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

NISTIR 7046

A Framework for Multi-mode Authentication: *Overview and Implementation Guide*

Wayne Jansen
Vlad Korolev
Serban Gavrila
Thomas Heute
Clément Séveillac

NISTIR 7046

A Framework for Multi-mode Authentication:
Overview and Implementation Guide

Wayne Jansen
Vlad Korolev
Serban Gavrila
Thomas Heute
Clément Séveillac

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

August 2003

U.S. Department of Commerce
Donald L. Evans, Secretary

Technology Administration
Phillip J. Bond, Under Secretary of Commerce for Technology

National Institute of Standards and Technology
Arden L. Bement, Jr., Director

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
30 pages (2003)

Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

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. Enabling adequate user authentication is the first line of defense against unauthorized use of a lost or stolen handheld device. Multiple modes of authentication increase the work factor needed to attack a device, however, few devices support more than one mode, usually password-based authentication.

This report describes a general Multi-mode Authentication Framework (MAF) for applying organizational security policies, organized into distinct policy contexts known as echelons, among which a user may transition. The approach is aimed at helping users easily comply with their organization's security policy, yet be able to exercise a significant amount of flexibility and discretion. The design of the framework allows various types of authentication technologies to be incorporated readily and provides a simple interface for supporting different types of policy enforcement mechanisms. Details of the implementation of the framework are provided, as well as two example authentication mechanisms.

Table of Contents

Introduction 1

Overview 1

Top-Level Design 3
- Kernel Module 3
- User Interface Components 4
- Authentication Handlers 5
- Echelon Level Selector 5
- Interfaces 6
- Process Flow 8

Authentication Mechanism Details 9
- Non-polled Handler Example 10
- Non-Polled Handler Example User Interface 14
- Polling Handler Example 16

Kernel Details 20
- Functional Description 20
- Basic Interaction with an Authentication Handler 22
- Automatic Transition from Level 0 to Level 1 23
- Transitioning Between Echelon Levels 23
- Automatic Transition with Polling Handlers 23
- Penalties for Failed Authentication 24
- Processing of Successful Authentication 24
- Processing of Failed Authentication 24
- Manual Transition to a Level 24
- Ensuring All Handlers are Running 25

Appendix A – Handler Library 26
- Types 26
- Functions 26

Appendix B – Mandatory Access Control Settings 28
- Hardware Related 28
- Software Related 29

Appendix C – MAM Global Variables 30
- Types 30
- Global variables 30

A Framework for Multi-mode Authentication: Overview and Implementation Guide

Introduction

The purpose of this guide is to provide an overview of the design of a Multi-mode Authentication Framework (MAF) intended for handheld devices. It provides sufficient details for a developer to maintain the implementation, add new authentication mechanisms, and implement any needed extensions. The solution was implemented in C and C++ on an iPAQ Personal Digital Assistant (PDA), for a Linux operating system distribution from handhelds.org and the Open Palmtop Integrated Environment (OPIE), 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.

Overview

From an operational view, MAF is not only a facility for multi-mode authentication, but also a general framework for applying PDA security policies that involve multiple sets of rules organized into distinct policy contexts, known as echelons, among which a user may transition. For this discussion, we consider only single-user systems. That is, the user is the sole operator of the device, once it is issued.

Conceptually, echelons are graded into sensitivity levels, level 1 being the least sensitive and level 3 being the most sensitive. Level 0 represents the most restrictive policy for device lockdown. While the framework allows the user discretion in selecting among echelon levels at which to operate, it also can be used to impose one or more authentication steps, where needed, before permitting transition to higher sensitivity levels. Thus, authentication steps are cumulative and hierarchical – all lower level authentication steps plus those required at some desired level must be successfully completed to reach the desired level. While the implementation uses our own policy enforcement engine, its design allows it to interface with other policy enforcement engines that may be available, such as LIDS or SE Linux.

Figure 1 gives an example of a 3-echelon configuration. Each distinct set of policy rules and prerequisite authentication steps comprise an echelon level. For level 1, authentication steps 1A and 1B are needed, while for level 3, authentication steps 3A is needed, and the condition for authentications steps 1A and 1B must still hold. No additional authentication steps are needed to transition from level 1 to level 2. Transition among levels is initiated at the discretion of the user. Though echelon levels range from low to high to differentiate escalating sensitivity, hierarchical policy rules among are not a requirement.

The framework supports both polled and non-polled forms of authentication. Non-polled authentication is discrete; once the verdict is determined it is inviolate until the next authentication attempt. Examples of non-polled authentication include password, fingerprint, and voice verification. Polled authentication is continuous; the presence or absence of some token or signal determines the authentication status. Examples of polled authentication include

smart card, memory token, and communications signal, whereby the absence of the device or signal triggers a non-authenticated condition.

	Required Authentication	Effective Policy
Level 3	3A	Policy for L3
Level 2	None – user choice	Policy for L2
Level 1	1A, 1B	Policy for L1
Level 0	None – default at power on and boot up	Most Restrictive

Figure 1: Echelon Example

The transition flow between echelons for the example configuration is illustrated in Figure 2. On reboot and power on events, the system begins at the default level, level 0, and automatically attempts to transition the user to echelon level 1. Level 1 may require the successful completion of one or more authentication steps for entry. If any of the authentication steps fails, the user remains at level 0 and the system automatically reattempts to transition again to level 1. However, access is blocked for a period of time, as a penalty for failing to successfully transition there in the previous attempt. Similarly, if the user successfully transitions to level 1, but the status of a polled authentication step used to reach that level changes (e.g., 1B authenticated the user through a smart card, which is then removed), triggering a non-authenticated condition, she is returned to level 0. The system then automatically reattempts to transition her to level 1, after blocking access for a period of time.

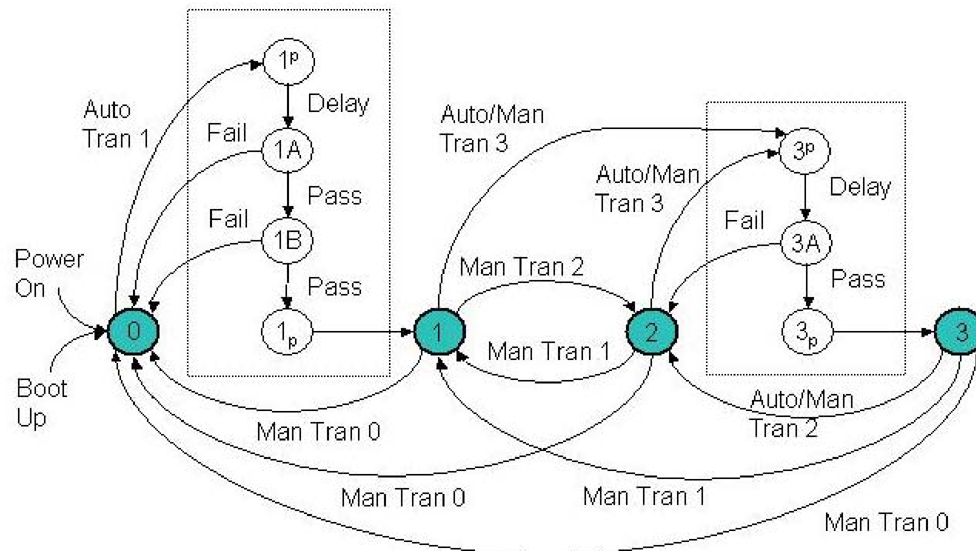

Figure 2: Transition Flow

Blocking access during a transition attempt for which an authentication failure previously occurred is done by bracketing the execution of those authentication handlers with pre- and post-handlers (e.g., 1^p and 1_p). The pre- and post-handler are a useful but optional technique for the framework. These pseudo authentication handlers work in tandem to allow authentication failures to be dealt with collectively by the framework, rather than individually by each authentication mechanism. The pre-handler maintains a penalty file where the number of failures is recorded. If the file does not exist, it creates the file and sets the value to 1, in anticipation of an upcoming failure. Otherwise, failures have previously occurred, and the pre-handler imposes a delay penalty commensurate with the recorded value, before incrementing the value by 1. The delay penalty can be programmed for linear, exponential, or whatever scheme the implementer chooses. If the authentication steps proceed through successfully, the post-handler deletes the penalty file when it executes to clear out the count.

Once the user reaches echelon level 1, transition among the remaining levels is done at her discretion. Downward transitions do not involve any authentication steps. It may require the successful completion of one or more additional authentication steps, however, to make the transition upwards to a higher level. The entire set of authentication steps from the current echelon level up to the requested level, including any intervening levels, are initiated in their logical sequence. If any one of the authentication steps fails, the user transitions to the highest level where she has successfully authenticated. For example, a user attempting to transition from level 1 to level 3 could attain level 2, should she successfully complete all authentication steps for level 2, but fail an authentication step for level 3. However, because the user unsuccessfully attempted a transition to level 3, access will be blocked for a period of time on the next transition attempt to that level. Note that the blocking is done selectively, permitting the user to continue operation at level 2, as the penalty time expires.

While the user typically initiates a transition among echelon levels, the framework also allows the possibility for polled authentication mechanisms to request that a transition attempt be initiated. For example, the presence of a smart card in the smart card reader could be made to trigger the system to attempt a transition to the level for which this authentication step is associated. This facility is fairly intuitive and appropriate for most authentication mechanisms; however, it could be troublesome for tokens that are not under the control of the user. For example, if the presence of a signal broadcast from a trusted beacon is used as one of the authentication steps for completing a transition to a higher level, an unwanted transition attempt might occur. The framework allows the user to control whether a polled authentication mechanism can initiate a transition attempt, by providing the option to cap the level from which requests from polled handlers are enacted upon by the system, so that those requests are ignored.

Top-Level Design

The design of the multi-mode authentication solution involves three main types of components: kernel modules, authentication handlers, and user interface (UI) components. Figure 3 below illustrates the different components of the solution and the flow of data between them.

Kernel Module

The kernel has been augmented with two key modifications: 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 the MAF. For example, it blocks the I/O ports on the device and other means to bypass the authentication sequence until the user is authenticated at level 1. It also protects authentication information files, the user interface and handler components, and policy enforcement information (see Appendix B for more details). Moreover, it also has the means to register and start up registered components if they are not running or restart them if they terminate for some reason, which is used for the authentication handlers.

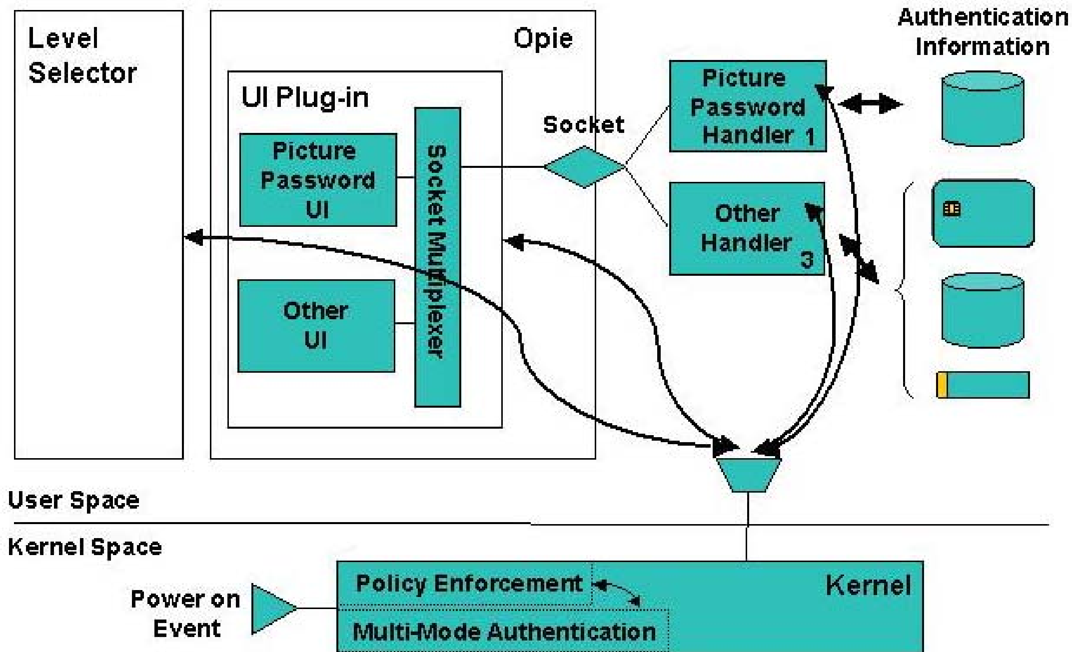

Figure 3: Top-Level Components

The MAF's main responsibility is to govern the authentication process as it relates to the various echelon levels that are configured. It is the source of all knowledge about the mappings between authentication mechanisms and echelon levels, simplifying the complexity 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 recognized handlers are accepted and processed.

User Interface Components

The user interface for the various authentication mechanisms is implemented as components of an OPIE plug-in module. Their function is to interact with the user, for example, to get the picture password image sequence, or to prompt the user to insert a smart card. The plug-in module supports a socket interface to receive commands from the authentication handler components that run as separate processes, and route the commands to the correct user interface component within itself. Similarly, the reverse response process is also supported between components and the module. The communication between different user interface components and the socket interface module is done using Qt's Signal and Slot facility. Since the user interface module is implemented as a plug-in to the desktop environment, it is started automatically.

Authentication Handlers

Handlers embody the mechanism that performs the actual authentication. They interact with the OPIE user interface components to tell them to bring up the specific screens, accept input, display messages, etc. For example, in the case of picture password, its handler accesses the theme identifier, button code mapping, and the password hash and verifies the sequence of user-selected images against the hashed password. Handlers communicate with the kernel module, listening for when to initiate authentication, and reporting back whether authentication was successful.

Echelon Level Selector

The echelon level selector is the end user's control panel for the MAF. It is visible as an active icon in the system tray that shows the current status and expands to a main window when selected. The main window has two interrelated selection items as shown in Figure 4: a slider and a group of four buttons. The slider is used to select the maximum echelon level to which a polled authentication mechanism can automatically trigger a transition (e.g., by the insertion of a token or the presence of a trusted beacon). The maximum level setting is provided as a way for the user to control the behavior of the authentication mechanism, essentially preventing a polled handler from automatically attempting to transition a user up to a level that she does not yet want to attain. The buttons display the available echelon levels (i.e., locked, low, medium, or high) and the current active level is highlighted. The user can change to another echelon level by pushing the corresponding button, which may cause the slider setting to change, if the selected echelon level is higher than the current level of the slider.

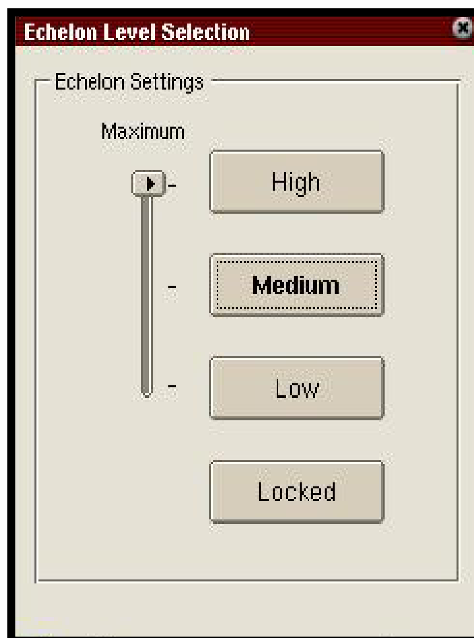

Figure 4: Echelon Level Selector

When the slider is accessed, the echelon selector interacts with the kernel to request that change. Similarly, if one of the buttons is pressed, the echelon selector interacts with the kernel to request a transition to that corresponding level.

Interfaces

The details of the various interfaces illustrated in Figure 3 are discussed in the subsections that follow. Note that this discussion is at a conceptual level. Further details appear later in the paper.

Between Authentication/Echelon-related Components and MAF

The communication between the kernel and a handler is done via `/proc/mAuth` file. There are several different messages that can be communicated between the authentication handler and the kernel, as well as the echelon selector and the kernel. In general, two classes of messages exist: those to control the authentication process and those to control echelon transitions. The purpose of these messages is described below. To simplify developing handlers, a common messaging programming interface has been established for all communications and the according routines have been developed and consolidated into a library (see Appendix A for more details). Note that due to its nature the `/proc/mAuth` file does not work very well with certain implementations of high level I/O libraries including standard GNU libraries. Thus, low-level I/O functions (open, close, read, write) should be used instead.

- *Handler Ready* - When a handler process is ready to perform authentication it signals the kernel with this message. When kernel receives this message it first verifies that this process is indeed a registered handler and, if it is, it puts the process on a wait queue until there is a need to perform the authentication step.

- *Authenticate* - The kernel uses this message to wake up authentication handlers and have them perform an authentication step. The handlers are awakened one at the time in the same order they occur in the handler's table.

- *Poll* - This message is used by the kernel to wake up authentication handlers periodically to have them check for the presence of a token.

- *Error* - With this message, the kernel signals a specific handler to exit.

- *Authentication failed* - This message is conveyed to the kernel to indicate a failed authentication attempt. The kernel verifies that the message comes from a registered handler.

- *Authentication successful* - This message is conveyed to the kernel to indicate a successful authentication attempt. The kernel checks if this message indeed comes from the handler that is supposed to be running at a present time. When all of the registered handlers report about successful authentication the kernel will unlock the device.

- *Level n* - This message is conveyed to the kernel from the echelon level selector and indicates that a transition to echelon level *n* should be attempted. An authentication handler may also use the message in special situations (e.g., a polling handler sensing the presence or absence of a token and needing to trigger a transition).

- *Maxl n* - This message is conveyed to the kernel from the echelon level selector and indicates that the maximum echelon level for automatic transition attempts should be set to *n*.

Note that the echelon level selector simply reads the `/proc/mAuth` file to obtain and display the current echelon status information.

Between the User Interface and Handler Authentication Components

Handlers and user interface components communicate through the UNIX domain socket `/tmp/mAuthUI`. From the handler side, all communication is done using standard system calls for sockets, which have been encapsulated in the handler library (see Appendix A for more details). The UI plug-in contains an object that handles all socket communications, routing incoming messages to the appropriate UI objects, and sending outgoing messages to handlers.

The format of the message is `prefix:msg`, where `prefix` is a message prefix and `msg` is the actual message to be delivered. The format of the actual message is implementation dependent, determined on the communications required between a particular authentication handler and user interface component. The only constraint is that a message should not contain a string terminator character `'\0'` (hex 00). When a developer adds a new component to the UI plug-in for some new functionality of an existing handler or a new authentication handler, she must choose a message prefix for this component, and then put a reference to the component into the `init.cpp` file.

All message routing is based on the message prefix. Messages are delivered to the UI components via Qt's Signal and Slot mechanism, simplifying their development. Note that, after it makes the routing decision, the routing object strips the prefix from the messages for delivery.

Between the UI Plug-in and the Kernel

In order for the UI plug-in to ensure that it is communicating with only registered handlers, it must obtain a valid list. The list of registered handlers is maintained by the kernel and can be read directly by the UI plug-in from the `/proc/mAuth` file entry.

Between MAF and Policy Enforcement

The interface between MAF and the policy enforcement mechanism is accomplished through two functions: `PolicySelect` and `MultiModeUpdateHandlers`. `PolicySelect` is implemented by the policy enforcement mechanism and `MultiModeUpdateHandlers` is implemented by MAF.

The policy enforcement mechanism calls `MultiModeUpdateHandlers` only once, during the boot process. The input to this function is a list of registered handlers. MAF uses this list to initialize internal MAF data structures. For the reasons of security all subsequent calls to the `MultimodeUpdateHandlers` are ignored.

`PolicySelect` is called by MAF when there is a need to change the current policy, typically upon successfully completing the necessary authentication steps for a requested level. This function has one integer argument that specifies the new effective policy level. In the current

implementation, policy levels are represented as bitmaps. The bitmap values for different policy levels are shown in Table 1 below.

Table 1: Policy Level Bitmap Representation

Level	Binary	Hex
Level 0 (default)	00000000	00
Level 1	00000001	01
Level 2	00000011	03
Level 3	00000111	07
Level 4	00001111	0F

Between the Startup Script and the Kernel

Communications with the kernel at system initialization is done via the `/proc/policy` file, using several messages:

- *Load Policies* - script initializes the policies to be enforced by the kernel on the device.

- *Load Handler List* - script initializes the list of registered handlers to be maintained by the kernel.

Process Flow

To understand how the components work together, the process flow of initializing the system from boot up is summarized. The start up and synchronization process among components proceeds as follows:

- On system boot-up, the kernel module loads and enforces its default policy, which blocks I/O ports on the device, hardware keys, and access to the authentication handler's code, as well as restricts access to authentication information within the file system exclusively to the appropriate authentication handler. The Linux proc file system (`/proc`) provides a communication channel between user space processes (UI components and handlers) and the kernel module. The kernel module registers a file in `/proc` file system (`/proc/mAuth`) for user space processes to trigger actions in the module.

- The system start-up script tells the kernel module (through the `/proc/policy` file) the filenames of the handler and any other related programs that need to be active. The kernel module sees that the handler programs are not running and starts them.

- Upon startup, each handler program performs all necessary initialization and then reads from the /`proc/mAuth` file, which causes their execution to be suspended.

- OPIE and its plug-ins are also loaded during boot-up. The authentication module plug-in signals the kernel via the `/proc/mAuth` file entry that all components are ready and that the device should begin a user authentication process.

- At this point all the components of the system are running and the default least privileges are being enforced.

- The kernel module wakes up the first authentication handler, picture password, to resume processing.

- The picture password handler reads the authentication information from the file system and signals its user interface component via a socket interface with the identity of the theme to display.

- The user interface component displays the theme, accepts the image sequence, and returns that information to the handler.

- The handler uses the button to alphabet mapping to compute and verify the password. If successful, it signals its success to the kernel module via the `/proc` interface. If unsuccessful, it continues to use the user interface component to have the user retry.

- When the kernel module receives an indication of success from the handler, it suspends the handler, and initiates any other registered handlers. If this is the last handler, the kernel unlocks the device.

Authentication Mechanism Details

As mentioned earlier, each authentication mechanism consists of two parts, the authentication handler and authentication user interface. The authentication handler is a stand-alone process that provides the actual authentication, while the user interface part performs all necessary interactions with the user. In the current implementation, all user interface components are implemented as components of a plug-in for the OPIE desktop environment. To create additional authentication mechanisms, new user interface objects can be added to the existing plug-in. Note that it is possible to write an authentication mechanism, which neither interacts with the user nor requires a user interface component. For example the mechanism could be based on a proximity beacon whose signal triggers a transition, both upward (presence) and downward (non-presence).

Recall that authentication handlers and user interface components communicate with each other using UNIX domain sockets. The framework provides high-level communication primitives for the authentication handlers and user interface components. Authentication handlers use high-level functions from the handler library (Appendix A), while user interface components use the Signal/Slot facility of the Qt library.

Since the user interface plug-in maintains a single socket for communication with all installed handlers, each handler must use message prefixes to identify the user interface component that is supposed to receive the message. A message prefix can be any alphanumeric string followed by a colon. When the user interface component receives the message, the prefix is already stripped by the message routing component of the plug-in.

Each authentication handler contains an infinite loop such as the one shown below.

```
For (;;)
{
     code =  HandlerReady (PollInterval);
     if ( code == Error )  EXIT  // done automatically by HandlerReady
     if ( code == Poll  )
                PerformPollAction
     Else
                PerformAuthentication
}
```

The `HandlerReady` function notifies the kernel that the handler is ready to proceed. The argument of this function specifies a poll interval. When the kernel receives this call, it suspends the process until the handler is needed to perform an authentication step, the poll interval elapses, or some error condition occurs. The poll interval is an integer value that specifies the frequency in seconds for the kernel to awaken the handler. If the value of poll interval is zero, it means that the kernel should keep this handler suspended until the authentication is required or an error condition occurs. When this function returns to the process, its return code indicates the reason for the kernel awakening this handler: an error condition occurred, the poll interval elapsed, or an authentication step is needed.

The handler process must always terminate itself when it receives an error code from the kernel. There are two reasons for an error condition to occur. One reason is that the kernel is unable to process the handler because either there is another handler already registered or the handler is not listed in the handlers' table, which is loaded into the kernel during system boot up. The other reason for an error condition is that a synchronous event signal is received while the process is in the `HandlerReady` call.

When a handler process receives the return code indicating that the poll interval has expired, it must perform necessary actions such as checking the state of the physical token or a proximity to the beacon and repeat the loop again. Note that the polling actions taken should not consume a lot of time; otherwise, it will lead to poor system performance. The handler process can indicate a failed or successful authentication step to the kernel as a result of the polling action. When the process receives the return code indicating authentication is required, it must perform an authentication step, which might involve interacting with the user or interrogating the physical token. When the result of authentication step is decided, the handler process must tell the kernel about positive or negative outcome of the authentication step (See example in the next section).

In order to write an authentication mechanism for the MAF, a considerable number of details need to be addressed. The remainder of this section touches on these details by reviewing two example handlers and their user interface components: one, a non-polled interactive authentication handler, and the other, a polled token based handler.

Non-polled Handler Example

The first example is an interactive authentication mechanism that is non-polled. This mechanism is very simple. When authentication step is needed the handler sends a message to the user interface component of the mechanism asking it to show a dialog box to the user. The dialog

box contains two buttons as shown in Figure 5: Get In and Stay Out. Clicking on the top button, "Get In," results in positive authentication, while clicking the bottom button, "Stay Out," results in negative authentication.

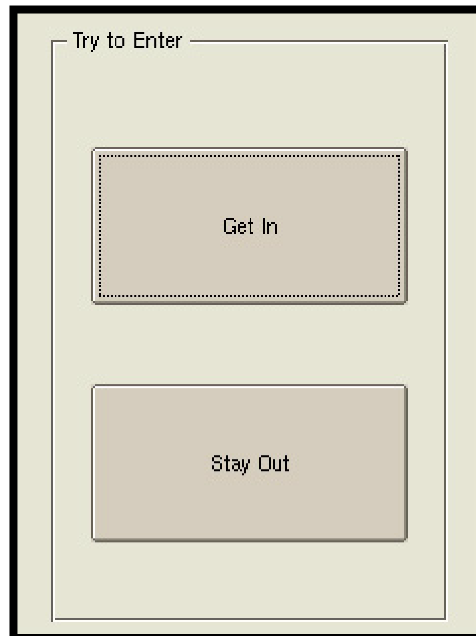

Figure 5: Simple Non-Polled Mechanism UI

When either button is pressed, the user interface component returns a reply to the authentication handler. Depending on the reply, the authentication handler signals the kernel a positive or negative authentication indication and then asks the kernel to suspend its execution until the next time authentication is needed. Since this handler does not need or use any polling actions, it calls the `HandlerReady` function with the polling interval of zero. The source code for this handler is located in the `HandlerEX` directory and the user interface component is located in the `ExtendUI.cpp`. The annotated fragments of the relevant parts of the code are as follows.

External library reference

Lines 30 and 31 contain the reference to the standard IO library and the handler library.

```
30 #include <stdio.h>
31 #include "hndlib.h"
32
```

Authentication Function

Lines 34 through 58 contain authentication step. The function starts with the declaration of the message buffer (line 38), which is used to communicate with the user interface part, and an integer variable that holds the length of the received message.

The line 42 uses the handler library function to set up the UNIX domain in socket that is used for further communication with the user interface component.

The line 45 sends the message to the user interface component. The user interface component is designated by the prefix ‘EX:’ and the message is `shw:Try to Ente`r. Upon reception of such message, the user interface component brings up the dialog box shown in Figure 5.

```
34 int ex_Login ()
35                 {
36
37         int rLen;
38         char msgfromUI[100];
39
40         puts ( "handlerEX: exLogin()" );
41
42         UIAttach ();
43
44         //send msgtoUI to UI
45         TellUI ( "EX:shw:Try to Enter" );
46
47         //wait for reply and let msgfromUI be the recieved string
48         rLen = RecvUI ( msgfromUI, sizeof(msgfromUI ));
49
50         printf ( "Got: [%d] %s \n", rLen,msgfromUI );
51
52         TellUI ( "EX:clr:" );
53
54         if( strcmp( msgfromUI+3, "Login" ) != 0 )  return 0;
55         puts ( "Login Success" );
56
57         return -1;
58 }
```

The code on line 48 waits for the reply message from the user interface component. When the message is received it is placed in the buffer `msgfromUI` and its length will be stored in the variable `rLen`.

The code on line 52 tells the user interface component to remove the dialog box from the screen.

The code on lines 54 and 55 processes the received message. If the message is not `Login` it means that the user did not press the ‘Get-In’ button, therefore the result of this authentication is negative and the return value of this function is FALSE. Otherwise the control continues to line 57 and the return value of the authentication step is FALSE.

Shutdown Function

The code on lines 62-65 contains a `ShutDown` function. This function is called when the handler is terminated by the kernel. This function tells the user interface component to remove the dialog box from the screen so that the PDA’s palmtop is left in a consistent state.

```
62 void ShutDown (int x) {
63   TellUI ( "EX:clr:" );
64   exit(0);
65 }
```

Main Function

The code on lines 68-98 contains the main function. This function can be used as a template for any non-polling handler.

```
68 int main (int argc, char * argv[])  {
69
70   if ( argc > 1 )
71     {
72       if ( !strcmp ( argv[1], "-d" ))
73            ex_Login();
74       else puts ( "Invalid argument " );
75       exit(1);
76     }
77
78   signal ( 15, ShutDown );
79
80   KrnlAttach ();
81
82   while(1)  {
83     int result;
84
85     puts ( "Waiting for Event");
86     HandlerReady ( 0 );
87
88     puts ( "Running Authorization" );
89     result =  ex_Login();
90
91     puts("Waiting for other handlers to finish\n");
92
93     if ( result )
94       TellKernel ( "AUTH-OK" );
95     else
96       TellKernel ( "AUTH-FAIL" );
97   }
98 }
```

The code on lines 70-76 determines whether the handler was started by the kernel to handle an authentication step or started by the user for debugging purposes. If the later is true, it runs the authentication step and then terminates the process. Otherwise, it registers the `ShutDown` function as a handler for the 'KILL' signal (line 78) and initializes communication with the kernel (line 80). Note that if the kernel does not have the modifications implemented to support the multi-mode authentication framework, the `HandlerReady` function terminates the process at this point.

If the process gets to line 82, it is ready to enter the authentication handler loop. The first action of the loop (line 86) is to signal the kernel that this handler is ready to perform its authentication step. Since polling functionality is not needed, the polling interval is set to 0. The exact description of the `HandlerReady` function is given in Appendix A. This function returns only when the kernel portion of MAF will determine that this handler should perform an authentication action. If the kernel signals an error condition to this handler, the `HandlerReady` function terminates this process.

The code on line 89 calls the authentication function described above. This function performs user interaction and returns TRUE if the authentication was positive or FALSE if it was negative. The code on lines 93-96 tells the kernel about the result of this authentication and then goes back to the beginning of the handler loop.

Non-Polled Handler Example User Interface

The user interface component for the example non-polled mechanism is derived from the standard Qwidget class. This class contains four member functions: `incomingMessage`, `let_in`, `keep_out`, and the class constructor. Also, this class contains four private fields that are described later.

Constructor

The constructor function of this class is defined on lines 40-60. The function of this constructor is to initialize internal variables, create a dialog box is used for user interaction, and set up all necessary Signal/Slot connections. The constructor accepts three arguments:

- `Mux` – a reference to the message router object.
- `Parent` – a reference to the parent widget
- `Name` – name of the user interface widget.

Only the first argument is required during creation of the object, if the other two are not specified the default values is used. The value of the first argument (`Mux`) is discussed later in this section.

The code on lines 43-47 sets up the dialog box to be used in full screen mode. Note that this is preferred method to create full screen dialog boxes from the plug-in. All other methods might leave PDA desktop in inconsistent state.

The code on lines 48-57 creates two push buttons and connects their Signals to the Slots implemented by this component.

```
40  extendUI::extendUI (MsgListener & Mux, QWidget * parent = 0,
                const char * name =   0) : QWidget( parent, name ), msgMux(&Mux)
41 {
42
43        reparent ( 0, Qt::WStyle_Customize |
44                      Qt::WStyle_NoBorder |
45                      Qt::WStyle_StaysOnTop, QPoint(0,0) );
46        setFixedSize( qApp->desktop() -> size() );
47
48        buttongroup_login = new QButtonGroup( tr("Hello"), this );
49        buttongroup_login -> setGeometry( 20, 10, 200, 300 );
50              QPushButton *pushbutton_getin = new QPushButton( tr( "Get In" ),
buttongroup_login );
51              pushbutton_getin -> setGeometry( 20, 60, 160, 80 );
52              pushbutton_getin -> setFocus();
53              connect( pushbutton_getin, SIGNAL( clicked() ), this, SLOT( let_In() ) );
54
55              QPushButton *pushbutton_stayout = new QPushButton( "Stay Out",
buttongroup_login );
56              pushbutton_stayout -> setGeometry( 20, 180, 160, 80 );
57              connect( pushbutton_stayout, SIGNAL( clicked() ), this, SLOT( keep_Out() )
);
58
59
60 }
```

Message Processing Function

The code on lines 65-83 processes incoming messages from the authentication handler. This function is called by the message router component of the plug-in when every time it receives a

message with the destination prefix EX. There are two possible messages: shw and clr. The shw message causes dialog box to be displayed on the PDA palmtop. The message could have additional text argument, which is displayed in the top part of the dialog box. The clr message causes dialog box to be removed from the screen.

The code on lines 68-70 separates command part of the message from the attached text. Note that every command has to be followed by the colon even if there is no additional text attached.

The code on lines 72-77 processes the shw message. Lines 73-74 save the socket address of the handler that send this message, which is required for sending replies back to the handler. The code on lines 75-77 sets the text to be displayed in the top part of the dialog box, displays the dialog box, and makes it the topmost window on the PDA palmtop.

```
65 void extendUI::incomingMessage (struct sockaddr * addr, unsigned aLen, const char * msg) {
66         qDebug( "Got Msg '%s'\n", msg );
67
68         char * pfxEnd = strchr ( msg , ':' );
69         if ( pfxEnd == NULL ) return;
70         int prefixLen = pfxEnd - msg;
71
72         if ( strncmp ( "shw", msg , prefixLen  ) == 0 )  {
73                 memcpy ( rAddr, addr, aLen );
74             rAddrLen = aLen;
75                 buttongroup_login -> setTitle( msg + prefixLen + 1 );
76                 show();
77                 raise ();
78         } else if ( strncmp ( "clr", msg , prefixLen  ) == 0 )  {
79                 memcpy ( rAddr, addr, aLen );
80             rAddrLen = aLen;
81                 hide();
82         }
83 }
```

The code on lines 78-81 handles the clr message. It erases the return address of the handler and hides the dialog box.

Button Click Processing Functions

The two functions on lines 86-98 handle the user interaction. The function let_In handles the click on the 'Get In' button and the function keep_Out handles the click on the 'Stay Out' button. Both of these functions are bound to their buttons by the constructor function described above.

These functions work very similar to each other. The message is prepared on lines 88 and 95, and then sent to the handler by the code on the lines 89 and 96. The public member function of the message router is used to send the message. The address of the handler was saved by the message processing function.

```
86 void extendUI::let_In ()                         {
87         char Seq[256];
88         sprintf ( Seq, "EX:Login");
89         if ( -1 == msgMux->SendMsg ( Seq , (struct sockaddr*)rAddr,rAddrLen) )
90           perror ( "Error Sending the sequence" );
91 }
```

```
93 void extendUI::keep_Out ()                          {
94         char Seq[256];
95         sprintf ( Seq, "EX:Stay Out");
96         if ( -1 == msgMux->SendMsg ( Seq , (struct sockaddr*)rAddr,rAddrLen) )
97           perror ( "Error Sending the sequence" );
98 }
```

Creating and Registering User Interface Objects

The following code was taken from the file `init.cpp` in the 'PlugIn' directory. The function 'InitializeModules' is responsible for creation of all user interface component objects defined in the plug-in. It is also responsible for assigning message prefixes to the user interface components. The component described above is created by the code on line 48, and it is bound to the prefix `EX` by the code on line 55.

```
44 void InitializeModules (MsgListener & msgMux, QWidget *){
45
46         picPassUI *picPass = new picPassUI( msgMux,0,"picPass" );
47         picPassChUI *picPassCh = new picPassChUI( msgMux, 0,"picPassCh" );
48         extendUI *extend = new extendUI( msgMux, 0, "extend" );
49         supplUI  *suppl  = new supplUI ( msgMux, 0, "suppl" );
50         suppl1UI  *suppl1  = new suppl1UI ( msgMux, 0, "suppl1" );
51         MMCAuthUI *mmcAuth = new MMCAuthUI ( msgMux, 0, "mmcAuth" );
52
53         msgMux.connect ( "PP", picPass , SLOT(incomingMessage(struct sockaddr
*,unsigned,const char *)));
54         msgMux.connect ( "PPC", picPassCh , SLOT(incomingMessage(struct sockaddr
*,unsigned,const char *)));
55         msgMux.connect ( "EX", extend , SLOT(incomingMessage(struct sockaddr
*,unsigned,const char *)));
56         msgMux.connect ( "FH", suppl , SLOT(incomingMessage(struct sockaddr
*,unsigned,const char *)));
57         msgMux.connect ( "PRE1", suppl1 , SLOT(incomingMessage(struct sockaddr
*,unsigned,const char *)));
58         msgMux.connect ( "MMC", mmcAuth, SLOT(incomingMessage(struct sockaddr
*,unsigned,const char *)));
59 }
```

Polling Handler Example

The second example handler is a polling handler. It uses a MultiMedia Card (MMC) as a physical token. This handler indicates successful authentication if it detects a card in the slot with the file `LetMeIn` in the root directory of the card. This handler uses the polling facility of the multiple authentication framework to detect insertion and ejection of the card. When the insertion of the card is detected, the handler triggers an automatic transition to the higher echelon level. When the card is removed, the handler immediately notifies the kernel that it's positive authentication is not valid anymore, and the kernel should switch to the lower echelon level. The source code for this handler is located in the file `handlerMMC.c`.

Although this authentication mechanism has a user interface component, its functionality is not substantially different from the user interface component of the non-polling handler described above. In fact, as shown in Figure 6, the user interface is simpler, only displaying a message to prompt the user to either insert the token or press the 'Cancel' button to terminate the authentication step. Therefore, no further details are provided in this document.

Figure 6: Simple Polled Mechanism UI

Card Interface Functions

Two functions, `SlotEmpty` and `CheckCard`, provide all necessary interfaces to the MMC card. Under the "Familiar" distribution of Linux the MMC card is treated as a removable storage device. When the card is inserted into slot, the kernel mounts it under the `/mnt/card` directory. When the card is removed, the kernel automatically unmounts this directory.

The function `SlotEmpty` checks if there is a card currently inserted into the slot by going through the table of mounted file systems and looking for an entry specific to the MMC card. If such entry exists, the function returns FALSE, indicating that MMC slot is not empty; otherwise, it returns TRUE. The function `CheckCard` checks if the card contains the `LetMeIn` file in its root directory.

```
88 int SlotEmpty ()  {
89   int f = open ( "/proc/mounts", O_RDONLY );
90   char buf[8192];
91
92   memset ( buf, 0, sizeof(buf));
93
94   if ( f >= 0 )
95     {
96       char *b;
97       read ( f, buf,4096);
98       close (f);
99       b = strstr ( buf, "/dev/mmc/part1 /mnt/card vfat rw,sync" );
100       if ( b == NULL ) return -1;
101       return 0;
102     }
103   else return -1;
104 }
```

```
107 int CheckCard ()  {
108   return ! access ( "/mnt/card/LetMeIn", R_OK );
109 }
```

Authentication Function

The authentication function performed by this handler is very similar in structure to the authentication function of the non-polling handler. The code on line 45 checks if there is a valid card in the slot. If there is a valid card in the slot, the function returns TRUE indicating successful authentication. Otherwise the function enters the interactive part. The code on line 47 asks the library to prepare the communication channel to the user interface part, and the code on line 51 sends a message to user interface component, which causes the dialog box shown in Figure 6 to appear on the screen. After dialog box is shown to the user, the function enters a loop that waits for the card insertion or cancellation from the user.

```
38  int ex_Login () {
39
40           int rLen;
41          char msgfromUI[100];
42
43          puts ( "handlerMMC: exLogin()" );
44
45          if ( CheckCard () ) return -1;
46
47          UIAttach ();
48
49
50          //send msgtoUI to UI
51          TellUI ( "MMC:shw:Insert token" );
52
53          while ( SlotEmpty ())
54              if ( PollUI () )
55                {
56                  puts ( "Have Events" );
57                  rLen = RecvUI ( msgfromUI, sizeof(msgfromUI ));
58                  printf ( "Got: [%d] %s \n", rLen,msgfromUI );
59                  if( strcmp( msgfromUI+4, "Cancel" ) == 0 )
60                    {
61                      TellUI ( "MMC:clr:" );
62                      return 0;
63                    }
64                }
65              else { sleep (1); puts ( "No Event" ); }
66
67          sleep ( 1 );
68
69          if ( ! CheckCard ())
70            {
71              int i;
72              for ( i = 0 ; i < 3 ; i ++ )
73                {
74                  TellUI ( "MMC:shw:Wrong Card" );
75                  sleep ( 1);
76                  TellUI ( "MMC:shw:" );
77                  sleep ( 1);
78                }
79              TellUI ( "MMC:clr:" );
80              return 0;
81            }
82
83          TellUI ( "MMC:clr:" );
84          return -1;
85 }
```

The code on line 54 checks if there are any messages from the user interface. If the user has pressed a cancel button, the function `PollUI` will return true and the code on lines 56-63 will execute. The lines 57-59 receive the message from the user interface and check if it is indeed a

cancel message, in which case the code on lines 61and 62 removes the dialog box from the screen and returns FALSE. If there were no messages from the user interface the handler sleeps for one second and then returns to the beginning of the loop (lines 53-65).

When the user inserts an MMC card into the slot the loop is terminated and the code on the lines 69-84 determines if it was an authorized card and returns TRUE in the event of the successful authentication. Otherwise, it notifies the user that the inserted card is not valid (lines 71-80) and returns FALSE. Regardless of the result of this authentication this function always removes the dialog box before returning back to the main authentication loop.

Main Function

The main function of the handler contains the authentication loop. Lines 116-123 are exactly the same as for the non-polling handler. The code determines whether the handler was started by the kernel to serve as an authentication handler or launched by the user for debugging purposes. Lines 126-133 are exactly the same as for the non-polling handler. However, in this case the `HandlerReady` function is called with non-zero polling interval. Since the polling interval is a constant, the kernel schedules polling of the handler to occur every three seconds.

```
113 int main (int argc, char * argv[]) {
114   int lastState = 0;
115   int cardState = 0;
116
117   if ( argc > 1 )
118     {
119       if ( !strcmp ( argv[1], "-d" ))
120            ex_Login();
121       else puts ( "Invalid argument " );
122       exit(1);
123     }
124
125
126   KrnlAttach ();
127
128   while(1)  {
129     int result;
130
131     lastState = cardState;
132
133     puts ( "Waiting for Event");
134     result = HandlerReady ( 3 );
135
136     if ( result == mmPoll )
137       {
138         puts ( "Checking for card \n" );
139
141         cardState = CheckCard ();
142
143         if ( cardState != lastState )
144           { puts ( "Card State Changed" );
145           if ( !cardState )
146             {
147               TellKernel ( "AUTH-FAIL" );
148             }
149           else
150             {
156               TellKernel ( "LEVEL 1" );
157             }
158           }
159         continue;
160       }
161
```

```
162
163     TellKernel ( ex_Login() ? "AUTH-OK" : "AUTH-FAIL" );
164   }
165 }
```

The `if` statement on line 136 determines whether the handler process was awakened to perform a polling step or to perform an authentication action. The code on lines 138-159 checks if the there is card in the slot and if the slot state is different than it was during the last poll cycle. It sends a message to the kernel to either initiate a transition to the higher echelon level (if the card was inserted) or to signal that the positive authentication given by this handler is no longer valid. Note that although the message to the kernel on the line 156 asks the kernel to transition to echelon level 1, the integer value on the level is ignored because of the special treatment of `LEVEL` messages coming from handlers. The effect of sending a `LEVEL 1` message to the kernel causes the next call to `HandlerReady,` at the beginning of the loop, to return with the indication that an authentication step should be performed. The code on the line 163 calls the authentication step function and then informs the kernel about the outcome of the authentication.

Kernel Details

The Multi-mode Authentication Module (`MultiAuth/MultiMode/MultiMode.c`) is at the heart of MAF. It is the kernel component of MAF and uses a table driven approach for controlling handlers and transitioning between echelon levels. The following sections describe it in detail.

Functional Description

The main task of the Multi-mode Authentication Module (MAM) is to monitor and schedule the authentication handlers' execution. At initialization, MAM establishes a communication channel with the handlers, the echelon level selector, and the UI plug-in through the `mAuth` entry of the `/proc` pseudo-file system. Programs running in user space can send the kernel the following one-line messages by writing them into `/proc/mAuth`:

- `ATTACH n`
- `LEVEL n`
- `MAX n`
- `AUTH-OK`
- `AUTH-FAIL`

User space programs can also read the following multi-line status information maintained by MAM from the kernel using the `/proc/mAuth` entry:

```
Level: MaxLevel/CurrentLevel/DesiredLevel
Lvl   Req   AState   PState   PInt   PID   Com
 l     r      a        p      poll   pid   name
...
```

The status information represents the contents of the handler table MAM uses to schedule handlers and manage echelon state. The first line contains the current echelon level of the framework, the desired level to which a transition request has been made, and the maximum echelon level to which an automatically initiated (versus manually initiated) transition can occur. Each entry in the list that follows includes the handler's executable name (`name`), its process

identifier (pid) and state (p), its authentication status (a), a polling interval (poll), and an authentication request status (r), for some echelon level (l) (see Appendix C for details).

The entire operation of MAM revolves around the handler table. The table is similar in structure to the standard Unix process table and its purpose is to represent the current state of the MAF system. The name of each handler and its level are pre-loaded at boot time and never change. The other entries change continually to represent the current state of the authentication framework. The kernel constantly examines the handler table to determine if a certain handler has to be restarted or needs to perform an authentication stop, if the system is ready to advance to the higher echelon level, or if the effective echelon level has to be lowered.

At initialization, MAM resets the handler table to zero. The handler table is partially built by the Policy Enforcement Module – PEM, whose code is located in PDAEnforcement/PolicyModule/Policy.c. The start-up script tells PEM the filenames of the handlers and any other programs that need to be continually active, by writing them into the /proc/policy entry of the /proc file system. The handler list written into /proc/policy has the following format:

```
<watch>
</usr/bin/HandlerA 1>
</usr/bin/HandlerB 1>
</usr/bin/HandlerC 2>
...
```

where watch is a keyword that identifies the /proc/policy contents, and each subsequent line contains a handler's executable filename and a echelon level. For example, the line:

```
</usr/bin/handlerPP 2>
```

specifies that the picture password handler, /usr/bin/handlerPP, is one of the handlers required to perform a successful authentication in order to transition the system to echelon level 2.

After parsing the list, PEM invokes a function (i.e., MultiModeUpdateHandlers ()) of the MAM module that partially builds the entries in the handler table based on the information read from /proc/policy. PEM further builds the environment in which the handlers are to execute and schedules all handlers not currently running (initially, each one in the list) for execution in user space. PEM periodically checks to determine whether a handler is running by walking through the current task list and comparing a task's command name to the handler's name. Any inactive registered handlers or applications are restarted.

MAM monitors the handlers' execution by processing the messages written by handlers and the echelon level selector into the /proc/mAuth entry of the /proc file system (i.e., ATTACH n, LEVEL n, MAX n, AUTH-OK, AUTH-FAIL). The handlers signal the kernel they are ready for processing by means of ATTACH n messages, where n is a polling interval (when n > 0). While processing the ATTACH messages, MAM completes the entries in the handler table with polling information, pid, and status. The polling handlers can send the kernel LEVEL n messages in

order to automatically attempt to transition to a higher level. Also, the OPIE plug-in can send a `LEVEL 0` message to automatically transition from level 0 to level 1 after power-on.

The echelon level selector sends the kernel `LEVEL n` messages in order to transition the system to level n, based on the user manual selection. The echelon level selector sends the kernel the message `MAX n` to limit the level to which the system can transition via a `LEVEL n` message from a polling handler.

The authentication handlers can send the kernel `AUTH-OK` or `AUTH-FAIL` messages signaling a successful or failed authentication process. Based on processing these messages, MAM transitions the system to the appropriate level, selects the policy to be enforced for that level, and indicates the policy to the PEM by calling PEM's function `SelectPolicy()`.

Basic Interaction with an Authentication Handler

The following code fragment can illustrates the handler-programming model:

```
while(1)  {
    TellKernel("ATTACH n");
    if ( errno == 2 ) exit(4);
    if ( errno == 3 ) TellKernel(Poll() ? "AUTH-OK" : "AUTH-FAIL");
    else TellKernel(Login() ? "AUTH-OK" : "AUTH-FAIL");
}
```

where `TellKernel(msg)` writes `msg` into `/proc/mAuth`, and `Login()` is a Boolean function that performs the authentication.

Normally, the handlers are launched into execution by PEM, but the information in the handler table (e.g., the pid) is not completed until the handlers attach themselves to the kernel by sending MAM the `ATTACH n` message to enable processing under the framework. MAM starts by checking the `ATTACH` message to determine whether there is an old task with the same name and a pid different from that of the current handler. If such a task is found, MAM sets the `errno` to 2 (`mmError`), the control flow returns to the user space portion of the handler, and the handler's execution is aborted. Otherwise, MAM completes the information about the handler process (the pid and polling interval) in the handlers' table and puts the handler to sleep.

The handler process will be woken up by an interrupt, a polling event, or processing of a pending request when trying to raise the level. If the handler was interrupted, MAM resets the handler pid and polling interval to 0 in the handler table, and sets the `errno` to 2 (`mmError`). The control flow returns to the user space portion of the handler, and the handler's execution is aborted. In the case of a polling event, MAM resets the handler's polling interval to 0, sets the handler's state to `authPoll` ("polling"), and sets the `errno` to 3 (`mmPoll`). The control flow returns to the user space portion of the handler, and the handler starts to perform `Poll()`, then signals the kernel the result (`AUTH-OK` or `AUTH-FAIL`). In the case of a pending request, MAM sets the handler's state to `authRun` ("authenticating") and sets the `errno` to 0 (`mmAuthenticate`). The control flow returns to the user space portion of the handler, and the handler starts to perform `Login()`, then signals the kernel the result (`AUTH-OK` or `AUTH-FAIL`). After polling or authentication, successful or failed, the handler reattaches to the kernel.

Automatic Transition from Level 0 to Level 1

Initially, right after system boot-up, the echelon level is at 0. The authentication handlers, which depend on the OPIE plug-in, wait for OPIE to load and run before proceeding (this is taken care of automatically by the handler library). One of the first actions OPIE takes is to invoke the `applet()` method that is implemented by the OPIE plug-in. This method, among other initializations, writes a `LEVEL 0` message into `/proc/mAuth`. By processing this message, MAM triggers an attempt of automatic transition from level 0 to level 1 (it calls `setDesiredLevel(0)`, which eventually calls `TryToRaiseLevel()`). Of course, the attempt may fail if an authentication handler for level 1 fails. The same automatic transition to level 1 takes place every time the system is downgraded to level 0.

Transitioning Between Echelon Levels

When MAM receives a "LEVEL n" message, where n is a level higher than the `CurrentLevel`, it initiates level advancement. The pseudo code for level advancement is as follows:

1. Set `DesiredLevel` to n.

2. Lookup a handler with `Level` <= n and `AuthState` == FALSE, if no such handler exist set the `CurrentLevel` = `DesiredLevel`, and notify the policy enforcement component about the level change.

3. Set the request pending (`ReqPending)` to True for this handler and wake up the handler process.

4. Wait for a response from the handler. If response is negative then abort the process. If the response is positive, set the `AuthState` of the handler to TRUE and go to step 2.

Note that if the process is aborted and the `CurrentLevel` is zero, then the system will restart it again with the `CurrentLevel` set to 1.

When the MAM receives an `AUTH-FAILED` message from one of the defined handlers, it sets the `CurrentLevel` to the one below the level of the handler that send this message. The level is also dropped when the system is going into stand-by mode, or when the kernel receives an explicit `LEVEL n` message with the `n` being less then the `CurrentLevel`.

Automatic Transition with Polling Handlers

While in general the attempt to transition to a higher level is triggered by user intervention (using the Level Selector GUI), in the case of a polling handler this attempt can also be done automatically. When sensing a token's presence or absence, the polling handler can issue a `LEVEL n` message to the kernel; this message will be processed by MAM's `setDesiredLevel(n)`, which eventually calls `TryToRaiseLevel(m)`, where `m` is the level associated with the polling handler in the handler table. Note that `m` is set in the handlers' table by PEM based on input from the start-up script.

Penalties for Failed Authentication

MAF allows imposing blocking penalties for failed attempts to transition to a higher level. The list of handlers that have to perform authentication for that level (called here proper handlers) must be "bracketed" by two pseudo handlers (called here pre-handler and post-handler). Thus, any attempt to transition to the higher level starts with the pre-handler's "attempt" to authenticate and, if the proper handlers are successful, ends with the post-handler's "attempt" to authenticate. The pre-handler attempts to authenticate by recording into a file the attempt number. The first time the authentication is attempted this number is 1, subsequent times is 1 plus the previous attempt number read from that file. The pre-handler always send the `AUTH-OK` message to the kernel. In case of successful authentication of all proper handlers, the post-handler wipes out the file with the attempt number, so the next time the attempt number will start again from 1. In case of unsuccessful attempt, the post-handler does nothing, so the next time the attempt number will increase, and the pre-handler will block the device for a period of time which increases with the attempt number.

Processing of Successful Authentication

A handler signals its successful authentication to the MAM by writing the `AUTH-OK` message into `/proc/mAuth`. MAM starts processing this message (in `AuthOK()`) by finding the slot in the handler table that corresponds to the signaling handler (the current task). If this handler's authentication is a required step on the road to the `DesiredLevel`, MAM sets the handler's authentication status to `authOK`. Otherwise, if the handler's level is higher than the `DesiredLevel`, the authentication status is left unchanged. In any case, the handler's state is set to `authDone` and the `ReqPending` field is set to false. MAM invokes the `TryToRaiseLevel()` function.

Processing of Failed Authentication

A handler signals its failed authentication attempt to the MAM by writing the `AUTH-FAIL` message into `/proc/mAuth`. MAM starts processing this message (in `AuthFail()`) by finding the slot in the handler table that corresponds to the signaling handler (the current task). If the handler's level is higher than the `DesiredLevel` (the authentication result is not a required step on the road to the `DesiredLevel`), MAM returns from processing. Otherwise, MAM sets the handler's state to `authDone` and the `ReqPending` to false. Also, MAM sets the authentication status to `authNone` for all handlers with levels higher or equal to this handler's level. Next, MAM decides the level to which the system is to drop. Currently, MAM drops to the failed handler's level minus 1, by setting the `DesiredLevel` to that value and invoking `SetCurrentLevel(DesiredLevel)`.

Manual Transition to a Level

The user can manually transition to a level by selecting the level in the echelon level selector GUI. The echelon level selector signals the kernel the user's attempt to raise/lower the level to the value `n` by writing the message `LEVEL n` into `/proc/mAuth`. MAM starts processing this message (in `SetDesiredLevel(n)`) by trying to find the slot in the handler table that corresponds to the signaling (current) task. The current task being the echelon selector, no slot will be found. First, MAM sets the `DesiredLevel` to `n`. Then, for all handlers with levels higher than `DesiredLevel`, MAM sets the authentication state to `authNone` and the `ReqPending` to

false. If this is a downgrade, MAM calls `SetCurrentLevel(DesiredLevel)` in order to set `CurrentLevel` to `n` and perhaps further raising the level automatically (because `SetCurrentLevel()` calls `TryToRaiseLevel()`). If this is an upgrade, MAM calls directly `TryToRaiseLevel()`.

Ensuring All Handlers are Running

PEM runs a low priority thread that examines the system process table and the handler table. If this thread notices that there is a handler defined in the handler table that is not present in the system process table, it spawns this handler process again.

Appendix A – Handler Library

The handler library is a collection of routines that provide a common API for authentication mechanism handlers to communicate with both its user interface components and the kernel. The echelon selector also uses it to communicate with the kernel.

The library code is located in `MultiAuth/MultiMode/hndlib.{h,c}`

Types

`ErrorCode` – Is an enumeration type that represents the values returned by the `HandlerReady()` function. It has the following values: `mmError`, `mmAuthenticate`, `mmPoll`.

Functions

`void KrnlAttach()` – Establishes a communication channel from handler to kernel through the `mAuth` entry of the `/proc` pseudo-file system. Namely, it tries to open `/proc/mAuth` for read/write. A failure means that the kernel does not support the Multi-Authentication Framework, and then a null handle is stored in the static variable `pHndl`. In case of success, the handle returned by `open()` is stored in `pHndl`.

`void UIAttach()` – Establishes a communication channel from handler to UI through sockets.

`void TellKernel (char *pMsg)` – Resets `errno` to zero and then sends the kernel a message (pointed to by `pMsg`) by writing it to the `mAuth` entry in the `/proc` pseudo-file system (using the handle obtained by calling `KrnlAttach()`). The echelon-level selector can send the kernel the following messages:

```
LEVEL n
MAX n
```

The `LEVEL n` message signals the kernel that a transition to level `n` should be attempted. The `MAX n` message signals that the maximum level for automatic transition attempts should be set to `n`.

The authentication handlers can send the kernel the following messages:

```
AUTH-OK
AUTH-FAIL
ATTACH n
```

`AUTH-OK` and `AUTH-FAIL` signal that the authentication succeeded or failed respectively. The `ATTACH n` message signals the kernel that the handler is ready to process and indicates a polling interval of `n` seconds. When `n` is zero, no polling will take place.

As a side effect, the kernel might set **`errno`** to a certain error number, which can be consulted by the handler on return from `TellKernel`.

`int TellUI(char *pMsg)` – Sends the UI a message (pointed to by `pMsg`) over the communication channel opened by `UIAttach()`. Returns the number of characters in the sent message or -1 in case of an error.

Each message has a prefix that identifies the message source; for example, the prefix `'PP'` identifies the Picture Password handler, while `'MMC'` identifies the Multimedia Memory Card handler. The prefix is separated from the message body by a colon `':'`. The message body is a null-terminated string and its contents depend on the particular handler and user interface. The message reaches the UI stripped of its prefix and the separator `':'`.

`int RecvUI(char *pMsg, int length)` – Receives a message from the UI in the buffer pointed to by `pMsg` of maximum length `length`. Returns the length of the received message, or –1 in case of an error.

`ErrorCode HandlerReady(int pollingRate)` – Signals the kernel that the handler is ready for action and sets the handler's polling rate to `pollingRate` by sending the kernel the message `ATTACH pollingRate` (see `TellKernel()` API). Returns an `ErrorCode` according to the `errno` set by the kernel as response to the message.

`void GetLevels ( int *pMax, int *pCurrent , int *pDesired )` – Obtains the maximum, current, and desired security level from the kernel (by reading the entry `mAuth` of the pseudo-file system `/proc`). These data arrive from the kernel in a message with the following format:

```
<some_string> [':' max_level ['/' current_level ['/' desired_level
<some_string>]]]
```

with the levels expressed in decimal. The levels are returned in the integer variables pointed to by `pMax`, `pCurrent`, and `pDesired`. A level missing from the kernel's message is returned as –1 by `GetLevels()`.

Appendix B – Mandatory Access Control Settings

Below is a summary of mandatory access control settings that are imposed to secure the multi-mode authentication framework. Typical `/dev` filenames are included in the list. However, additional files could exist that have different names, but the same function. For example the touch screen is both `/dev/touchscreen/0` and `/dev/h3600_ts`. The actual function of the files that are in the /dev directory is determined by the major/minor numbers not by the filenames. Therefore, the major/minor numbers are also provided. When the minor number is specified in the form `[n-m]`, it means the whole range from `n` to `m`.

Another convention used is: `level 0` for the lowest most restrictive policy level, and `level 1` and higher for less restrictive policy levels. Though we tend to think of higher echelon levels as being increasingly less restrictive (i.e., `level n` policy granting privileges that encompass all those granted at `level n-1`), in practice, policies might not necessarily be so neatly nested.

Hardware Related

Touch screen – Access is granted at all echelon levels. One exception is that during system boot up access is denied briefly, until the UI plug-in initializes for the level 1 authentication handlers. Its device filename is `/dev/touchscreen/0`, and its major/minor numbers are `254, [0-1]`.

Hardware keys – Access is denied at level 0 and granted only at level 1 and higher. The device filename is `/dev/touchscreen/key`, the major/minor number is `254,2`.

Serial port – Currently access is granted at all levels for development and testing purposes; for production, access should be denied at level 0 and granted only at level 1 and higher, as appropriate, to enable communications through this interface. The device filenames for serial ports are `/dev/ttyS*`, `/dev/cua*/*`, and many others – essentially everything with major numbers `204, 205`. One exception is that Bluetooth, with device filename `/ttySB0` and major/minor `204,60`, is not really a serial port.

IRdA port[1] – Access is denied at level 0 and granted only at level 1 and higher, as appropriate, to enable communications through this interface. It is not clear that IRdA has a device filename on the iPAQ, but its interface is `irda*`.

Bluetooth port[2] – Access is denied at level 0 and granted only at level 1 and higher, as appropriate, to enable communications through this interface. The Bluetooth port has the device filename `/dev/ttySB0` and also the interface `hci*`. Most programs don't access it through the device file (not certain whether you can even do it). The major/minor numbers are `204,60`.

PCMCIA/CF/MMC slots – Access is denied at level 0 and granted only at level 1 and higher, as appropriate, to selectively enable known sets of hardware modules. These cards do not have device files. However scripts sometimes create such files when a card is inserted, then delete them when the card is removed.

---

[1] Most programs use IRdA through the socket interface. The socket family is `AF_IRDA`.
[2] Most programs use Bluetooth through the socket interface. The socket family is `AF_BLUETOOTH`.

Software Related

Initialization scripts – All MAF start-up scripts are protected against modification. The startup script is `/etc/rc2.d/S05policy`.

Policy information – `/root/defaultpolicy`, `/root/handlers` – Only MAF startup scripts are able to read policy information files.

Kernel policy enforcement interface – `/proc/policy` – Once start-up scripts have loaded policy information and authentication handler registration information through this interface, further access is denied at all levels (i.e., a loaded policy entry restricts access).

Kernel MAF interface – `/proc/mAuth` – Only registered handlers can use the interface to communicate authentication failure and success, at all levels, while other programs can signal requests to attempt a change in authentication level.

OPIE plugin files – `/opt/Qtopia/plugin` – Only the OPIE process can read these files at any level.

Handler executables – No process can access the handler executables, at any level, other than for execution. The handler executables generally reside in `/usr/sbin/handler*`, but that depends entirely on the system configuration.

Handler authentication information – (e.g., `/root/picPass.dat` for picture password) – Only handler processes can access their associated authentication information.

OPIE UI plugin socket – Any process can access this socket, however, messages from other than registered handlers are ignored.

Network sockets – Access is denied at level 0 and granted only at level 1 and higher, as appropriate, to filter socket communications.

Appendix C – MAM Global Variables

Types

`ProgState`

This is an enumeration type and defines an authentication handler's state. Its values are:
- `authWait`: the handler is in a wait state;
- `authRun`: the handler is in the authentication process;
- `authPoll`: the handler is in the polling process;
- `authDone`: the handler has finished the authentication.

`AuthResult`

This is an enumeration type and defines the result of the authentication process. Its values are:
- `authNone`: authentication not yet performed;
- `authFail`: authentication failed;
- `authOK`: authentication succeeded.

Global variables

`static struct authInfo authTable[11]` - `authTable` is the table of authentication handlers. An entry in the table is an `authInfo` structure with the following fields:
- `ReqPending`: a Boolean; true indicates an authentication request for the corresponding handler is pending;
- `AuthState`: a field of type `AuthResult`; indicates authentication status, i.e., the result of the authentication performed by the corresponding handler;
- `pState`: a field of type `ProgState`; indicates the handler's state;
- `Level`: an integer indicating the echelon level; the handler is intended to perform the authentication required for transition (from a lower level) to this level;
- `PollInt`: an integer specifying whether the handler is a polling handler (if `PollInt` > 0) and the polling interval in seconds; its initial value comes from the argument to `HandlerReady` and is decremented every second;
- `pid`: the process id of the handler task;
- `comm`: the handler's name (the command name as it appears in the `struct task_struct`).

`int MaxLevel, CurrentLevel, DesiredLevel` - These global variables define respectively the maximum, current, and desired level of the system. The desired level, if different from the current level, is the one the user or the system is attempting to reach.