NIST

**National Institute of Standards and Technology**
Technology Administration
U.S. Department of Commerce

**NISTIR 7200**

# Proximity Beacons and Mobile Device Authentication: *An Overview and Implementation*

Wayne Jansen
Serban Gavrila
Vlad Korolev

NISTIR 7200

# Proximity Beacons and Mobile Handheld Devices: Overview and Implementation

**Wayne Jansen**
**Serban Gavrila**
**Vlad Korolev**

C O M P U T E R   S E C U R I T Y

Computer Security Division
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20988-8930

June 2005

## Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analysis to advance the development and productive use of information technology. ITL's responsibilities include the development of technical, physical, administrative, and management standards and guidelines for the cost-effective security and privacy of sensitive unclassified information in Federal computer systems. This Interagency Report discusses ITL's research, guidance, and outreach efforts in computer security, and its collaborative activities with industry, government, and academic organizations.

**National Institute of Standards and Technology Interagency Report**
**31 pages (2005)**

## Abstract

The use of mobile handheld devices within the workplace is expanding rapidly. These devices are no longer viewed as coveted gadgets for early technology adopters, but have instead become indispensable tools that offer competitive business advantages for the mobile workforce. While these devices provide productivity benefits, they also pose new risks to an organization's security by the information they contain or can access remotely.

Enabling adequate user authentication is the first line of defense against unauthorized use of an unattended, lost, or stolen handheld device. This report describes an innovative type of authentication mechanism that relies on the presence of a signal from a wireless beacon for access to be granted. Such proximity beacons can be either organizational or personal oriented, and require only that handheld devices support a common standard wireless interface for Personal Area Network (PAN) communications, such as Bluetooth. Details of the design and implementation for both personal and organizational proximity beacons are provided.

## Table of Contents

## Introduction

With the trend toward a highly mobile workforce, the use of handheld devices such as Personal Digital Assistants (PDAs) is growing at an ever-increasing rate. These devices are relatively inexpensive productivity tools that are quickly becoming a necessity in government and industry. Most handheld devices can be configured to send and receive electronic mail and browse the Internet using wireless communications. While such devices have their limitations, they are nonetheless extremely useful in managing appointments and contact information, reviewing documents and spreadsheets, corresponding via electronic mail and instant messaging, delivering presentations, and accessing remote corporate data.

Manufacturers produce handheld devices using a broad range of hardware and software. Unlike desktops and notebook computers, handheld devices typically support a set of interfaces that are oriented toward user mobility. Handheld devices are characterized by their small physical size, limited storage and processing power, and battery-powered operation. Most Personal Digital Assistant (PDA) devices provide adequate memory (at least 32MB Flash ROM and 64MB RAM) and processing speed (200Mhz or higher) for basic organizational use. Such devices come equipped with a LCD touch screen (one-quarter VGA or higher) and a microphone, soundcard, and speaker, but usually lack a QWERTY keypad. One or more wireless interfaces, such as infrared or radio (e.g., Bluetooth and WiFi) are also built-in for communication over limited distances to other devices and network access points; so too are wired interfaces (e.g., serial and USB) for synchronizing data with a desktop computer. Many high-end PDA devices also support Secure Digital (SD) and Compact Flash (CF) card slots for feature expansion. Over their course of use, such handheld devices can accumulate significant amounts of sensitive corporate information (e.g., medical or law enforcement data) and be configured for access to corporate resources via wireless and wired communications.

One of the most serious security threats to any computing device is unauthorized use. User authentication is the first line of defense against this threat. Unfortunately, management oversight of user authentication is a persistent problem, particularly with handheld devices, which tend to be at the fringes of an organization's influence. Other security issues related to authentication that loom over their use include the following items:

- Because of their small size, handheld devices are easily lost or stolen.
- User authentication may be disabled, a common default mode, divulging the contents of the device to anyone who possesses it.
- Even if user authentication is enabled, the authentication mechanism may be weak or easily circumvented.
- Once authentication is enabled, changing the authentication information regularly is seldom done.
- Limit processing power of the device, may preclude the use of computationally intensive authentication techniques or cryptographic algorithms.

Authentication using passwords is perhaps the best-known example of a proof by knowledge mechanism. Other classes of authentication mechanisms include proof by possession (e.g., smart cards) and proof by property (e.g., fingerprints). Two additional factors that can apply to each class of authentication mechanism are location and time of day. They refer respectively to whether the authentication is being attempted at either an acceptable location or an acceptable time. The mechanisms described in this report involve location as a facet of user authentication.

Establishing location benefits user authentication in several important ways:

- If a user attempts to authenticate from an unauthorized location, the authentication mechanism can reject the attempt.
- If a user attempts to authenticate from a location outside a defined boundary, the authentication framework can require that additional authentication mechanisms be satisfied before granting access.
- If a user instantiates a new activity, such as accessing a specialized application, the authentication framework can require that access to the functionality and related data be conducted from with an appropriate location.
- If a user moves within or outside of a defined boundary, the authentication mechanism can be triggered automatically to grant or deny access.

This report provides an overview of two kinds of location-based authentication mechanisms involving proximity beacons that interface and communicate through standard interfaces supported by most handheld devices. The report describes how each kind of beacon is used to authenticate users on handheld devices and provides details of the solutions' design and implementation.

The authentication mechanisms were implemented in C and C++ on an iPAQ Personal Digital Assistant (PDA), running the Familiar distribution of the Linux operating system from handhelds.org and the Open Palmtop Integrated Environment (OPIE). OPIE is an open-source implementation of the Qtopia graphical environment of TrollTech. OPIE and Qtopia are both built with Qt/Embedded, a C++ toolkit for graphical user interface (GUI) and application development for embedded devices, which includes its own windowing system. The Familiar distribution was modified with MAF, a framework for multi-mode authentication [Jan03a]. The framework includes a policy enforcement engine, which governs the behavior of both code modules and users [Jan03b], and the facility to add new authentication mechanism modules and have them execute in a prescribed order.

## Background

Physical location sensors come in many shapes and sizes and use many different techniques for determining position. Physical sensor systems typically have two kinds of components: appliances and infrastructure. An appliance is the equipment associated with an entity (e.g., a Global Positioning System (GPS) receiver or mobile phone), while the infrastructure is the set collection of sensor equipment, usually fixed, which needs to be in place for the appliances to function (e.g., GPS satellites or mobile phone towers) [Ind03]. A communication medium through which the devices and infrastructure communicate is also required. Other classes of location systems, where the user carries no appliance and the solution relies entirely on infrastructure components (e.g., infrared cameras or floor sensors), are outside the scope of this discussion.

Physical location sensors can provide either position or proximity information. Position sensors attempt to provide the coordinates of an entity (or more usually, an appliance) relative to some coordinate system. The coordinate system may be fixed and global (e.g., the latitude, longitude and altitude reported by a GPS receiver), or mobile and local (e.g., "3 meters to my right"). Proximity sensors are less exact (e.g., within close or distant range of a sensor) [Ind03]. While latitude-longitude-altitude coordinates are suitable for describing points on the globe, they do not work as well for describing points indoors. Proximity sensors with overlapping detection regions can form the basis of position sensors. Position information can be determined via triangulation or trilateration. Both techniques use the geometry of triangles to calculate the relative position between points. Triangulation uses both distance and angle measurements, whereas trilateration uses only distance measurements.

Different sensors have different resolutions and associated errors, ranging from centimeters (e.g., the ultrasound positioning of the Active Bat system) to tens of meters (e.g. raw GPS) [Hig01, Ind03, Haz04]. Different sensors also operate over different scales of distance, ranging from zero (e.g. contact sensors and card readers) to global (e.g. GPS). Sensors may also be limited to indoor or outdoor use. For example, the GPS, perhaps the best-known technology for establishing location, requires a clear view of at least three of the two-dozen satellites orbiting above the Earth to determine position [Hig01]. Because satellite reception in buildings is poor to nonexistent, GPS is ineffective indoors [War97].

Thus, location can be treated in two ways: by position, where geographical or other physical coordinates of a unit are resolved to some degree of accuracy, or by proximity, where a unit's presence, relative position, or absence within an area is determined. Determining positional coordinates typically requires an extensive sensor infrastructure able to cooperate with an appliance to estimate position algorithmically through monitored signals, using triangulation or some other technique. Determining proximity, while less precise, typically requires a less extensive infrastructure.

Two classes of solution prevail for resolving location. The first is where location information is initially known only by the appliance, but not the infrastructure. The second is the reverse by which location information is initially known only by the infrastructure and then released to the appliance [Gru03].

The first class of solutions makes the appliance more independent of infrastructure components and services. It also has privacy benefits, since the approach can allow the user of the appliance to decide when or whether to release the location information system wide. However, it requires the appliance to be not only compatible with the infrastructure beacons, but also powerful enough to make the needed computations and access control decisions. The second class of solutions is less demanding on the appliance, since the device does not have to be powerful enough to perform such computations and access control decisions (e.g., RFID or the Active Bat [War97]), relying instead on infrastructure components and services. For example, pervasive systems fall into this category, since they are by their very nature context-aware, one type of context information being location information gathered from a variety of location sources and sensors (e.g., the location of users, devices and services) [Gru03].

This report describes two kinds of authentication mechanisms that rely on proximity, which is determined using a small number of proximity beacons for the infrastructure. The authentication mechanisms are distinguished as either organizational or personal oriented, and in both cases require only that participating handheld devices, which function as the appliances, support a common standard wireless interface for Personal Area Network (PAN) communications, such as Bluetooth. The mechanisms are designed to establish the relative location of a mobile device with respect to a trusted beacon that, once discovered, serves as a security token, which is contacted periodically to confirm its presence and to verify its authenticity.

## The Multi-mode Authentication Framework (MAF)

MAF was developed previously in a related effort to provide a structured environment for the protection and execution of one or more authentication mechanisms operating on Linux handheld devices [Jan03a]. The authentication mechanisms described in this report were implemented specifically for this framework. Each authentication mechanism consists of two parts: an authentication handler and a user interface (UI). Figure 1 illustrates these elements within a Linux operating system environment, enhanced with kernel support for MAF.

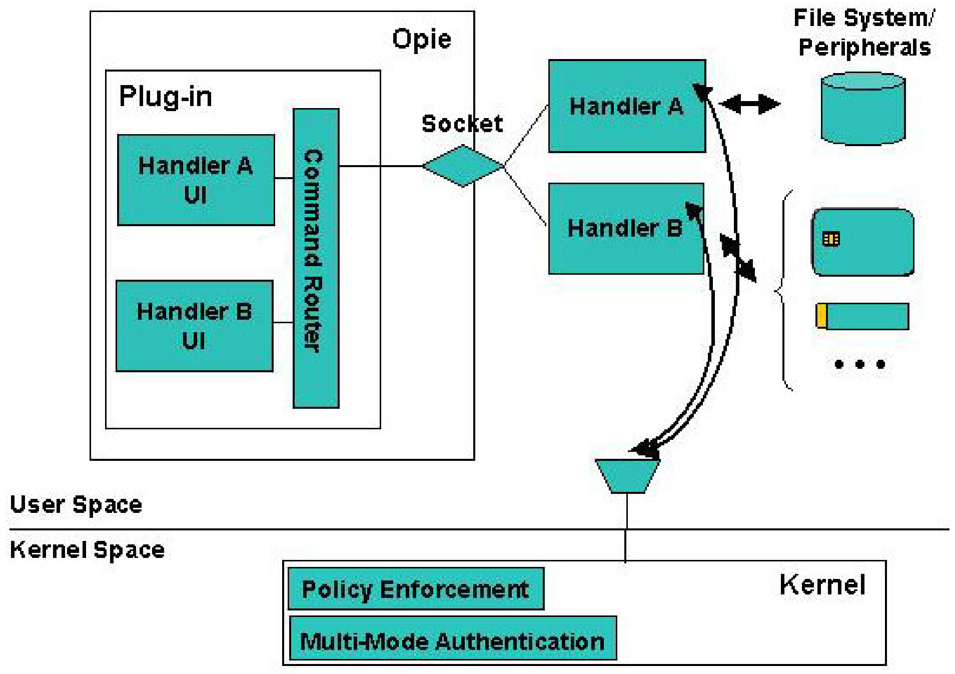

**Figure 1: Multi-mode Authentication Framework**

Authentication handlers embody the procedure that performs the actual authentication. They communicate with the kernel, listening for when to initiate authentication and reporting whether authentication was successful. They communicate with the user interface components to bring up specific screens, accept input, display messages, etc. on the device. Handlers also communicate with any peripheral hardware devices needed for authentication, such as a security token, and access the file system to store and retrieve information as needed. Handlers run in user space as do their respective user interface.

The user interface for an authentication mechanism is implemented as a set of components of a plug-in module for the OPIE desktop environment. Their function is to perform all necessary interactions with the user. For example, with beacon applications they can be used to notify the user of errors that occur. The plug-in module supports a socket interface to receive commands from an authentication handler that runs as a separate process, and to route the commands to the correct user interface component. Similarly, reverse routing is also supported for responses from user interface components to an authentication handler.

The kernel has two key modifications to support the framework: the multi-mode authentication functionality and the policy enforcement functionality.

- Policy enforcement's main responsibility is to impose different sets of policy rules on the device, as signaled by multi-mode authentication, for one or more defined policy contexts referred to as policy levels. For example, it can block hardware buttons and certain I/O ports on the device until the user is authenticated at the lowest policy level, policy level 1. Policy enforcement is also used to protect authentication information files, the user interface and handler components, and policy enforcement information against improper access. Moreover, it has the means to register and start up authorized handlers, if they are not running, or restart them, if they terminate for some reason.

- The main responsibility of the multi-mode authentication functionality within the kernel is to govern the authentication steps as they relate to the various policy levels that are configured. Communication between the kernel and an authentication handler is done via the /proc file system. The multi-mode authentication functionality maintains complete knowledge about the mappings between authentication mechanisms and policy levels, simplifying the development of the authentication handlers. One of its key functions is to initiate user authentication when the device is powered on. It also controls the order and frequency in which the handlers are awakened from suspended state and begin execution, and ensures that messages from only legitimate handlers are accepted and processed.

Together, the kernel policy enforcement and multi-mode authentication extensions are essential for securing authentication applications.

To create an additional authentication mechanism, a developer needs to create a new authentication handler along with any required user interface objects and the policy rules to protect the mechanism. Policy rules include limiting access to the storage objects used, the user interface objects within the plug-in module, and the authentication handler itself. They also can limit communications to peripheral devices and among the handler, the user interface, and kernel. Note that writing an authentication mechanism that neither interacts with the user nor requires a user interface component is possible. For example, the mechanism could be based on a sensor that is continually monitored and whose input automatically triggers an authenticated or non-authenticated transition.

## Personal Beacon Authentication

The personal beacon authentication mechanism relies on a security token in possession of the user to satisfy authentication. A PDA is either in or out of the proximity of the beacon as determined by the footprint of the communications signal. The mechanism periodically checks on connectivity with the beacon and reports successful authentication if present and able to be verified; otherwise, it reports failure. Conceptually, the mechanism operates somewhat like a garage door opener that keeps open the door as long as the opener is on and its signal is received. Roam too far from the door with the opener or turn it off and the door closes automatically.

A personal beacon supports a single PDA to which it is uniquely enrolled. A PDA and personal beacon communicate using Personal Area Network (PAN) communications. Two variants of the personal beacon were developed: one using near-field magnetic communications, the other using Bluetooth radio communications. The solution could be readily adapted for other types of wireless PAN communications technologies.

Bluetooth is a short-range wireless communications protocol for mobile devices, such as PDAs, cell phones, and headsets, which operates in the globally available 2.4GHz frequency band. Many models of mobile devices are often manufactured with built-in Bluetooth radios, which allow short-range communication and have low power consumption compared to other wireless technologies. Two PDAs with built-in Bluetooth radios were used for the Bluetooth prototype implementation: one as the beacon and the other as the mobile device. The PDA simulating the personal beacon token displays a fully functional virtual token via the touch screen.

Similarly, for the near-field magnetic communications prototype implementation, a pair of data evaluation boards produced by Aura Communications was used by the two PDAs to communicate.[1] The boards create a 1.25-meter communication bubble that enables private communications over a channel bit rate of 204.8 kbps and fully reusable frequencies only a few meters away. The time-division-duplexed (TDD) digital communication system uses magnetic induction technology, operating in the ISM band at 13.556 MHz, and requires very little power, making it suitable for use by handheld devices in short-range wireless applications.

### Operation

The personal beacon mechanism was designed for use with organizational handheld devices. Therefore, the design incorporates a public key infrastructure (PKI) and the use of X.509 certificates. It is also easily amenable to work solely with public key pairs generated on the token, if a PKI is unavailable for this application.

In the operation of the personal beacon, three phases are distinguished.

- *The beacon setup phase* – During this phase, the administrator generates a pair of private/public Rivest-Shamir-Adelman (RSA) keys, obtains a user certificate for the beacon, and stores the certificate and the private RSA key on the beacon. The administrator also stores the root certificate (chain) of the Certificate Authority (CA) that

[1] More information can be found at http://www.auracomm.com/

issued the beacon certificate onto the client PDA. In the case of Bluetooth personal beacons, the PDA and personal beacon can be paired to establish a long-term trusted association between the two. A special class identifier is used to distinguish beacons from other Bluetooth devices and simplify the pairing operation to the preferred beacon assigned to the user.

- *The beacon enrollment phase* – During this phase, the (client) PDA tries to authenticate the beacon for the first time. It consists of the following steps:

  - The PDA tries to identify and connect to a personal beacon. Note that in the case of the Bluetooth variant, the beacon's identity is already known and could be used to connect to the beacon directly. However, the PDA inquires for all Bluetooth devices available and tries to connect only to devices of class "Access Point." If no such device exists, this phase fails. The enrollment phase also fails if such devices exist, but no connection can be successfully made to any of them.
  - The PDA starts the high-level protocol with the connected device, whereby it requests and receives the user certificate. If the certificate transfer fails, or the certificate does not verify, the PDA closes the connection and restarts the previous step with another device as the prospective token.
  - The PDA tries to authenticate the prospective token through a challenge-response protocol, based on the certificate information. If the authentication exchange fails, the PDA retries the above steps with another device. If the authentication succeeds, the PDA concludes that the prospective token is the intended personal beacon, saves the certificate (i.e., as a flag attesting that enrollment completed successfully) and the communications address of the token (e.g., a MAC Address for Bluetooth) for subsequent use in the authentication phase, and closes the connection.

- *The authentication phase* – During this phase, the (client) PDA initiates a challenge-response exchange with the beacon and checks that its certificate remains in effect. The authentication succeeds if and only if the verification is successful. This phase takes place periodically to ensure that the beacon is present and enabled.

### *Challenge-Response Protocol*

The underlying mechanism used to authenticate users via a personal beacon relies on a challenge-response protocol compliant with FIPS 196 between the device and the beacon. The PDA challenges the beacon for an appropriate and correct response that can be used to verify that the token is the one originally enrolled by the device owner. The PDA relies on credential information, obtained earlier from the beacon when the PDA owner initially enrolled the beacon with the device.

Figure 2 illustrates a typical exchange between the PDA and beacon. The upper part of the diagram shows the enrollment information exchange used to register a token (at right) with the PDA (at left), while the remainder shows the exchanges used to verify the claimed identity.

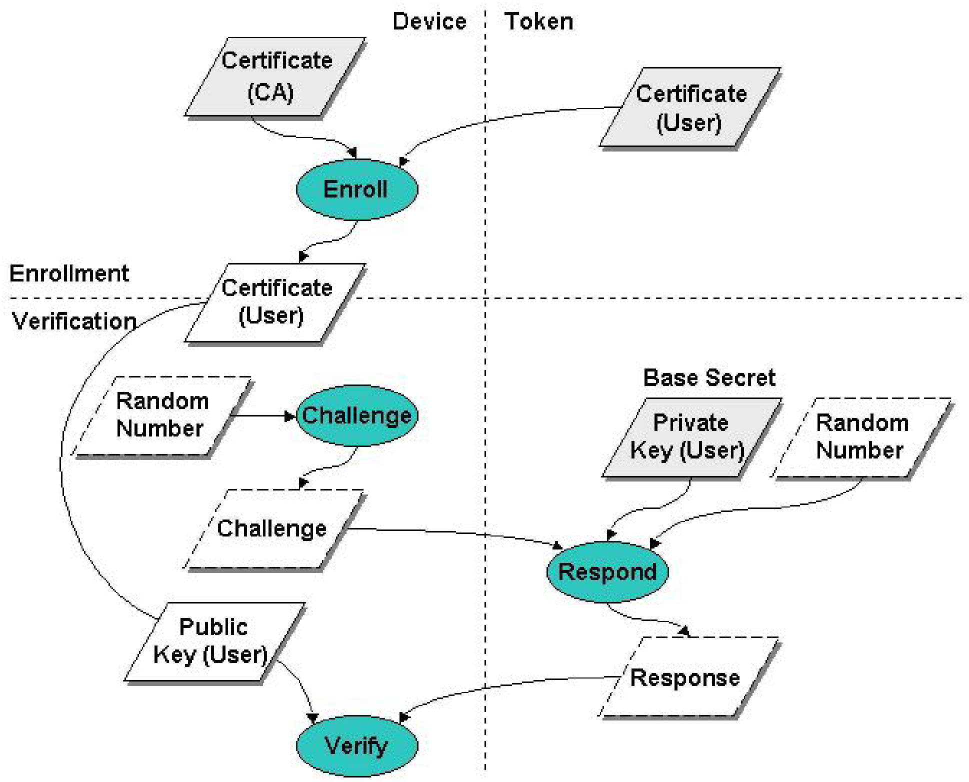

**Figure 2: PDA and Beacon Exchange**

Before the personal beacon token can be enrolled at the device, it must first be issued to the user. A security administrator populates the token with the user's credentials and other needed information. Those credentials are issued by a Certification Authority (CA) and validated using the certificate of the CA. Once the credentials are validated, they are retained at the device and used to verify the user's identity following FIPS 196 procedures.

For verification, the device and the token adhere to the following protocol [FIPS196]:

- The device, acting as the verifier, generates a random challenge "B" and passes it to the token for signing with the private key associated with the enrolled identity certificate;
- The token, acting as the claimant, generates a random value "A", signs A||B with its private key ("||" denotes concatenation), and returns A and the signature to the PDA;
- The device retrieves the enrolled identity certificate, verifies it, then verifies the token's signature over A||B using the public key in the certificate;
- If everything successfully verifies, authentication succeeds; otherwise, the authentication attempt fails.

The authentication of an entity depends on two things: the verification of the claimant's binding with its key pair, and the verification of the claimant's digital signature on the random number challenge. Using a private key to generate digital signatures for authentication makes it computationally infeasible for an attacker to masquerade as another entity, while using random number challenges prevents an intruder from copying a valid response signed by the claimant and replaying it successfully at a later time. Including a random number of the claimant in the response before signing it precludes the claimant from signing data that is solely defined by the verifier. The security of the FIPS 196 protocol also hinges on the generation of random numbers that have a low probability of being repeated.

#### *Bluetooth Pairing*

Bluetooth pairing is basically a process that consists of exchanging passkeys and setting up a trusted connection between the PDA and the personal beacon. The following steps are required to set up Bluetooth pairing between a PDA and the personal beacon:

- The PDA searches for Bluetooth enabled beacons in the area. The beacons must be set up to be discoverable when other Bluetooth devices search. During the discovery process, discoverable devices usually broadcast what they are (such as a beacon, a printer, a mobile phone, a handheld, etc.), and their Bluetooth Device Name. Depending on the device, its Device Name may be able to be changed to something more specific. If multiple Bluetooth devices are in range, and they are all discoverable, identification helps to select a specific beacon from other devices.

- Once the beacon is enrolled it can toggle off the discoverability setting, since the PDA retains the address of the beacon. When discoverability is off, the beacon does not respond when other devices search for it. Undiscoverable devices can still communicate with other Bluetooth devices, but they must initiate the communications themselves, if not paired with the device.

- After selecting the beacon, the PDA prompts for a passkey or PIN, which is shared by both devices to prove that their respective owners agree to be part of the trusted pair. With more advanced devices, such as mobile phones, both participants must agree on a passkey and enter it on each device. With other types of devices, such as hands-free headsets, where no interface exists for changing the passkey on the device, the passkey is fixed. For such devices, their associated documentation provides the default passkey, and how to change it, if possible. Often, the passkey is simply zero.

- Once the passkey is entered on the PDA, it is sent over to the beacon for comparison. If the beacon is an advanced device that needs the user to enter the same passkey, it asks for the passkey; otherwise, the beacon uses its standard, fixed passkey. If the beacon's passkey is the same as that entered by the PDA, a trusted pair is formed. Each device automatically accepts communication from the other, bypassing the discovery and authentication process that normally happens during Bluetooth interactions.

## Safeguards

For user authentication, the fundamental threat is an attacker impersonating a user and gaining control of the device and its contents. Tokens such as the personal beacon should be designed to resist physical tampering and avoid disclosing its base secret, the private key used to sign challenges it receives. Presuming the design and implementation are effective, the following vulnerabilities are the main candidates for exploitation:

- The authentication mechanism can be bypassed
- Weak authentication algorithms and methods are used
- The implementation of a correct and effective authentication mechanism design is flawed
- The confidentiality and integrity of stored authentication information is not preserved

The Personal Beacon handler uses the challenge-response mechanism described in FIPS 196 in order to authenticate the personal beacon token. In signing the challenge and verifying the signature, the handler and the token use OpenSSL v0.9.7APIs that comply with the PKCS #1 standard. The private key held in the token should be used exclusively for authentication.

The Personal Beacon authentication mechanism embodied in the handler relies on MAF, which in turn relies on the security of the underlying operating system implementation. The handler is protected from substitution and overwrite respectively through the multi-mode authentication and policy enforcement functionalities of MAF. Substitution is prevented through an entry in the list of registered handlers (e.g., </usr/bin/handlerPB 2>) identifying its location, while overwrite is prevented through policy rules in the MAF policy file (/etc/MAF/defaultPolicy).

The personal beacon handler uses the following data files stored on the PDA, which must also be protected through policy enforcement functionality of MAF:

- /etc/MAF/cacert.pem: contains the X.509 certificate of the root CA that issued the user's certificate on the token. This file is installed on the device through security administration.
- /root/Settings/PBcert.pem:[2] contains the user's X.509 certificate token. This file is written at token enrollment, and afterwards it is read only when the handler is restarted.
- /root/Settings/PBaddr.txt:[3] contains the token's Bluetooth address. This file is written at token registration, and afterwards it is read only when the handler is restarted.

The MAF policy file (/etc/MAF/defaultPolicy) must contain the following policy rules that grant exclusive permission to the handler to read/write these files at any security level and also prevent the handler from being overwritten [Jan03b]:

- <file /etc/MAF/cacert.pem /usr/bin/handlerPB 0>
- <file /root/Settings/PBcert.pem /usr/bin/handlerPB 0>
- <file /root/Settings/PBaddr.txt /usr/bin/handlerPB 0>

Policy rules specifying access to the Bluetooth stack are not yet available.[4] If device pairing would be used, then the binary file /etc/bluetooth/link_keys, where the BlueZ Unix Bluetooth stack appends the symmetric key used in pairing, should also be protected by the following rules:

[2] For the NFM variant the file name of the token's certificate is NFMcert.pem.
[3] Currently no counterpart file name is required in the NFM variant, since the addresses are not exposed in the evaluation data boards used in the implementation.
[4] For the NFM variant, the MAF policy file must instead contain policy rules that grant handlerNFM the permission to access the serial interface to the evaluation data boards, from any policy level: <interface serial /usr/bin/handlerNFM 0>.

- <file /etc/bluetooth/link_keys /usr/bin/handlerPB 0>
- <file /etc/bluetooth/link_keys /usr/sbin/hcid 0>[5]

## Handler Implementation

The Personal Beacon authentication handler operates as a polling handler, periodically checking the status of the token, as well as initiating authentication with it. The following code excerpt shows the main execution loop of the handler.

```
crtBeaconState = 0;
...
while (1)
{
    previousState = crtBeaconState;

    kernelResponse = HandlerReady(11);
    if (kernelResponse == mmPoll)
    {
        crtBeaconState = Authenticate(previousState);
        if (enrollfailed) continue;

        if (crtBeaconState!= previousState)
        {
            if (!crtBeaconState)
            {
                TellUI("PB:shw:Authentication failed");
                sleep(2);
                TellUI("PB:clr:");
                TellKernel("AUTH-FAIL");
            }
            else
            {
                TellKernel("LEVEL 1");
            }
        }
    }
    else    // kernelResponse = mmAuthenticate
    {
        crtBeaconState = Authenticate(previousState);
        if (enrollfailed) {
            TellKernel("AUTH-OK");
            continue;
        }
        TellKernel(crtBeaconState? "AUTH-OK" : "AUTH-FAIL");
    }
}
```

The variable crtBeaconState stores the beacon current state as reported by the Authenticate() function. The state "authenticated" (1) means the beacon is on and the handler has successfully authenticated it. The state "not authenticated" (0) means no beacon is present, or the beacon authentication has failed.

[5] This rule lets Bluetooth use the link key when it connects, since the previous rule prevents any other library from reading the keys.

The variable previousState maintains the beacon state as detected at the previous polling or authentication moment.  Comparing the two state variables allows the handler to detect a change in the beacon state from one polling moment to the next.

After saving the previous beacon state in previousState, the handler tells the kernel that it is a polling handler by calling the **HandlerReady()** function with the polling interval of 11 seconds.[6] Both the polling and the authentication procedures are performed by the same function, **Authenticate()**, whose code is listed below:[7]

```
int Authenticate(int previousState)
{
    int res, dev, s;
    static int failedattempts = 0;

    now = time(NULL);
    if (kernelResponse == mmAuthenticate)
    {
        if (now < lastTry) lastTry = now;
        if (now - lastTry < AUTH_TRY)
        {
            // Too soon to try authenticate again
            return 0;
        }
    }

    // If last successful authentication was less than AUTH_CONFIRM
    // seconds ago, assume it's still valid.
    if ((now - lastAuth) < AUTH_CONFIRM)
    {
        // Consider authentication still valid
        return 1;
    }

    // If token not registered
    if (!TokenRegistered())
    {
        if (kernelResponse == mmPoll) return 0;
        res = RegisterAndAuth();
        if (res) lastAuth = now;
        return res;
    }
    lastTry = now;

    // Verify certificate if more than CERT_VERIF seconds
    // passed since last verification.
    if (now - lastVerif > CERT_VERIFY)
    {
        res = VerifyCert(pcert);
        if (res == 0) {
```

[6] This is a default value; a different polling interval can be specified in a configuration file, as shown later in this section. For the NFM variant, a 5 second interval is used.
[7] The Authenticate() function for the NFM variant functions similarly, but has some slight differences.

```
            errmsg = "Invalid certificate";
            return 0;
        }
        lastVerif = now;
    }

    // Connect to the stored token address.
    res = TryToConnect(-1, &s);
    if (res == 0)
    {
        if (!previousState) return 0;
        if (++failedattempts > maxfailedattempts) return 0;
        return 1;
    }

    // Authenticate.
    res = TryToAuth(s);
    if (res == 0)
    {
        if (!previousState) return 0;
        if (++failedattempts > maxfailedattempts) return 0;
        return 1;
    }
    failedattempts = 0;
    lastAuth = now;
    return 1;
}
```

First, this function refuses to perform authentication and returns the state “not authenticated” if the last try was less than AUTH_TRY (about 4 seconds). The reason behind this is the following: In the case of a failed authentication with the handler associated with level 1, the policy level drops to 0, and the kernel tries to raise it to 1, instructing the handler to authenticate the token. If the authentication fails again, this process repeats itself, flooding the token with connection requests. Limiting authentication attempts to only every 4 seconds or so, avoids such flooding.

The next few lines after that show that **Authenticate()** considers a successful authentication to be still valid for at least the AUTH_CONFIRM period (about 20 seconds), to avoid costly connections and data transfers.

If the personal beacon token is not yet registered, **Authenticate()** tries to enroll and authenticate the token by calling the function **RegisterAndAuth()**. If the token is registered, **Authenticate()** continues by verifying the user certificate (but not too often), in order to detect the eventual expiration of the certificate. No revocation information is available to the handler.

Finally, **Authenticate()** tries to connect to the token, by using its stored Bluetooth address (for the case where the token is already enrolled and its address has been stored in the handler’s file system), then to authenticate. If the authentication is successful, the handler saves the time of this authentication. Otherwise, if the token is currently authenticated (as reflected by the parameter previousState), a limited number of failure attempts are allowed, before declaring the authentication unsuccessful. The maximum number of failed attempts is by default 2, but it can be set to another value in a configuration file, /root/Settings/Pbconf.txt, using the format "key=value". The same configuration file can be used to specify the polling interval (which by default is 11 seconds), as the following entries illustrate:

- maxFailedAttempts=1
- pollingInterval=10

## Token Implementation

The personal beacon operates as a server to the client authentication handler on the PDA. The personal beacon itself is simulated on the display of an iPAQ PDA. Its code executes as an OPIE application whose graphical user interface is shown in Figure 3. The virtual token pictured represents a key fob form factor, containing two LEDs and an on/off switch. The actual token could have formats other than a key fob. A command line C-language version for PDA/laptop/desktop computers is also available.

The beacon has three internal states: "off", "on and unconnected", and "on and connected". The on/off switch is used to power on or off the beacon and transition between the "off" and consolidated "on" states. The state is reflected by the color of the left LED – green means on, yellow means off. Turning the switch on causes the beacon to start listening for connections and the LED at left to change color. Once a connection is established, the right LED blinks whenever an authentication exchange occurs. The switch itself is labeled "On" or "Off" to indicate whether pressing it will cause the beacon to start up or shutdown respectively.

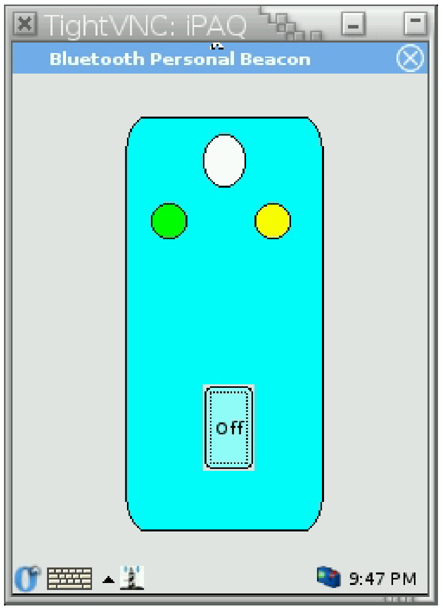

**Figure 3: Simulated Personal Beacon**

### *Procedural Steps*

The Personal Beacon starts up by initializing the OpenSSL libraries and reading the user PKI credentials from files in PEM format (the private key is protected with a password, which for now is hard-coded). These files are located in the $HOME/Setting/ directory, and their name is prefixed with PB or NFM (depending on the variant), where $HOME denotes the real user's home directory.

In the next step, the Personal Beacon builds and displays its interface according to the beacon's initial state (off, with both LEDs off and the On/Off button displaying "On"). When the user clicks on the "On" button, the personal beacon creates an L2CAP server socket, enters the "on" state (unconnected), and starts listening for connections from remote devices.[8] It repeatedly polls the server socket for connections with a timeout of 0.5 seconds. When it detects a connection request, the beacon accepts the connection, creates a client socket, and enters the "on connected" state.

In this new state, the beacon repeatedly polls the client socket for input requests from the device, and the server socket for new connections. The beacon processes the input received on the client socket with priority. If a new connection request arrives on the server socket, the old connection is closed and a new one is established.[9]

Input on the client socket is interpreted as a request from the client PDA. The server processes the request and returns the answer on the client socket. The next section describes the high level protocol between the client PDA and the Personal Beacon. The last request from the client PDA in a session should be "bye"; the server will close the connection with the client PDA.

When the user clicks on the button labeled "Off", the server closes any open connection and repaints its GUI so that the LEDs reflect the beacon state.[10] Whenever the user suspends the PDA on which the Personal Beacon runs, the following script located in the /etc/suspend-scripts/ directory automatically terminates the server:[11]

```
#!/bin/sh
LOGNAME=root
QTDIR=/opt/QtPalmtop
export LOGNAME QTDIR
/opt/QtPalmtop/bin/qcop "QPE/Application/serverPB" "close()"
```

Note that "serverPB" or "serverNFM" should be the name of the application executable. Using the absolute path of qcop can be avoided by adding its location to the PATH variable and exporting PATH in the script.

[8] For the NFM variant, the personal beacon instead opens the serial port /dev/ttyC0 on the supporting PDA, saves the port's current parameters, and configures the serial port with the following parameters: 9600 bps, 8 data bits, no parity, 1 stop bit. Note that the getty process running on the PDA may interfere with the personal beacon trying to read from the serial port and needs to be disabled in /etc/inittab.
[9] Instead of waiting for commands from the client (the authenticating PDA) arriving on the serial line, in the NFM variant the server starts a timer that issues timeout signals to repeatedly call a function that processes the input coming from the client on the serial line. That function polls the serial port file descriptor and returns after 0.5 seconds if no input is available during that interval. If there is some input, the server repaints the GUI with the right LED turned on (green) for about 0.2 seconds (a "single shot" timer is used to turn off the led after that interval). The server reads and processes the input data as a server command, and sends an answer back to the client.
[10] For the NFM variant, the server stops the timer that was issuing the timeout signals to read input.
[11] For the NFM variant, serverNFM replaces serverPB in the last line of the script.

***Personal Beacon Client/Server Protocol***

The high-level protocol used by the authentication handler and the personal beacon server comprises a few commands. Their descriptions and the request to and responses from the server are described below.

- *getCertLength* – The command requests the length of the user's certificate in bytes. The server maintains the certificate in the PEM format. The server returns the certificate length as a decimal value.

- *getCertData|offset|length* – The command asks for a chunk of the certificate, starting at the indicated offset and for length bytes long. The offset and length are expressed in decimal. The server should return a string of length bytes.[12]

- *signChallenge|challenge* – The command asks the server to sign a challenge, which is a string (called B) of 16 bytes randomly generated by the client. When issuing the command, the client must translate each byte of the challenge into two characters representing its hexadecimal digits, resulting in a 32-byte string for the challenge. The server returns the results of the signing operation, which is a 289-character string obtained as follows:[13]
  - The server generates a random string A of 16 bytes and translates each byte into two characters representing its hexadecimal digits.
  - The server signs (A || B) using the user's private RSA key, and translates each byte of the 128-bye signature into two characters representing its hexadecimal digits.
  - The server concatenates the 32-byte representation of A, the character "|", and the 256-byte representation of the signature. This is the result of the signing operation, and it has 32 + 1 + 256 = 289 characters.

- *bye* – The command requests the server to disconnect from the remote device by closing the client socket, resulting in the beacon state maintained at the client to become "on unconnected".[14] The server returns the string BYE.

The following is an example of the client/server dialog conforming to this protocol, as seen from the server side:

[12] For the NFM variant, to avoid sending control characters over the serial line, the server translates every certificate byte into two characters representing its hexadecimal digits. Thus, the command actually returns a string of 2*length characters, each being a hexadecimal digit. For example, the character '7' (i.e., the byte with the value 0x37) is returned as a string of two characters "37".

[13] The challenge signing protocol for the NFM is slightly different. Instead of returning the results of the signing operation, the command returns "ok" if signing succeeds or "Error: errmsg" if signing fails (the returned string does not contain quotes). The server stores the result of the signing operation internally, which can then be retrieved via additional commands: getSignLength and getSignData
getSignLength - The command asks the server for the length in bytes of the result of the signing operation. It should return 289 in decimal (see the command signChallenge).
getSignData|offset|length - The command returns a chunk of the signChallenge result, of specified length and starting at the specified offset.

[14] The NFM variant does not use the bye command, because of the characteristics of the serial connection.

```
getCertLength
1472
getCertData|0|128
...
getCertData|128|128
xCzAJBgNVBAYTAlVTMREwDwYDVQQI...
...
getCertData|1408|64
...
signChallenge|8120D79AD6231DF4C7FB2C2DD924036B
1E03120C57AED6555C9705EA32422D33|51BBBF778989B5D1A319BB9ABDA5504D...
bye
BYE
```

## Organizational Beacon Authentication

The organizational beacon is a small device that is placed in an area to establish a perimeter where a distinct policy is in effect. To accomplish this, the organizational beacon offers an area location service for discovery and use by PDAs and other mobile devices. One or more organizational beacons define the area. Location is determined relative to a beacon. Mobile devices equipped with an organizational beacon authentication mechanism sense the locale of the organizational beacons and adjust their security policies accordingly. A device is either in or out of the vicinity of the beacons, as determined by the footprint of their communications signal.

The organizational beacon authentication mechanism periodically checks for proximity to a beacon and reports successful authentication if a beacon is detected and able to be verified; otherwise, it reports failure. Multiple organizational beacons can be used to improve service above that of a single beacon, or arranged to service a larger area. An organizational beacon provides credential information for a PDA to verify using the Transport Layer Security (TLS) protocol over Bluetooth. Many mobile devices are manufactured with built-in Bluetooth radios, which allow short-range communication and have low power consumption. The solution could also be adapted for other types of wireless PAN communications technologies.

Intrinsyc CerfCubes serve as the platform for the prototype organizational beacons.[15] The CerfCube 255 includes a PXA255 microprocessor, 32MB Flash ROM, and 64MB SDRAM, in a convenient 3" x 3" x 3" form factor. It comes loaded with a Linux kernel and the Familiar Distribution, including device drivers for all on-board peripherals. Peripheral support includes Ethernet and several serial ports (one exposed). CerfCubes come equipped with a Compact Flash connector that supports Type I and II cards, and can be used to add Bluetooth, WiFi, or wireless WAN communications, local storage, etc.

### Operation

The organizational beacon authentication mechanism operates in two distinct modes: unauthenticated and authenticated. The mobile device is pre-configured with the specific policy settings that are applied in both the presence and absence of a beacon. In the unauthenticated mode, the following steps occur:

- The mobile device periodically scans for the available organizational beacons in the area.
- When the mobile device finds a prospective beacon, it establishes the wireless connection to the beacon, and then tries to set up a secure TLS connection over that physical channel, using the X.509 certificate supplied by the beacon.
- If the beacon is successfully authenticated and a TLS connection established, the mobile device enters a readiness exchange with the beacon to verify that it is indeed a functional organizational beacon.
- Once the mobile device verifies that the beacon is functional, it enables the policy on the device for that location and switches to the authenticated mode.
- Otherwise, the mobile device blacklists the beacon for a period of time and retries the above steps.

[15] More information can be found at http://www.intrinsyc.com

Once in the authenticated mode, the following steps occur:

- The mobile device periodically tries to reestablish a physical and TLS connection with the last beacon to which it successfully authenticated.
- If the beacon is again successfully authenticated and a TLS connection established, the mobile device verifies that the beacon is functional.
- Once the mobile device successfully verifies that the beacon is functional, it maintains the associated policy on the device for that location and remains in the authenticated mode.
- Otherwise, the mobile device retries the above steps again, allowing for a momentary out of range condition.
- If the beacon cannot be successfully authenticated and vetted within a preset time period (approx. 2 – 3 minutes), the mobile device switches to the unauthenticated mode and changes policy accordingly.

The beacon operates as a server to the mobile device client, listening to the inquiries from mobile devices and responding as needed. The software on the beacon interacts with a mobile device only in the second and third steps of the unauthenticated mode and in the first and second steps of the authenticated mode. The rest of the activities (e.g., responding to inquiries, establishing Bluetooth connections, etc.) are performed entirely by its Bluetooth hardware. The range of the beacon may also be tied to the capabilities of the Bluetooth hardware to increase or decrease the strength the radio signal.

Beacons support specific types of policy, denoted by an identifier in their credentials. Assorted beacons may be configured to support distinct policies for different areas. A mobile device, running the client side of the organizational beacon authentication mechanism, is configured to observe a specific policy in the presence of an associated beacon and disregard beacons that identify other policies.

## Safeguards

The authentication mechanism must ensure that the messages it receives from a beacon have been created recently for the particular purpose intended and by the beacon claiming to have sent it. The mechanism must be able to detect when a message has been modified or forged by an attacker with access to the wireless network, or when a message issued previously (or for a different purpose) is being replayed on the network by an attacker.

The security of organizational beacon relies on the TLS protocol and MAF. The TLS protocol provides the assurance that the beacon is genuine. The security of the TLS protocol is based on the challenge response mechanism and public key cryptography. The protocol is widely accepted by the Internet community and is currently considered secure for financial transactions. As with the personal beacon, the OpenSSL library (version v0.9.7) was used for cryptographic functions and the TLS implementation. The OpenSSL is a widely accepted implementation of cryptographic functions and the TLS protocol, under constant scrutiny by the open source security community.

The authentication mechanism assumes that beacon is physically secure and situated at the correct location it identifies. The policy enforcement functionality of MAF is used for the protection of sensitive files. The substitution of the handler program is prevented through the entry in the list of registered handlers (e.g., </usr/bin/handlerBB 3>). The following rules prevent overwrite of the handler and also grant it exclusive access to the CA's public key certificate and the governing policy identifier adhered to by the handler [Jan03b]:

- <file /etc/MAF/cacert.pem /usr/bin/handlerBB 0>
- <file /etc/MAF/OB-PolicyID /usr/bin/handlerBB 0>

Blocking access to the CA's public key certificate and the governing policy identifier prevents an attacker from substituting them with ones from a different organization to gain unauthorized access to the mobile device.

## Handler Implementation

The organizational beacon handler is a user space program that runs on the mobile device. It communicates with the MAF mechanism in the kernel and the remote device that claims to be a Bluetooth organizational beacon. The remote device must prove to the Bluetooth MAF handler that it is a legitimate organizational beacon by successfully establishing a TLS channel, using its X.509 certificate signed by the organization certificate authority. If the handler determines that the remote device is indeed a legitimate beacon associated with a policy level it protects, the handler tells the kernel that a successful authentication has occurred, allowing the kernel to activate the policy if all other required conditions are met. Otherwise, the handler continues to search for a legitimate beacon.

The organizational beacon handler is a polling handler, which means that it instructs the kernel to make periodic contact, awakening it to perform the necessary operations. As explained earlier, the handler has two modes of operation: authenticated and unauthenticated mode. In the unauthenticated mode, the handler periodically performs a Bluetooth inquiry to find prospective beacons. If inquiry process results in finding a Bluetooth device with the class of Access Point (i.e., 0x082311), the handler attempts to establish L2CAP connection to the predetermined Protocol Service Multiplex (PSM) (i.e., a designator similar to a TCP/IP port number). When the L2CAP connection is established, the handler tries to set up a TLS session over this connection and verify that the organization's certificate authority signed the X.509 certificate used by the server to set up the TLS session. If the signature verification succeeds, the handler switches to the authenticated mode and signals the kernel that it can raise the current policy level.

When the handler runs in the authenticated mode, it periodically tries to establish connection with the last known beacon and authenticate the beacon using the same authentication steps as in the unauthenticated mode. If the handler is unable to communicate successfully and verify the beacon during a 2-minute interval, it switches to the unauthenticated mode and signals the kernel that the authentication is no longer valid.

An excerpt of the main loop of the handler is provided below. Before each iteration of the loop the handler records its current authentication state and then calls the **HandlerReady** function.

The **HandlerReady** function is a part of MAF API; it puts the handler in the suspended state for between 3 and 8 seconds or until the user requests authentication. When the handler resumes its operation, the return code **result** contains the reason code for why it is being resumed. If the **result** equals “mmPoll,” it means that the handler was awakened for the periodic status check. In this case, the handler checks the current authentication is still valid by calling the **Authenticated** function. This function returns TRUE if the authentication is still valid or FALSE if it no longer authenticated. The handler then compares the current and last known authentication values. If the values are different and the new authentication state is negative, the handler sends the “AUTH-FAIL” signal to the kernel. If the values are different and the current value is positive, the handler sends the “LEVEL 1” request to the kernel, which signals the kernel to attempt a transition to the policy level of the handler.

Regardless of whether an authentication state changed or not the handler executes the **Worker** routine. This routine confirms that the current beacon is still in the vicinity or performs a search for a new beacon.

If the reason for resuming the handler was not “mmPoll,” it means that the kernel wants the handler to return the current authentication status. The handler calls the **Authenticate** routine to determine the current status of the handler, which sends “AUTH-OK**”** or **“**AUTH-FAIL**”** message accordingly.

```
while(1)  {
 int result;

 // This is needed to know if the beacon state has changed
 lastState = beaconState;

 // Suspend the handler…
 result = HandlerReady ( 3 + rand() % 5 );


 // What was the reason we got woken up
 if ( result == mmPoll )
   {

     /// Check the authentication status
     beaconState = Authenticated ();


     /// Has the state changed??
     if ( beaconState != lastState )
       { puts ( "Beacon State Changed" );
       if ( !beaconState )
       {
         // Authentication no longer valid send the messages
         // to the kernel
         TellKernel ( "AUTH-FAIL" );
       }
       else
       {
         /// Since the message comes from the kernel...
         /// the numeric level value is ignored...
```

```
           /// so we can put any number we want.. but we
           /// there must be a number here otherwise kernel
           /// will not accept this message
           TellKernel ( "LEVEL 1" );
         }
         }
       // Perform periodic maintenance
      Worker ();
       // Restart the loop
       continue;
     }
   // We got woken up because kernel wants us to authenticate
   TellKernel ( Authenticated () ? "AUTH-OK" : "AUTH-FAIL" );

  }
}
```

The code for the **Authenticated** function is given below. The function first obtains the current time value and then compares how long a period passed since the last successful communication with the beacon. If it took more then a 100 seconds, the function returns negative authentication; otherwise, it returns positive authentication

```

int Authenticated ( )
{
  Now = time(NULL);
  if ( (Now - LastAuth) < AUTH_TIMEOUT )
   return TRUE;
  else
   return FALSE;
}
```

***Beacon Table***

The handler maintains a table of prospective beacons to carry out its function. The table contains the information about all Bluetooth devices in the vicinity of the mobile device. The table has the following fields: MAC Address, Last Seen, Last Contact, and Status. An example of such a table is shown below.

**Table 1: Prospective Beacon Table**

| MAC Address | Last Seen | Last Contact | Status |
|---|---|---|---|
| 00:02:92:21:AB:C8 | 20 | 20 | Beacon |
| 00:22:11:22:33:11 | 30 | 30 | Not Beacon |
| 00:22:99:11:11:11 | 20 | 20 | Unknown |

The MAC Address field contains the address of the Bluetooth device, while the fields Last Seen and Last Contact contain the time value of when the device was last seen and when the last successful communication with the device took place. The Status field contains the handler's idea of the device's purpose. The Status field can be one of the following: "Beacon," "Not

Beacon," and "Unknown." When the remote Bluetooth device is initially entered into the table, it is assigned the "Unknown" status. Later, when a successful communication with the remote device takes place, the device is assigned the "Beacon" status. If the handler can establish a connection to the remote device, but the device does not follow the beacon readiness protocol, the device is assigned the "Not Beacon" status.

The handler populates the table by performing a Bluetooth inquiry process every 50 seconds. The inquiry discovers Bluetooth devices in the vicinity and returns a list of their MAC Addresses. The handler looks up each MAC Address received during the inquiry process to see if it already exists in the beacon table. If the address does not exist, it is entered into the table. For every MAC Address received during the inquiry process, the handler updates the corresponding Last Seen entry in the handler table.

When the handler is not doing an inquiry, it tries to contact the devices in the beacon table whose status is either "Beacon" or "Unknown." The devices with "Beacon" status are contacted before the devices with "Unknown" status. During the contact, the handler first tries to establish the L2CAP connection to the remote device. The Last Contact value is updated before every attempt to establish an L2CAP connection is made. If the connection succeeds, the handler performs the TLS exchange described in the section below. If a failure occurs after the L2CAP connection has been established, the handler sets the Status field of that beacon to "Not Beacon," which temporarily blacklists the beacon. If the TLS exchange results in the successful authentication, the handler sets the Status to "Beacon," sets the lastAuthentication variable to the current time, and does not try to contact the other devices in the table.

The lastAuthentication variable is used to determine whether the current authentication is still valid. If the time value stored in this variable is less then 120 seconds before the current time, the handler considers the state to be unchanged, remaining valid. When the kernel sends an authentication request to the handler, the handler checks the current time and the value of the lastAuthentication variable and returns the positive response if the value is within 120 seconds of the current time, or otherwise responds with negative authentication.

The Beacon table is periodically swept for stale entries. If the handler sees an entry with the Last Seen value older than 60 seconds, the entry is removed from the table. The handler uses the Last Contact column in conjunction with the Status column to prevent permanent blacklisting of beacons that did not correctly follow the beacon readiness protocol previously. For example, it could be the case that the beacon was just starting up and not all the software was fully operational and able to complete the exchange. When the Status column for a particular entry contains a "Not Beacon" value and the Last Contact time value is older than 20 seconds, the handler changes the Status value to "Unknown." The above process is implemented by the **Worker** function, which is called from the main loop. The code for the **Worker** function is shown below.

```
// This is a worker thread...
void Worker ()
{
  Now = time(NULL);

  printf ( "Time Now is %d \n", Now );
```

```
if ( (Now - LastAuth) < AUTH_TIMEOUT )
  {

    if ( (Now - LastAuth) > AUTH_CONFIRM )
 {
   int res;
   puts ( "Have to re-authenticate" );
   LastAuthAttempt = Now;

   printf ( "Last known beacon %d\n", LastKnownBeacon );

   if ( LastKnownBeacon>=0 && LastKnownBeacon < N_BEACONS )
     // if ( BeaconTable[LastKnownBeacon].State == sBeacon )
       {
       puts ( "Last known beacon is still there" );
       res = Authenticate ( LastKnownBeacon );
       UpdateBeaconEntry(LastKnownBeacon,res,Now,&LastAuth);
       }
 }
    else
 puts ( "Authentication Still Valid" );
  }
else
  {
    int res = 0;
    int i;
    puts ( "Not authenticated" );
    /// Time To Do Inquiry?
    if ( (Now - LastInquiry) >  INQ_INTRVL )
 { puts ( "Time To Do Inquiry" );
 PerformScan ();
 LastInquiry = Now; }

    /// Attempt to connect to beacons..
    for ( i = 0 ; i < N_BEACONS ; i ++ )
 {
   if ( BeaconTable[i].State != sBeacon ) continue;
   res = Authenticate ( i );
   UpdateBeaconEntry(i,res,Now,&LastAuth);
   if ( res == 1 ) { LastKnownBeacon = i; break;}
 }


    /// Attempt to connect to unknown devices if the previous loop
    /// did not yield any good results..
    if ( res != 1 )
    for ( i = 0 ; i < N_BEACONS ; i ++ )
 {
   if ( BeaconTable[i].State != sUnknown ) continue;
   res = Authenticate ( i );
   Now = time(NULL);
   UpdateBeaconEntry(i,res,Now,&LastAuth);
   if ( res == 1 ) { LastKnownBeacon = i;break;}
 }
  }
  // Check the current
  Sweeper ();
}
```

### *TLS Protocol*

Bluetooth organizational beacon handler uses the TLS protocol [RFC2246] to authenticate prospective beacons. The TLS is a well-established and carefully scrutinized protocol for secure transactions. In the current implementation, both the mobile device and organizational Beacon use OpenSSL library (www.openssl.org) to provide the protocol functionality. Authentication of the beacon is performed as a part of the initial TLS handshake.

The server and handler use two basic I/O (BIO) object pairs provided by the OpenSSL library. Each BIO object pair contains both a source and a sink stream object. The first BIO object pair corresponds to the high level of the TLS connection where unencrypted text is submitted and the decrypted text is received. The second BIO object pair is a low level pair that conveys encrypted text and TLS-specific messages.

Both the server and handler portions of the organizational beacon code manage the Bluetooth specific aspects of the communication, such as establishing and tearing down connections, determining the Message Transmission Unit (MTU) size, etc., as well as actual data transmission. At every iteration, the source BIO object is asked whether it has any data to send. If such data exist, the organizational beacon code extracts the data packets making sure that they are smaller than the MTU size. Those packets are transmitted to the other side of the Bluetooth connection. The organizational beacon code then determines whether there is any incoming data on the Bluetooth connection. If such data is present, it is given to the sink part of the BIO pair. This process is repeated until the disconnection request is received from the higher level or the Bluetooth connection. The operation of the various protocol segments is illustrated in Figure 4.

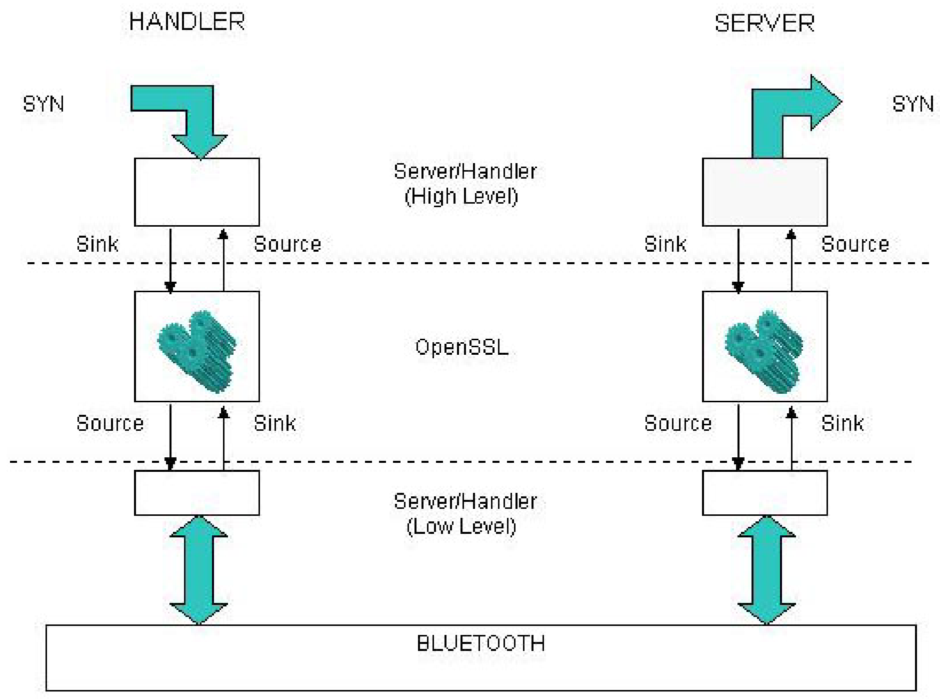

**Figure 4: TLS Over Bluetooth Protocol Stack**

When the TLS handshake is completed, the OpenSSL library executes a callback function provided by the handler code. The callback function examines the credentials that were used to establish the TLS session. The handler requires that the beacon uses an X.509 certificate signed by the certificate authority known to the handler. The callback function checks that this condition is satisfied and also checks the PolicyID against the policy ID stored on the device. If all these conditions are satisfied, the handler accepts the connection and allows the high level protocol to proceed. Otherwise, it issues a disconnection request and marks the remote device as not a beacon.

***Readiness Protocol***

The purpose of the beacon readiness protocol is to confirm that the TLS connection established is operational, and that the remote device is a functioning organizational beacon. The beacon protocol is a simple three-way handshake. Three messages communicate the different stages of the protocol: “SYN,” “ACK,” and “SYN-ACK.” Once the TLS connection is established, the handler sends the “SYN” message to the server and goes into a waiting state. When the server receives the “SYN” message, it responds with the “ACK” message and goes into the final stage.

The handler, receiving the “ACK” message, concludes that the session was completed successfully and sends the final “SYN-ACK” message, and then issues a disconnection request to the lower level. It also updates the lastAuthentication variable with the current time. If the handler does not receive an “ACK” message as expected it resets back to the initial state. When the server receives the “SYN-ACK” message, it also issues a disconnection request to the low level and resets itself into the initial state. If the server does not receive the “SYN-ACK” message as expected, it times out the connection and issue a disconnection request to the low level. When the low level segment receives a disconnection request, it empties all the sink objects, flushes all the transmission buffers, and drops the Bluetooth connection.

At the same time, the high level segment of the protocol runs a similar process. The source object of the high level BIO pair is examined to see if it has any outgoing data. In case the incoming data is present, the handler or server code tries to interpret it as an element of the beacon protocol. If it succeeds doing so, the code changes the internal state of the protocol machine, puts the response token into the sink object, or if it is a last token in the protocol it signals the low level to disconnect.

## Token Implementation

The Bluetooth organizational beacon program, referred to as the beacon server, is a user space program that listens to the inquiries from mobile devices, and responds to these inquires. The beacon server proves its identity to mobile devices, but it does not require mobile devices to do the same. The beacon server proves its identity by establishing the TLS channel between the beacon server and the mobile device using the private certificate located on the server. The beacon server's certificate must be valid and be issued by the organization's certificate authority (or by a certificate authority having a valid certificate chain from the organization's root certificate authority). The mobile device must hold the public key of the organization's certificate authority to verify the authenticity of the beacon server certificate.

The Bluetooth stack on the organizational beacon is configured to respond to incoming inquires and connections, known respectively as inquiry scan and page scan modes. The current implementation requires the device class identifier to be set to 0x082318, a unique identifier defined for beacon class devices. The new device class is used to improve the performance, by filtering out other types of devices that may be present in an area (e.g., cell phones, printers, etc.) and eliminating unnecessary connections to such non-beacon devices.

The beacon contains the server program that controls the authentication process. The beacon is a simple server that listens to the incoming L2CAP connections. Once such connection occurs the beacon server establishes TLS protocol connection and then observes its part of the beacon exchange protocol. The TLS connection is handled in exactly the same way as for the handler. The code for the main loop of the organizational beacon server is provided below.

```
     while(1)  {
       char BA[20];

       if ( (s1 = accept(s, (struct sockaddr *) &client_addr, &opt)) < 0)
    {
      perror("Error in accept call\n");
      exit(1);
    }

       ba2str ( &client_addr.l2_bdaddr, BA );

       printf ( "Connection from %s \n", BA );
```

```
// Authenticate device
      BeaconAuthClient ( s1 );
      close(s1);

    }
```

The beacon is a slave device that can only accept one connection at the time. Fortunately, the TLS exchange is fairly quick, significantly less than the Bluetooth connection time out. Therefore, two devices could easily connect during that period. For example, if two devices try to connect at the same time, the beacon picks the first device at random, processes the request, and then disconnects. Meanwhile, the second device keeps sending the connection request packets (Page Packets in BT terms). Once the transaction with the first device is complete, the beacon sees the connection request from the second device and processes it. A third device, however, would likely receive a connection time out and need to reattempt the connection.

## Summary

While mobile handheld devices provide productivity benefits, they also pose new risks associated with the information and network access capabilities they acquire over time. Robust user authentication provides safeguards against the risk of unauthorized use and access to a device's contents. This paper demonstrates how proximity-based authentication can be implemented as either a primary authentication method or a supplemental technique used in conjunction with another. The approach provides users the flexibility to perform their tasks unimpeded within the bounds set by an organization. The methods used depend on available PAN communications built into most handheld devices and require only a simple infrastructure of as few as a single proximity beacon.

## References

[FIPS196] Entity Authentication Using Public Key Cryptography, Federal Information Processing Standards Publication (FIPS PUB) 196, U.S. Department of Commerce, National Institute of Standards and Technology, February 1997, <URL: http://csrc.nist.gov/publications/fips/fips196/fips196.pdf>.

[Gru03] Marco Gruteser, Graham Schelle, Ashish Jain, Rick Han, Dirk Grunwald, Privacy-Aware Location Sensor Networks, USENIX 9th Workshop on Hot Topics in Operating Systems (HOTOS IX), May 2003, pp. 163-167, <URL: http://systems.cs.colorado.edu/Papers/Generated/2003PrivacyAwareSensors.pdf>.

[Haz04] M. Hazas, H.Scott, J. Krumm, Location-Aware Computing Comes of Age, IEEE Computer, 37, 2, February 2004.

[Hig01] J. Hightower, G. Borriello, Location Systems for Ubiquitous Computing, IEEE Computer, 34, 8, August 2001.

[Ind03] Jaga Indulska, Peter Sutton, “Location management in Pervasive Systems,” Workshop on Wearable, Invisible, Context-Aware, Ambient, Pervasive and Ubiquitous Computing, February 2003, Adelaide, Australia. In Conferences in Research and Practice in Information Technology series, Vol. 21, pp.143-152, <URL: http://www.itee.uq.edu.au/~peters/papers/indulska_sutton_wicapuc2003.pdf>.

[Jan03a] Wayne Jansen, Vlad Korolev, Serban Gavrila, Thomas Heute, Clément Séveillac, A Framework for Multi-Mode Authentication: Overview and Implementation Guide, NISTIR 7046, August 2003, <URL: http://csrc.nist.gov/publications/nistir/nistir-7046.pdf>.

[Jan03b] Wayne Jansen, Tom Karygiannis, Michaela Iorga, Serban Gavrila, Vlad Korolev, Security Policy Management for Handheld Devices, The 2003 International Conference on Security and Management (SAM'03), June 2003, <URL: http://csrc.nist.gov/mobilesecurity/Publications/SecurityPolicyManagementForPDAs-IEEEformat.pdf>.

[RFC2246] The TLS Protocol, Version 1.0, IETF Network Working Group, Request for Comments 2246, January 1999, <URL: http://www.ietf.org/rfc/rfc2246.txt>.

[War97] Andy Ward, Alan Jones, Andy Hopper, A New Location Technique for the Active Office, IEEE Personal Communications, Vol. 4, No. 5, October 1997, pp 42-47, <URL: http://www-lce.eng.cam.ac.uk/publications/files/tr.97.10.pdf>.