0/[Gg]s[Gg]@/[@0]
^[0-9]^[0-9]/[oO]s[oO]0/[Ee]s[Ee][\[{]/[\[{0]
^[0-9]^[0-9]u/[oO]s[oO]0/[Ee]s[Ee][\[{]/[\[{0]
^[0-9]^[0-9]l/[oO]s[oO]0/[Ee]s[Ee][\[{]/[\[{0]
^[0-9]^[0-9]/[oO]s[oO]0/[Ss]s[Ss][\$5]/[\$50]
^[0-9]^[0-9]u/[oO]s[oO]0/[Ss]s[Ss][\$5]/[\$50]
^[0-9]^[0-9]l/[oO]s[oO]0/[Ss]s[Ss][\$5]/[\$50]
^[0-9]^[0-9]/[oO]s[oO]0/[Cc]s[Cc][\(\[]/[\[\(0]
^[0-9]^[0-9][ul]/[oO]s[oO]0/[Cc]s[Cc][\(\[]/[\[\(0]
^[0-9]^[0-9]/[oO]s[oO]0/[lLIi]s[lLIi][!1|]/[!1|0]
^[0-9]^[0-9]u/[oO]s[oO]0/[lLIi]s[lLIi][!1|]/[!1|0]
^[0-9]^[0-9]l/[oO]s[oO]0/[lLIi]s[lLIi][!1|]/[!1|0]
^[0-9]^[0-9]/[Gg]s[Gg]@/[lLIi]s[lLIi][!1|]/[!1|@]
^[0-9]^[0-9]u/[Gg]s[Gg]@/[lLIi]s[lLIi][!1|]/[!1|0@]
^[0-9]^[0-9]l/[Gg]s[Gg]@/[lLIi]s[lLIi][!1|]/[!1|0@]
^[0-9]^[0-9]/[Gg]s[Gg]@/[Ee]s[Ee][\[{]/[\[{@]
^[0-9]^[0-9][ul]/[Gg]s[Gg]@/[Ee]s[Ee][\[{]/[\[{@]
^[0-9]^[0-9]/[Gg]s[Gg]@/[Ss]s[Ss][\$5]/[\$5@]
^[0-9]^[0-9][ul]/[Gg]s[Gg]@/[Ss]s[Ss][\$5]/[\$5@]
^[0-9]^[0-9]/[Gg]s[Gg]@/[Cc]s[Cc][\(\[]/[\[\(@]
^[0-9]^[0-9][ul]/[Gg]s[Gg]@/[Cc]s[Cc][\(\[]/[\[\(@]
^[0-9]^[0-9]/[Ee]s[Ee][\[{]/[Ss]s[Ss][\$5]/[\$5\[{]
^[0-9]^[0-9][lu]/[Ee]s[Ee][\[{]/[Ss]s[Ss][\$5]/[\$5\[{]
^[0-9]^[0-9]/[Ee]s[Ee][\[{]/[Cc]s[Cc][\(\[]/[\[\({]
^[0-9]^[0-9][lu]/[Ee]s[Ee][\[{]/[Cc]s[Cc][\(\[]/[\[\({]
#
#
# wordword
#
d
#
#
# word(gc)word
#
1!?A$i$n$g2
1!?A$e$r2
1/?a$s2
1/?a$'$s2
1/?a$4$u2
1/?a$4$y$o$u2
1)s]$4$u2
1)s]$4$y$o$u2
1/?a$2$b2
1/?a$2$b$e2
#
# wordword(gc)
#
12!?A$i$n$g
12!?A$e$r
```

```
12/?a$s
12/?a$'$s
12/?a$4$u
12/?a$4$y$o$u
12)s]$4$u
12)s]$4$y$o$u
12/?a$2$b
12/?a$2$b$e
#
# word(gc)word(gc)
#
12+!?A$i$n$g
12+!?A$e$r
12+/?a$s2
12+/?a$'$s2
12+/?a$4$u2
12+/?a$4$y$o$u2
12+)s]$4$u2
12+)s]$4$y$o$u2
12+/?a$2$b2
12+/?a$2$b$e2
#
# word(ns)word
#
1^[0-9]2
1^[0-9]^[0-9]2
#
# wordword(ns)
#
12$[0-9]
12$[0-9]$[0-9]
#
# word(gc)(ns)word
#
1!?A$i$n$g$[0-9]2
1!?A$e$r$[0-9]2
1/?a$s$[0-9]2
1/?a$'$s$[0-9]2
1/?a$4$u$[0-9]2
1/?a$4$y$o$u$[0-9]2
1)s]$4$u$[0-9]2
1)s]$4$y$o$u$[0-9]2
1/?a$2$b$[0-9]2
1/?a$2$b$e$[0-9]2
#
# word(ns)word(gc)
#
1$[0-9]2!?A$i$n$g
1$[0-9]2!?A$e$r
1$[0-9]2/?a$s
1$[0-9]2/?a$'$s
1$[0-9]2/?a$4$u
1$[0-9]2/?a$4$y$o$u
1$[0-9]2)s]$4$u
1$[0-9]2)s]$4$y$o$u
1$[0-9]2/?a$2$b
1$[0-9]2/?a$2$b$e
```

```
#
# word(gc)(ns)word(gc)
#
1$[0-9]2$[0-9]+!?X$i$n$g
1$[0-9]2$[0-9]+!?X$e$r
1$[0-9]2$[0-9]+/?a$s2
1$[0-9]2$[0-9]+/?a$'$s2
1$[0-9]2$[0-9]+/?a$4$u2
1$[0-9]2$[0-9]+/?a$4$y$o$u2
1$[0-9]2$[0-9]+)s]$4$u2
1$[0-9]2$[0-9]+)s]$4$y$o$u2
1$[0-9]2$[0-9]+/?a$2$b2
1$[0-9]2$[0-9]+/?a$2$b$e2
#
# lc(word(gc)word)
#
# lc(wordword(gc))
#
# lc(word(gc)word(gc))
#
# lc(word(ns)word)
#
# lc(word(gc)(ns)word)
#
# lc(word(ns)word(gc))
#
# lc(word(gc)(ns)word(gc))
#

# Some pre-defined word filters
[List.External:Filter_Alpha]
void filter()
{
      int i, c;

      i = 0;
      while (c = word[i++])
      if (c < 'a' || c > 'z') {
            word = 0; return;
      }
}

[List.External:Filter_Digits]
void filter()
{
      int i, c

      i = 0;
      while (c = word[i++])
      if (c < '0' || c > '9') {
            word = 0; return;
      }
}

[List.External:Filter_LanMan]
void filter()
{
```

```
	int i, c;

	word[7] = 0;			// Truncate at 7 characters

	i = 0;				// Convert to uppercase
	while (c = word[i]) {
		if (c >= 'a' && c <= 'z') word[i] &= 0xDF;
		i++;
	}
}

# A simple cracker for LM hashes, similar to L0phtCrack
[List.External:LanMan]
int length;			// Current length

void init()
{
	word[0] = 'A' - 1;		// Start with "A"
	word[length = 1] = 0;
}

void generate()
{
	int i;

	i = length - 1;			// Start from the last character
	while (++word[i] > 'Z')		// Try to increase it
	if (i)				// Overflow here, any more
positions?
		word[i--] = 'A';	// Yes, move to the left, and repeat
	else				// No
	if (length < 7) {
		word[i = ++length] = 0; // Switch to the next length
		while (i--)
			word[i] = 'A';
		return;
	} else {
		word = 0; return; // We're done
	}
}

void restore()
{
	length = 0;			// Calculate the length
	while (word[length]) length++;
}

# Useful external mode example
[List.External:Double]
/*
 * This cracking mode tries all the possible duplicated lowercase
alphabetic
 * "words" of up to 8 characters long. Since word halves are the same,
it
 * only has to try about 500,000 words.
 */
```

```
/* Global variables: current length and word */
int length, current[9];

/* Called at startup to initialize the global variables */
void init()
{
      int i;

      i = length = 2;               // Start with 4 character long
words
      while (i--) current[i] = 'a'; // Set our half-word to "aa"
}

/* Generates a new word */
void generate()
{
      int i;

/* Export last generated word, duplicating it at the same time; here
"word"
 * is a pre-defined external variable. */
      word[(i = length) << 1] = 0;
      while (i--) word[length + i] = word[i] = current[i];

/* Generate a new word */
      i = length - 1;               // Start from the last character
      while (++current[i] > 'z')    // Try to increase it
      if (i)                        // Overflow here, any more
positions?
            current[i--] = 'a';     // Yes, move to the left, and
repeat
      else {                        // No
            current = 0;            // Request a length switch
            break;                  // Break out of the loop
      }

/* Switch to the next length, unless we were generating 8 character
long
 * words already. */
      if (!current && length < 4) {
            i = ++length;
            while (i--) current[i] = 'a';
      }
}

/* Called when restoring an interrupted session */
void restore()
{
      int i;

/* Import the word back */
      i = 0;
      while (current[i] = word[i]) i++;

/* ...and calculate the half-word length */
      length = i >> 1;
}
```

```
# Simple parallel processing example
[List.External:Parallel]
/*
 * This word filter makes John process some of the words only, for
running
 * multiple instances on different CPUs. It can be used with any
cracking
 * mode except for "single crack". Note: this is not a good solution,
but
 * is just an example of what can be done with word filters.
 */

int node, total;              // This node's number, and node count
int number;                   // Current word number

void init()
{
      node = 1; total = 2;          // Node 1 of 2, change as
appropriate
      number = node - 1;            // Speedup the filter a bit
}

void filter()
{
      if (number++ % total)         // Word for a different node?
            word = 0;         // Yes, skip it
}
```

Appendix B

References used:

Klein, Daniel V., " "Foiling the cracker": A survey of, and Improvements to, Password Security", Feb 22nd 1991, http://packetstorm.securify.com/papers/password/klein.ps

Thompson, Ken & Morris, Robert, "Password Security: A Case History", http://www.alw.nih.gov/Security/FIRST/papers/password/pwstudy.ps

Feldmeier, David C., Karn, Philip R. "UNIX Password Security – Ten Years Later", http://www.alw.nih.gov/Security/FIRST/papers/password/pwtenyrs.ps

Appendix C

Resources available:

Solar Designer, "John the Ripper", v1.6, http://www.openwall.com/john

Wordlists, ftp://sable.ox.ac.uk/pub/wordlists