Approaches to Open, Interoperable, and Interchangeable Interfaces for Managing Private-Key Cryptographic Data

Case Larsen and William Johnston

Lawrence Berkeley National Laboratory
CTLarsen@lbl.gov, WEJohnston@lbl.gov

December 9, 1996

Abstract

One of the current impediments to the wide-spread use of public-key infrastructure is the practice by commercial-sector suppliers of public-key infrastructure of using proprietary and non-standard interfaces for the management of private-key data. The ability to interchange components is required to allow research to continue to occur in a production environment through the use of alternate implementations of components. We argue that open and interoperable interfaces must be provided for all widely-used components of PKI in order to support diverse security architectures and applications, and that sufficient practice and draft standards exist that provide a substantial basis for these required open interfaces.

Motivation

In order to provide public-key services to a wide variety of applications, several interfaces and data formats must be standardized. In particular, open interfaces that use the private keys for standard signing and authentication; standard encodings of identity certificates and digital document signatures; the message contents and interpretation of request-response protocols must be defined. All must be standardized and completely documented so that cryptographic as well as higher-level security services may be provided through a collection of independent and independently supplied modules.

There are already certificate encoding and directory protocol standards that address interoperability between users of different certificate authority (CA) infrastructures so that an application can make use of a certificate regardless of what CA issued it. The PEM and distinguished encodings (DER form) of x.509v3 [HFS96] certificates and the LDAP directory protocol [LDA96] are examples of standards that are being widely adopted to address interoperability in these areas. The PGP "certificate" (a combination of public-key, user-ids and signature elements) [ASZ95], and associated WWW-based PGP "certificate"-servers are other examples of de-facto standards.

But, in practice, applications share and access data not only in the wide-area infrastructure (public-key certificates and directory services), but data present in the local infrastructure as well (private-keys and trusted verification certificates). The interface to the local infrastructure has been examined to some degree [RSA95, Mic96, Int96, MZB96], but has not been adequately implemented or adopted. Barriers to the use of the local infrastructure can be barriers to the use of the wide-area infrastructure as well because local and wide-area data are intimately related. Locally stored private-keys are paired with public-key certificates distributed in the large, and the locally generated and encoded results of digital-signature are used as input to public-key certificate based verification. The use of proprietary signature encodings or lack of ability of diverse software to fully utilize the private-keys for generating signatures, for example, and the absence of APIs that provide access to local data all serve as barriers.

This white paper addresses the requirements for open interfaces to the use of private-key credentials and other sensitive data: the potential operations and information that an application may require and the degree to which the infrastructure makes the operations and information available. We also address the interchangeability of the private-key management component since being tied to a single implementation on a large scale makes evolution to different architectures through the replacement of components difficult, if not impossible, to achieve. The last requirement is more aimed to enable future research rather than to provide the functionality needed now.

The necessity can be further motivated by two concrete examples. Currently, each suite of applications manages the private-keys that are used by the applications in an ad-hoc manner that prevents or complicates their use elsewhere. For example:

> *Netscape's web browser maintains the user private-keys and trusted certificates in an undocumented encoding format. It is effectively impossible to make use of the private-keys and trusted certificates in another application that is not intimately associated with the web browser*[1]*. Neither the encodings, nor an API for gaining access to the data are provided. Moreover, since access to the data does not occur through a (replaceable) API, credentials acquired through another application cannot be used in place of those initially acquired through Netscape. In addition, we are prevented from switching to another browser since that browser, most likely, will not understand the undocumented formats.*
>
> *PGP maintains the private-keys in an encoding specific to PGP and accessed from within PGP through an internal API not intended for general use with other applications. To make use of a PGP private-key in another application, such as an ACL management system, an application would need to emulate (or incorporate) part of the PGP logic to gain access to the private-keys. This is possible given the source code or data description, but this is not a good solution. The API for the basic cryptographic operations involving the private-key (such as signature and key-exchange) needs to be available to the developer so that they may use the private-key for more general applications than sending and receiving PGP encoded files.*[2]

1. Possibly, not even an applet running on the web browser would be able to make use of private-key credentials. The Java SDK currently does not address the ability to choose private-keys for the purposes of signing.
2. Viacrypt does sell an API toolkit to make use of PGP keyring [Inc96]. If it were to support x.509v3, we'd have the functionality we need. Single copies licenses are $300 for the toolkit and $150 for the required PGP base software. But, one could argue that a solution that costs $450/seat is effectively a "closed" solution.

The currently supported uses of the private-key that these example applications provide, namely to connect to a WWW site through a GUI interface, or to encrypt and sign files for transfer between implementations of PGP, are too high a level of function to be useful for other purposes, such as session establishment protocols like SSL or SPKM. Such protocols use digital-signature and key-exchange as basic operations.

One effect of an "open", common interface to private-key management is that we can acquire credentials from any infrastructure without introducing new client-side software. The other, and more important effect of an "open" interface is that different applications, security libraries, and protocols can share the same private-key credentials (issued by various authorities for various purposes) without prior arrangement (licensing, recompiling, etc.). There can be a common set of private-key services that applications can rely on to be present and that aren't tied to the choice of infrastructure.

Introduction

The IETF draft on Architectures for Public-Key Infrastructure [BA96] outlines the breadth and depth of what makes up a public-key infrastructure. It also points out the drawbacks, exemplified above, of hiding low-level cryptographic interfaces. Here, we concentrate on one of the lowest-levels, namely the way applications store user-specific long-term data, such as private-keys and trusted certificates, to be used with cryptographic services.

Before considering requirements, we first set down definitions of our terminology, and at the same time, touch on issues related to "openness," "interoperability," and "interchangeability".

Definitions:

Openness: - Provides the ability to remove one piece of software and replace it with another (independently produced) implementation because there is enough information available to allow an implementation. Open components interact through the use of:

- <u>Protocols</u>, e.g. TCP, LDAP, HTTP, X11, NFS
- <u>Data encodings</u>, e.g. RFC-822, ASCII, XDR, x.509v3, the input and output of cryptographic transforms
- <u>APIs</u> and explicit ways of linking to or making use of implementations, e.g. POSIX, Java SDK, Motif, RSA BSAFE (but not if its calling interface is hidden away in a monolithic application as it typically is.)

The degree of openness is affected both by how easily an implementor can get information (e.g. freely-available documentation) about the protocols, APIs, and data encodings; and by how restrictive (e.g. patents and licensing agreements, royalties) the designers are about the usage of their protocols, APIs and data encodings.

Interoperability: - The ability of two different implementations (using "open" interfaces) to work together correctly. E.g., the ability to negotiate the algorithms, data encodings, supported extensions, etc., are necessary to enable interoperability. E.g., SPKM with algorithm negotiation allows two different implementations to agree on common algorithms even if most of the algorithms used by both implementations are different.

Interchangability: - The ability to replace one implementation with another. A factor affecting this is the storage of persistent state, such as private-key data, which cannot be understood by another implementation but is required in order to duplicate the behavior. Import and export of keying data, for example, using a common format can allow the interchange of implementations. Another factor is the presence of proprietary features which are needed by other components.

"Abstract" Service vs. Implementation: - We try not to use the terms "service," "operation," or "function," when we actually mean a particular "implementation" instead of "something that is needed regardless of how it is implemented". For example, encryption can be performed by any of a number of algorithms. When talking of encryption as a "service," we are not focusing so much directly on the implementation as we do on its properties (e.g. strength, speed, interoperability with other decryption and key-generation services). Interoperability, though, does depend upon implementation to a large degree. For example, data encodings, if not specified by the API, can prevent interoperability if they vary incompatibly across implementations.

Application Programming Interface (API): - An API stands as a barrier between the service (as it is defined by the API) and its actual (ideally hidden) implementation as it is defined by the implementor. A common API eases the interchange of implementations sharing the same API. However, with cryptography, data such as signature and key encodings and the actual cryptographic data that are passed into and out of the API, almost always reflect the underlying implementation. If two implementations communicate, as is often the case, then these implementations become significant to interoperability. In other words, with cryptographic implementations, common APIs do not buy us much for interoperability -- in fact, by allowing the easy choice among multiple implementations, they provoke the need for interoperability[3]. In addition, many cryptographic implementations keep persistent state, such as private-key data, which may not be understood by other implementations. This "hidden" persistent state prevents the interchange of implementations since the state, private-keys in this example, is essential to the duplicating the behavior of the implementation that is replaced.

Encapsulation: - Hiding individual components within a larger component. Both the interface (API) and the implementations (data encodings and protocol) of the individual components may be hidden, or accessible only through the interface of the larger component. Where one of the encapsulated components provides a function that is commonly-needed by other components, such as digital-signature, this prevents or complicates the use of that component for that purpose.

Private-key and sensitive data management: - The manipulation, storage, and use of private-

3. For example, the "Simple GSS-API Negotiation Mechanism" [BP96] is a protocol that allows two implementations of GSS-API, each supporting multiple protocols to decide if there is a common protocol that they can use. But, there are some choices that cannot be negotiated such as use of a protocol, data encoding, or algorithm that the other end has no knowledge of.

keys and sensitive data, and the provision of operations involving this data to applications and security libraries. Some examples of sensitive data are: long-term encryption keys, trusted CA verification certificates, any configuration information used by the security libraries. Some examples of such operations are: digital-signature, key-exchange, generation of certificate-authority requests.

Public-key infrastructure: - This consists of the CA and the services it provides, the directory service, the application-level cryptographic services, and the protocols, APIs and data encodings for talking between the various pieces. Although the PKI components can be separated by openly documented protocols and data encodings, and can be replaced with different implementations to a large extent, certain aspects of these components cannot be changed without widespread effect. The products of certain services provided by one area of the PKI, such as private-key digital-signature, are used significantly by other parts of the PKI, such as for certificate verification and session establishment. Certain implementation choices, such as whether or not digital-signatures can be produced, thus have widespread impact -- preventing the interoperation with the many components that use digital signatures. In a better, but still less ideal case, these services may be available for use, but only by applications closely tied to the implementations because the private-key management component which provides this service is encapsulated (hidden) in the implementation which provides security services. We concentrate on the private-key management component and argue that this needs to be made open and unhidden.

Key exchange: - Transmission of a symmetric encryption key, in a confidential way, to one (or more) recipients. If a public-key technique is used, the public-key of the recipient is used to encrypt the key data so that only the recipient can decrypt it. The recipient then uses its private-key to decrypt and retrieve the key. This is a basic service provided by public-key algorithms, but unless one, or a select few, data encodings of the key-exchange material can be agreed upon, two implementations will not be able to exchange keys -- even if they both use the same underlying mathematical algorithm.

Digital signature: - proof that data (or as is usually implemented, the one-way hash of the data) originated from the signator.[4] In the public-key technique, the signator uses its private-key to encrypt a hash of the data. Any recipient can then use the public-key of the signator to verify that the (decrypted) hash matches the data, and thus that both the data and hash match and that the data was signed by the signator. This is another basic service provided by public-key algorithms, and the same issue as in “key exchange” applies for “digital signature” encodings.

Requirements

By limiting the available private-key related functions to a high-level interface, such as a message enclosure interface, or to implementations emitting proprietary encodings, the interoperability of one private-key implementation with another is made difficult or impossible. For example, not all

4. Assuming that private-key has never been revealed, the digital signature on a piece of data can only have been generated by the possessor of the private-key.

applications can be expected to use PGP messages as basic, primitive encodings -- especially if the applications are quite different than E-Mail, such as a network based packet-level protocol, or ACL database entries. Not all applications can be expected to adopt a proprietary encoding[5]. The RSA PKCS #11 [RSA95], the Microsoft CryptoAPI [Mic96], and Intel CDSA [Int96] proposals address private-key and sensitive data management functions. Regardless of the interface that is eventually adopted, at least the following low-level information about private-keys needs to be made available to the application that makes use of them.

> *Given multiple private-keys (and associated management and identification information) per entity, the application should be able to obtain the following information about the private-keys of the entity it is representing:*
> - *algorithm usage (e.g. DSS, RSA)*
> - *restrictions on usage (e.g. signature-only, encryption-only)*
> - *associated identifier or certificate -- who or what identity does this key represent?*
> - *key issuer, i.e. where to go to reacquire private-keys.*
> - *probably many more...*

Not all types of sensitive data will fit well into the interface recommendations cited above. For example, a digital-coin system may have no equivalent of “signature” or “key-exchange” and may have additional operations such as debiting and depositing, and providing information such as cash balance that don’t fit well into “signature”, “key-exchange”, or “identity” oriented APIs. Future application uses may have the same mismatch.

> *Domain-specific APIs and protocols are acceptable as long as they are documented and as long as they follow the recommendations below.*

Each application may require different uses of the private-keys. One common denominator is the mathematical operation on the key, whether it be RSA, DSA, Diffie-Helman or any future algorithm. One area of possible incompatibility is the encoding of the bits representing the input, output and the keys used in the various algorithms. The RSA PKCS #1 document [RSA93] suggests an encoding for the keys and data used with the RSA algorithm for example.

> <u>Recommendation:</u> *Basic cryptographic operations involving the private-key should be made available, and should use a widely documented (and implemented) data encoding, e.g. according to the PKCS standards in the case of RSA.*

A vendor could require that the smartcard or private-key management service implement proprietary algorithms or understand proprietary data formats. For example, private-keys could be issued by a CA, but encoded in a proprietary format that only one, or a few, cooperating implementations could understand. To address this problem, Microsoft, for example, spells out the data encodings and algorithms necessary for another implementation to import or export private-key data to the RSA provided implementation. (Appendix A of the CryptoAPI Programmer’s Guide

5. Licensing is one way of acquiring access to a proprietary encoding. “Reverse-engineering” is another way. Both tend to be expensive.

[Mic96]). This having been said, it is acceptable to explicitly hide the encodings from the developer under certain circumstances. The GSS-API [Lin96] specifies the use of opaque "tokens" as input and output parameters. But, this opacity is acceptable because the underlying encodings, e.g. SPKM, have been well-documented elsewhere.

> Recommendation: *Any algorithm or data encoding used in the interface, i.e. as an input or output parameter to the API calls, must either be widely documented or must not be required for normal operation, i.e. negotiable. E.g. the algorithms and data encodings that are used to implement key-exchange, session key derivation, digital signature, hashing.*

Data encodings that are intended to be extensible, such as x.509v3, SPKI [SPK], etc. can suffer from being extended in a proprietary way such that two implementations do not share a common way of communicating. Many protocols specify a set of functionality that must be implemented and allow two implementations to negotiate down to a common set of functionality. [BA96] suggests that "profiles" of algorithms, encodings, and protocols be spelled out for various environments so that within a given environment, implementations adhering to the "profile" will be able to interoperate with one another. [NIS96] begins to profile the choice of signature algorithms and encodings, and the presence and contents of the x.509v3 attributes and extensions for the public-key certificate infrastructure, but there remains a need to reflect these requirements for the infrastructure (such as supported signature algorithms and encodings) back into the requirements for private-key management.

> Recommendation: *Where a "profile" of commonly available algorithms and encodings has been suggested, it should be openly available and followed by implementors.*

A vendor could split keying data among a proprietary infrastructure and an open infrastructure, negating the ability to interoperate with independently produced components. An implementation for sensitive-data management could acquire private-keys through an undocumented protocol and store them in a proprietary form, but provide a signature service through an open, documented API. The integrators would not be able to make use of alternate implementations that knew nothing of the proprietary format or protocols. They would be faced with reverse-engineering the protocol that implements the "acquire private-key" function. It is not enough that an open API is used -- the API must allow other implementations to be substituted in its place with a minimum of effort. If a replacement implementation cannot get access to the keying and certificate data associated with the implementation to be replaced, then effectively, we cannot replace this implementation.

> Recommendation: *Any management of private-key, or otherwise sensitive data, that is essential to normal operation must occur through open APIs so that the underlying implementation may be replaced. At a minimum, persistent state necessary to duplicate behavior should be both importable and exportable via a defined format. E.g. KeyBlobs in [Mic96] and [Int96]. Other APIs may need to be modified. E.g. PKCS #11 does not specify how RSA private-keys are exported, for instance.*

Potential threats for software management of private-keys

The model we assume is:

1. The OS and "root" owned parts of local configuration are trusted. The "root" user is trusted. System libraries and other files that are only writable by "root" are "safe" if the filesystem is local (non-networked)[6]. Although the user is trusted to maintain the integrity of their own environment, applications may want to take extra precautions such as encryption of sensitive data, to ensure this -- especially on multi-user systems where files may be readable by others. In the best case, no stored data sensitive to the user will be readable by anyone other than the user and "root".
2. The user must trust certain applications and libraries completely. E.g. applications and libraries that see the user's password and manipulate their private-data need to be trusted.
3. Many other applications will be granted a restricted interface: e.g. no disclosure of private-data, access to selected credentials, and only a selected few operations with those credentials.
4. The user must take precautions to so that an untrusted application or library is not run in place of a trusted one. E.g. by hardcoding the location of the trusted application into the user's environment and refraining from running any applications that haven't been previously inspected.

Some of the many threats we cannot protect against are:

- Exploiting bugs or misconfiguration in the OS or applications to gain "root" or other user privileges.
- Substituting dynamically linked libraries.
- Masquerading or trojan-horse applications.

Inadequate configuration and protection of the "root" account, inevitable software bugs, or lack of individual user and system administrator vigilance will allow any of these to happen.

We believe that compared to these threats, the added risk due to opening up the private-key management interface is negligible as long as item #3 of the model, a restricted interface for most applications, is provided. The risk is probably no higher than the current level of risk if the threats above continue to remain unaddressed. In the meantime, both users and system administrators need to remain vigilant about monitoring their environment and the applications they run.

Techniques and Approaches

Possible means of providing private-key for use by applications:

1. Provide raw key data
    - + all envisioned uses of keys can be supported.
    - - applications need to understand key encodings, implement algorithms which must conform. In other words, raw data doesn't imply a "standard" encoding. We are still left with the task of deciding on a set of encodings.
    - - private-key is disclosed to application, and possibly other parties. This most likely

6. [BGGW95], for instance, describe NFS specific attacks which undermine an infrastructure that requires filesystem integrity.

weakens the assurance of the digital signature.

2. Provide function-call interface, e.g. PKCS #11
   - + private-key can be hidden (in hardware or through software techniques)
   - + applications do not need to know about algorithm implementations, instead the uses of algorithms: signature, key-exchange.
   - - only provided functions can be performed with private-key. (A lowest common denominator must be chosen. If not, some potential application uses will be left out.)
   - - We may need more than a function-call interface. For example, an application operating on a remote machine may need to make use of private-key credentials stored on a different machine. This is effectively a remote procedure call (RPC) interface, and since the data involved is sensitive, the RPC must be secure.
   - issues: function API (input and output encodings) must be open, e.g. PKCS standards for message encodings, to allow interoperation across implementations. The data encoding and algorithms can be thought of part of a protocol that communicates between cryptographic implementations.

Potential implementations:

1. Hardware smart card (PC-MCIA cards)
   - + private-key (and other sensitive) data is secured in tamper-proof hardware
   - + there are recommendations for API (PKCS #11)
   - - token readers are not widely deployed.
2. Hardware Smart floppy (GSA, SmartDisk(TM))
   - + existing readers and writers: 3.5 inch floppy drives
   - - standards: will the industry support this in the long run?
3. Software “smart card”
   - + cheap to install (does not require additional hardware).
   - - no widely adopted or implemented standard for software smart-card API. PKCS #11 or equivalent could be used. SecuDE’s PSE (personal security environment) [SECUDE] is another example. The SSH agent protocol [Ylo96] can provide networked (as well as local) access to private-keys, but requires an API and support of widely used encodings, such as PKCS #1, to be used as a general-purpose “smartcard”.
   - - key data is not protected as well as a hardware solution.
   - issue: applications, such as Netscape, Microsoft, and Nortel’s Entrust, currently have their own private-key, and sensitive data storage formats. There is no commonly available implementation of an API to store and retrieve private cryptographic data, so each vendor performs their own implementation. While an API is useful, it seems there will be as many APIs as vendors -- at least until a marketable software “smartcard” product appears that vendors can assume the presence of. Is there a market for a “software smartcard” when the hardware approaches address physical security so much better? An API such as PKCS #11 could be adopted as the interface to a software “smart card”.

Recommendations

- Adopt appropriate parts of PKCS #11 (or Microsoft CryptoAPI, or Intel CDSA) to provide private-key management functions to both the application and the security infrastructure.
- Provide basic cryptographic operations using widely documented data encodings and algo-

rithms. Examples include: digital signature and key exchange using RSA and PKCS #1 encodings.

- Conform to the “profile” for the environment. Always have at least the required algorithms and data encodings available before providing experimental or vendor-specific extensions.
- If storage formats are not documented, provide a way to export and import keying data and other persistent state so that a component may be interchanged.

References

[ASZ95] D. Atkins, W. Stallings, and P. Zimmerman. PGP message exchange formats. Internet Draft, January 1995. , Available at http://world.std.com/~franl/pgp/draft-rfc.html.

[BA96] Bob Blakley and APKI Working Group. Architecture for public-key infrastructure. Internet Draft, November 1996. Available at ftp://ftp.es.net/pub/internet-drafts/draft-ietf-pkix-apki-00.txt, Internet Draft (PKIX Working Group).

[BGGW95] Eric Brewer, Paul Gauthier, Ian Goldberg, and David Wagner. Basic flaws in internet security and commerce, 1995. Available at http://HTTP.CS.Berkeley.EDU/~gauthier/endpoint-security.html,

[BP96] Eric Baize and Denis Pinkas. Simple GSS-API negotiation mechanism. Internet Draft, October 1996. , Available at ftp://ietf.org/internet-drafts/draft-ietf-cat-snego-02.txt.

[HFS96] R. Housley, W. Ford, and D. Solo. Internet public-key infrastructure, Part I: X.509 Certificate and CRL profile. Internet Draft, February 1996. Available at ftp://ftp.ietf.cnri.reston.va.us/internet-drafts/draft-ietf-pkix-ipki-part1-% 01.txt, Internet Draft (PKIX Working Group).

[Inc96] Pretty Good Privacy, Inc. Viacrypt PGP for windows toolkit. product summary, 1996. , Available at http://www.pgp.com/products/viacrypt-toolkit.cgi.

[Int96] Intel. Common Data Security Architecture, 1996. Available at http://developer.intel.com/ial/security/index.htm, An architecture for security services: algorithms, certificates, trust, and data storage. The architecture is designed to accommodate multiple implementations for different services.

[LDA96] Lightweight Directory Access Protocol, 1996. Available at http://www.umich.edu/~rsug/ldap/ldap.html,

[Lin96] John Linn. Generic security service application program interface, version 2 (draft). Internet Draft, August 1996. Available at ftp://ftp.ietf.org/internet-drafts/draft-ietf-cat-gssv2-08.txt, Internet Draft (CAT Working Group).

[Mic96] Microsoft. CryptoAPI version 2.0, 1996. Available at http://www.microsoft.com/intdev/security/capi/capiref.zip and http://www.microsoft.com/intdev/security, An architecture for security services: algorithms, certificates, trust, and data storage. The architecture is designed to accommodate multiple implementations for different services.

[MZB96] Montserrat Mane and Sylvie Zapata-Brel. Introduction to the personal security module (PSM), March 1996. , Available at http://www.gr.osf.org/book/psm-wppr.htm.

[NIS96] NIST. Minimum interoperability specifications for PKI components (MISPC), December 1996. Available at http://csrc.nist.gov/pki/mispc.ps,

[RSA93] RSA. PKCS #1, 1993. Available at ftp://ftp.rsa.com/pub/pkcs/ascii/pkcs-1.asc, This describes an encoding of RSA private and public-keys, encryption and signature data formats.

[RSA95] RSA. PKCS #11, 1995. Available at ftp://ftp.rsa.com/pub/pkcs/ps/pkcs-11.ps, This describes an API for management of sensitive data and provision of cryptographic functions.

[SPK] Simple public-key infrastructure (IETF working group). , Send "subscribe spki" to majordomo@c2.org to join. Another public-key infrastructure working group has started in the IETF (SPKI). This group has formed to explore public-key certificates that are attribute-based instead of name-based, for use in a variety of Internet applications.

[Ylo96] Tatu Ylonen. The SSH (Secure Shell) remote login protocol. Internet Draft, 1996. , See also "SSH (Secure Shell) Remote Login Program" (http://www.cs.hut.fi/ssh/).