"Crack Version 4.1"
A Sensible Password Checker for Unix

Alec D.E. Muffett

Unix Software Engineer
Aberystwyth, Wales, UK
(aem@aber.ac.uk or alec_muffett@hicom.lut.ac.uk)

ABSTRACT

Crack is a freely available program designed to find standard Unix eight-character DES encrypted passwords by standard guessing techniques outlined below. It is written to be flexible, configurable and fast, and to be able to make use of several networked hosts via the Berkeley `rsh` program (or similar), where possible.

1. Statement of Intent

This package is meant as a proving device to aid the construction of secure computer systems. Users of Crack are advised that they may get severly hassled by authoritarian type sysadmin dudes if they run Crack without proper authorisation.

2. Introduction to Version 4.0

Crack is now into it's fourth version, and has been reworked extensively to provide extra functionality, and the purpose of this release is to consolidate as much of this new functionality into as small a package as possible. To this end, Crack may appear to be less configurable: it has been written on the assumption that you run a fairly modern Unix, one with BSD functionality, and then patched in order to run on other systems.

This, surprisingly enough, has led to neater code, and has made possible the introduction of greater flexibility which supercedes many of the options that could be configured in earlier versions of Crack. In the same vein, some of the older options are now mandatory. These, such as *feedback mode* and `CRACK_PRINTOUT` are no longer supported as options and probably never will be again. There is just a lot of wastage in not running with them, and too many dependencies in other functions to bother programming around them.

The user interface is basically identical to the previous versions, although some people have asked about providing X-windows GUI's to Crack, I think it would be a waste of time to do so. Crack has far less options than your ordinary version of `/bin/ls`.

3. Introduction to Version 4.1

Version 4.1 of the Crack program is an attempt to extend the features introduced in v4.0 and provide hooks for external libraries such as Michael Glad's wonderful **UFC** crypt() implementation, which (on some platforms) can outperform my fcrypt() by a factor of 3. I have also been burdened with the task of making Crack's memory handling bombproof (hah!) in the vague hope that it will survive running out of memory on small machines.[1]

The extensions that I mention above regard the addition of extra primitives to the dictionary processing language which permit the production of more concise dictionaries containing words, more of which are likely

---

[1] - or even on large ones. Brian Tompsett at Hull tweaked Crack v3.3 until it could run to completion after filling the swapspace on each of a network of SparcStation2's. Due to restructuring work on v4.0, I have had to write my own sorting algorithm & re-implement all of his tweaks from scratch, and can only hope that I have emulated the bombproofness of this desirable (?) functionality.

to be passwords. The idea is to gain efficiency by removing some of the dross from the generated dictionaries.

Crack should (generally) be more disk-space efficient now that the program can spot dictionaries which have been compressed using *compress* or *pack* and will uncompress them on the fly as necessary (using *zcat* or *pcat* respectively).[2]

4. Crack Methodology - Part 1: Internals

Crack takes as its input a series of password files and source dictionaries. It merges the dictionaries, turns the password files into a sorted list, and generates lists of possible passwords from the merged dictionary or from information gleaned about users from the password file. It does **not** attempt to remedy the problem of allowing users to have guessable passwords, and it should **NOT** be used in place of getting a really good, secure `passwd` program replacement.[3]

The above paragraphs define the purpose of Crack, and embody a great deal of hard work, screams of *Eureka!*, drunkeness, and a fair amount of swearing too. There is a lot of thinking, philosophy, and empirical guesswork behind the way that Crack attacks password files, and although it is not perfect, I certainly hope that Crack will out-do most of it's competitors.

Crack works by making many individual passes over the password entries that you supply to it. Each pass generates password guesses based upon a sequence of rules, supplied to the program by the user. The rules are specified in a simplistic language in the files `gecos.rules` and `dicts.rules`, to be found in the `Scripts` directory. The distinction between these two files will be made clear later.

The rules are written as a simple string of characters, with one rule to a line. Blank lines, and comment lines beginning with a hash character **#** are ignored. Trailing whitespace is also ignored. The instructions in the rule are followed from left to right, and are applied to the dictionary words one by one, as the words are loaded. Some simple pattern matching primitives are provided for selection purposes, so that if the dictionary word does not match the pattern, it is ignored. This saves on time and memory. Before carrying on, I suggest that you browse through `Scripts/dicts.rules`, take a look at the rules supplied as defaults, and try to work out what they do.

The rules are stored in two different files for two different purposes. Rules in `Scripts/gecos.rules` are applied to data generated by Crack from the pw_gecos and pw_gecos entries of the user's password entry. The data fed to the gecos rules for the user *aem*, who is *Alec David Muffett, Systems* would be: *aem*, *Alec*, *David*, *Muffett*, *Systems*, and a series of permutations of those words, either re-ordering the words and joining them together (eg: *AlecMuffett*), or making up new words based on initial letters of one word taken with the rest of another (eg: *AMuffett*).[4]

The entire set of rules in gecos.rules is applied to each of these words, which creates many more permutations and combinations, all of which are tested. Hence testing the password gecos information under Crack v4.0 and upwards takes somewhat longer than previously, but it is far more thorough.

After a pass has been made over the data based on gecos information, Crack makes further passes over the password data using successive rules from the `Scripts/dicts.rules` by loading the whole of `Dicts/bigdict` file into memory, with the rule being applied to each word from that file. This generates a *resident dictionary*, which is sorted and uniqued so as to prevent wasting time on repetition. After each pass is completed, the memory used by the resident dictionary is freed up, and (hopefully) re-used when the next dictionary is loaded.

The `Dicts/bigdict` dictionary is created by Crack by merging, sorting, and uniq'ing the source dictionaries, which are to be found in the directory `DictSrc` and which may also be named in the Crack shellscript, via the `$STDDICT` variable. (The default value of $STDDICT is `/usr/dict/words`).

[2] Note to people who are short on memory or swap: do remember that to do this Crack will have to *fork()* (via *popen()*) and might not be able to create the uncompressing process. Hence, if you intend to swaplock your machine, don't compress the dictionaries. Switch this off by editing the `Crack` shellscript.

[3] See the end of ths document for more information about *passwd* replacements.

[4] - and *ASystems* and *DSystems*, and *MSystems*, etc... because Crack does not differentiate. Hence, care should be taken to check for redundancy when adding new rules, so as not to waste time during the gecos pass.

The file DictSrc/bad_pws.dat is a dictionary which is meant to provide many of those common but non-dictionary passwords, such as *12345678* or *qwerty*.

If you wish to provide a dictionary of your own, just copy it into the DictSrc directory (use compress on it if you wish to save space; Crack will unpack it whilst generating the big dictionary) and then delete the contents of the Dicts directory by running Scripts/spotless. Your new dictionary will be merged in on the next run. For more information on dictionary attacks, see the *excellent* paper called "Foiling the Cracker: A Survey of, and Improvements to, Password Security" by Daniel Klein, available from *ftp.sei.cmu.edu* in *~/pub/dvk/passwd.**. Also, please read the APPENDIX file supplied with this distribution.[5]

Having described the method of cracking, perhaps we should now investigate the algorithm used to overlay the cracking mechanism.

5. Crack Methodology - Part 2: Feedback Filters

As is stated above, Crack permutes and loads dictionaries directly into memory, sorts and uniques them, before attempting to use each of the words as a guess for each users' password. If Crack correctly guesses a password, it marks the user as *done* and does not waste further time on trying to break that users password.

Once Crack has finished a dictionary pass, it sweeps the list of users looking for the passwords it has cracked. It stores the cracked passwords in both plaintext and encrypted forms in a *feedback file* in the directory Runtime. Feedback files have names of the form Runtime/F*.

The purpose of this is so that, when Crack is next invoked, it may recognise passwords that it has successfully cracked before, and filter them from the input to the password cracker. This provides an *instant* list of crackable users who have not changed their passwords since the last time Crack was run. This list appears in a file with name out* in the $CRACK_OUT directory, or on *stdout*, if foreground mode is invoked (see *Options*, below).

In a similar vein, when a Crack run terminates normally, it writes out to the feedback file all encrypted passwords that it has **NOT** succeeded in cracking. Crack will then ignore all of these passwords next time you run it.

Obviously, this is not desirable if you frequently change your dictionaries or rules, and so there is a script provided, Scripts/mrgfbk which sorts your feedback files, merges them into one, and optionally removes all traces of 'uncrackable' passwords, so that your next Crack run can have a go at passwords it has not succeeded in breaking before.

Mrgfbk is invoked automatically if you run Scripts/spotless.

6. Crack Methodology - Part 3: Execution and Networking

Each time Crack is invoked, whether networked or not, it generates a *diefile* with a name of the form Runtime/D* (for network cracks, this file is generated by RCrack, and is of the form Runtime/DR* which points to a **real** diefile, named Runtime/RD* - see below for details).

These diefiles contain debugging information about the job, and are generated so that all the jobs on the entire network can be called quickly by invoking Scripts/plaster. Diefiles delete themselves after they have been run.

As you will read in the sections below, Crack has a -network option: This is designed to be a simple method of automatically spreading the load of password cracking out over several machines on a network, preferably if they are connected by some form of networked filestore.

When Crack -network is invoked, it filters its input in the ordinary way, and then splits its load up amongst several machines which are specified in the file Scripts/network.conf.

This file contains a series of hostnames, power ratings, flags, etc, relevant to the running of Crack on each machine. Crack then calls Scripts/RCrack to use the rsh command (or similar) to invoke Crack on

[5] Extra dictionaries (those detailed in Dan Klein's paper) can be obtained via anonymous FTP from *ftp.uu.net* (137.39.1.9) as *~/pub/dictionaries.tar.Z*; or check an *Archie* database for other possible sources of dictionaries.

the other hosts. See the RCrack script, and the example network.conf file for details.

7. Installation

Crack is one of those most unusual of beasties, a self-installing program. Some people have complained about this apparent weirdness, but it has grown up with Crack ever since the earliest network version, when I could not be bothered to log into several different machines with several different architectures, just in order to build the binaries. Once the necessary configuration options have been set, the executables are created via `make` by running the Crack shellscript .

Crack's configuration lies in two files, the `Crack` shell script, which contains all the installation specific configuration data, and the file `Sources/conf.h`, which contains configuration options specific to various binary platforms.

In the Crack shellscript, you will have to edit the `CRACK_HOME` variable to the correct value. This variable should be set to an absolute path name (names relative to *~username* are OK, so long as you have some sort of `csh`) through which the directory containing Crack may be accessed on **ALL** the machines that Crack will be run on. There is a similar variable `CRACK_OUT` which specifies where Crack should put its output files - by default, this is the same as `$CRACK_HOME`.

You will also have to edit the file `Sources/conf.h` and work out which switches to enable. Each `#define` has a small note explaining its purpose. Where I have been in doubt about the portability of certain library functions, usually I have re-written it, so you should be OK. Let me know of your problems, if you have any.

If you will be using `Crack -network` you will then have to generate a `Scripts/network.conf` file. This contains a list of hostnames to `rsh` to, what their *binary type* is (useful when running a network Crack on several different architectures), a guesstimate of their *relative power* (take your slowest machine as unary, and measure all others relative to it), and a list of per-host *flags* to **add** to those specified on the `Crack` command line, when calling that host. There is an example of such a file provided in the Scripts directory - take a look at it.

If ever you wish to specify a more precise figure as to the relative power of your machines, or you are simply at a loss, play with the command `make tests` in the source code directory. This can provide you with the number of fcrypt()s that your machine can do per second, which is a number that you can plug into your `network.conf` as a measure of your machines' power (after rounding the value to an integer).

8. Usage

Okay, so, let's assume that you have edited your `Crack` script, and your `Sources/conf.h` file, where do you go from here ?

`Crack` [*options*] [*bindir*] `/etc/passwd` [...other passwd files]

`Crack -network` [*options*] `/etc/passwd` [...other passwd files]

Where **bindir** is the optional name of the directory where you want the binaries installed. This is useful where you want to be able to run versions of Crack on several different architectures. If **bindir** does not exist, a warning will be issued, and the directory created.

Note: **bindir** defaults to the name `generic` if not supplied.

Notes for Yellow Pages (NIS) Users: I have occasional queries about how to get Crack running from a YP password file. There are several methods, but by far the simplest is to generate a passwd format file by running:-

```
ypcat passwd > passwd.yp
```

and then running Crack on this file.

9. Options

-f Runs Crack in *foreground* mode, ie: the password cracker is not backgrounded, and messages appear on stdout and stderr as you would expect. This option is only really useful for very small password files, or when you want to put a wrapper script around Crack.

Foreground mode is disabled if you try running `Crack -network -f` on the command line, because of the insensibility of `rsh`ing to several machines in turn, waiting for each one to finish before calling the next. However, please read the section about *Network Cracking without NFS/RFS*, below.

-v Sets verbose mode, whereby Crack will print every guess it is trying on a per-user basis. This is a very quick way of flooding your filestore, but useful if you think something is going wrong.

-m Sends mail to any user whose password you crack by invoking `Scripts/nastygram` with their username as an argument. The reason for using the script is so that a degree of flexibility in the format of the mail message is supplied; ie: you don't have to recompile code in order to change the message.[6]

-nvalue

Sets the process to be `nice()`ed to *value*, so, for example, the switch `-n19` sets the Crack process to run at the lowest priority.

-network

Throws Crack into network mode, in which it reads the `Scripts/network.conf` file, splits its input into chunks which are sized according to the power of the target machine, and calls `rsh` to run Crack on that machine. Options for Crack running on the target machine may be supplied on the command line (eg: verbose or recover mode), or in the network.conf file if they pertain to specific hosts (eg: `nice()` values).

-r<pointfile>

This is only for use when running in *recover* mode. When a running Crack starts pass 2, it periodically saves its state in a *pointfile*, with a name of the form `Runtime/P.*` This file can be used to recover where you were should a host crash. Simply invoke Crack in **exactly** the same manner as the last time, with the addition of the `-r` switch, (eg: `-rRuntime/Pfred12345`) switch. Crack will startup and read the file, and jump to roughly where it left off. If you are cracking a very large password file, this can save oodles of time after a crash.

If you were running a *network* Crack, then the jobs will again be spawned onto all the machines of the original Crack. The program will then check that the host it is running on is the same as is mentioned in the pointfile. If it is not, it will quietly die. Thus, assuming that you supply the same input data and do not change your `network.conf` file, Crack should pick up where it left off. This is a bit inelegant, but it's better than nothing at the moment.

The method of error recovery outlined above causes headaches for users who want to do multiprocessing on parallel architectures. Crack is in no way parallel, and because of the way it's structured (reading stdin from shellscript frontends) it is a pain to divide the work amongst several processes via `fork()`ing.

The hack solution to get several copies of Crack running on one machine with *n* processors at the moment is to insert *n* copies of the entry for your parallel machine into the `Scripts/network.conf` file. If you use the `-r` option in these circumstances however, you will get *n* copies of the recovered process running, only one of them will have the correct input data.

The old solution to this problem (see old documentation if you are interested) has been negated by the introduction of feedback mode, so the best bet in this particular situation is to wait until the other jobs are done (and have written out lists of uncrackable passwords), and then re-start the jobs from

[6] I'm uncertain about the wisdom of mailing someone like this. If someone browses your cracked user's mail somehow, it's like a great big neon sign pointing at the user saying "This Is A Crackable Account - Go For It!". Not to mention the false sense of security it engenders in the System Manager that he's "informed" the user to change his password. What if the user doesn't log on for 3 months? However, so many people have wired it into their own versions of Crack, I suppose it **must** be provided... AEM

scratch. Anyone whose password was not cracked on the first run will be ignored on the second, if they have not changed it since. This is inelegant, but it's the best I can do in the limited time available.

10. Support Scripts

The `Scripts` directory contains a small number of support and utility scripts, some of which are designed to help Crack users check their progress. Briefly, the most useful ones are:-

Scripts/shadmrg

This is a small (but hopefully readable) script for merging `/etc/passwd` and `/etc/shadow` on System V style shadow password systems. It produces the merged data to stdout, and will need redirecting into a file before Crack can work on it. The script is meant to be fairly lucid, on the grounds that I worry that there are many shadowing schemes out there, and perhaps not all have the same data format.

I have not wired this facility into the Crack command itself because the world does **NOT** revolve around System V yet, regardless of what some people would have me believe, and I believe that the lack of direct support for NIS outlined above, sets a precedent. There are just too many incompatibilities in shadow password schemes for me to hardwire anything.

Scripts/plaster

which is named after a dumb joke, but is a simple frontend to the `Runtime/D*` diefiles that each copy of the password cracker generates. Invoking `Scripts/plaster` will kill off all copies of the password cracker you are running, over the network or otherwise.

Scripts/status

This script `rsh`es to each machine mentioned in the `Scripts/network.conf` file, and provides some information about processes and uptime on that machine. This is useful when you want to find out just how well your password crackers are getting on during a `Crack -network`.

Scripts/{clean,spotless}

These are really just frontends to a makefile. Invoking `Scripts/clean` tidies up the Crack home directory, and removes probably unwanted files, but leaves the pre-processed dictionary `bigdict` intact. `Scripts/spotless` does the same as `Scripts/clean` but obliterates `bigdict` and old output files too, and compresses the feedback files into one.

Scripts/nastygram

This is the shellscript that is invoked by the password cracker to send mail to users who have guessable passwords, if the `-m` option is used. Edit it at your leisure to suit your system.

Scripts/guess2fbk

This script takes your `out*` files as arguments and reformats the 'Guessed' lines into a slightly messy *feedback* file, suitable for storing with the others.

An occasion where this might be useful is when your cracker has guessed many peoples passwords, and then died for some reason (a crash?) before writing out the guesses to a feedback file. Running

```
Scripts/guess2fbk out* >> Runtime/F.new
```

will save the work that has been done.

11. Network Cracking without NFS/RFS

For those users who have some form of `rsh` command, but do not have a a networked filestore running between hosts, there is now a solution which will allow you to do networked cracking, proposed to me by Brian Tompsett at Hull. Personally, I consider the idea to be potty, but it fills in missing functionality in a wonderfully tacky manner.

From the documentation above, you will note that Crack will undo the `-f` *(output in foreground)* option, if it is invoked with the `-network` switch at the same time (see the *Options* section above). This is true, but it does not apply if you specify `-f` option in the `network.conf` file.

The practical upshot of doing this is that remote copies of Crack can be made to read from *stdin* and write to *stdout* over a network link, and thus remote processing is accomplished. I have tweaked Crack in such a way, therefore, that if the `-f` option is specified amongst the crack-flags of a host in the network.conf, rather than backgrounding itself on the remote host, the `rsh` command on the **server** is backgrounded, and output is written directly to the files on the server's filestore.

There are restrictions upon this method, mostly involving the number of processes that a user may run on the server at any one time, and that you will have to collect feedback output together manually (dropping it into the `Runtime` directory on the server). However, it works. Also, if you try to use `rsh` as another user, you will suffer problems if `rsh` insists on reading something from your terminal (eg: a password for the remote account). Also, recovering using checkpointing goes out the window unless you specify the name of the pointfile as it is named on the remote machine.

12. UFC Support and notes on fast crypt() implementations

The stdlib version of the `crypt()` subroutine is incredibly slow. It is a *massive* bottleneck to the execution of Crack and on typical platforms that you get at universities, it is rare to find a machine which will achieve more than 50 standard crypt() s per second. On low-end diskless workstations, you may expect 2 or 3 per second. It was this slowness of the crypt() algorithm which originally supplied much of the security Unix needed.[7]

There are now **many** implementations of faster versions of crypt() to be found on the network. The one supplied with Crack v3.2 and upwards is called `fcrypt().` It was originally written in May 1986 by Robert Baldwin at MIT, and is a good version of the crypt() subroutine. I received a copy from Icarus Sparry at Bath University, who had made a couple of portability enhancements to the code.

I rewrote most of the tables and the KeySchedule generating algorithm in the original *fdes-init.c* to knock 40% off the execution overhead of fcrypt() in the form that it was shipped to me. I inlined a bunch of stuff, put it into a single file, got some advice from Matt Bishop and Bob Baldwin [both of whom I am greatly indebted to] about what to do to the `xform()` routine and to the fcrypt function itself, and tidied up some algorithms. I have also added more lookup tables and reduced several formula for faster use. Fcrypt() is now barely recognisable as being based on its former incarnation, and it is 3x faster.

On a DecStation 5000/200, fcrypt() is about 16 times faster than the standard crypt (your mileage may vary with other architectures and compilers). This speed puts fcrypt() into the "moderately fast" league of crypt implementations.

Amongst other crypt implementations available is **UFC** by Michael Glad. UFC-crypt is a version of the crypt subroutine which is optimised for machines with 32-bit long integers and generally outperforms my fcrypt() by a factor of between 1 and 3, for a tradeoff of large memory usage, and memory-cache unfriendliness. Hooks for even more optimised assembler versions of crypt() are also provided for some platforms (Sun, HP, ...). Getting UFC to work on 16 bit architectures is nearly impossible.

However, on most architectures, UFC generates a stunning increase in the power of Crack, and so, from v4.1 onwards, Crack is written to automatically make use of UFC if it can find it. All that you have to do is to obtain a suitable copy of UFC (preferably a version which mentions that it is compatible with `Crack v4.1`, and unpack it into a directory called `ufc-crypt` in `$CRACK_HOME`, and then delete your old binaries. UFC will then be detected, compiled, tested and used in preference to fcrypt() by the Crack program, wherever possible.

13. Conclusions

What can be done about brute force attacks on your password file ?

You must get a drop-in replacement for the `passwd` and `yppasswd` commands; one which will stop people from choosing bad passwords in the first place. There are several programs to do this; Matt Bishop's `passwd+` and Clyde Hoover's `npasswd` program are good examples which are freely available. Consult an **Archie** database for more details on where you can get them from.

[7] See: "Password Security, A Case History" by Bob Morris & Ken Thomson, in the Unix Programmer Docs.

It would be nice if an organisation (such as **CERT**?) could be persuaded to supply skeletons of *sensible* passwd commands for the public good, as well as an archive of security related utilities[8] on top of the excellent `COPS`. However, for Unix security to improve on a global scale, we will also require pressure on the vendors, so that programs are written correctly from the beginning.

[8] `COPS` is available for anonymous FTP from *cert.sei.cmu.edu* (128.237.253.5) in *~/cops*