Pass or Fail: A New Test for Password Legitimacy*

Andrew Cherry, Mark W. Henderson, William K. Nickless,
Robert Olson, and Gene Rackow†‡

September 25, 1992

Abstract

While other programs check for bad passwords after the fact, it is important to have good passwords at all times, not just after the latest `Crack` run. To this end we have modified Larry Wall's Perl password program and added, among other features, the ability to check a sorted list of all the "bad passwords" that `Crack` will generate, given all the dictionaries that we could get our hands on (107 MB of unique words, so far). The combination of improvements has turned publicly available code into a powerful tool that can aid sites in the maintenance of local security.

1 The Problem

Our motivation for developing a new test for passwords was based on the following assumptions:

- There is some cracker out there on The Net who has our password file and nothing better to do.
- Every so often we find out that someone is trying to break into our systems.
- Having weak passwords on a system is an invitation for trouble, especially when connected to the Internet.
- Stock `passwd` as it exists out of the box has many problems.

Two of the more persistent issues that we faced—and continue to face—were security and technology advances.

*This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

†Authors can be reached via e-mail with "LastName@mcs.anl.gov" addresses.

‡We express our sincere appreciation to Gail Pieper for help in the preparation of this document.

1.1 Security Disruptions

Security disruptions cannot be ignored. They place a large drain on the systems administrators, who must drop everything and make sure that things are sane. Such disruptions thus cost time and computational resources, put at grave risk work that hasn't yet reached backup media, and play havoc with a systems group's morale. And of course, these disruptions tend to happen at the worst possible time.

1.2 Advances in Cracker Technology

Programs like `Crack` are available from many sites. Since `Crack` has been posted to The Net, some crackers have found ways of improving on the algorithms and methods used. We have done some of that for our own use. The current password algorithm was created when speeds of hardware got to the point of doing approximately 800 encryptions a second using the old technology. This was when a DEC PDP11/70 was considered a fast machine [2] and before networking and parallel computers. Currently a Sun SparcStation-2 can do approximately 1200 encryptions a second; a Sequent Symmetery can do approximately 18,000 encryptions per second; and machines like the Intel DELTA and Thinking Machines CM-200 have capabilities in excess of half a million encryptions per second. (Of course, most crackers do not have access to machines like the DELTA or TMC, but they can get access to large networks of workstations that perform just as well.)

One of our assumptions was that, at the minimum, we would have to protect ourselves against a complete run of `Crack`. `Crack` is designed to check the most likely passwords first rather than simply going through a list. By precomputing the list, we were able to cut down the memory requirements of each `Crack` process dramatically.

We took the set of all the dictionaries we could find and applied all the `Crack` rules to each word in this set of dictionaries. This list was sorted to remove duplicates, as well as to facilitate a binary search when searching for the plain text of a candidate password.

A useful side-effect of this precomputed dictionary was that the initial password check could be made much more efficient. The current version of `Crack` would have taken over 8 megabytes of resident memory per process because the dictionary is distributed to each process checking passwords. Our version only required 240 kilobytes of resident memory per process.

1.3 Cracking Users Isn't Nice

The problem with cracking one's own user base is that it becomes necessary to disable accounts with weak passwords. This tends to be rather traumatic to the users, especially if the password that they were using had previously been

considered safe (most likely, however, finding one's account disabled is less traumatic than trying to remember all the work one has done since the last backup). As many reported attacks come from foreign countries, the set of words they may consider standard probably are different from "/usr/dict/words." So, what may look random here in the United States may not be so random elsewhere.

1.4 The Intent

The idea is to reduce the amount of time that a bad password is on the system. The ideal is, of course, zero; and that is what we achieved by designing a technique that checks the passwords *before* they are entered into the actual password file. This means that even if someone has the password file (a situation that experience has shown to be a reasonable assumption), he can't `Crack` any accounts.

1.5 Computing Environment Issues

The Advanced Computing Research Facility (ACRF) at Argonne comprises various Unix-based machines. These machines are administered as two distinct groups: workstations and research machines. Currently, almost all of the machines within the ACRF run YP/NIS. Some people have accounts via YP/NIS on the research machines, and some have accounts directly, and/or only, on the research machines.

Any replacement program has to function in a YP/NIS mode as well as directly on the local /etc/passwd file. Given the confusion of just where their account comes from, it would be nice if the password program would follow through with the intent of the users and change their (yp)password for them. People who change their password should be rewarded, not hassled.

It would also be nice if a password that is acceptable on one of our machines is acceptable on all of our machines. Vendors have varying checks built into their `passwd` programs. A constant user interface is most desirable.

1.6 Our Philosophy

The philosophy of the Systems Support Group at Argonne, as well as the culture of our research division, has shaped our decisions in putting this program together. We try to be as accommodating as possible; but we realize that if we are to get anything done, we have to set down some guidelines. Some of the issues that we decided on are as follows.

1.6.1 Other People Have More Time

Ideally, password security should be something that just happens in the background, much the same way that backups should. This viewpoint is based on

our experience that automating things is much preferred over dealing with the same old problem repeatedly. Like most sites, we don't *have* time for password security: we have to *make* time for it. So, if we don't have to make time to deal with the almost inevitable results of a weak password, we can do other things.

1.6.2 Password Aging

One of features that we disabled from the stock perl password program was the password aging code. This is a religious issue. While we don't agree with the school of thought that says passwords should be aged and expired, we plan to implement something in the near future to notify a user that he has had his current password for over some as-yet-undetermined time. This will serve as a gentle reminder for the user who wishes to change their password.

1.7 Forcing a Password Change

Forcing a user to change his password isn't friendly. What is even worse is when a user tries to change a password, discovers that the new password is weak, and has to come up with a different password before the program will let him exit. (A user can't very well leave a window lying around with an "Enter new password" prompt!) Coupled with password aging, this situation commonly results in having a really unnatural password thought up on the fly.

We have taken a different approach. If a new password isn't "good enough" (read "strange enough"), the user can try later, after he's had less sleep, more coffee, or whatever. The program doesn't trap and force the user to make up a password on the spot. This means that if a user wants to see whether a possible new password will work, he can try it out without worrying about the program forcing him to change his password in the event that his candidate was weak.

2 Our Solution

What we did was combine several available tools and programs, throw in some of our own code, and forge a tool that has lots of promise.

This program checks only new passwords; it does not do anything for the passwords currently on a computing system.

To put such a system in place imposes a few requirements. Foremost is that no bad passwords can be on the system. This requirement is easy to meet if one is starting with a new system, but it's a little harder when 3,500 accounts already exist in a dozen different password files. We had two options: force everyone to change passwords with this as-yet-unwritten program, or `Crack` all of our accounts. Given the fact that we were under yet another `Crack` attack, we `Cracked` ourselves—with full rules. Our collection of mutated dictionaries totals some 13,160,566 "words." It took eleven days to complete the `Crack` runs. The processes were `niced` so as to have little effect on normal daily activities.

2.1 Added Features to Perl

We started with the password program that Larry Wall has in the Perl [1] nutshell book, and added some needed features to it. (Perl has the advantage that it runs on all our machines.)

2.1.1 YP/NIS Smarts

Since we have an existing YP/NIS setup, we had to work within its constraints. We made the program smart enough to do the right thing—regardless of whether it was called as `yppasswd` or just `passwd`. This little feature alone should go a long way to helping people figure out how things are set up. Again, our approach was to reward the users' intent, not harass them with details that they don't need to know.

The actual interactions with the YP/NIS server occur with the aid of an auxiliary C program `ypstuff` invoked by the password program. The `ypstuff` program takes as an argument the name of the YP/NIS server and reads from standard input the current password of the account for which the information is to be changed, and the password file information to be sent to the YP/NIS server. It then connects to the specified YP/NIS server, sets up the RPC arguments, and invokes `callrpc` to perform the password change.

We contemplated integrating the RPC mechanism into Perl itself through the user-defined subroutine mechanism provided by Perl. This was judged to be too much of a hassle, however, as it requires a custom copy of Perl for each architecture. It is easier to maintain the small `ypstuff` program for each architecture.

We also contemplated implementing remote procedure call in Perl, using the basic networking calls that Perl provides. However, since the internals of `callrpc` are less than obvious, it was not worth the time.

2.1.2 Gecos Changes

We needed some way to allow users to change their gecos information without getting stuck with a case of "silly names." We went for the following compromise: when a user changes his gecos information, the old and the new information is sent off to our "accounts" e-mail alias. This is looked at by a live human to determine whether the new gecos is missing any critical information. (At some point in the future, this is going to get logged to a file.) This information is important for identification and contact in the case of run-away jobs. Of course, there are checks so that people can't put ":" or a <CR> in their gecos entry, thereby protecting the actual password file while not imposing a rigid format.

2.1.3 Shell Changes

The program is now also capable of changing a user's shell. Choices are limited to the set of supported shells. This way, a user can't inadvertently lock himself off of a machine.

2.1.4 Clear Text Check

In addition to the various regular expression checks that came with the original perl password program, we've added a binary search through our 13,160,566 "words." This determines whether the clear text word is crackable, without getting bogged down in all the encryption calculations. This involves a nontrivial amount of disk space, and there are some less-than-ideal implications that we'll address later.

2.2 Security Holes

Of course, it is possible to find something wrong with anything, and this approach is certainly no exception.

2.2.1 Packet Watching

While the new clear text password doesn't actually travel the network, someone with promiscuous access to the network (which requires root access) could watch the blocks of the dictionary coming across and determine, within a relatively small range, what the new password is. This means that if there is an unguarded PC on the network, that network is wide open. If crackers have that sort of access to a physical network, they could collect more passwords by watching `telnet` and `ftp` packets. Also, if either `telnet` or `rlogin` is being used to access another machine across a network, those key strokes are open for inspection if there is a device on the wire in promiscuous mode. The basic message is that if the physical transport media is accessible, nothing—short of packet end to end encryption—will prevent someone from watching everything on that physical transport.

2.2.2 "I'll Do It Myself..."

It is still possible, for some determined/demented soul, to put a bad YP/NIS password on the system. Any program that will talk to a `ypserv` daemon can be used to change a password. For example, NeXT computers come with a GUI application for changing one's password. These aren't run through the various checks for good passwords: we didn't write them! To avoid the problems of a weak password, we will begin logging the encrypted changes to a file and then check to make sure that's what is in the password file. If there is a difference, it's time to have a very serious talk with that user.

2.2.3 Full Attacks

It is of course also still imaginable for someone to do a "full" attack on password information. Rough calculations point out that it will still take approximately 2,000 machines the power of the DELTA (42.2 Gflops, 17 bips peak) a full year to break a password, barring of course a lucky strike. This amount of activity would most likely not go unnoticed.

3 Wouldn't It Be Neat If...

We are contemplating adding a feature that was brought out in Matt Bishop's paper on a similar program, `passwd+`. [3] The feature is the distribution of the authority over a set of accounts. We have several schools using our machines for parallel programming classes, and it might make sense to enable the instructor to change student accounts. At this point, however, we haven't decided whether we are going to take the trivial way out by limiting this to our paradigm for classes, or whether we will go with Matt's more general solution.

We started this work prior to Matt's program being available. Matt's work is good, but we needed to solve this problem and could not wait. (`Passwd+` is in beta test at this writing.) By using Perl, we gain a superset of regular expressions that Matt has programmed into `passwd+`.

Another program of interest is the `npasswd` [4] program by Clyde Hoover. It too has similar features to our program.

A problem that we see with both `passwd+` and `npasswd` is that they are written in C and therefore not as portable or easily extensible as a Perl version.

In any case, we recommend that individuals interested in this a priori password-checking approach read these papers.

4 Coming Soon to an Anonymous FTP Site Near You!

We've been running this code since July 20th and it's worked fine. To help other people get started, we'll be putting the following up for anonymous ftp in the "pub/systems/passwd" directory on "info.mcs.anl.gov":

- copies of the above mentioned papers,
- locations of ftp-able dictionaries,
- shell script for generating the large dictionary,
- the C source that communicates with `ypserv`, and
- our modified Perl password program.

References

[1] Larry Wall and Randal L. Schwartz, *Programming Perl*, O'Reilly & Associates, Inc., 1990

[2] Daniel Klein, "A Survey of, and Improvements to, Password Security," *Proceedings of the USENIX Second Security Workshop, Portland*, Oregon, 1990, pp. 5–14

[3] Matt Bishop, "Anatomy of a Proactive Password Changer," *Proceedings of USENIX UNIX Security III*, Baltimore, Maryland, 1992

[4] Clyde Hoover, `npasswd`, Posted in comp.sources.unix, Vol. 25, issues 74–76, 1991