

# Hacking Fingerprint GUI

(Version 0.15)

For developers who want to understand and possibly 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 5 executables and one library used:

- “fingerprint-gui” – 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.
- “fingerprint-identifier” – An application for testing the identification of users by their fingerprints and to be used in special cases for any user-defined scripts in which a fingerprint identification is required.
- “fingerprint-helper” -- A helper application to be called out of PAM for requesting fingerprints while identifying or authenticating a user. This application should not be called from the command line.
- “fingerprint-suid” – Another helper application to be called from “fingerprint-gui” when a new user directory has to be created in “/var/lib/fingerprint-gui/”, where user settings and fingerprint (bir) data are stored. This application should not be called from the command line.
- “fingerprint-plugin” -- 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 top of their own screen. This application should not be called from the command line.
- “pam\_fingerprint-gui.so” -- A PAM module that is called by PAM 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 fingerprint-gui

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

## 3 fingerprint-identifier

This program first tries to collect all stored fingerprint data of all users from user-specific directories in “/var/lib/fingerprint-gui/<username>/<drivename>”. If a directory named after the user 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 directories of other users are not 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 fingerprint-helper

All Fingerprint GUI modules are developed using the Qt system, except the “pam\_fingerprint-gui.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 “pam\_fingerprint-gui.so” library this would cause a crash, when “pam\_fingerprint-gui.so” would try to create a (second) QApplication or QCoreApplication object while prompting for a finger swipe. Therefore “pam\_fingerprint-gui.so” is not a Qt application but forks a child process (the fingerprint-helper) to prompt the user for a finger swipe and handle fingerprints.

## 5 fingerprint-suid

If a user acquires (enrolls) his fingerprints the first time the user's data directory in “/var/lib/fingerprint-gui/” does not exist yet. Because the directory “/var/lib/fingerprint-gui/” is owned by root and has mode 755 it is required to create the user's data directory from a process running suid root. In this case “fingerprint-gui” executes “fingerprint-suid” (which is owned by root and has the mode u+s Bit set). “fingerprint-suid” creates then the required <user> directory, makes it owned by this user and his primary group and sets its mode to 700.

## 6 fingerprint-plugin

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 “pam\_fingerprint-gui.so” module is called by PAM. This module forkes the “fingerprint-helper” 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 “fingerprint-plugin” 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 “fingerprint-plugin” module. This module plugs into the screensaver and is displayed below the normal unlock prompt. Then the “fingerprint-plugin” listens for “display commands” at a named pipe “/tmp/fingerprint-plugin”. The “fingerprint-helper” process, forked by “pam\_fingerprint-gui.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 “fingerprint-plugin” to exit before “fingerprint-helper” 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 “fingerprint-plugin” via its named pipe and then displayed on its GUI window.

## 7 “pam\_fingerprint-gui.so”

This is the PAM module to be called out of PAM 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 command line of the X display process. If it was found “pam\_fingerprint-gui.so” sets the XAUTHORITY environment variable.

The module then creates an anonymous pipe and forkes into a child process. This process executes “fingerprint-helper” 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 (fingerprint-helper) prompts the user for swiping their finger.

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

### **7.1 Case 1 (username was given by keyboard)**

The PAM conversation function returns to the parent process (pam\_fingerprint-gui.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.

### **7.2 Case 2 (user was identified by fingerprint)**

In this case the child process (fingerprint-helper) 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..

### **7.3 Case 3 (empty username was given by keyboard)**

The PAM conversation function returns to the parent process (pam\_fingerprint-gui.so) with an empty username (maybe the user has typed <enter> only). The parent then kills the child and returns PAM\_AUTHINFO\_UNAVAIL.

### **7.4 Case 4 (user was known already; authentication)**

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

The “fingerprint-helper” 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 (fingerprint-helper), 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.

## 7.5 *Special case 5 (the real password of the user matches the random number; very unlikely)*

In this very unlikely case the parent process compares a match between the random number (given to “fingerprint-helper”) 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.

## 8 Saving Passwords to Removable Media

With “fingerprint-gui” (“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 “/var/lib/fingerprint-gui/<username>” 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 with root permission 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 “Install-step-by-step” manual) and the removable media is connected while fingerprint login the system takes the following steps:

- After the user is identified (by fingerprint) the “pam\_fingerprint-gui.so” module looks for the user's “/var/lib/fingerprint-gui/<username>/config.xml” 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.

## 9 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 “qmake-qt4” command to create the makefiles for your system. Then call “make” and it will create all executables in “./bin/...” subdirectories. After binaries are successfully created you can use “sudo make install” to copy them to their proper locations. If you have a fingerprint scanner manufactured by UPEK inc. or SGS Thomson you can install the bundled proprietary driver “libbsapi.so” by executing “sudo make install-upek”. This is needed because the open source driver (in libfprint) lacks the ability of comparing one-to-many fingerprints.

---