2-1-2005

SHEMP: Secure Hardware Enhanced MyProxy

John Marchesini
Dartmouth College

Sean Smith
Dartmouth College

SHEMP: Secure Hardware Enhanced MyProxy

John Marchesini and Sean Smith
Technical Report TR2005-532
Department of Computer Science
Dartmouth College
{carlo,sws}@cs.dartmouth.edu *

Abstract

While PKI applications differ in how they use keys, all applications share one assumption: users have keypairs. In previous work, we established that desktop keystores are not safe places to store private keys, because the TCB is too large. These keystores are also immobile, difficult to use, and make it impossible for relying parties to make reasonable trust judgments. Since we would like to use desktops as PKI clients and cannot realistically expect to redesign the entire desktop, this paper presents a system that works within the confines of modern desktops to shrink the TCB needed for PKI applications. Our system (called Secure Hardware Enhanced MyProxy *(SHEMP)) shrinks the TCB in space and allows the TCB's size to vary over time and over various application sensitivity levels, thus making desktops usable for PKI.*

1 Introduction

Because public-key cryptography can enable secure information exchange between parties that do not share secrets a priori, *Public Key Infrastructure* (PKI) has long promised the vision of enabling secure information services in large, distributed populations. A number of useful applications become possible with PKI. While the applications differ in how they use keys (e.g., S/MIME uses the key for message encryption and signing, while client-side SSL uses the key for authentication), all applications share one assumption: users have keypairs.

Where these user keypairs are stored and used is the primary focus of this research. Traditionally, users either put their key on some sort of hardware device such as a smart card or USB token, or they place it directly on the hard disk such as in a browser or system keystore. Most modern operating systems (such as Windows and Mac OSX) include a keystore and a set of *Cryptographic Service Providers* (CSPs) which use the key. In fact, many cross-platform software systems, such as the Java Runtime and the Netscape/Mozilla Web browser include their own keystore so that they may use a user's keypair without having to rely on the underlying OS (thus enhancing portability).

1.1 Keystores

Most keystores fall into one of four basic categories:

- A *software token* stores the key on disk (most likely in some sort of encrypted format). Examples of this approach include the default CSP for Windows and the Mozilla/Netscape Web browser.
- A *hardware token* stores the key and performs key operations. The interaction between an application and the key is typically mediated by the OS (although in some cases, the application may interact with the device directly). In order for the OS or application to be able to speak to the token, the token vendor must provide a driver for the device which adheres to one of the two common standards for communicating with cryptographic devices: the CryptoAPI (CAPI) for Microsoft, and RSA's PKCS#11 for the rest of the world. Examples of hardware tokens include the Aladdin eToken and Spyrus USB tokens, as well as more powerful devices (sometimes referred to as *cryptographic accelerators* or *Hardware Security Modules* (HSM)).
- A *secure coprocessor* stores the key, can perform key operations internally using cryptographic hardware, and can even house the applications directly, such as the IBM 4758 [3, 30]. These devices can also be used as cryptographic accelerators or HSMs.
- A *credential repository* is a dedicated machine that stores private keys for a number of users. When a user Alice wishes to perform key operations, she

*Preliminary versions of some of this research appear in Dartmouth College Technical Report TR2004-525 [11].

must first authenticate to the repository. The repository then certifies a temporary key with Alice's permanent key via a digital signature, or actively participates in the requested key operation. Examples of credential repositories include MyProxy [18], hardened MyProxy [9], and SEM [2].

In previous work, we examined the security aspects of some of the standard keystores and the their interaction with the desktop [15, 16]. We concluded that software tokens are not safe places to store private keys, and we demonstrated the permeability of keystores such as the Microsoft default CSP and the Mozilla keystore. Our experiments showed that in many cases, an attacker can easily *keyjack*: either steal the private key or use it at will.

In addition to being unsafe, standard software keystores have the disadvantage of being immobile. Once a private key is installed on a desktop, the only way to transport it to another machine is to export it and re-import it on the new machine. Since this process can make the key vulnerable to attack, such solutions may often offer mobility at the expense of security. As user populations become more mobile and begin to use multiple devices, this immobility becomes more problematic.

Hardware tokens claim to solve both of these problems; they get the key off of the desktop and give users mobility. We experimented with these devices and found that an attacker is typically still able to use the key at will. However, with respect to mobility, devices such as USB tokens can add some benefit, provided that the appropriate software is installed on each machine and that users use supported OSes (but the tokens we experimented with did not have Apple or Linux support at the time of our experimentation).

We concluded that the security problems of software and hardware tokens stem from the facts that the *Trusted Computing Base* (TCB) is too large and ill-defined, and that usability issues make it hard for users and application developers to "do the right thing" [15, 16]. These shortcomings make it impossible for relying parties to make reasonable trust judgments about the system.

Secure Coprocessors Secure coprocessors are a combination of physical armor and software protections that create a device that possesses a different security domain from its host machine. Such devices can be used to shrink the TCB, and have been shown to be feasible as commercial products [3, 30] and can even run Linux and modern build tools [6]. We have explored using secure coprocessors for *trusted computing*—both as general designs (e.g., [21]) as well as real prototypes (e.g., [7])—but repeatedly were hampered by their relatively weak computational power. Their relatively high cost also inhibits widespread adoption, particularly at clients. Their lack of ubiquity, coupled with their sometimes awkward programming environments lead us to conclude that secure coprocessors are difficult to use, especially for application developers.

In other previous work, we used the *Trusted Computing Group's* (TCG) specifications and hardware (a device known as the *Trusted Platform Module* (TPM)) to secure an entire desktop [10, 13, 14]. While the security properties of our platform (called *Bear*) are not as strong as a secure coprocessor such as the IBM 4758, our approach shrinks the TCB of a general purpose desktop.

Credential Repositories Credential repositories can provide safe storage facilities for private keys as well as give users mobility. The repository approach allows an organization to focus security resources on the repository, thus providing economies of scale. In terms of secure key storage, repositories significantly shrink the TCB. The private key no longer relies on a general purpose and buggy desktop for safe storage, but instead on a dedicated server which is presumably administered by a professional. Repositories allow users to access their private key from multiple machines, thus giving them mobility.

However, when a user Alice wishes to use her private key to perform some operations, she must either bring it to her desktop, or design or use a protocol which allows her to use the private key on the repository and rewrite her application to use the new protocol. Thus, repositories can be difficult to use, especially for application developers.

Recently, a credential repository has been developed by the Grid computing community which provides both security and mobility to clients. Their repository is called *MyProxy* [18], and there have even been efforts to harden a MyProxy repository by using an IBM 4758 for key storage and cryptographic operations [9].

1.2 The Problem

As we will establish in Section 2, a usable key storage solution must be secure, must be usable, must give users mobility, and must allow relying parties to make reasonable trust judgments. As we have discussed in this section, none of the current approaches meet this criteria. Table 1 summarizes the status quo.

SHEMP The status quo is not satisfactory. Ideally, we need a way to use a desktop as a PKI client that answers "yes" in all of the columns of Table 1. Since we cannot redesign the entire desktop and expect anyone to use it, our solution must operate within the confines of modern desktops. Additionally, in order to remain usable to appli-

Keystore	Secure	Usable	Mobile
Software Token	no	no	no
Hardware Token	no	no	maybe
Coprocessors	yes	maybe	no
Repositories	maybe	no	yes

Table 1: A summary of modern keystores.

cation developers, it must adhere to common development paradigms and practices.

Our solution (SHEMP) builds on MyProxy, secure hardware, and policy tools. We extend the MyProxy approach by taking advantage of potentially heterogeneous secure hardware on the client and repository. We also extend the MyProxy design by exploring the use of *Proxy Certificates* (PCs) for applications beyond mere authentication. We use the *eXtensible Access Control Markup Language* (XACML) to provide a mechanism which allows users to specify their key usage options based on the client and repository properties. We have built a SHEMP prototype and constructed a testbed and have conducted performance and user studies. The repository and one client currently run on our Bear TCPA/TCG platform. All of this code will be available for public download.

This Paper Section 2 examines the problem in detail and Section 3 discusses the criteria that a solution to the problem should meet. Section 4 discusses the SHEMP toolkit and Section 5 applies those tools to build the SHEMP system. In Section 6, we offer an evaluation of SHEMP and Section 7 concludes.

2 The Problem

The problem that we are attempting to solve is that modern desktops are unsuitable for use as PKI clients. They can allow a user's private key to be stolen or used at an attacker's will, they make it difficult for users (and application authors) to do the "right thing", they are inherently immobile, and they do not allow relying parties to make good trust judgments about the system (i.e., they allow the key to be used for transactions which the user was not aware of or did not intend). A more detailed description of the experiments used to draw this conclusion can be found in previous work (see [15, 16]); this section presents a brief analysis of some of those results.

Software A major cause of the problem is the complexity of modern software. This complexity makes it difficult or impossible for users to draw conclusions about a given computation's results. Complexity also decreases the system's usability, and decreased usability often results in decreased security. Complexity also expands the set of software that must be trusted in order for the system to operate correctly. This set of software is often referred to as the TCB. A good discussion of the TCB can be found in the "Orange Book" [19]. A small TCB minimizes the attacker's target and maximizes the chance for developers to build secure systems.

Placing a private key on such a complex system is problematic. By exploiting the complexity, it is possible for an attacker to trick users into giving away their key directly, or use it for purposes which they are unaware of or did not intend. By exploiting the fact that so much of a complex system needs to be trusted in order for it to behave correctly, it is possible for an attacker to either get the key directly, or be able to use it at will without alerting the key's owner. We found that getting one user-level executable (i.e., our *keyjacking malware*) to run on the client is enough to accomplish a successful attack.

Hardware Many in the field suggest getting the private key off of the desktop altogether and placing it in a separate secure device of some sort. Taking the key to a specialty device such as an inexpensive USB token would seem to reduce the likelihood of key theft as well as shrink the amount of software which has to be trusted in order for the system to be secure. At first glance, it would appear that just the device and the software which provides access to the device (i.e., its CSP) need to be trusted. We found that relying on such a device is also problematic. While putting the private key on a token gives some physical security and makes it harder to steal the key, we found that it does not shrink the TCB (since the adversary can still borrow the key via host-side attacks).

Secure coprocessing is an improvement from a security standpoint, but it is not a magic bullet either. From a practical standpoint, high end devices such as the IBM 4758 are far too expensive to deploy at every client. On the other end of the spectrum, lower priced devices (e.g., the TPM) cannot withstand many common attacks (such as hardware attacks, or attacks from root) without additional measures (e.g., aid from the processor, such as what is being considered in the literature [8, 17, 31, 32]).

Immobility In addition to the security and cost considerations mentioned above, the desktop PKI client paradigm suffers another problem: immobility. Modern computing environments are becoming increasingly distributed and user populations are becoming increasingly mobile. Moreover, the number of computing devices that a typical user owns is growing. It is not uncommon for someone to own a desktop, a laptop, a cell phone, and a PDA. Which device(s) should house the private key?

One proposal is to use inexpensive tokens such as USB tokens and allow users to carry their token with them across devices and computing environments. This approach has a number of drawbacks in addition to the security problems mentioned above. First, some devices may not have the proper hardware or software installed, or may not have support altogether. Second, a particular machine may not be trustworthy, or may have malware installed which abuses the private key. Again, putting the private key in a token does not shrink the TCB.

Another proposal is to move the key around on some removable media (e.g., a floppy) and export the key to some intermediate format (e.g., PKCS#12) and then import the key at the destination. This approach suffers a number of drawbacks as well. First, some devices may not support the media—e.g., we are unaware of cell phones with floppy drives. Second, the intermediate format may be insecure (e.g., as Peter Gutmann has demonstrated with his `breakms` [5] tool).

3 Criteria for a Solution

In order for *any* proposed solution to succeed in making desktops usable for PKI, it must address a range of issues including security, usability, and mobility. For the solution to be of any practical interest, it must safely store and use the private key, give application developers flexibility while maintaining security, match the model of real world user populations, and allow relying parties to make reasonable judgments about the system.

3.1 Security

The notion of security is difficult or impossible to quantify in a practical system. Within a formal framework, one can prove that a system is secure, but once the formal frameworks give way to implementations, problems often arise. We let the operating definition of security in this paper involve minimizing the risk, impact, and window of opportunity for misuse of a user's private key.

3.1.1 Minimizing Risk

The TCB and Security We define the TCB for PKI applications to be the private key and the set of software which stores and uses the private key directly (e.g., libraries that make up constructs such as the CryptoAPI). The security trouble of Section 2 results from the fact that this set of software is intertwined with the OS and applications (such as Internet Explorer), and no clear boundaries exist. The result is that the entire system must be trusted in order for the system to be secure; just one well-crafted piece of malware can subvert the entire desktop, rendering it ineffective as a PKI client.

The TCB and Secure Hardware Secure hardware can reduce the size of the TCB. Highly secure devices such as the IBM 4758 [3, 30] can effectively create an entirely separate security domain from their host. Since the device has a general-purpose OS, it can house applications as well as critical data (e.g., private keys), thus eliminating the need for the private key to come in contact with the host desktop at all. The end result is that the entire TCB can be placed in such a device and be totally protected. However, as discussed in Section 1, devices like the IBM 4758 are expensive, which prohibits their widespread use on client platforms.

While few devices can isolate the TCB to the extent of the IBM 4758, other devices can reduce the TCB. For instance, many HSMs can get the key off of the desktop, perform key operations internally, and protect against physical attack. However, the applications may still live on the desktop, leaving the device only as secure as the CSP and leaving part of the TCB on the desktop (hardened MyProxy follows this approach [9]. Other approaches, such as our "Bear" [10, 13, 14] project and others [26], can use a TPM to extend a weaker level of security to the entire desktop. This may be sufficient, depending on the threat model.

Use Secure Hardware When Available In SHEMP, machines which house users' private keys are called *key repositories*. Machines which actually use the key on a user's behalf are called *clients*. We envision (and prototyped) a heterogeneous environment, where machines (both repositories and clients) may have very secure hardware such as an IBM 4758 secure coprocessor, less secure hardware such as our "Bear" platform, or no secure hardware at all.

Organizations which aim to provide high levels of security will adopt a threat model which assumes a powerful attacker. Under such a threat model, key repositories should be able to withstand a wide range of attacks. For instance, an organization may wish to assume that an attacker can get root privileges on the repository's host machine. This would imply that the attacker can watch any process's memory, and run any code of his choice on the host. Furthermore, the organization may wish to assume that an attacker has physical access to the secure hardware holding the private keys and can attempt to perform local hardware attacks. As a result, that organization's repository should be able to resist local physical and software attacks, and should refuse to disclose any user's private key,

even if the attack is running with root privileges. In practice, this may involve using a device such as an IBM 4758 to house the repository, thus giving the repository a different security domain than its host. The threat model for clients may be different, and a successful solution should allow for such variations and be flexible enough to accommodate clients with a range of security levels, as well as provide a means for expressing those security levels.

Moving the TCB off of the Desktop Roughly speaking, the larger the desktop-resident TCB, the greater the risk of key disclosure. If we assume that client machines will be running standard OSes, then we should minimize the amount of the TCB that resides on the client machine. The ideal scenario is one in which no part of the TCB comes into contact with the client machine, although this approach also makes desktops unusable as PKI clients (as no part of the desktop—including applications—can be used in any PKI operation). A compromise could consist of keeping the TCB out of the reach of a desktop until some portion of the TCB is needed, and only then, would we embed part of the desktop in the TCB.

3.1.2 Minimizing Impact

In order to minimize the impact of a key disclosure and the window of opportunity for an attacker to misuse the key, we need some way to control the lifespan during which a compromised key can be used. A short key lifespan reduces the opportunity for misuse. However, just issuing short-lived private keys to the population would increase the already cumbersome administrative burden of the PKI, as users would have to be re-keyed frequently. To remedy this, a number of systems rely on *delegation* to control the lifespan of a user's credentials. By using delegation, we can issue a temporary credential which, when evaluated in conjunction with a long-term credential such as a PKI certificate, allows a relying party to make a reasonable trust judgment. Should a short-term credential be compromised by an attacker, the attacker can only misuse the temporary credential for a short period of time, thus minimizing the impact and window of opportunity for misuse.

3.2 Usability

The second feature that a proposed solution should provide is usability. Users, administrators, and application developers must be able construct accurate mental models of the system. From a user's perspective, the system must be easy to use. A design strategy which can enhance usability involves hiding complexity from the user. Clearly, there is a balance to be achieved; hiding too much complexity can have adverse effects, as can exposing too much. The system should hide enough complexity so that users are not overwhelmed by configuration options (in which case, they are likely to misuse and/or misconfigure the system—most likely resulting in security trouble). However, enough complexity should be visible so that users can construct a valid mental model of the system.

The requirements for an administrator's view can be different. Typically, an administrator is a special entity who has a deeper knowledge of the system, and as a result, can be burdened with some of the complexity. In many scenarios, it is the administrator's role to insulate the end user from complexity. However, in order to deal with this complexity, administrators should be given tools which aid in system configuration and use. Furthermore, the toolkit should make it difficult for administrators to do the wrong thing, and easy for them to do the right one.

In order to get parties to write applications for the system, it should expose common programming paradigms to developers, and allow them to use the solution to build and deploy real applications. This requires that the platform must be easily programmable with modern tools, and must also allow easy maintenance and upgrade of its software. Forcing developers to conform to awkward or constraining mechanisms limits the usability of the system from a developer's viewpoint.

3.3 Mobility

The third feature that a proposed solution must provide is mobility. Modern user populations increasingly use multiple computing platforms from multiple locations. Many current PKI systems either make it difficult for the user to move their private key or make it vulnerable to attack during and/or after transit. A PKI solution must allow users to move throughout their domain, and across their computing platforms. Most importantly, the solution should not put the private key at risk of disclosure any time the user moves geographically or uses different devices. A good solution should take into account the trustworthiness of the client platform, thus disallowing the key to migrate to untrustable client machines (or severely limiting its use).

4 Our Building Blocks

Our primary building blocks consist of three categories: a *credential repository* (MyProxy) and *delegation framework* (Proxy Certificates (PCs)) which are used to get keys off of the desktops and give users mobility; *secure hardware* which can be used as the basic keystore, both at repositories and clients, when available; and a *policy language* which is used to express key usage and delegation

policies at the repository as well as express attributes of repositories and clients.

4.1 MyProxy and Proxy Certificates

The first component we use to build SHEMP is the MyProxy credential repository, which we use to shrink the TCB and give users mobility [18]. MyProxy was originally designed to allow Grid users to obtain and delegate access to their credentials from multiple locations on the Grid. Modern versions of MyProxy [9] use the repository to store a long-term credential, thus getting the private key off of the user's desktop altogether (in fact, Lorch et al. [9] store the key in an IBM 4758 at the repository). When a user, or process running on a user's behalf, needs to use a credential for authentication or authorization, it logs in to the MyProxy repository and requests that a short-lived PC be generated. The PC along with the user's long-term credential can then be used for authentication or authorization.

The MyProxy system is attractive for two reasons. First, it gets the user's private key off of the desktop entirely, and thus shrinks the TCB. When a user or process needs to use a credential, the TCB expands to include the desktop (via delegation)—but only for short period of time. This approach shrinks the TCB in space and time which, in turn, gives MyProxy a security advantages over the standard desktop PKI approach. Second, the MyProxy system gives users mobility. Since the user's private key is stored in a central location, it can be accessed from many locations without having to be transported by hand (i.e., exporting/re-importing or using a protocol like Sacred [23, 24, 25]).

Proxy Certificates The second component we use in SHEMP are PCs. PCs allow us to expand the TCB for to cover a client machine for a short period of time. We chose X.509 Proxy Certificates for a number of reasons. First, they are standardized by the IETF and are awaiting an RFC number assignment. Second, because they are X.509-based, they can be used in many places in the existing infrastructure that are already outfitted to deal with X.509 certificates. Third, they are widely used in the Grid community and are used in the MyProxy system and in the dominant middleware for Grid deployments: the Globus Toolkit [4]. Fourth, they allow dynamic delegation without the help of a third party, allowing clients to obtain a PC without having to endure the cumbersome vetting process at the *Certificate Authority* (CA). Last, the PC standard defines a *Proxy Certificate Information* (PCI) X.509 extension which can be used to carry a wide variety of (possibly domain-specific) policy statements (e.g., XACML statements, discussed below).

4.2 Secure Hardware

The third component we use in SHEMP is secure hardware, which allows us to shrink the TCB of each machine. Over the years, our lab has built a number of systems which involve and/or enhance secure coprocessors. Secure hardware is interesting in the context of SHEMP because it can be used to reduce the size of the TCB, thus reducing the risk of a key disclosure.

Most of our initial systems were constructed around the IBM 4758, as the second author brought it to the PKI Lab from IBM [3, 27, 30]. Members of our group have used these devices to enhance privacy [7], harden PKI [12, 28], and enhance S/MIME [21].

The IBM 4758 is a secure coprocessor which provides secure storage facilities, cryptographic acceleration, and a platform on which to run third-party applications. The IBM 4758 is a very secure device, having been validated to FIPS 140-1 Level 4. It can withstand both software and hardware attacks, and effectively provides a different security domain from its host machine. A useful feature of the IBM 4758 is what it calls *Outbound Authentication* (known as *attestation* in many other contexts), which enables applications running inside of the IBM 4758 to authenticate themselves to remote parties [29]. A good overview of the IBM 4758 and its capabilities can be found in the literature (e.g., [3, 27, 30]).

More recent projects have involved constructing a "virtual" coprocessor out of commodity hardware. Our initial design and prototype was based on the TCG specification (see [20, 33, 34, 35]) and was called "Bear" [13]. The Bear platform is less secure than the IBM 4758. It does provide a means to ensure file integrity for files which a possibly remote *Security Admin* decides are necessary. However, since the design is based on the TCG specification and hardware, it is susceptible to local hardware attacks, as well as attacks from root [10, 13]. Bear has a mechanism which allows it to "attest" to the integrity of the platform when challenged. The TCG specifications refer to this mechanism as *attestation*. More information about Bear can be found in previous work (e.g., [10, 13]), and a summary of the attestation mechanism can be found in earlier work [10] as well as the literature (e.g., [20, 26, 33, 34, 35]).

4.3 Policy

The last tool we use in SHEMP is a policy framework. In order to enhance SHEMP's the expressiveness and usability, we want to give users a way to relay their wishes regarding key usage to relying parties and applications—and to the SHEMP system itself. Further, we want the SHEMP system to be able to convey attributes of both key repositories and clients to relying parties.

In one role, the policy framework should allow a relying party Bob, upon receiving a PC from Alice, to be able to discover the conditions under which Alice's PC was generated. Then, Bob can decide for himself whether to trust Alice, given her current environment. As we will explore in detail in Section 5, SHEMP administrators assign attributes to clients and repositories. When Alice makes a request for the repository to generate a PC for her, the repository will include the attributes of the client desktop and the repository in the PC itself (in the PCI extension). These attributes essentially define Alice's TCB. When Alice presents her PC to Bob, he can examine the attributes himself, and then make a trust decision based on Alice's TCB.

In another role, the policy framework should allow a keyholder Alice to express her wishes about uses of her private key—potentially based on the security level of the repository and end client platform. For example, users may wish to restrict access to cryptographic operations that the repository will perform with their private key; applications may wish to restrict certain data or operations. Without this ability, a successful attacker could fully impersonate the victim or use the victim's key for any operation. The policy framework must be flexible enough to allow SHEMP administrators to specify domain-specific attributes to machines, and easy enough to use that users and application developers can construct policies which accurately govern their resources.

We chose to use XACML [36]. XACML is an XML-based language for expressing generic policies and attributes. A *Policy Decision Point* (PDP) takes a policy and a set of attributes, and makes an access control decision. We chose XACML because it is generic enough to express a wide range of attributes, and has an open-source implementation [22] which is implemented in the language of our prototype: Java. As we will show in Section 6, it is possible to build XACML-generating policy tools which make XACML easy enough to use for administrators and application developers.

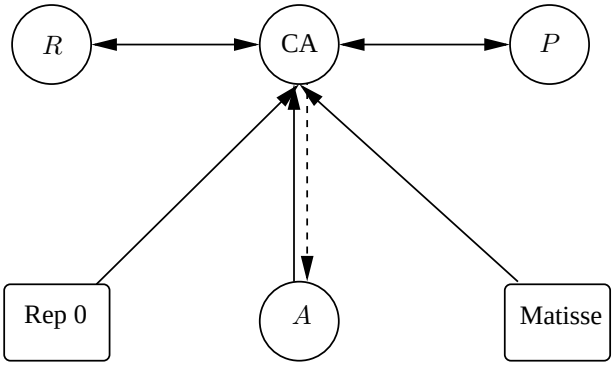

Figure 1: The parties in the SHEMP system. The circles represent individuals or organizations and the boxes represent machines. The arrows indicate trust relationships between the parties; an arrow from A to B means "A trusts B".

5 SHEMP

The goal of SHEMP is to allow a relying party Bob to be able to make valid trust judgments about Alice upon receiving a Proxy Certificate from her—and this validity must reflect the opinions of Bob and Alice about the desktop infrastructure involved. Bob should have some reason to believe that Alice authorized the issuance of her Proxy Certificate for the intended purpose(s), and that the private key described in the Proxy Certificate is authentic. Equipped with the tools of Section 4, we designed and implemented the SHEMP system.

When a user Alice wishes to use her private key, she logs into the SHEMP repository from her client desktop, generates a temporary keypair on her desktop, and then requests a PC which includes the public portion of her temporary keypair and is signed by her permanent private key on the repository. The PC is only valid for a short period of time, and includes a snapshot of the environment in which the PC was generated. This snapshot describes the security attributes of the repository and client desktop, and allows applications to decide for themselves how trustworthy the private key described by PC really is. The SHEMP system attempts to leverage secure hardware when it can, but it does not require secure hardware. Concretely, SHEMP allows keypairs on the repository and the client to be generated and used in secure coprocessors. Additionally, the framework for describing the security attributes of repositories and client desktops allows users and administrators to express the presence and quality of secure hardware—and relying parties to use this information when making their trust judgments.

5.1 The SHEMP Architecture

The Players Initially, there are three familiar parties involved in SHEMP: a CA, a user Alice (A), and a user's machine (Matisse). As in any typical PKI, Alice trusts her CA to certify members of her population (including herself). This relationship is depicted as a solid arrow from Alice to the CA in Figure 1. In order for the CA to trust Alice, it must believe her identity and that she has the private key matching the public key in her certificate request (typically a *Registration Authority* verifies Alice's identity on the CA's behalf). Once the CA believes Alice's identity is authentic and that she owns the private key, the CA will express its trust in Alice in the form of a CA-signed identity certificate. This relationship is depicted as a dashed edge from the CA to Alice in Figure 1.

For an application running on Alice's machine (Matisse) to trust certificates signed by the CA (such as Alice's), it usually needs to have the CA certificate installed. This relationship is represented by the edge from Matisse to the CA in Figure 1. To illustrate a concrete example of the necessity of this relationship, assume that some organization uses S/MIME mail. If Alice and Bob both have identity certificates signed by the CA and Bob sends Alice a signed message, then Alice's mail program needs to know Bob's certificate and it needs to trust the entity which vouched for Bob's identity (the CA).

In addition to the three familiar parties described above, the SHEMP system introduces three more: a *Repository Administrator* (R in Figure 1) who runs the key repository(s), a *Platform Administrator* (P in Figure 1) who is in charge of the platforms in the domain (such as Matisse), and at least one key repository (*Rep 0* in Figure 1).

The Repository Administrator is in charge of operating the key repository. Since the repository contains the entire population's private keys and is thus a target for attacks, it must be maintained with care. Concretely, the Repository Administrator is in charge of loading private keys into the repository and vouching for the repository's identity and security level (these will be discussed below). Thus, it is necessary for the CA to trust the Repository Administrator. Since the Repository Administrator is a member of the CA's domain (in fact, probably part of the same organizational unit—such as Dartmouth College Computing Services), it trusts the CA as well. This relationship is depicted by the edge connecting the Repository Administrator to the CA in Figure 1.

The Platform Administrator is in charge of the platforms that end users (e.g., Alice) will use. At the base level, the Platform Administrator has the same responsibilities as a typical system administrator: configuring machines, installing and upgrading software, applying patches, etc.

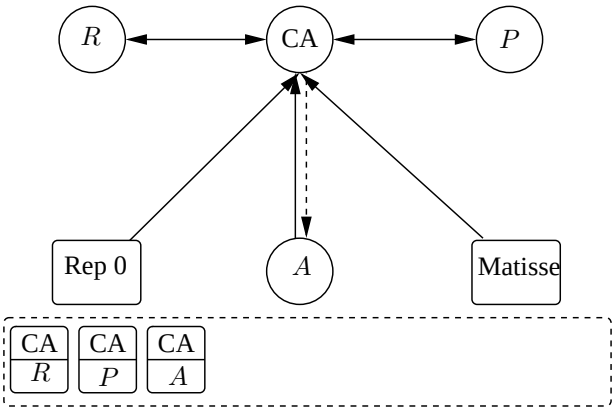

Figure 2: The entities, trust relationships, and initial certificates in SHEMP. The boxes inside of the dashed area represent certificates. In this figure, all three certificates are signed by the CA, and are issued to the Repository Administrator, Platform Administrator, and Alice respectively.

Additionally, the Platform Administrator is in charge of creating and vouching for platform identities and security properties (discussed below). Since the Platform Administrator is in charge of the nodes that will be using the keys stored in the repository, the CA must trust the Platform Administrator. Since the Platform Administrator is a part of the CA's domain (again, possibly part of the same organizational unit), it trusts the CA. The relationship is shown in Figure 1 as the edge connecting the Platform Administrator to the CA.

The last entity involved is the actual key repository which holds the users' private keys. As with individual platforms (e.g., Matisse), the repository trusts the CA. This relationship makes it possible for entities with CA-signed certificates to establish SSL connections to the repository. Since the repository trusts the CA, it believes the identity of an entity with a CA-signed certificate. This relationship is represented by the edge between the repository and the CA in Figure 1

There could be more entities involved in the system. For example, there will most certainly be multiple users (e.g., Alices) and platforms (e.g., Matisses). Further there could be any number of CAs in virtually any valid architecture (hierarchy, mesh, etc.). There could also be multiple repositories with different Repository Administrators, as well as multiple Platform Administrators. The only constraint that must be enforced is that the multiple parties form a valid chain of certificates. The set of entities in Figure 1 is the smallest set which is necessary and sufficient to describe the system.

Identity Certificates Setup The way SHEMP (and PKI in general) represents trust is via certificates. From the initial

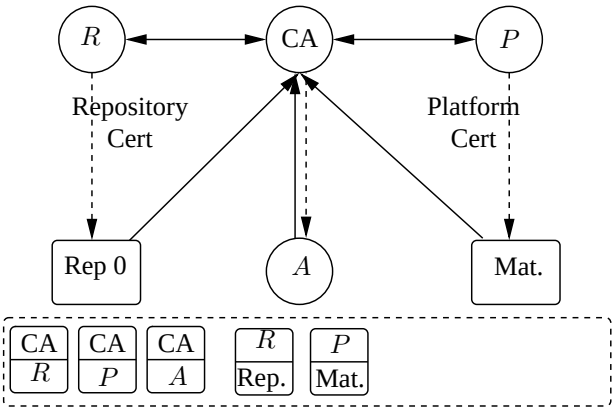

Figure 3: The administrators issue identity certificates to the repository and Matisse. The dashed edges indicate the issuing of a certificate, and the resulting certificates are added to the certificate store.

trust relationships between the entities in Figure 1, a number of certificates can be immediately issued. Figure 2 illustrates these initial certificates; they are contained in the dashed box which could possibly represent an LDAP directory where users go to locate certificates.

The certificates are issued from the CA to entities which have a mutual trust relationship with the CA. Since the administrators and Alice all have such a relationship with the CA, they are all issued identity certificates. The certificates not shown in Figure 2 are the CA certificates which are installed at the key repository and at the platform. As previously discussed, these certificates are necessary to allow things like client-side SSL connections, and are represented by the one-directional edges in Figure 2.

The first phase of setup begins when machines are added to the domain. As a repository is added, the Repository Administrator must take a number of steps to set it up. First, he must generate a keypair for the repository. This keypair can be generated in a number of ways depending on what type of platform the repository runs on. For instance, if the repository runs in a IBM 4758, then the keypair ought to be generated there, so as not to be compromised. If the repository runs on a Bear platform, then the keypair should be generated inside of the TPM.

Second, the Repository Administrator binds the public portion of that keypair to an identifier for the repository. A repository could be identified by a name, a hardware MAC address, the hash of the newly-generated public key, etc. The only restriction that SHEMP imposes is that this identifier uniquely identify the repository. The binding of the public key to the identifier is accomplished via the Repository Identity Certificate issued by the Repository Administrator (depicted as the certificate issued from the Repository Administrator to Repository 0 in Figure 3). A similar procedure is performed by the Platform Administrator each time a new machine is added to the domain.

First, the Platform Administrator generates a new keypair on the platform, using the most secure method available to it (e.g., an IBM 4758 or a TPM). Second, the Platform Administrator binds the public portion of the keypair to a unique identifier for the platform. This binding is represented as the Platform Identity Certificate (depicted as the certificate issued from the Platform Administrator to Matisse in Figure 3). As with the repository, SHEMP is agnostic to the specific mechanism used to identify the platform, but administrators should use the "least spoofable" identifier possible. For example, if a TPM is present, the TPM's Endorsement Key could be used, providing a more secure identifier than a hardware MAC address (which is easily spoofed).

Attribute Certificates Setup The final phase of setting up the system involves issuing attribute certificates to the appropriate entities. These attribute certificates are used to bind the security level of the machines (i.e., the repository and client platform) to the machine's identifier, and to bind a user's delegation policy to the user's identity.

As the Repository Administrator configures the repository, he must also assign some domain-specific security level to the repository. Concretely, the security level is expressed by the Repository Administrator generating and signing some XML attributes for the repository. The idea is for the administrator to make some signed XML statements such as "This repository runs on a Bear platform", "This repository is in a secure location and guarded by armed guards.", etc. These attributes can be arbitrarily complex, and are stuffed into a signed XML statement called the *Repository Attribute Certificate* (RAC). The RAC is identified by the same identifier that the Repository Administrator used in the Repository Identity Certificate, and thus binds the repository to its XML attributes. The RAC is then signed by the Repository Administrator and placed in a well-known location, such as an LDAP directory. This procedure is shown in Figure 4.

As the Platform Administrator configures new machines, she constructs some XML attributes for the platform and signs them. These attributes are expressed in XML, and can state any domain-specific properties that the Platform Administrator feels are important in determining the security level of the machine. Examples may include statements such as "This machine is inside the firewall", "This machine is a Bear platform", "This machine was patched on April 21, 2004", etc. Like the RAC, these attributes can be arbitrarily complex and are stuffed into a signed XML statement called the *Platform Attribute Certificate* (PAC). The PAC is identified by the same unique identifier that the Platform Administrator used to identify the platform

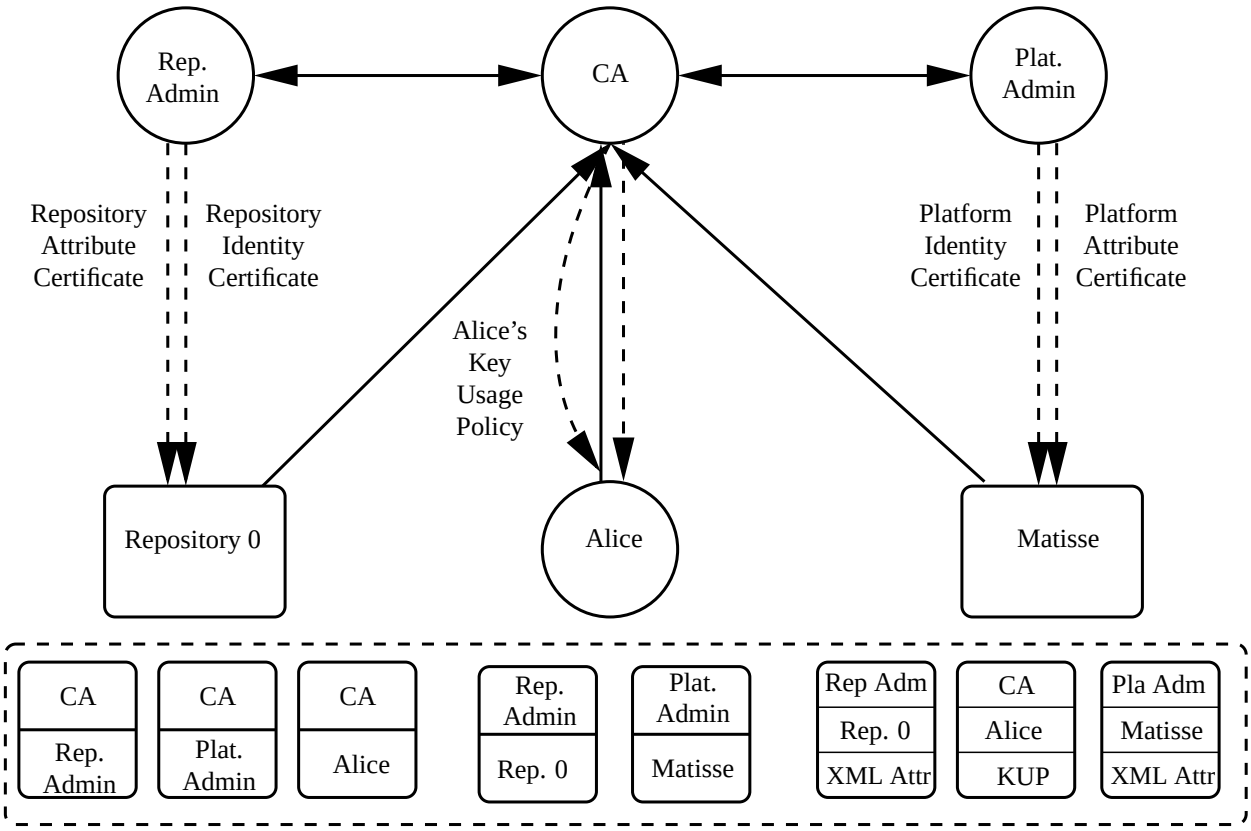

Figure 4: The administrators issue attribute certificates to the repository and Matisse which contains their security level expressed in XML. The CA issues an attribute certificate to Alice which contains her Key Usage Policy (KUP). The KUP is an XACML policy which specifies Alice's policy.

in the Platform Identity Certificate. Again, machines with no secure hardware may be identified by a hardware MAC address, whereas a Bear platform may be identified by the TPM's endorsement key. In any case, the PAC binds the client platform's identity to its XML attributes. The PAC is signed by the Platform Administrator and is placed in a well-known location such as an LDAP directory. This procedure is shown in Figure 4.

The last part of the setup occurs when a user Alice visits the CA for the first time in order to get her identity certificate issued. Alice goes through the standard identity vetting process, eventually proving her identity to the CA. At the CA, Alice also gets a chance to express her *Key Usage Policy* (KUP), which governs how her key is to be used. For example, Alice may specify "If my key lives in an IBM 4758 repository, and I request a Proxy Certificate from a Bear platform, grant the Proxy Certificate full privileges. If my key lives in a Bear repository, and I request a Proxy Certificate from any machine outside the firewall, allow my key to be used for encryption only. etc." This KUP is expressed as an XACML policy, and is signed by the CA. The signed KUP is identified by Alice's name and is placed into the LDAP along with her identity certificate.

The System in Motion Once setup is completed, Alice is free to wander throughout the domain and use her key. For example, assume that she needs to register for classes via an SSL client-side authenticated Web site. Alice begins by finding a computer which is acting as a client (i.e., has our SHEMP client software installed, and has an Platform Identity Certificate and PAC in the directory). For illustration, assume Alice walks up to the client named Matisse. Matisse first connects to the repository and establishes a client-side SSL connection. The Repository and Platform Identity Certificates are used to negotiate this connection. Recall that the Repository and Platform Identity Certificates are signed by the appropriate administrators (Repository and Platform, respectively), and that the administrators have CA-signed certificates (or a valid chain of certificates back to the CA). The implication is that there is a valid certificate chain from each of the platforms back to the CA. Since both the repository and platform trust the CA, they have good reason to believe the client-side SSL authentication.

The second step is for Alice to authenticate herself to the repository. SHEMP is agnostic with respect to how authentication is accomplished. For prototyping purposes, Alice uses a username/password, but for stronger security, Alice could use an authentication technique which cannot be intercepted by rogue processes on the client. For instance, Alice could use some other keypair (possibly stored on a token) for authentication purposes or she could use biometrics, etc.

Once both Matisse and Alice have authenticated, the repository software uses Matisse's identifier to look up Matisse's PAC. The repository may also fetch Alice's identity certificate and KUP if it is not locally stored on the repository (possibly to save space on the repository). Once the repository has gathered all of the policy information about the Matisse and Alice (e.g., the PAC, KUP, and Alice's identity certificate), it will acknowledge Alice's and Matisse's authentication, and waits for a Proxy Certificate request from Matisse.

Matisse will then generate a temporary keypair for Alice to use. Again, this may be generated a number of ways depending on the resources available to the client. For example, if the client is a Bear platform, it could generate a keypair in the TPM so that the key will never leave the TPM. If the client is a standard unarmed desktop, it may generate a keypair with OpenSSL. In any event, Matisse generates an unsigned Proxy Certificate containing the public portion of the temporary key, and sends it to the repository to be signed by Alice's private key.

The repository must then decide if it should sign the request with Alice's private key. The repository takes the security levels of itself and Matisse (contained in the RAC and PAC, respectively) and generates an XACML request containing the attributes. This XACML request and Alice's KUP are then evaluated to determine whether the operation is allowed. Concretely, an XACML PDP running on the repository as part of the repository software will make this decision. If the operation is allowed, the repository will place the attributes found in the PAC and RAC, along with Alice's KUP into the Proxy Certificate's PCI extension, and then sign the Proxy Certificate with Alice's private key. Placing the attributes and KUP into the PCI allows the Proxy Certificate's relying party to see the security properties of Alice's environment. The signed Proxy Certificate is then returned to Alice.

Alice now presents her Proxy Certificate which, along with her identity certificate, form a chain: one which includes her real public key which is signed by the CA, and an X.509 Proxy Certificate which contains a short-lived temporary public key, signed by her real private key.

Applications Traditional PKI uses of private keys include decryption, signing, and authentication. The PCs generated by SHEMP can by used for any of these operations, although the short lifespan of the Proxy Certificate adds some complexity. For example, if Bob encrypts something for Alice using her PC's public key, and the PC expires before Alice decrypts the message, then she loses the message. If Alice signs something with her temporary private key, and Bob attempts to verify the message after Alice's PC has expired, the signature is meaningless. Having Bob deal with Alice's long-term certificate would be ideal, but then Alice needs a way to ask the repository to perform private key operations on her behalf.

We designed and implemented decryption and signing proxies on the repository to solve this problem. They allow Alice to turn a message encrypted with her long-term public key into a message encrypted with her temporary public key, and turn a signature generated with her temporary private key into a signature generated with her long-term one. This way, Bob just has to deal with Alice's long-term certificate, and is not required to know anything about SHEMP. These applications do not explore novel cryptographic techniques such as proxy re-encryption schemes [1], but are still novel contributions in themselves, as they explore the use of PCs for standard private key operations; until now, they have only been used for authentication and dynamic delegation.

6 Evaluation

In Section 3, we argued that a solution to the desktop PKI problem must be secure, usable, and give users mobility. SHEMP meets these criteria, and thus makes desktops usable PKI clients.

6.1 Security

Before we offer a security analysis of SHEMP, we state an important assumption which holds throughout our analysis: the level of security in SHEMP (or any system) cannot be measured with a single bit. It is *not* the goal of our analysis to conclude some meaningless statement such as "SHEMP is secure." Rather, our analysis aims to illustrate how SHEMP can be used to increase security in a wide range of environments with possibly different threat models. We show how SHEMP creates a framework which makes it possible to build a secure PKI environment (i.e., one which minimizes the risk and impact of key disclosure) under an array of threat models.

6.1.1 Minimizing Risk

SHEMP decreases the risk of private key disclosure in a number of ways. First, SHEMP removes users' keys from the desktop and places them in a credential repository which is administered by a professional. Placing keys in a repository shrinks the TCB. The TCB is expanded to cover the desktop only when needed, and only for a short period of time. Second, by using secure hardware when available, SHEMP can reduce the TCB size even further. Finally, by including environmental information (i.e., repository and platform attributes) in each user's PC, relying parties can decide for themselves whether they should trust the request.

Getting Keys Off of the Desktop Since the TCB is a finite set of software and (possibly hardware) components, we can represent TCBs with set notation as the set TCB. We consider the TCB of the current client-side approach to be the union of the TCBs of all of the n client desktops in the domain. We denote the this total TCB as T, where:

$$T = \bigcup_{i=1}^{n} TCB_i \ .$$

If *any one* of the n desktops in Alice's domain have the keyjacking malware installed, then anyone who uses that desktop will have their key stolen or misused. Solutions which encourage mobility (i.e., allowing users to store their private keys on USB dongles) actually make matters worse, as a compromised machine is likely to service a number of users. In this case, all of the users of the compromised machine will have their key stolen or misused. If we assume that c of the desktops are infected with keyjacking malware, then Alice has a c/n chance of having her key stolen or misused. If the desktops are all roughly the same in terms of OS and software, and an attacker can compromise one of them, then it is likely that c can approach n very quickly (e.g., if the keyjacking malware were propagated by a worm or virus), leaving it almost certain that Alice will be keyjacked.

Under the SHEMP approach, there is only one machine which houses users' private keys: the key repository.[1] Centralization shrinks the TCB from the n desktops to just one key repository when no one is using the system. When Alice needs to use her key, she requests that the repository extend the TCB to cover her machine during the duration of her session. Concretely, this is accomplished by the repository signing a short-lived PC for a temporary key on Alice's current desktop. The TCB at some time t is the repository's TCB plus the TCB of whatever clients are involved in active sessions (i.e., have valid PCs) at time t. If we let TCB_{rep} be the repository's TCB and p be the number of valid PCs at time t, we can denote SHEMP's total TCB at time t as $T(t)$, where:

$$T(t) = TCB_{rep} \cup \bigcup_{i=1}^{p} TCB_i \ .$$

Assume that organization S uses SHEMP, and that organization O does not. Additionally, assume that they have the same number of users and desktops (denoted n), and that one desktop is serving as S's key repository (leaving S with $n-1$ clients). The TCB at S is never greater than O's TCB, i.e., $\forall t : |T(t)| \leq |T|$ because:

$$\left| TCB_{rep} \cup \bigcup_{i=1}^{p} TCB_i \right| \leq \left| \bigcup_{i=1}^{n} TCB_i \right| \ .$$

To see why this statement is true, assume that every user in S has a valid PC at some time t. In this case $p = n-1$, which yields the same size TCB as O. The implication is that if any client desktop does not have a valid PC, then the SHEMP approach shrinks the TCB. Furthermore, Alice's policy statement may disqualify some clients from using her key, thus shrinking the TCB even further.

SHEMP also minimizes the risk of private key disclosure by placing all of the private keys under the control of a trusted entity: the Repository Administrator. The Repository Administrator will likely be closely related to the organizational unit which issues certificates (i.e., the Certificate Authority). A specialist is more likely to protect the private keys than an individual user is. Thus, letting a specialist care for the private keys decreases the risk of private key disclosure.

Using Secure Hardware As we discussed in Section 4, secure hardware can shrink the TCB. Highly secure devices such as the IBM 4758 can provide a separate security domain from their host, while secure platforms such as Bear can provide some level of protection, and cost significantly less than an IBM 4758. SHEMP reduces the TCB (and hence, the risk of private key disclosure) further by taking advantage of secure hardware, when and where available.

Since the keys reside in a central location (i.e., the repository), we envision that the repository will utilize some form of secure hardware. The repository application

[1] Actually, the SHEMP design allows for a number of repositories, but we envision a small number of repositories in relation to clients.

could be running in secure hardware, and the private keys could be stored inside. The organization's threat model should dictate the level of secure hardware that they adopt. For maximum security, the repository should run in a device such as the IBM 4758, and the clients should minimally use something like Bear. The use of secure hardware allows organizations with SHEMP to shrink the TCB even further, thus further decreasing the risk of private key disclosure.

Describing the TCB Finally, SHEMP minimizes the risk of key disclosure through the use of the environmental attributes (found in the Repository and Platform Attribute Certificate) and KUPs found in each Proxy Certificate's PCI extension. SHEMP mandates that all of this information be included. This approach gives useful information to relying parties, allowing them to adjust their trust in the client based on the environment. Relying parties are thus aware when a client generates a temporary key under conditions which are likely to result in key disclosure, and have the possibility to limit the key's use. The usability of the policy statements in the context of building applications will be examined below.

6.1.2 Minimizing Impacts

In addition to minimizing the risk of a private key disclosure, SHEMP minimizes the impact of such a disclosure. First, a successful keyjacking-style attack only gives the attacker access to a temporary keypair, and only for a limited period of time. Second, SHEMP reduces the impact of a disclosure to the organization by simplifying and shrinking the size of Certificate Revocation Lists (CRLs). Finally, SHEMP makes forensics easier by consolidating (and possibly protecting) the audit trail.

Closing the Window SHEMP minimizes the impact of private key disclosure at the client by only allowing the temporary key to be used for a short time. Under SHEMP, the key issued on the client's desktop is valid for a number of hours (our prototype defaults to two hours). This small time window limits the opportunity for a successful attacker to use the victim's key. The set of operations that an attacker can perform with a stolen key is possibly further limited by the victim's KUP. A successful attacker may not have access to the encryption or signing proxies (or other resources in the domain) depending on how the victim has set her KUP. Therefore, a restrictive KUP can also limit the impacts of a private key disclosure.

Revocation SHEMP minimizes the impact to the organization in the case of a key compromise. In many status quo PKIs, compromised keys are revoked by placing their certificate into a CRL or an *Online Certificate Status Protocol* (OCSP) server. Keeping CRLs up to date and distributing them are non-trivial problems in PKI space.

Since only the SHEMP repository can use Alice's private key, SHEMP (like SEM [2]) can effectively revoke a user's keypair by changing the authentication information at the repository. Changing Alice's authentication information results in Alice (or anyone with Alice's login information) being unable to log on to the repository and make requests to use her private key. This approach reduces the size of CRLs and the amount of work for the administrative staff.

The Audit Trail SHEMP minimizes the impact of a private key compromise by consolidating the audit trail used for gathering forensic information. Since all accesses to use Alice's private key are received by the SHEMP repository, there exists a central log of Alice's private key activity on the repository. In the event that Alice's key is compromised, investigators need only look in one place for information. Furthermore, since the SHEMP repository software can run inside of secure hardware, SHEMP can secure the logs themselves by keeping them inside of the hardware. The logs could be cryptographically protected to prevent tamper or viewing by unauthorized individuals. In the event of a key compromise, the organization would not only have a central location for the logs, but can protect them against modification.

6.2 Usability

In order to show that SHEMP is usable, we need to show that developers and administrators can understand and construct valid policies to solve real security problems—i.e., the policy mechanism must be a valid medium for developers and users to express their mental models. Furthermore, we need to show that the computational overhead introduced by SHEMP's policy mechanism and use of extra keypairs does not make the system unusable from an end user's perspective.

User Study To see whether the policy mechanisms were usable, we conducted a user study consisting of eight subjects which are highly representative of the types of people who would be tasked with constructing SHEMP policies. Our user study outlined some real application designs taken from Dartmouth's Grid community. We gave the application designs to subjects who would likely fill the roles of the Repository Administrator, the Platform Administrator, and the CA. We were interested in evaluating whether the parties could generate a meaningful set of policies which represent a given mental model, how many tries it took them, and their feedback regarding the difficulty of their task.

Once users had completed the test, they were asked to complete and return a survey. Compiling the survey data led to a few interesting discoveries. First, there was an inverse correlation between the number of machines under the subject's control and the number of mistakes the subject made. The subjects who administered the most machines made the fewest mistakes. Second, of the subjects who had configured other application security policies (such as Apache or MySQL), all but one of them said SHEMP was easier to configure. The one who said it was harder recommended using a GUI, and giving a users a way to go back. Many thought that the structure of the tool was helpful; they liked the question and answer tone rather than having a random access configuration file to edit. Third, no one reported anything particularly confusing about SHEMP, and everyone mentioned in one way or another that they would like a GUI with the potential to go back to the previous set of options. Finally, the subjects leaned to use the tool rather quickly: no one reported running any of the scripts more than three times.

The overall results were positive. Half of the subjects built perfect policies, and of the remaining half, no one missed more than one operation out of eight. Every mistake resulted from a typographical error, such as a misspelled word or failure to respect case sensitivity. These results suggest that a policy generation tool which does not allow users to make such mistakes (possibly by doing input validation or presenting users with a graphical menu of options to choose from) would yield better results. The results indicate that the SHEMP policy mechanisms are usable, but a good policy construction tool is essential.

Performance In order to show that the overhead introduced by SHEMP does not make the system unusable to end users, we conducted a performance analysis. Performance is not the most interesting aspect of SHEMP, but since a third party is contacted for all private key operations, we expected a slowdown and wanted some quantification. If SHEMP keeps users waiting for long periods of time to perform key operations, then users are likely to find faster solutions, even at the expense of security.

We used our prototype to measure the overhead of PC generation and the decryption and signing proxies. As a baseline, we compared SHEMP to a simple Java application which we call the SHEMP `CryptoAccessory`. The `CryptoAccessory` performs the standard cryptographic operations (encryption, decryption, signing, and verification) using a locally-stored keypair, and without third-party involvement.

We measured the slowdown for three operations (Generate Proxy Certificate, Decrypt, and Sign) on three network configurations. The operations consisted of generating an RSA keypair, using it to decrypt a message, and then using it to sign a message. For the first configuration, we put the SHEMP client and the repository on the same machine, thus eliminating network delay altogether. In the second configuration, we placed the client and repository on the same Ethernet segment, so as to simulate a Local Area Network. For the final configuration, we put the client and repository on different networks by putting the client on our campus-wide wireless network.

Op	Local	Same Seg	Diff. Net	Avg.
Gen PC	3.69%	1.94%	4.33%	3.32%
Decrypt	54.95%	42.64%	54.42%	50.67%
Sign	40.22%	49.04%	57.51%	48.92%

Table 2: Slowdown of SHEMP compared to local private key operations.

We averaged ten runs with our `CryptoAccessory` and SHEMP, and then calculated the slowdown introduced by the SHEMP overhead as a result of using the proxies on the repository to perform the operations. The SHEMP overhead also includes the time for the policy check on the repository. The repository (and decryption and signing proxies) locate all of the policies and attribute certificates, verify their signatures, and pass the information to the PDP for evaluation. Performance results are given in Table 2.

The column labelled "Avg." is an average over the results of the different configurations. The results indicate that only 3.32% of the time spent generating a PC is used by SHEMP. This extra time that SHEMP introduces is used to transport the unsigned PC over the network, verify the current environment's attribute certificates, perform a policy check against Alice's KUP, sign the PC, and return the signed PC to Alice. The other 96.68% of the time is spent generating the temporary keypair on the client.

The performance results for the proxies are less impressive, indicating that roughly half of the time spent performing the operation is introduced by SHEMP. In these cases, SHEMP uses the time to transport the messages over the network, perform a policy check, perform a public key operation (either to verify the message or to encrypt the message with Alice's temporary public key), and perform a private key operation (either to sign a message with Alice's long-term private key or decrypt a message which was encrypted with her long-term public key). From s user's perspective, using SHEMP doubles the time it takes to perform a private key operation.

However, this is not as bad as it appears. First, the extra time needed for SHEMP may not be noticeable to humans. Over the average of the ten decryption operations performed in the "Different Network" configuration,

SHEMP takes the time of the operation from 222.3 milliseconds to 487.8 milliseconds. Human perception cannot detect the slowdown. If the network is lagging, then the time of the operation is likely to grow even more, but then any application using the network will feel a loss of performance as well. Second, it is possible to reduce the overhead by using cryptographic acceleration hardware. Our prototype repository used the default Java cryptographic provider to perform the operations. If we run the repository in an IBM 4758, we could exploit the cryptographic acceleration subsystem to improve performance.

Our performance analysis indicates that the overhead introduced by SHEMP does not make the system unusable. While the performance hit is statistically significant, it can be improved via specialized hardware, and users are unlikely to notice the slowdown anyway.

6.3 Mobility

SHEMP gives users mobility without sacrificing security. In our prototype testbed, we have three client desktops which are assigned a different set of security attributes. The desktops represent low-, medium-, and high-security machines (high-security machines being ones armed with a TPM). We are able to access our private key from each one, subject to the restrictions in our KUP. The mobility of SHEMP stems from the fact that it is based on the MyProxy design. The use of the credential repository approach allows SHEMP users to access their key from anywhere, provided that they can access the key repository. MyProxy's mobility is what led us to use it as a foundation for the SHEMP design in the first place. In all fairness, we can claim that SHEMP is as mobile as MyProxy.

SHEMP excels in the security properties which are maintained during migration. Again, the current client-side infrastructure makes migration risky by using unsafe transport formats. The use of a secure transport format such as Sacred [23, 24, 25] could provide some benefits, but it is not necessarily a part of what we consider the current client-side infrastructure. MyProxy is an improvement in that private keys typically stay on the repository, and only PC are given to the user. However, MyProxy does not consider Alice's environment when deciding whether or not to allow Alice to use her private key. As long as Alice (or anyone else) can authenticate to the repository, it will grant her full access to her key.

SHEMP takes the MyProxy approach a step further by actually checking the security properties of the current environment, and then consulting Alice's KUP to see if it should grant the request. Concretely, the SHEMP repository uses the platform authentication step to identify the requesting platform. As discussed in Section 5, the repository gives the attributes contained in the Platform and Repository Attribute Certificates along with Alice's KUP to a PDP for evaluation. If the PDP returns "Permit", then the request is granted. SHEMP's use of environmental information in making its access decision gives users the same amount of mobility as the MyProxy approach, while simultaneously providing extra security.

7 Summary

This research began when we discovered numerous problems with the current client-side approach to deploying PKI. By exploiting the large TCB of modern desktops, an attacker can either steal private keys outright or use them at will, leaving desktops unsuitable for PKI. Starting with the problems of the current approach, we derived a set of criteria for making desktops usable PKI clients. As we established in Section 3, any solution which claims to make desktops usable for PKI must address security, mobility, and usability.

Starting with the approach employed by the Grid community (i.e., the MyProxy online credential repository), we designed a system which makes desktops usable PKI clients: SHEMP. SHEMP meets the criteria we established in Section 3. In Section 6, we offered a security analysis of SHEMP, illustrated how it minimizes the risks and impacts of a private key disclosure, and how it can defend against the keyjacking attacks of Section 2. We discussed how SHEMP maintains security while providing mobility through the use of environmental attributes and Key Usage Policies. Finally, we showed that SHEMP is usable by presenting the results of our usability study and performance analysis. The results indicate the SHEMP's policy framework can be used to accurately capture a mental model of the system given the right tools, and that SHEMP's overhead is imperceivable by humans.

Acknowledgements

This research has been supported in part by the Mellon Foundation, NSF (CCR-0209144), AT&T/Internet2 and the Office for Domestic Preparedness, DHS (2000-DT-CX-K001). This paper does not necessarily reflect the views of the sponsors.

References

[1] G. Ateniese, K. Fu, M. Green, and S. Hohenberger. Improved Proxy Re-Encryption Schemes with Applications to Secure Distributed Storage. In *Network*

and Distributed System Security Symposium. The Internet Society, 2005.

[2] D. Boneh, X. Ding, G. Tsudik, and C. Wong. A method for fast revocation of public key certificates and security capabilities. In *10th USENIX Security Symposium*, pages 297–308. USENIX, 2001.

[3] J. Dyer, M. Lindemann, R. Perez, R. Sailer, S.W. Smith, L.van Doorn, and S. Weingart. Building the IBM 4758 Secure Coprocessor. *IEEE Computer*, 34:57–66, October 2001.

[4] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[5] P. Gutmann. How to recover private keys for Microsoft Internet Explorer, Internet Information Server, Outlook Express, and many others - or - Where do your encryption keys want to go today? `www.cs.auckland.ac.nz/~pgut001/pubs/breakms.txt`.

[6] IBM Research Demonstrates Linux Running on Secure Cryptographic Coprocessor, August 2001. Press release.

[7] A. Iliev and S.W. Smith. Privacy-Enhanced Credential Services. In *2nd Annual PKI Research Workshop*. NIST, April 2003.

[8] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the 9th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–177, November 2000.

[9] M. Lorch, J. Basney, and D. Kafura. A Hardware-secured Credential Repository for Grid PKIs. In *4th IEEE/ACM International Symposium on Cluster Computing and the Grid*, April 2004.

[10] R. MacDonald, S.W. Smith, J. Marchesini, and O. Wild. Bear: An Open-Source Virtual Coprocessor based on TCPA. Technical Report TR2003-471, Department of Computer Science, Dartmouth College, 2003.

[11] J. Marchesini. Secure Hardware Enhanced MyProxy. Technical Report TR2004-525, Dartmouth College, November 2004.

[12] J. Marchesini and S.W. Smith. Virtual Hierarchies - An Architecture for Building and Maintaining Efficient and Resilient Trust Chains. In *NORDSEC2002 - 7th Nordic Workshop on Secure IT Systems*, November 2002.

[13] J. Marchesini, S.W. Smith, O. Wild, and R. MacDonald. Experimenting with TCPA/TCG Hardware, Or: How I Learned to Stop Worrying and Love The Bear. Technical Report TR2003-476, Department of Computer Science, Dartmouth College, 2003.

[14] J. Marchesini, S.W. Smith, O. Wild, J. Stabiner, and A. Barsamian. Open-Source Applications of TCPA Hardware. In *20th Annual Computer Security Applications Conference (ACSAC)*, December 2004.

[15] J. Marchesini, S.W. Smith, and M. Zhao. Keyjacking: Risks of the Current Client-side Infrastructure. In *2nd Annual PKI Resarch Workshop*. NIST, April 2003.

[16] J. Marchesini, S.W. Smith, and M. Zhao. Keyjacking: The Surprising Insecurity of Client-Side SSL. *Computers and Security*, 2004. In Press.

[17] P. McGregor and R. Lee. Virtual Secure Co-Processing on General-purpose Processors. Technical Report CE-L2002-003, Princeton University, November 2002.

[18] J. Novotny, S. Tueke, and V. Welch. An Online Credential Repository for the Grid: MyProxy. In *Proceedings of the Tenth International Symposium on High Performance Distributed Computing (HPDC-10)*. IEEE Press, August 2001.

[19] Department of Defense. Department of Defense Trusted Computer System Evaluation Criteria. DoD 5200.28-STD, December 1985.

[20] S. Pearson, editor. *Trusted Computing Platforms: TCPA Technology in Context*. Prentice Hall, 2003.

[21] M. Periera. Trusted S/MIME Gateways, May 2003. Senior Honors Thesis. Also available as Technical Report TR2003-461.

[22] S. Proctor. Sun's XACML Implementation. `sunxacml.sourceforge.net/`.

[23] Securely Available Credentials-Framework. `www.imc.org/ietf-sacred/index.html`. draft-ietf-sacred-framework.

[24] Securely Available Credentials-Protocol. `www.imc.org/ietf-sacred/index.html`. draft-ietf-sacred-protocol-bss.

[25] Securely Available Credentials-Requirements. `www.imc.org/ietf-sacred/index.html`. RFC3157.

[26] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architectrue. Technical report, IBM Research Division, RC23064, January 16, 2004. to Appear at the 13th Annual USENIX Security Symposium.

[27] S.W. Smith. Secure Coprocessing Applications and Research Issues. Technical Report Los Alamos Unclassified Release LA-UR-96-2805, Los Alamos National Laboratory, August 1996.

[28] S.W. Smith. WebALPS: A Survey of E-Commerce Privacy and Security Applications. *ACM SIGecom Exchanges*, 2.3, September 2001.

[29] S.W. Smith. Outbound Authentication for Programmable Secure Coprocessors. *International Journal on Information Security*, 2004.

[30] S.W. Smith and S. Weingart. Building a High-Performance, Programmable Secure Coprocessor. *Computer Networks*, 31:831–860, April 1999.

[31] N. Stam. Inside Intel's Secretive 'LaGrande' Project. `www.extremetech.com/`, September 19, 2003.

[32] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant processing. In *In Proceedings of the 17 Int'l Conference on Supercomputing*, pages 160–171, 2003.

[33] Trusted Computing Platform Alliance. TCPA Design Philosophies and Concepts, Version 1.0. www.trustedcomputinggroup.org, January 2001.

[34] Trusted Computing Platform Alliance. TCPA PC Specific Implementation Specification, Version 1.00. www.trustedcomputinggroup.org, September 2001.

[35] Trusted Computing Platform Alliance. Main Specification, Version 1.1b. www.trustedcomputinggroup.org, February 2002.

[36] XACML 1.1 Specification Set. www.oasis-open.org, July 24, 2003.