Hacking Fingerprint GUI

(Version 0.11)

For developers who want to understand and possible change something in the Fingerprint GUI project it might be required to give some hints about how the system works. It is assumed the reader of this document has a certain understanding of PAM.

There are 4 executables and one library used:

- “fingerprintGUI” – The main application to be used for discovering fingerprint scanners on the USB bus, acquiring (enrollment) and verification of fingerprints and testing the PAM settings.
- “fingerprintIdentifier” – An application for testing the identification of users by their fingerprints and in special cases for any user-defined scripts in which a fingerprint identification is required.
- “fingerprintHelper” -- A helper application to be called out of PAM for requesting fingerprints while identifying or authenticating a user.
- “fingerprintPlugin” -- A helper application to be plugged in into some other applications (like gnome-screensaver) that don't allow other GUI applications to be displayed on to of their own screen.
- “libpam_fingerprint.so” -- A PAM module that is called from “libpam.so” when a user has to be identified or authenticated by their fingerprint.

1 Debug Output

All modules accept a “debug” (or “-d” or “--debug”) argument to be given for execution. This argument causes a lot of debug output printed to syslog. Depending on syslog settings the output might be printed to /var/log/auth.log, /var/log/messages or /var/log/syslog.

2 FingerprintGUI

This software should be self explanatory. So no further information is needed at the moment.

3 FingerprintIdentifier

This program first tries to collect all stored fingerprint data of all users from their home directories. Therefore it walks through the /etc/passwd file, extracts the home directories and looks in there for a “.fingerprints/” directory containing fingerprint data for this user. If this directory is available and can be read it collects all fingerprint data and tries to identify the user after a finger swipe was given. If it doesn't run with root permissions the home directory of other users might not be readable. In this case the fingerprints of the current user are accessible only. So only the current user can be identified or authenticated. After a successful authentication the user's login name is printed to stdout and the program exits.

4 FingerprintHelper

All Fingerprint GUI modules are developed using the Qt system, except the libpam_fingerprint.so module. This means the executable creates a QApplication or a QCoreApplication object when running. Qt doesn't allow more than one of these objects to be created in an executable. Now, if some other Qt application would call PAM for authenticating a user and PAM would call the

libpam_fingerprint.so library this would cause a crash, when libpam_fingerprint.so would try to create a (second) QApplication or QCoreApplication object while prompting for a finger swipe. Therefore libpam_fingerprint.so is not a Qt application but forks a child process (the fingerprintHelper) to prompt the user for a finger swipe and handle fingerprints.

5 FingerprintPlugin

Some applications (namely gnome-screensaver) don't accept for security reasons any GUI window to be displayed on top of their own screen. This means if gnome-screensaver calls PAM to prompt the user for authentication for the screensaver to be unlocked, the libpam_fingerprint.so module is called by PAM. This module forkes the fingerprintHelper process to request a finger swipe, but the window of this process is not visible (it is displayed “under” the locked screen). Therefore an additional fingerprintPlugin was created. This application is “plugged in” into the gnome-screensaver the same way an “embedded keyboard command” would be (by gnome-screensaver configuration).

When gnome-screensaver starts its unlock prompt, it starts the fingerprintPlugin module. This module plugs into the screensaver and is displayed below the normal unlock prompt. Then the fingerprintPlugin listens for “display commands” at a named pipe “/tmp/fingerprintPlugin”. The fingerprintHelper process, forked by libpam_fingerprint.so finds this named pipe and sends all strings to be displayed at the GUI to this pipe. If no fingerprints for the user are available a “stop” command is sent to the pipe that causes the fingerprintPlugin to exit before fingerprintHelper exits itself. If some other command has to be displayed to the user (e.g. “ready...” or “authenticating <username>”) this command will be received by fingerprintPlugin and then displayed on its GUI window.

6 Libpam_fingerprint.so

This is the PAM module to be called out of “libpam.so” in case some application requests authentication. When its “pam_sm_authenticate” entry is called the module determines whether DISPLAY and XAUTHORITY environment variables are available. If DISPLAY is available and XAUTHORITY is missing the module tries to find the xauthority filename by searching the commandline of the X display process. If it was found libpam_fingerprint.so sets the XAUTHORITY environment variable.

The module then creates an anonymous pipe and forkes into a child process. This process executes “fingerprintHelper” with pipe and username (if available) as arguments. After forking, a random number (10 digits) is sent to the child process via the anonymous pipe. This number will be given back as a “password” if the user was identified or authenticated by his fingerprint.

The parent process then calls the “PAM conversation function” of the calling application for prompting the username and/or password by keyboard while the child process (fingerprintHelper) prompts the user for swiping their finger.

Now the two processes wait for input in the following cases:

6.1 Case 1 (username was given by keyboard)

The PAM conversation function returns to the parent process (libpam_fingerprint.so) with a non empty username. Because the parent doesn't know, whether the username was typed by keyboard or given via libfakekey by the child process, it sends a SIGUSR1 to the child. This causes the child to exit, if the username was not identified by fingerprint. Then the parent stores the username in PAM environment and prompts for a password by calling the “PAM conversation function” of the calling application again. When this function returns, the password is compared with the random number

mentioned above. If the “password” matches the number this is case 2 below. If it doesn't match, the password might be given by keyboard. It is stored to the PAM environment and a PAM_IGNORE is returned to PAM. This causes PAM to call the next module in stack (probably pam_unix.so) for authenticating the given username/password.

6.2 Case 2 (user was identified by fingerprint)

In this case the child process (fingerprintHelper) writes the username to the prompt using libfakekey, waits for 1 second to make sure the parent prompts for the password meanwhile, then writes the random number (received from parent via the pipe) to the password prompt via libfakekey and then exits.

The parent then stores the username to the PAM environment and compares the “password” with the random number given to the child process. If the “password” matches the random number it returns PAM_SUCCESS and PAM is finished. If it doesn't match this is case 1 above..

6.3 Case 3 (empty username was given by keyboard)

The PAM conversation function returns to the parent process (libpam_fingerprint.so) with an empty username (maybe the user has typed <enter> only). The parent then kills the child and returns PAM_AUTHINFO_UNAVAIL.

6.4 Case 4 (user was known already; authentication)

In this case the parent gives the already known username to the child process when calling fingerprintHelper as an argument and sends the random number via pipe to the fingerprintHelper. After that it prompts for a password by calling the “PAM conversation function” of the calling application.

The fingerprintHelper collects the stored fingerprints of this user and prompts for a finger swipe. If one of the fingerprints matches the swipe, the random number is sent to the password prompt via libfakekey.

If the “PAM conversation function” returns to the parent, it compares the “password” with the random number. If it matches, the user was authenticated by fingerprint. The parent waits for the child to exit and then returns PAM_SUCCESS and PAM is finished. If it doesn't match, the password might be given by keyboard. Then the parent kills the child, stores the password to the PAM environment and returns PAM_IGNORE. This causes PAM to call the next module in stack (probably pam_unix.so) for authenticating the given username/password.

6.5 Special case 5 (the real password of the user matches the random number; veriy unlikely)

In this very unlikely case the parent process compares a match between the random number (given to fingerprintHelper) and the real password (given by the user via keyboard). Then the parent process “thinks” the user was authenticated by fingerprint and returns PAM_SUCCESS without storing this password to the PAM environment. This can cause further PAM modules in stack to prompt for a password again (e.g. gnome-keyring). This is not a false authentication (it was the correct password) but an unexpected behavior of the PAM system because the user knows that he has given the password already and is now prompted again. This is the same behavior like if a user was identified by their fingerprint.

7 Saving Passwords to Removable Media

With FingerprintGUI (“Password” Tab) the user can chose some directory on a mounted removable media, invoke his login password and save it to this media. This way the system can provide the login password to PAM while the user logs in with fingerprint to avoid a password request when e.g. gnome-keyring has to be opened.

This login password information is split into 2 different locations:

- A file “<username>@<machinename>.xml” in a subdirectory “.fingerprints” on the chosen removable media, containing the encrypted password;
- A file “config.xml” in the directory “~/.fingerprints” in the user's home directory, containing the path to the “<username>@<machinename>.xml” file, the UUID of the chosen partition on removable media and the key for decrypting the password.

The password is encrypted with a random symmetric key (AES128-CBC-PKS7).

THERE IS A SECURITY RISK when this user is not the only one who has root access to the machine! Someone could connect to this machine (e.g. via ssh) and copy the “config.xml” file and the “<username>@<machinename>.xml” from the connected removable media and then decrypt the user's password.

When configured (see “StepByStep” manual) and the removable media is connected while fingerprint login the system takes the following steps:

- After the user is identified (by fingerprint) the FingerprintPAM module looks for the “~/.fingerprints/config.xml” file in the user's home directory and, if found, reads the UUID, the decryption key and the initialization vector;
- Then creates a temporary directory “/tmp/<UUID>” and tries to mount the partition with this UUID there;
- Then reads the encrypted key from “<username>@<machinename>.xml” file and unmounts the media immediately;
- If the password can be decrypted it is given to PAM by a “pam_set_item()” call;
- If this call was successful it returns PAM_IGNORE. Then PAM calls the next module in stack (pam_unix.so) to validate username and password and complete the login process.

I'M NOT A CRYPTO EXPERT! If you are, please have a look at the sources (UserSettings.cpp) and let me know if there are possible problems.

8 Compiling the Sources

You need the Qt4 environment (incl. libqca2) and the “developer” packages of the used libraries installed on your system for being able to compile the sources. You can then use the “configure.sh” script to create the makefiles for your system. The “configure.sh” uses “qmake” in combination with the “*.pro” files in each subdirectory and creates a “Makefile”. Then call “make” and it will create all executables in “32bit” or “64bit” subdirectories depending on your system architecture. Don't use “make install” because an “install” target is not available. Use the “install.sh” script instead to copy the binaries to their proper locations.

8.1 Compiling on 64-bit Systems

The “fprint-only” modules should be properly created in the 64bit subdirectories if the Makefile was created by the “configure.sh” script on your 64bit machine.

Unfortunately the proprietary library “libbsapi.so” from UPEK Inc. is only available as a 32-bit

version. If you have a device from UPEK or SGS Thomson you will not be able to use the 64bit binaries of “fingerprintGUI”, “fingerprintIdentifier” and “fingerprintHelper”. This means you need a 32bit system in order to compile these components and then copy them into the 32bit subdirectories of your 64bit machine before you can install the new compiled binaries.

Please note, that the “upekts” driver of the current “libfprint” project is not capable to “identify” fingerprints. It can only “verify” what is not sufficient for login. So you need to use the “libbsapi.so” in combination with the 32bit binaries like mentioned above.