Biometric Fingerprint Checking With Ipaq SDK

North Carolina State University
Real Time Systems (CSC714)
By: Michael Noeth and Jyothish Varma

1.0 Introduction

Biometrics is the collaboration of multiple scientific fields which attempt to automate human recognition based on a unique physical attribute. Currently, a person can be identified by multitude of different metrics including their fingerprints, facial structure, iris of their eye, hand geometry, voice, or written signature (see [1] for more details on each metric). A sensor is used to capture a particular metric and store it in an electronic format. This information can be used later to identify or authenticate a person to a system.

Biometrics is being used increasingly throughout technology as a means of authentication [2]. Industry has begun to include biometric sensors as standard additions on laptops (i.e IBM's Thinkpad T42), personal digital assistants (i.e. HP's iPAQ Pocket PC h5400), and other various devices. Currently, many of the devices simply use the biometric authentication as an initial login requirement (both the ThinkPad and iPAQ use their biometrics solely for this purpose). Software must catch up to the current trends in hardware by providing protocols and applications that support biometric hardware.

We've implemented a simple client / server authentication protocol that uses the iPAQ H5550's biometric hardware. Instead of using the traditional user name and password pair to authenticate users, a user name and finger print pair will be used in its place. We propose using this API as a replacement for the user name and password pair for SSH daemons, FTP daemons, and other various client / server authenticated applications. Our simple API provides the building blocks necessary to replace the clunky old user name / password pair for distributed system authentication.

2.0 Motivation & Background

Biometric hardware is currently showing up on many new hardware platforms as mentioned previously. For the most part – it lacks widespread support in software. On a single user platform, biometrics is used to authenticate a user before he / she can use the system. This authentication typically comes bundled with the biometric hardware.

Other possible extensions of the biometric hardware (not implemented to the best of our knowledge) could use it to establish the identity of a user in order to load personal settings for an application. Although this would be nice – biometrics is best suited for authentication purposes. Since authentication on a single platform usually comes standard on systems sporting the new biometric hardware, the next domain to explore is distributed computing.

The iPAQ platform provides a fairly comprehensive API based on the BioAPI Consortium standard [3]. This rising industry standard provides an interface that allows application programmers to scan, store, and authenticate finger prints in their applications. Since the BioAPI Consortium is a standard, it provides a means for portable code on any supporting hardware platforms. The API is geared towards authentication on only a single platform. We propose creating a protocol / API to authenticate users in a distributed computing environment based on the BioAPI Consortium standard. Thus – our API could be ported to any system supporting the BioAPI Consortium standard.

To explore this idea further – we have decided to use SSH as our target application (although any client / server application should work well with our protocol). SSH provides protection from the following (more details on each attack from [4]):

- Eavesdropping: An attacker who has gained access to the data path between the client and server can "listen in" and read your traffic potentially stealing passwords or sensitive data. Eavesdropping is prevented because the client and server negotiate a secure channel (encrypted) before authentication takes place.
- Data Modification: An attacker who has gained access to the data path between the client and the server can modify data being sent back and forth. Data modification is also prevented by the secure channel discussed in Eavesdropping.
- Identity Spoofing: The attacker uses a client's IP address to pretend to be that client after a client / server relationship has been established. Identity spoofing is prevented by the secure channel discussed in Eavesdropping.
- Password / Brute Force: The attacker tries random passwords until they've guessed a correct user name / password combination. SSH can be configured to ignore excessive password attempts for user accounts.
- Man in the Middle: The attack pretends to be the server. SSH prevents this by using a public key which it provides to all connecting clients. If the public key (which must be originally verified by a trusted 3rd party) is incorrect – the client knows not to disclose their password.

It is generally accepted that SSH is a secure means of communication as long as the user name / passwords are secure. An attacker can obtain the user name / password by means of social manipulation, easily guessing simple passwords (birthdays, family members' names, etc), finding the password written down, etc. Since this appears to be the weakest link in SSH (and many other applications using user name / password pairs), biometrics provides an innovative alternative for authentication. A finger print allows each user to have a unique identifier which cannot be easily stolen or reproduced. In addition to the heightened security – users no longer have to remember their password (or risk having it seen if they write it down).

3.0 Network Authentication API

We developed an API to support network based authentication. The API can be used as the basic building blocks in arbitrary applications to ensure that legitimate clients are interacting with the server. In this section, we provide details about the API and discuss the limitations and tradeoffs. The API provides the following functions (see bio_lib.h as appendix A for more details):

WriteToFile
This function allows the application developer to write a finger print to file. The file is a binary file that contains the data structure used by BioAPI Consortium standard. It can be broken down into two sections: (1) meta information about the finger print such as length of the data section, and (2) binary data corresponding to the finger print itself. This function is necessary for the server to generate new users.

ReadFromFile
This function allows the application developer to read a finger print from file (written by *WriteToFile*). The application developer is responsible for identifying the target file and instantiating a file handle. This provides the application developer the most flexibility in naming and storing of the finger print. *ReadFromFile* allocates memory for the data section (which can be arbitrarily large). To ensure the memory is reclaimed, the application developer must use our function, *Reclaim* (more details follow). This function will typically be used on the server to authenticate clients.

WriteToSocket
This function allows the application developer to send finger print data over a socket. The application developer is responsible for opening their own socket (and securing it as necessary). A finger print stored in a data structure conforming to the BioAPI Consortium standard will be marshaled and the socket will be used to send the data. Meta information is transmitted in the first send. A second send is used to transmit the actual finger print data. This function will typically be used by clients to send finger print data to the server for authentication.

ReadFromSocket
This function allows the application developer to receive finger print data over a socket. Again, the application developer is responsible for opening their own socket. The receive function call will block and wait for a corresponding *WriteToSocket* using the same socket. The meta data is transmitted first. This is used to fill in the data structure conforming to the BioAPI Consortium standard. It is also used to coordinate the size of the data to be received and allows for the allocation of the necessary memory. Since *ReadFromSocket* allocates memory for the data section (which can be arbitrarily large) the application developer must use our function, *Reclaim* to get the memory back. This function will typically be used on the server to authenticate clients. This function will typically be used by the server to read finger print data from the clients for authentication.

Init
This function must be called before performing any biometric function calls. It serves two purposes: (1) a call to the BioAPI Consortium init function, and (2) allocates and initializes a state tracker. The state tracker's purpose is to keep track of buffers allocated using the *ReadFromFile* and *ReadFromSocket* function calls.

Reclaim
This function must be performed after all biometric function calls. It de-allocates all memory used in *ReadFromFile* and *ReadFromSocket*. It uses the data structure "state tracker" to perform this task.

This simple API provides the basic building blocks necessary to replace the old user name and password pair with a user name and finger print pair in client / server applications. There are a few things the application programmer should do to ensure security (these were not covered by our API to provide the most flexibility):

1. The application programmer is responsible for opening their own sockets. It is recommended that the socket be a secured channel (via encryption). Thus it is the responsibility of the application developer to ensure their application is not susceptible to any of the attacks mentioned in the Methodology and Background section.
2. The application programmer is responsible for opening their own file handles for reading and writing. The files for writing must be opened in write and binary modes. The files for reading must be opened in read and binary mode.

4.0 API Verification
The end goal was to have our API replace SSH's user name / password pair but due to time constraints – this has been put off to future works. To test our API, we developed a simple toy system. Our toy system uses the iPAQ H5550 as both clients and servers. Servers store files containing finger print data named after the corresponding user. In order for a client to authenticate itself, it sends a user name and finger print pair to the server. The server opens the user's file (based on the name provided by the client) and verifies that the passwords match. If the user name and password were valid, the server grants access to the requested resources.

Our test system consists of three applications: (1) a client, (2) a server, and (3) a create user program:

The CreateUser application uses the *WriteToFile* function from our API. Its purpose is generating user files. In our system, the files' names correspond to the user name. See figure 1 for a screen shot. The simple application consists of a place to write in a user name, and a button to begin the user creation process. After a user name is typed in – pressing the add user button results in the following:

1.	User:	Type in user name.
2.	User:	Press add user button.
3.	System:	Message box appears asking for a finger swipe.
4.	User:	Clicks “OK” and swipes finger
5.	System:	Message box appears asking for a finger swipe
6.	User:	Clicks “OK” and swipes finger
7.	System:	Writes data to file

Since this program is essentially the equivalent of the Linux passwd command, we decided the same safety measures should be taken. A new password is always verified in the event that something was mistyped. In our system – a finger print if verified to ensure everything was read correctly. If the finger prints do not match, the process starts again at step 2. Michael Noeth wrote this application.

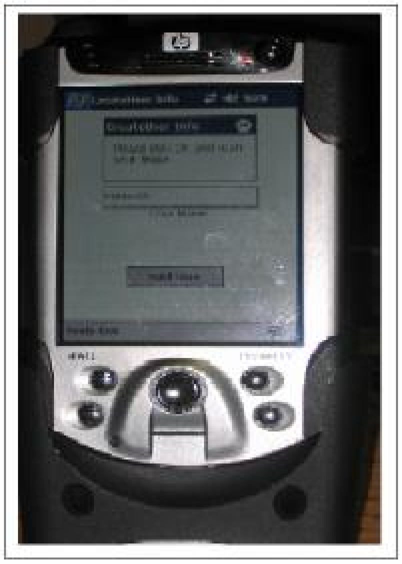

Figure 1: Screen shot of the CreateUser application.

The Client application uses the *WriteToSocket* function from our API. Its purpose is to attempt authentication with the server. See figure 2 for an idea of how the GUI is laid out. At the top, there are two input boxes where the user can specify the server’s IP address and their user name. There is a status bar at the bottom which indicates what is currently happening in the authentication process. Once these have been filled out, the client follows these steps to authenticate:

1.	User:	Type in server IP address.
2.	User:	Type in user name.
3.	User:	Click “Connect to Server” button.
4.	System:	Message box appears asking for a finger swipe.
5.	User:	Swipes finger.
6.	System:	Sends finger print to server.
7.	System:	Waits for ACK or NOACK and displays status.

Since this program is essentially the equivalent of the SSH client, our goal was to simulate this as closely as possible. In an SSH client, a user must specify their user name and the target host. Once this is done, the client attempts to connect with the server. At this point, authentication takes place. In the SSH client, this would be done by password. In our client application, this is now done by finger print. The server responds with whether the authentication was successful or not. If the authentication fails, you can begin again at step 3. Jyothish Varma wrote this application.

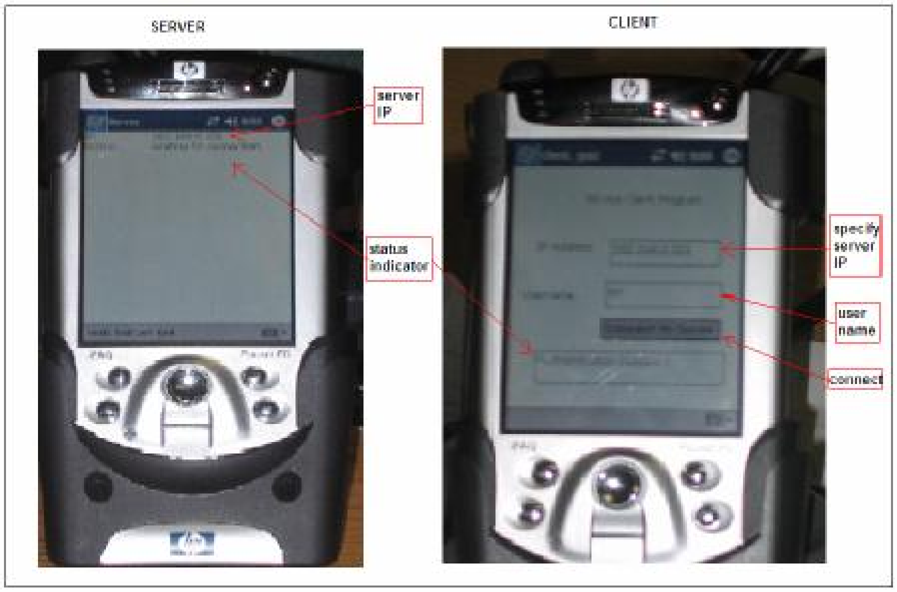

Figure 2: Screen shot of Client and Server applications

The Server application uses the *ReadFromSocket* and *ReadFromFile* functions from our API. Its purpose is to attempt authentication clients. See figure 2 for an idea of how the GUI is laid out. At the top, are two status indicators: (1) the IP address and (2) a connection status indicator. The server opens up a socket and waits to accept connections. When a connection is present, the server does the following:

1. Accepts connection.
2. Receives target user's name over the open socket.
3. Opens corresponding finger print file (file name is the same as the user name).
4. Receives finger print over the open socket.
5. Compares stored finger print with client's finger print.
6. Sends back status (i.e. whether the finger prints matched or not).

Since this program is essentially the equivalent of the SSH server, our goal was to simulate this as closely as possible. The SSH daemon on the server waits for connections and follows essentially the same procedure as our toy server application. The only difference is in the authentication method. Rather than verifying a user password from /etc/passwd and /etc/shadow, the new authentication uses the finger print files. Michael Noeth wrote this application.

5.0 Future Work

The future work can be broken into two sections: (1) implementation of our API in a real client / server application for complete verification, and (2) possible additions to the API that would increase security.

As mentioned before – SSH was the target application to replace the old user name / password pair with a user name / finger print pair. Due to time constraints – we knew that we would not be able to complete this task. Ideally – three applications must be modified: (1) the SSH client, (2) the SSH daemon and (3) the passwd command. The client modification should follow closely to our simple client. It will require the replacement of the password portion with a request for a finger print scan. The SSH daemon also requires a similar change. The portion of code which receives the password and verifies it against /etc/passwd and /etc/shadow must be replaced with calls to our API. The process for enrolling users via the passwd command should probably be approached by providing a different application to create users (there is not much overlap between the passwd command and a new enrollment / user creation).

The real problem with modifying the SSH client, SSH daemon, and the enrolling process is that Linux (where most SSH daemons are implemented) does not support the BioAPI Consortium standard. Since our API is built on top of this layer, certain functions must be made available. The standard is available on line along with the algorithms necessary – we just have to wait until it has been implemented.

There are two additions to the API itself that could increase security. The first is a history feature. The second is encryption of binary files containing finger print data.

A history feature would allow the server to better fend off replay attacks. A replay attack occurs when an attacker captures a sequence of packets that correspond to an authentication. The attacker then replays or resends these packets in order to gain access to the server. By keeping a history of finger print scans, the server could avoid a replay attack. Since the verification of matching finger prints is a heuristic algorithm (match based on closeness threshold) – we know that no two finger print scans (even of the same person) will result in the same binary data. Thus, if we use the same heuristic algorithm that determines if the finger print has matched a minimum threshold in reverse (i.e. cannot exceed a maximum threshold when matched against finger prints stored in a history data structure) – we could fend off replay attacks. Storage-wise, this improvement would not be very costly (each finger print file is only approximately 1 Kb).

Encryption of the binary files containing finger print data is fairly straight forward. This measure is taken for the single user login on the iPAQ currently. It is only decrypted for authentication to gain access to the system when the iPAQ is turned on.

6.0 Conclusion

We developed an API which allows authentication of an iPAQ on a remote server using biometric fingerprint checking. We have extended the scope of authentication using a username – fingerprint pair on a single system to a client-server model. This method can effectively reduce brute force attacks and when combined with good encryption algorithms will ensure a better way to protect system from hackers. It also eliminates the need for a user to remember passwords. The password to a remote system is stored with in himself. A few important points we noted during our experiments with iPAQ was the high power consumption of iPAQ's when it makes use of the wireless network. During the experiments, the iPAQ's had to be kept in its cradle for the network to work properly. Embedded Visual C++ served as a good platform for programming and it offers lot of features which makes a programmer's life lot simpler. Since the iPAQ's had to communicate over the network, the emulator did not help us to verify the programs. Each time a program was modified, it had to be validated and tested manually. With the addition of secure channel, we foresee the replacement of username – password pair with username – biometric pair for remote authentication.

7.0 References

[1] Blackburn, Duane M. "Biometrics 101." Federal Bureau of Investigation, March 2004.

[2] HP Smart Handheld Group, tech. white paper. "Biometric Security with the iPAQ Pocket PC h5400 series." Hewlett-Packard Company, Palo Alto, CA, January 2003.

[3] HP iPAQ Pocket PC Developers, tech. specification. "Biometrics API version 0.6." Hewlett-Packard Company, Palo Alto, CA, November 11, 2002.

[4] Microsoft Developers Network. "Common Types of Network Attacks." Microsoft, 2005.

Appendix A – bio_lib.h

```
#include "BioAPI_iPAQ3.h"
#include <stdio.h>
#include <stdlib.h>
#include <winsock.h>

void WriteToFile(BioAPI_BIR Target, FILE *Output);
/***************
This function writes the header and the image captured using BioAPI_Capture() function
to the file pointer specified by Output. Usually, *Output is a pointer to a file which
is the username of the user who has an account in the server machine. This function is
called by a create user application which adds a username to the server. The file written
is in the format of the BioAPI header followed by the data (which is the image itself).
No encryption features or shadowing has been currently implemented to the file written.
This function helps in storing the acquired data in a database which can be further
send across machine if the application demands for it.
***************/

void ReadFromFile(BioAPI_BIR_PTR Target, FILE *Input);
/**************
This function reads the sored template from a file pointer specified by file pointer *Input.
The parameters stored in the file are in the following format - all of which are integers.
  --      Header.Length,
  --      Header.HeaderVersion,
  --      Header.Type,
  --      Header.Format.FormatID,
  --      Target.Header.Format.FormatOwner,
  --      Header.Quality,
  --      Header.Purpose,
  --      Header.FactorsMask.
This is followed by the finger image data which is stored in binary format in the file
***************/

void WriteToSocket(BioAPI_BIR_PTR Target, SOCKET Socket);
/**************
The image captured from the finger print sensor along with the header information is
send to the server in two steps. Initially, the client sends the username to the server to
verify if the user of specified name exists in the server. This prevents heavy data transfer
which is required for authentication. This data transfer is heavy since it involves transfer
of  image of the fingerprint. If the username exists, the client proceeds with sending
header information followed by image data to the server for authentication. If the
username and the finger print data is verified correctly, the server sends back an ACK to
the client which indicates that server has successfully authenticated the client.
***************/
```

```
void ReadFromSocket(BioAPI_BIR_PTR Target, SOCKET Socket);
/**************
This function reads the data send by the client to server side and fills the BioAPI
structure.  The server proceeds with calling this function only if the username which was
initially send by the client exists in the server. The BioAPI header is initially received
followed by the image of the finger print. This data fills in the BioAPI data structure.
Futher, a caparison is made against the existing image of the server against the image
obtained from the client. The server sends an ACK message if the finger print data
matches. If the match fails, it sends a NOACK and server rejects the connection from the
client.
***************/
```