

Choosing correct passwords: a reminder*

David Mentré

2006-08-03 (v3)

This short document presents some rules to choose a new password as secure as possible. It is difficult to give some always-true rules so this document rather tries to give some general information about potential issues, in order to help one makes his/her own decision.

1 Strength of a password

Firstly, the strength of a password can only be measured against a given *attack scenario*, i.e. a description of how an attacker might guess the password, with which tools, etc. Trying to guess a password over a wide area network is quite different from having the `/etc/passwd` file of a Unix system and applying a password cracker program on it.

Secondly, it is important to know the system on which the password will be used: is the length of a password limited (e.g. 8 characters on old Unix systems), is salting¹ and stretching² used, etc. This helps avoid making wrong assumption about system behaviour.

Thirdly, power of current computer systems renders most passwords that a human can memorize sensitive to brute force attack (i.e. test of eah possible password in turn). So, if one needs very strong security, a password should be combined with another security token (smartcard, public cryptographic key, random bits stored on a hardware device, ...).

2 Some rules to choose a good password

They are three major ways to find a good password:

1. **Use a long sentence:** find a rather long sentence (at least six words of more that four characters), if possible not taken from a book and very specific to you, like "There is three years between my son Adam and his sister Beatrice". For added security, you can add numbers or punctuation characters, use your native language, change characters to upper or lowercase, etc., like in "Mon fils AdaM a 3 ans de + que sa soeur BéatricE.". However, be careful to use simple transformation rules otherwise you won't remember your password!

*This document is under GNU GPL license, see <http://www.gnu.org/copyleft/gpl.html>

¹Salting: use some public but random bits with the password to impact dictionary attacks.

²Stretching: repetitive use of a cryptographic hash function to make more lengthy the computation of a password.

2. **Use a program to generate a random password and memorize it.** It is quite important to check that the program is using a good source of random (i.e. a strong random source, like `/dev/random` on Unix systems). An example of such a program is given in Appendix A;
3. **Use random to generate a passphrase and memorize it.** The Diceware³ system proposes such an approach. By using a dice, one selects several words in a dictionary and this list of words should be memorized.

3 Additional points

Some points that you should consider:

- **How strong is a password?** The strength of a password is measured by the number of bits of information entropy⁴ (i.e. randomness) it contains. While it is easy to compute it for a program generating random password, it is much more difficult to estimate it for a long sentence. However, one should know that words taken from the dictionary or sentences having a real meaning are not that strong⁵. Moreover, usual transformations applied on sentences (e.g. like substituting a “o” with “O” or “0” or using a foreign language) is known and used by password crackers;
- **How frequently change a password?** Once again, answer to such a question is related to security policy and a given attack scenario. If a password can be tested by an attacker repetitively on his local machine, one should consider changing password every few months (one to three months). If the attacker has no access to the password file (e.g. through the Internet), the password can be used for a longer time;
- **Should I write down passwords?** Once again, it depends on your setting and the importance of not losing data. The general rule is to *never ever write down passwords!* That’s said, if losing your password means losing your precious data, it is better to write down the password and store it in a secure place (from your wallet to a safe box). If you write down a password, do not store it in the same place as your data.

4 Further readings

A subjective list of articles and web pages to read:

- The Great Debates: Pass Phrases vs. Passwords, Jesper M. Johansson, Security Program Manager, Microsoft Corporation
<http://www.microsoft.com/technet/security/secnews/articles/itproviewpoint091004.msp>

³<http://world.std.com/~reinhold/diceware.html>

⁴http://en.wikipedia.org/wiki/Information_entropy

⁵C.E. Shannon, calculated the entropy per letter of an 8-letter chunk of English as 2.3 bits per letter (Shannon, C.E., Predication and Entropy in Printed English, Bell System Technical Journal, v. 30, n. 1, 1951, pp. 50-64), as given in <http://www.microsoft.com/technet/security/secnews/articles/itproviewpoint100504.msp>.

- The Passphrase FAQ
<http://www.iusmentis.com/security/passphrasefaq/>
- Enforcing Strong Password Usage Throughout Your Organization
http://www.microsoft.com/smallbusiness/support/articles/enforce_strong_passwords.msp
- Bruce Schneier: Write down your passwords
http://www.schneier.com/blog/archives/2005/06/write_down_your.html
- Bruce Schneier: Practical Cryptography book, chapter 22, Storing Secrets
<http://www.schneier.com/book-practical.html>

A A program to generate a random password for Unix systems

Program available at:

<http://www.linux-france.org/~dmentre/code/dev-random-pass-gen.c>

```
/* dev-random-pass-gen: generation of user password with /dev/random
Version: 1.1 - 2006-07-28
```

```
Original author: Thomas PORNIN (see below)
Modifications by David MENTRE <dmentre@linux-france.org>
```

```
This program is in the Public Domain. Original author as provided his
code as-is and don't want to hear of it (source: private
communication).
```

```
For any comment on this program, send them to David MENTRE. Reviews are
welcome.
```

```
Compile with: gcc -Wall -o dev-random-pass-gen dev-random-pass-gen.c
```

```
Should work on any Unix-like system. Run the program only once! Don't
repeat it until you find an easy password as you would loose the
randomness of this approach.
```

```
*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
```

```
#include <unistd.h>
#include <fcntl.h>
```

```
/* return a random integer x in 0 <= x < max */
static unsigned randval(int rand_fd, unsigned max)
{
    unsigned val;

    do {
        unsigned char x;

        for (;;) {
            if (read(rand_fd, &x, 1) <= 0) {
                if (errno == EINTR)
                    continue;
            }
        }
    } while (x < 0);

    val = x % max;
}
```

```

                                perror("read");
                                exit(EXIT_FAILURE);
                            }
                            break;
                        }
                    /* after this point: x has a random value such that
                       0 <= x < 256 */

                    val = x;
                } while (val >= max * (256 / max));

                /* after this point: 0 <= val < max * | - 256 / max -|. In other
                   words, there is an integer k so that 0 <= val < max * k. So
                   each integer n in 0 <= n < max has as much probability to be
                   chosen by val. */

                return val % max;
            }

static int randletter(int rand_fd)
{
    return "abcdefghijklmnopqrstuvwxyz"[randval(rand_fd, 26)];
}

static int randdigit(int rand_fd)
{
    return "0123456789"[randval(rand_fd, 10)];
}

int main(void)
{
    static int rand_fd = -1;

    /* we use /dev/random as it is supposed to be more secure than
       /dev/urandom and the blocking case is a non issue for manual
       password generation. */
    rand_fd = open("/dev/random", O_RDONLY);
    if (rand_fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    printf("%c%c%c%c%c%c%c%c%c\n",
           randletter(rand_fd), randletter(rand_fd),
           randdigit(rand_fd), randdigit(rand_fd),
           randletter(rand_fd), randletter(rand_fd),
           randdigit(rand_fd), randdigit(rand_fd),
           randletter(rand_fd), randletter(rand_fd));
    /* Generated password has:
       log2(26*26 * 10*10 * 26*26 * 10*10 * 26*26) > 41 bits of entropy

       If each password takes 1ms to test, it would take to try all
       passwords:
       241 * 10e-3 / (3600 * 24 * 365) > 69 years
    */

    close(rand_fd);

    return EXIT_SUCCESS;
}
/* The above program is a modification of following Usenet post:

```

From: pornin@nerim.net (Thomas Pornin)
Newsgroups: fr.misc.cryptologie
Subject: Re: Quel logiciel libre de gnration de mots de passe ?
Date: Sun, 23 Jul 2006 11:39:33 +0000 (UTC)
Message-ID: <e9vn5l\$1lil\$2@biggoron.nerim.net>
**/*