RZ 3219 (# 93265) 20/03/00
Computer Science/Mathematics .. pages

Research Report

James Riordan

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).

IBM Research
Almaden · Austin · Beijing · Delhi · Haifa · T.J. Watson · Tokyo · Zurich

Access Control for PDAs

James Riordan

IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

Abstract

Traditional access control mechanisms focus on separating users from one another and are designed with the assumption that there is a competent systems administrator or security manager controlling who may do what. These mechanisms are inappropriate for personal digital assistants (PDAs) where there is a single user who is probably not a professional systems or security administrator. This paper examines what access control features are needed in a PDA and what assumptions are reasonable to make about its administration. It then presents an access control system to provide these features subject to the administrative constraints.

1 Introduction

As the size, power, and cost of computing devices has changed, so have the methods with which and the purposes for which we interact with them. These methods and purposes determine what sort of security policy is appropriate for a particular device. At the advent of a new form of computing, it makes sense to reconsider what sort of security policy is appropriate.

Traditional larger machines support numerous users and are managed by systems administrators. The access contol mechanisms have focused on separating the users from one another based upon a security policy determined by the systems administrators. Some, primarily military, systems have allowed finer-grained access control policies allowing the separation of different aspects of an individual user, but the complexity of these systems makes them prohibitively expensive to administer. As a result these system access control mechanisms have not been widely adopted.

The access control mechanisms available in various databases and in Java offer finer-grained control of data and objects but do now solve the general problem of access control at the system level.

Most personal computers solved the problem of security by declaring it a non-problem. This approach worked reasonably well for CPM, TRS/DOS, and other operating systems which lived in a kinder, gentler period of computer history. Subsequent PC operating systems running on modern hardware, including DOS, Windows, and MacOS, have been assaulted by a barrage of viruses, Trojan horses, and other malicious software (collectively *malware*). The release and use of such malware is essentially a form of vandalism in that there is generally little value gained by the perpetrator beyond a certain idiotic entertainment.

As we begin to use systems for economically meaningful transactions, there is far greater benefit and hence incentive for the attacker. With the increased motivation, it is reasonable to expect that such systems will be attacked by increasingly skilled and knowledgeable adversaries. Thus the need for security is increasing sharply.

Traditional commerce required appropriate levels of security to develop and thrive (lest thieves would have stolen everything). Similiarly e-commerce requires appropriate levels of security. Expections to the contrary are like thinking that banking would have developed in a world devoid of safes, guards, and alarms.

The form factor and usage characteristics of PDAs makes them extremely desirable for use in many e-commerce applications [RW, PPSW97, BF99]. Unfortunately, current PDA operating systems[1] do not offer the necessary security for e-commerce applications. The very fact that PDAs are powerful and general-purpose computing devices renders them vulnerable to attack. E-commerce systems based upon PDAs are potentially vulnerable to an entire range of attacks that do not pose danger for more primitive systems (e.g., credit cards).

This paper presents some ideas about what appropriate access control mechanisms might look like.

2 Security Generalities

It is naturally the case that access control mechanisms are about bringing into existence certain security-related qualities. It is also important to understand that *secure* only makes sense with respect to a specific collection of threats. When designing and building secure systems it is of the utmost importance to understand these threats and to design the systems accordingly. As such, we will here explicitly state the relevant security assumptions, goals, and generalities.

2.1 Secure Systems

A system that can be brought to an insecure state through normal user action is not secure. This is particularly problematic when the user is also the administrator. Given the choice between entertainment and security, users will almost always choose entertainment. It is, thus, necessary to construct a system in which entertainment and security are not mutually exclusive.

Even security-conscious users are hard pressed to make security-related decisions. Users can probably make reasonable decisions about “Cut a check to Bob Air for 1,000,000£ (Yes/No)?” but might not be able to make reasonable decisions about “Install new font (Yes/No)?” thereby signing a check for 1,000,000€ instead of 1,000,000£.

2.2 Unstructured complexity

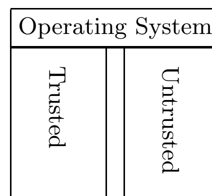
Unstructured complexity is necessarily insecure. There seems to be little that one can do to completely eliminate errors as to eliminate vulnerabilities. Rather it is necessary to isolate errors by isolating individual components, thus limiting the ability of an attack to propagate through these vulnerabilities. Security mechanisms must exist redundantly in layers so as to minimize the number of points where a single error results in vulnerability.

This dictates that high-security systems must be built upon a small, uncomplicated foundation (e.g. a microkernel) and that the access control mechanisms, configuration, and policies must be very simple.

[1] nor, for that matter, do current PC desktop operating systems

2.3 Open

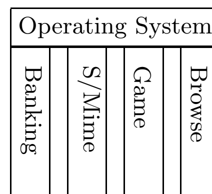
For a system to be useful, it must be open. Recognizing the need for hand-held devices that are both secure and general purpose, a variety of schemes have been proposed that support two compartments, one for secure and trusted applications and the other for insecure and untrusted applications with the operating system to isolate the two.



A user visable LED[2] on the device indicates which state the device is in.

The problem with such schemes is that real-world trust relationships do not fit into two compartments: a home banking application clearly needs to be in the secure compartment but still should not have access to the user's personal correspondence keys.

One instead needs a system with multiple compartments in which each is protected from the other by the operating system.



There are several difficulties here: the various compartments need to intercommunicate, the user needs to know which compartment is active, and it is not clear how to manage compartments.

Traditional access control mechanisms have been hierarchical in the sense that absolute power is given to the sytem administrator who subsequently delegates limited power to users and subsystems. Hierarchical structures have the problem that there may be serveral groups who do not commonly trust any third party sufficiently to administer the system. They are additionally expensive and error- prone to administer.

2.3.1 Open Source

Another sense of open in the collection of security concerns and principles is Open Source [Com, Cox]. Open design and implementation have long been security requirements in the cryptographic community. Their value for operating is becoming apparent for operating systems.

2.3.2 Code Signing

Security mechanisms are about who can do what; code signing is about where a piece code probably originated. The two should not be confused.

The most common code-signing based security mechanism is allowing application to be installed only when the application has been signed with some key known to the device. On single-function systems, such as smartcards,[3] the system itself provides isolation: it is not likely that a malicious piece of code on one card will attack code existing on a completely different card. On multi-function systems, such as a PC or a PDA, there are a number of serious problems related to code signing:

- Trust is a complicated quantity that rarely fits into the clean hierarchies required by this technique. In particular, it is unlikely that a single signing organization will be trusted by all parties. It is additionally extremely difficult to make a reasonable judgment as to whether a given source is trustworthy (or even what trustworthy means in a given context).

- Code signing is a social process rather than a technical one. Social processes are much more difficult to secure than technical ones. In addition to being open to normal technical attacks, they are open to social attack and human error.

- There is an implicit belief that a well meaning source will produce vulnerability free code (and data). Evidence to the contrary is ample and uniform.

- Code signing schemes require a global registry and key hierarchy of all developers and security policies for the developers. These don't exist.

Systems in which code signing has been linked to code verification, either automatically or by human intervention, provide improved yet questionable security. The code verification environment is not generally the same as the code execution environment. As a result, code signing schemes tend to suffer from labelling and concurrency problems. This is especially true when more than one developer is involved and is completely intractable when the developers do not trust one another.

In summary, this downloading- enabling mechanism itself cannot offer the security needed for a PDA.

[2] light emitting diode

[3] Even multi-function smartcards have an owner who controls what is downloaded onto the card.

2.4 Painful security

Security mechanisms that are painful to use tend to be bypassed. It is generally the case that the mechanisms used to bypass the mechanism are completely insecure. There are numerous examples of this sort of revenge effect in security: difficult-to-remember passwords are written on post-it notes, overly restrictive firewalls are tunneled through, intrusive software monitoring systems are lied to by the users whom they are supposed to protect.

3 Access Control Needs

Various operating system abstractions have lead to different means of interacting with data. In the Unix world everything is a file and tools are built to manipulate arbitrary files. In the Windows and Macintosh worlds, files are generally associated with applications, and applications do not generally share files. Each abstraction has its own advantages. The latter model is more appropriate for PDAs where different aspects of an individual user of a PDA are roughly represented by the different applications running.

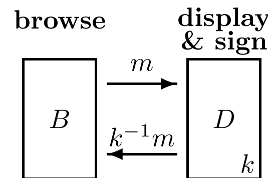
These aspects need to be protected from one another. The access rights needed by Alice to run a calendar program are entirely different from those needed by Alice to sign a digital check. Naturally, the access control mechanisms should prevent the calendar application from accessing Alice's check-signing keys. At the same time, applications need to intercommunicate.

3.1 Usage Scenario

A good example is provided by a scenario in which a user wants to select and purchase an item. We will develop our access control model by developing this example.

The item will be selected using a browser, such as WAP [Con], running on the PDA. A browser is a general purpose and extremely sophisticated piece of software that acts upon complex data supplied by untrusted users. Browser evolution is driven primarily by the addition of new features and is still both active and rapid. In short, it is unlikely that a browser that satisfies consumer demand could be made sufficently secure to safely authorize payments (e.g. to manipulate cryptographic keying material in a safe and meaningful way).

It is possible, however, for the browser to generate a request, which is handed to a simple, static, secure compartment for display of terms and payment authorization. We denote the signature on m under the key k as $k^{-1}m$.

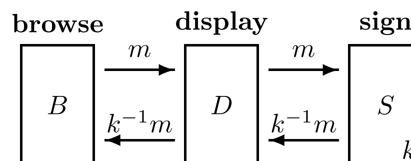


By separating the two we can create a hybrid architecture that permits the otherwise conflicting requirements: feature richness and general purpose abilities for the browser and simplicity for security.

Indeed, the browser needn't run on the PDA but can be an external device communicating with the PDA (i.e. a *TINGUIN* [PPSW96]). The browser runs on the user's (insecure) PC. When a signature is needed, the document is passed to the secure PDA for display, authorization, and signature generation.

Only the sign and display compartment needs access to the signing key k. The security of the scheme is dependent only upon the sign and display compartment and its ability to display information to the user.

One could further split the display and sign functionality



to allow the use of a secure token such as a smart card.

In this architecture, the security depends only on the **sign** and **display** compartments, the **display** compartment's ability to display information to the user, and the **sign** compartments ability to accept requests from **display** (other trusted compartments). Only **sign** needs access to the signing key k.

3.2 Analysis

Whereas the security of each compartment clearly depends upon the nature of that particular compartment, there are low-level security concerns common to all [GS91]. These concerns are well understood although still quite difficult to address. It is a tenet of this paper that browser security is dubious if not impossible; we thus address only the particular needs of the **display** and **sign** compartments.

Proper interlinking of compartments is also essential and, indeed, will form the basis of the architecture.

3.2.1 Display

The *display compartment's* ability to display data to the user has two primary requirements: that the compartment can obtain a resource lock on the display and that the data itself have a single, well-defined meaning.

The ability to lock the display is needed to diminish the threat of Trojan horses. The granting of exclusive locks on system resources allows malicious code to either soft or hard lock the system, thereby staging a denial-of-service attack. Assuming that we can force all system locks to be soft locks, this threat is not of interest. It is thus the case that the primary issue is that the system must be able to lock a sufficient number of resources. These resources include the display, touch screen, various other I/O devices, memory pages, etc..

Semantic invariance of the data is somewhat more problematic. The data must have a simple format as the signature applies only to the document as a number and not to a semantic entity. It thus cannot allow conditionals, include files, dynamic generation of fields, changeable style sheets, etc.). Complicated formats, such as MS Word or PostScript, can be made to display differently when viewed under different conditions. This makes it unclear as to whether a signature on a Word document or a PostScript file has any meaning.

One must further guarantee that the system resources, including libraries and fonts, remain fixed. Although this could lead to inflexibility for the system, generality can be maintained by compartmenting system functionality. Subsystems requiring security use only simple stable resources, whereas subsystems requiring higher functionality can use more complex and dynamic (separate) features.

3.2.2 Sign

The sign compartment must be able to protect and manage its key k and to ensure that a request to sign a document came from **browse**. Protecting and managing these data means that they should only be accessible to other compartments though **sign**'s external interfaces.

This implies certain low-level properties of the system: the system cannot allow raw access to memory, the integrity of messages (IPC) must be maintained, and access to system resources does not require complete privileges (as with Unix's `suid-root` mechanism).

It also requires a naming system so that there is a well defined difference between compartments. The naming issue is the essence of the problem.

3.2.3 Application Identification

Generally, *application identification* is provided by the names in a configured and maintained registry (e.g. Unix places naming information in `/etc/password` and `/etc/group` files, in the kernel itself, and in the file system).

The security of the system depends upon the registry and hence upon its configuration and maintenance. This poses a problem on PDAs, which are not typically maintained by trained system administrators. It is thus the case that the security of the device cannot require that the owner make good decisions from a security standpoint.

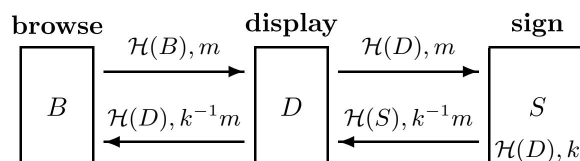
We nonetheless, as a fundamental security construct, need to enable applications, equivalently different user roles, to identify one another.

4 Primitives

We have seen that naming is the fundamental issue in access control. We use cryptographic hashes to generate unique and unforgeable names.

With each executable E we associate the name $\mathcal{H}(E)$ where the executable is regarded as a byte stream $E = \{b_0 b_1 b_2 \ldots\}$. This provides us an automatic, if seemingly inflexible, registry. When the executable is run, it is run with the label $\mathcal{H}(E)$. Persistent data created by E is accessible only to E and also bears the label $\mathcal{H}(E)$.

The operating system ajoins to all requests m made by an application A to an application B the hash $\mathcal{H}(A)$, thereby identifying the requestor to B.

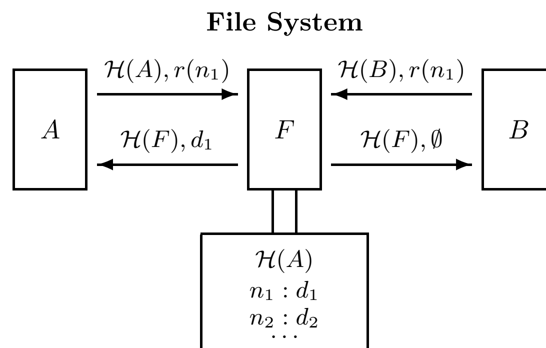


The **sign** application honors requests from the application whose checksum is $\mathcal{H}(D)$, which is to say the application D. Thus, if **display** has been written correctly, **sign** generates signatures only for documents that have been authorized by the user.

4.1 Objects and File Systems

Naturally, there is the need for various applications to share data. We adopt the viewpoint of persistent objects to do so.

The following illustrates a file system object with access control using the hash:



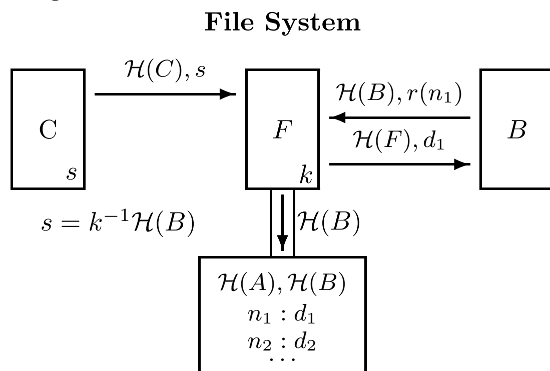
Object F receives two read requests $r(n_1)$. The first comes from A, which appears in F's access control list and is granted (d_1 is returned). The second comes from B, which does not appear in F's access control list and is denied ($\emptyset$ is returned). Different access control lists could be kept for read and write privileges.

More complex objects F with rich method sets can use the same type of contsruction to implement desired access control policies in generality.

Whereas this is a static setup which does not allow us to update the collection of trusted executables, it is easy to create a more dynamic setup.

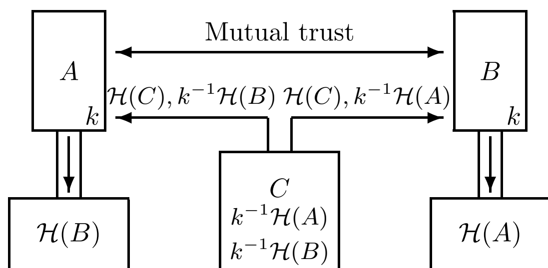
4.2 Access Update

We can update the configuration of the system using digital siguatures:



Here, a helper application C is used to deliver $k^{-1}\mathcal{H}(B)$ to F, which verifies the validity of the signature and adds $\mathcal{H}(B)$ to its access control list. Access requests from B will now be granted by F. The construction depends upon the fact that the bearer of a digital signature does not need to be trusted so long as the signature is valid.

One can use this trick to set up arbitrarily complicated trust relationships using a helper applications:



The helper application C delivers $k^{-1}\mathcal{H}(A)$ and $k^{-1}\mathcal{H}(B)$ to each of A and B. This allows A and B to set up a mutual trust relationship.

Note that this is *not* the same as traditional code signing, which requires an intractable hierarchy of keys, certificates, developer registration, and so forth. The system does *not* use code signing to determine system privileges but rather uses signatures as credentials in a developer software coterie.

An example of where this might be useful is if a bank has several payment schemes that wish to share a common key. The individual components can be updated independently.

4.3 Passwords

There are two serious problems with using passwords to protect several applications on a single device: the first is that it forces the user to remember several passwords and the second is that such schemes are generally vulnerable to Trojan horse attacks. Our architecture allows a simple solution to both problems.

The problem of using a single LED to distinguish between *secure* and *insecure*, as discussed in Section 2.3, can be solved by having a single LED to indicate the secure password entry mode. The password entry program reads in a password p for the excutable E and passes on the value $\mathcal{H}(\mathcal{H}(E), p)$. A Trojan horse application E' will have a different checksum than E means that it will be given a derivative password $\mathcal{H}(\mathcal{H}(E), p) \neq \mathcal{H}(\mathcal{H}(E'), p)$

This eliminates the problem of Trojan horses and allows a single password to be shared by several different applications in a perfectly secure manner (irreproducible, unsniffable, unspoofable, etc.).

4.4 Standard Functions

In order to maintain full PDA functionality, standard interfaces must be provided by each application. A good example of such an interface is that of the function *find*; the feature is sufficiently useful as to be necessary. At the same time the feature should not be abusable, for example, to steal keys (e.g. to find "password").

To provide this functionality, a *find* method must be provided by each application (either by searching or returning the data to search). Those wishing to keep all data private need only return an empty result. In order to maintain a consistent API, these basic methods would likely be supplied in the form of a standard library.

4.5 Higher-Level Functions

Certain higher-level functionality, such as a secure display program (discussed in Section 3.2.1), is not properly an access control issue but is necessary for the creation of a secure system. We discuss this functionality insofar as it has special access control needs.

Source of randomness that collects entropy from the system in usage similar to the device */dev/random* in the Linux [Ts'] kernel.

In many settings it will be useful for the device to have its own key pair used to identify the device, not

the user, and to assist in the installation of applications. This is to help solve the boot strapping problem of public key infrastructures.

Secure backup and restore may also prove problematic.

5 Implementation

To complete the technical section of this paper, we give an overview of the design of a payment system that uses digital signatures employing these access control mechanisms. We design this system to be used through WAP without depending upon the security of WAP itself. We assume that a bank implements the system.

5.1 Global Setup

The initial setup for the bank is not too intrusive. We stress that the bank need not contact the creators of the device.

1. The bank generates a public key/private key pair to sign individual user's keys. We denote this key pair (mk/mk^{-1}). This will often be the bank's master key or some derivative thereof.

2. The bank generates a public key/private key pair to identify membership in the banks suite of applications. We denote this key pair as (ak/ak^{-1}).

3. The bank writes a signing program S that contains the public keys ak and mk.

4. The bank writes a display program D. The display program accepts as a simple description of that which is to be signed (e.g., payee, amount, date, and description). The display program then requests a *lock* on the physical display device. Once obtained, the program displays the necessary information to the user. If the user agrees, the program releases the lock and passes the terms to the signing program S

5. The bank computes $ka^{-1}\mathcal{H}(D)$ and places this in a registration program C (as described in Section 4.2).

5.2 Individual Setup

The particulars of how key hierarchies are established is beyond the scope of this paper. We will assume that the bank wishes to generate and distribute keys, for the users as that is the most difficult scenario. We enumerate the list, as the order in which the tasks are carried out is important.

1. For each user U, the bank generates a key pair (uk/uk^{-1}) and an application CU carrying the signed key pair $mk^{-1}(uk/uk^{-1})$.

2. The bank gives user U the applications D, S, C, and CU.[4] Note that only CU depends upon U and is the only component requiring secrecy.

3. The user installs this applications and the system automatically sets up four new security domains corresponding to the names $\mathcal{H}(D)$, $\mathcal{H}(S)$, $\mathcal{H}(C)$, and $\mathcal{H}(CU)$.

4. The user executes application CU which sends S the message $mk^{-1}(uk/uk^{-1})$. S verifies that uk/uk^{-1} is a valid user key using the key mk. CU then calls C and deletes itself. The application C sends S the message $ka^{-1}\mathcal{H}(D)$. Then S uses ka to verify that D is a trusted application. Hence forth S trusts D.

When an application such as the WAP browser needs to generate a signature, it passes the text to D for display and approval. If the user approves, the request is passed on to S which will actually sign it. S knows that the request reflects the user's desires because it comes from the trusted application D. The signature is eventually returned to the initial application.

If the bank wishes to generate a new application B, say for home banking, trusted by S then it need only generate a helper application C' carrying $ka^{-1}\mathcal{H}(B)$.

5.3 Smartcard

If the bank wishes to use a smartcard to protect the private portion of the user's key pair, then the user steps can be replaced as follows:

1. For each user U, the bank generates a key pair (uk/uk^{-1}) and puts it on the smartcard (possibly signing it $mk^{-1}(uk/uk^{-1})$ with the bank's master key).

2. The bank gives user U the applications D, and C.

3. The user installs this application and the system automatically sets up four new security domains corresponding to the names $\mathcal{H}(D)$, $\mathcal{H}(S)$, and $\mathcal{H}(C)$.

4. The user executes application C, which sends the smartcard the message $ka^{-1}\mathcal{H}(D)$. The smartcard uses ka to verify that D is a trusted application and hence forth trusts D.

In this scheme when the application D makes a request m to the smart card, the request is delivered along with $\mathcal{H}(D)$ as though it were simply another application.

[4] The bank may wish to split the secret in some way.

6 Conclusions

We have analyzed the security requirements of personal digital assistants, introduced a new mechanism for creating security labels, and shown how this mechanism can be used to create secure application suites. The introduced constructions have the salient feature that no security administrator is needed. We feel that this is a necessary property for end user security. Further work will focus on using more sophisticated cryptographic primitives to allow attitional functionality.

References

[BF99] Dirk Balfanz and Edward W. Felten. Hand-held computers can be better smart cards. In *8th USENIX Security Symposium*, August 1999. To appear.

[Com] Internet Community. Open source frequently asked questions. Web page. `http://www.opensource.org/faq.html`.

[Con] WAP Consortium. Wap forum web site. `http://www.wapforum.org/`.

[Cox] Alan Cox. The risks of closed source computing. OS Opinion web site. `http://www.osopinion.com/Opinions/AlanCox/AlanCox1.html`.

[GS91] S. Garfinkel and G. Spafford. *Practical UNIX Security*. O'Reilly & Associates, Inc., 1991.

[PPSW96] A. Pfitzmann, B. Pfitzmann, M. Schunter, and M. Waidner. Mobile user devices and security modules: Design for trustworthiness. Research Report RZ 2784 (#89262), IBM Research, February 1996. accepted for: IEEE Computer.

[PPSW97] Andreas Pfitzmann, Birgit Pfitzmann, Matthias Schunter, and Michael Waidner. Trusting mobile user devices and security modules. *Communications of the IEEE*, 30(2):61–68, February 1997.

[RW] James Riordan and Michael Waidner. *SURE* secure user representation. Project Proposal.

[Ts'] Theodore Ts'o. `/usr/src/linux/drivers/char/random.c`. Linux source code. available from `ftp://ftp.kernel.org`.