Securing the Core with an Enterprise Key Management Infrastructure (EKMI)

Arshad Noor
StrongAuth, Inc.
550 Lakeside Drive, Suite 10
Sunnyvale CA 94085
arshad.noor@strongauth.com

ABSTRACT

The last twenty-five years has witnessed an emphasis on protecting the network and computing host as a proxy for protecting data from unauthorized access. While this was a reasonable strategy at the dawn of network-based computing, given the state of the internet today with its security issues, this strategy is proving to be hopeless.

This paper advances the notion that the time has finally come to begin what we should have done initially – protect the core of our computing infrastructure: the data – in addition to protecting the network and computing host.

The paper describes an architecture - and a specific implementation of that architecture - to enable the encryption of data across the enterprise in a platform and application-independent manner. The architecture describes the use of a Public Key Infrastructure (PKI) and a Symmetric Key Management System (SKMS) within an Enterprise Key Management Infrastructure (EKMI), to securely - and centrally - manage the life-cycle of the symmetric encryption keys used for data encryption.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and protection – *cryptographic controls.*
E.3 [**Data Encryption**]: *Public key cryptosystems, Standards.*

General Terms

Management, Design, Security, Standardization.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

IDtrust '08, March 4-6, 2008 Gaithersburg, MD
Copyright 2008 ACM 1978-1-60558-066-1…$5.00

Keywords

Enterprise Key Management Infrastructure (EKMI)
Key-management (KM)
Public Key Infrastructure (PKI)
Symmetric Key Client Library (SKCL)
Symmetric Key Management System (SKMS)
Symmetric Key Services (SKS)
Symmetric Key Services Markup Language (SKSML)
XML Encryption (XENC)
XML Signature (DSIG)

1. INTRODUCTION

Most security professionals are familiar with symmetric-key based cryptography when presented with terms such as **Data Encryption Standard (DES)**, **Triple DES (3DES)** and the **Advanced Encryption Standard (AES)**. Some are also familiar with **Public Key Infrastructure (PKI)** as an enterprise-level solution for managing the life-cycle of digital certificates used with asymmetric-key cryptography.

However, the term **Symmetric Key Management System (SKMS)** - the discipline of securely generating, escrowing, managing, providing access to and destroying symmetric encryption keys - almost always draws blank stares. Given the number of applications that needed to encrypt data in the past, this is not surprising; symmetric encryption key management has traditionally been buried within the business applications performing encryption. These applications primarily focused on business functions, but managed encryption keys as an ancillary function. Consequently, there was no reason to emphasize key-management. This paper advances the notion that the time has come to address SKMS as an application-independent, enterprise-level defense mechanism. Additionally, this article advocates the use of a PKI for securing an SKMS within an **Enterprise Key Management Infrastructure (EKMI)**.

While encryption has been in use for centuries[1], computer-based cryptography entered the general-computing field with the advent of the DES algorithm. The primary business uses for this technology was within the military and later banking. Given the nature of what encryption tech-

nology was protecting, implementers were willing to live with custom key-management solutions, however contrived they may have been. With the explosion of the world-wide web, businesses have been racing to implement business processes on the Internet, bringing sensitive information significantly closer to attacks. Although businesses have invested billions in firewalls, intrusion detectors, intrusion prevention systems and other network and host-based defense mechanisms, the US has witnessed more than 400 breach disclosures[2] since the passage of California's Breach Disclosure law[3]. Of some note are the disclosures by the University of California over the years, with the most recent one in Los Angeles (UCLA)[4]. Given that this is the *seventh* breach disclosure by the University of California across all their schools, it's reflective of a situation spiraling out of control.

All data breaches pale in comparison to the one at TJX[5], which is currently ranked as the largest breach ever with nearly 95M credit card numbers exposed at this Massachusetts-based retailer[6]. A quarterly financial statement[7] filed by this company in August 2007 shows that, so far, it has taken a charge of US $216M against this single breach. At least three lawsuits are pending against TJX with the full extent of damage yet unknown.

Breaches at retailers such as TJX, Ralph Lauren, BJ's, DSW and credit-card processing companies such as CardSystems have prompted credit-card giants Visa, Mastercard, American Express and Discover to standardize on security requirements for merchants and card-acquirers through the **Payment Card Industry's Data Security Standard (PCI-DSS)**[8]. One critical element required within PCI-DSS is the encryption of credit-card numbers and a robust key-management system to accompany it. This effort is aimed at strengthening controls protecting sensitive data on systems that have potential for causing financial damage to customers and the credit-card brands.

While the Retail sector has been particularly battered by data-breaches, the need to encrypt sensitive data in companies is also driven by the following regulation across the world:

- The **Health Insurance Portability and Accountability Act of 1996 (HIPAA)**[9], which specifies rules for how medical information must be secured within the US when used within computerized systems;
- The **Financial Modernization Act of 1999,** aka the **"Gramm-Leach Bliley Act" (GLBA)**[10], which specifies rules for how financial information must be secured within the US when used within computerized systems;
- The **Personal Information Protection and Electronic Documents Act (PIPEDA)**[11], which specifies rules for how personal information must be secured in Canada, when used within computerized systems;
- **Directive 95/46/EC of the European Parliament**, aka the European Union Directive or EU Directive[12], which specifies rules for how personal information must be secured in the EU, when used within computerized systems;
- Thirty-eight US "computer breach disclosure" laws requiring companies that have breached sensitive information of US residents, to disclose those breaches to affected individuals. Most of these laws provide a "safe-harbor" to the company by not requiring a disclosure, if the affected data affected was encrypted;

With the publication of each new computer breach, governments across the world are reacting with increasing legislation that regulates the protection of sensitive data. Companies are also becoming sensitive to adverse publicity and private lawsuits when stolen data or lost laptops and computer tapes, etc. are publicized in the media. As a result, security professionals now accept that sensitive data needs to be encrypted across the enterprise – on desktops, laptops, Personal Digital Assistants (PDAs), mobile telephones, etc. - and not just on servers and on-line or off-line storage within the Data Center.

1.1 Organization of Paper

This paper begins by describing some of the problems with current key-management systems and architectures in Section 2. In Section 3 an architecture for an SKMS is presented with an explanation for why this particular architecture makes sense and the rationale for the design decisions that were made. Section 4 describes the SKSML protocol and how applications are expected to use it. In Section 5, the paper goes on to describe a specific implementation of this architecture in an open-source product and the experience gained from such an implementation. Section 6 discusses the security required for such an architecture. Section 7 provides a high-level plan for how an SKMS can be built for production use in an IT environment and what are the issues that implementers must take into consideration. The paper finally concludes in Section 8.

2. PROBLEMS WITH CURRENT SYSTEMS

Why is symmetric key-management a problem? After all, applications seem to have addressed the problem within the applications for decades, and appear to be continuing to do so. The problem becomes obvious from the perspective of a manager in IT Operations. As an illustration, if the IT Operations Manager was responsible for the following in a retail enterprise that accepted credit-cards for payment:

- A Point-of-Sale (POS) application used in hundreds of stores across the country;

- An e-commerce application that required credit-card numbers for payment;
- A payment-processing application in the back-office that communicated with the credit-card network for settling transactions;
- A back-office database that consolidated transactions for accounting; and
- A business analytics application for determining retail fraud;

the IT Operations manager would have five applications that required encryption and thus, key-management. With the proliferation of laptop and PDA losses or thefts, companies are now mandating encryption on these devices, thus adding two more key-management schemes to the infrastructure. Finally, add database and operating system-specific encryption to the mix, and you round out the picture with 8-10 key-management infrastructures.

Since applications are typically purchased from multiple vendors, each vendor, focusing primarily on their own business application, implements encryption and perform key-management functions using its own design. As a result, the IT Operations staff are forced to manage at least 8-10 distinct symmetric key-management infrastructures, each with its own technology, training, documentation, procedures and audits (PCI-DSS regulated entities are required to perform annual audits of any system that stores credit-cards).

Not only does this border on the ridiculous as it raises Total Cost of Ownership (TCO) for the company, but more importantly, one could argue that there is the potential for a compromise in such a security strategy because "with so many pots cooking on the stove simultaneously, something is bound to get burned".

Presented with the problem in this perspective, the logical solution springs to clarity: the key-management capability needs to be abstracted from applications that need the capability, and managed independently in its own infrastructure. Applications need only have access to a key-management service, thus enabling encryption and decryption, without having to be aware of implementation details. Such a solution is not unlike the architecture of the Domain Name System (DNS) for hostname-IP-address resolution, the Dynamic Host Configuration Protocol (DHCP) for dynamic IP-address allocation or a Relational Database Management System (RDBMS) for data management.

2.1 File, Database and Disk-based Encryption

There are many commercial technologies on the market today that are capable of transparently encrypting data at the file, database or storage-media layer. These technologies, which include encrypting file systems, full-drive encryption, database encryption, etc., have their own built-in key-management.

In the presence these technologies what advantage would an abstract key-management system offer implementers?

Aside from the issue that current implementations of such technologies offer incompatible key-management designs , the single biggest issue with such techniques is that it does not address the problem completely.

An application, typically, consists of many layered technologies in a stack, through which data must pass before it is stored on storage media. The layers vary depending on the complexity of the application, its architecture, operating system and physical implementation. File, database and storage-media encryption occur at some of the lowest layers of such a technology stack, oblivious to the applications that create the data. Implementers of such encryption schemes offer this feature as a benefit, because it does not require applications to be modified to take advantage of the encryption capabilities; all applications can immediately start using it.

However, the lower in the technology stack the encryption occurs, it leaves open the possibility for attackers to compromise a layer higher up in the stack and avail themselves to plaintext data. Even in a perfectly functioning encryption system (consisting of file, database or media-based encryption),, data could be compromised in one of many layers above the encrypting layer.

On the contrary, encrypting at the application layer allows implementers to encrypt data at the highest possible layer in the stack, leaving little "wiggle-room" for the attacker to compromise the data. While there is no guarantee that an attacker cannot compromise the application layer and still get to the data before it is encrypted (or after it is decrypted), it, nonetheless, reduces the attack surface to the smallest possible target within the stack. It allows implementers to focus their resources on protecting just one layer - the application layer - towards making the most effective use of encryption.

Encrypting at the application-layer also has the added benefit, that once data is encrypted at this highest stack-layer, implementers need have little concern for the safety of data in lower layers of the stack: the data is protected no matter how many technology layers it must pass through on the host or the network.

3. ARCHITECTURE

An Enterprise Key Management Infrastructure (EKMI) is defined as a collection of technology, policies and procedures for managing <u>all</u> cryptographic keys within an enter-

prise – both symmetric and asymmetric. Therefore, an EKMI consists of a PKI and an SKMS.

Note: In the short-to-medium term, PKI and SKMS will be managed as distinct entities within enterprises. However, the two infrastructures will merge towards a cohesive infrastructure in the not-too-distant future. This paper dispenses with detailed discussions of PKI except where it integrates with an SKMS, and focuses more on the discussion of the SKMS itself.

The PKI part of an EKMI is a standard public key infrastructure issuing and managing X.509 and PKIX-compliant digital certificates. It uses industry-standard protocols for communicating with client requests: CMS, PKCS#10, PKCS#7, etc., and the digital certificates issued out of the PKI are managed per the PKI's Certificate Policy (CP) and Certification Practices Statement (CPS). Nothing unusual here, so we won't dwell on this too much. We will point out where digital certificates from a PKI are needed to enable the capability described in this paper.

The SKMS, on the other hand, has an architecture based on the the following business, technical and operational requirements:

- Centralized policy-definition and key-management;
- Platform, application and language independent;
- Highly-available; yet KM client applications were required to continue functioning - i.e., encrypt and decrypt data - even in the absence of the KM service;
- Highly-scalable;
- Very secure;
- Leverage existing standards and security certifications of cryptographic components;

Given these requirements, the following design decisions were made for the SKMS architecture:

3.1 Client-Server

While key-management can be easily abstracted from applications even while running locally on the same machine as the application, the requirement that KM policies be defined centrally and symmetric keys be managed centrally led the design towards a client-server architecture for key-management.

The client is implemented as a library, named the **Symmetric Key Client Library (SKCL),** and is much like the name-service library in DNS or the database connectivity libraries - ODBC, JDBC, etc. - for RDBMS. Client applications use the SKCL to request and receive KM services from the server.

The **Symmetric Key Services (SKS)** server functions as a centralized service-provider on the network, listening for and responding to KM requests. When requested, it generates all keys centrally based on predefined policies, escrows them, and then sends the symmetric key to the authorized client. The client and server communicate with each other using a secure protocol: **Symmetric Key Services Markup Language (SKSML)** (discussed later).

By using the client-server architecture, not only are the centralized policy definition requirements addressed, but also the centralized key-management requirements.
An alternative architecture is to define policies centrally and push them down to the clients, and similarly have the clients generate keys locally and push them up to the server. However, the SKMS architecture avoided this design for one reason: to avoid the possibility of catastrophic data-loss.

If a client were to generate a symmetric key locally, encrypt the plaintext, delete the plaintext (to eliminate the vulnerability), but cannot persist or send the generated symmetric key to the server for any reason, the plaintext might be lost forever.

While it is possible to design around such conditions, the complexity of the SKCL increases significantly because it is difficult to predict potential catastrophic conditions on a client machine - especially mobile devices. With centralized policy-definition and key-generation, this loss is avoided altogether by escrowing the symmetric key first, and then sending it to the client for use.

Given that a client-server architecture makes it possible to have multiple servers servicing numerous clients using hundreds of application, the following schema was chosen to uniquely identify every symmetric key in the system. This is how applications will map the symmetric key they've used, to the ciphertext:

- Every enterprise is assigned a unique **Domain-ID** - a monotonically increasing sequential number - to distinguish its SKMS from others'. The design chose to reuse the Private Enterprise Numbers assigned by IANA[13] for this purpose. While using DNS-style domain-names was an option, the design opted for sequential numbers as it was similar to other identifiers in the SKMS; besides, since humans would rarely need to know these identifiers, it was more efficient to maintain them as numerals;
- Every server within an SKMS is also assigned a unique **Server-ID** - once again, a monotonically increasing sequential number - to distinguish a server from others within an SKMS domain;
- Every symmetric key generated by a server is assigned a unique **Key-ID** - a monotonically increasing sequen-

tial number - to distinguish it from every other key generated by that SKS server;

Concatenating the Domain ID, the Server ID and the Key ID, with simple hyphens to separate them, allows the SKMS to create a unique **Global Key-ID (GKID)** that can be referenced by a client, anywhere on the internet. An example of a GKID would be **10514-3-34348**, indicating that this is key ID 34348, generated on SKS server number 3 within an SKMS for the organization whose domain is represented by the unique private enterprise number, 10514, as issued by IANA.

It is necessary for applications to be modified to maintain the GKID with the ciphertext so that applications know which symmetric key to use to decrypt the ciphertext. For applications that use a structured database of any kind, this is relatively easy by modifying the database schema as a one-time enhancement activity. For standalone ciphertext files, it is recommended to use the World Wide Web Consortium's XML Encryption standard whose schema allows for identifying the unique identifier of the encryption key, the URL for locating the key and many other encryption parameters along with the ciphertext. The implementation experience described in Section 5 shows how this design was used successfully for relational databases as well as standalone ciphertext files.

3.2 XML-based protocol

To support platform, application and programming-language independence, the SKCL and the SKS server communicate using an eXtensible Markup Language (XML)-based protocol, named the **Symmetric Key Services Markup Language (SKSML)**. Details of this protocol are defined in the next section.

SKSML is designed to be a very thin layer, leveraging the Simple Object Access Protocol (SOAP) for sending and receiving messages. SOAP is well-understood, accepted and supported industry-standard for sending and receiving complex messages in client-server communications. SOAP libraries are available for every major programming language and platform, allowing almost every modern operating system to support applications using SOAP and thus, SKSML.

The question arises - why was Abstract Syntax Notation (ASN) not used for the protocol?

The PKI community has been using ASN, successfully, across all major platforms for a number of years. ASN is well-understood, accepted and supported by the PKI community and has the advantage of being compact, thus allowing for significant efficiencies in communication, as well as by small devices where space is at a premium.

Despite these advantages of ASN, the design chose to use XML for the protocol for the following reasons:

i. XML is equally well-understood, accepted and supported by the *general computing community* - not just the PKI or protocol-development community;
ii. It is extremely easy to understand and explain XML - even to non-computer professionals;
iii. There are significantly larger number of developers who know and use XML than ASN, thereby increasing the probability of adoption and making a larger pool of candidates available for its development;
iv. XML is the lingua-franca of Service Oriented Architecture (SOA) - the architecture for a new class of applications that has the support of every major software vendor in the world;
v. XML does not require special tools; its messages can be assembled and debugged using simple text editors;
vi. While ASN definitely is more compact than XML, with the exception of a small number of use-cases, the verbosity of XML is not a handicap in an environment whose bandwidth only keeps increasing every few years. With gigabit networking now showing up in new computers for LANs, 108Mb/s for WiFi-enabled devices expected to become the standard by the end of this decade, and broadband connectivity to be near the 1Mb/s range for small mobile devices, bandwidth is not a constraint for the vast majority of computing devices.

Primarily driven by the ease-of-use feature and industry trends, it was decided to use XML for the representation of the key-management protocol.

3.3 Scalability and Availability

There are well-known and proven design architectures for creating highly-scalable and highly-available software systems for servers. However, most of the work in creating such software requires addressing basic infrastructure requirements such as authentication, authorization, logging, performance management, scheduling, persistence, etc.

This architecture chose to leverage the capability built into the Java 2 Enterprise Edition (J2EE) for these services rather than develop them from scratch.

The J2EE architecture was created to address precisely such infrastructure issues, while scaling to address the requirements of extremely large and demanding infrastructures. It has evolved over the last decade and continues to improve with the input of millions of Java developers. Java Enterprise Edition 5 (JEE5), the latest incarnation of this architecture, has once again improved on this design through the community-development process.

By choosing J2EE, and the newer JEE5, the architecture addressed every infrastructure requirement for a scalable and highly-available service, allowing the designers to concentrate on the core functionality required within an SKMS.

On the client side, scalability of the SKCL isn't expected to be a major issue (even for an e-commerce transaction server, which would still be an SKMS client) because the client library would execute within a thread of the client application. It is expected that the designers of the client application will have addressed performance issues for their application in general, and the SKCL will benefit from those design improvements.

The SKMS architecture abstracted the cryptographic processing of the SKMS service to execute outside the service. This allows implementations to use third-party cryptographic accelerators for enhancing the performance of SKMS clients and servers, if desired. Using well-known interfaces such as Public Key Cryptography Standard #11, Cryptography API (CAPI) and Java Cryptography Extension (JCE) allows the SKMS architecture to provide flexibility to implementers for dealing with scalability.

The architecture defines the notion of a **"global" SKS (GSKS)** server, used for defining policy for the SKMS domain. All other SKS servers, named "local" SKS servers, are expected to replicate symmetric keys generated by them to the GSKS. Since every symmetric key has a unique GKID within an SKMS domain, a single GSKS, appropriately sized, is capable of storing every local SKS servers' symmetric keys for redundancy. The GSKS itself never generates any symmetric keys itself; it only serves as a centralized repository within an SKMS infrastructure. In the event of an outage of a local SKS server, the client just contacts the GSKS and requests the symmetric key, much as the nameservice library contacts a different DNS server when the first is unavailable. SKMS implementations must ensure that they have implemented sufficiently redundant GSKS servers to accommodate their business requirements.

Finally, to ensure clients can continue processing – encrypt and decrypt – even in the face of network failures, the SKMS architecture includes "secure key-caching" on the client side. How and when a client may cache keys is based on "key-caching policies" defined on the SKS server, centrally. All SKMS clients can be directed by centralized "key-caching policies" to either cache or not cache symmetric keys on the client-side, securely. While implementers have the flexibility to choose the strategy that works best for them, this design uses the "message-level" security built into the SKMS architecture to enable secure key-caching.

3.4 Application Integration

Since the SKCL is merely a library, it is necessary for an application to integrate the SKCL to take advantage of the benefits offered by an SKMS. However, unlike the DNS model, an application cannot just use the SKCL, acquire the symmetric key, use it and carry on without maintaining some state of the operation it performed.

In order for an application to be able to decrypt its ciphertext, it must maintain knowledge of the GKID of the symmetric key it used in the cryptographic operation. Without the GKID, the application will never know which symmetric key to request from the SKMS.

This architecture recommends the modification of the database schema of the application, to include the storage of the GKID in the same database or file where the ciphertext is stored. If the symmetric key was requested from a non-default domain – one in which the client normally does not belong – the client application may also need to maintain the URL of the SKS service where the key can be located.

For example, where a database schema before SKMS integration might look like the following:

Employee table

Name	Type	Comment
ID	Long	Unique identifier
Name	Char	
DOB	Date	
SSN	Char	
.....		

After including the GKID, the schema might resemble the following:

Modified Employee table

Name	Type	Comment
ID	Long	Unique identifier
GKID	Char	Unique key identifier
NAME	Char	
DOB_CIPHERTEXT	Date	
SSN_CIPHERTEXT	Char	
SSN_SHA256	Char	SHA-256 hash of SSN
.....		

By associating a GKID with every ciphertext in the database, an application can now retrieve the required sym-

metric key from an SKMS to decrypt data when it needs it. However, when ciphertext is transmitted from one application to another, it must carry the GKID with it. It is recommended that the W3C XML Encryption standard be used for the transmittal of such information, since the XML Encryption schema allows for specifying details such as the GKID, SKS server, etc., along with the ciphertext.

An alternative choice to this design was to use a complex data structure in the ciphertext – such as the Cryptographic Message Syntax (CMS) – to embed many cryptographic elements into a binary "blob" that can be stored within a single data element in the database. This has the advantage of carrying many required pieces of information for cryptographic processing, together.

However, this design avoids this for the following reasons:

i. Most applications expected to use the SKMS will be the traditional client-server, enterprise applications that use an RDBMS for data-storage. Developers of such applications expect to see the database schema laid out in its entirety rather than to have data structures collapsed into blobs within a single element;
ii. Collapsing data-structures into a single element hides information that might help improve performance of the client application. For instance, if the application determined that the same GKID was in use for the next 100 records, it might be able to cryptographically process the next 100 records after fetching the symmetric key once, as opposed to fetching it once per record;
iii. It allows the application developers to use standards such as XML Encryption by "exporting" the data elements into XML elements more easily and efficiently than if additional processing had to be performed to "explode" the data-structure for conversion to XML. Once again, this design assumes that XML significantly eases the adoption of complex technologies by neophytes, so decisions were made to favor XML.

Applications link in the SKCL as any other library in the standard software development process. Once linked in, the client application need two additional pieces of information before they can make requests of an SKS server:

1. A list of the SKS/GSKS servers that the client may contact when it needs SKMS services. This information is provided in a text file (much like the /etc/resolv.conf file for DNS clients) and simply lists the URL's of the SKS/GSKS servers that the client may contact.
2. A digital certificate with a corresponding key-pair to be used for signing requests and for decrypting server responses. The key-pair and digital certificate is provisioned to the client in an out-of-band process using any of the traditional PKI mechanisms. It is assumed that the organization implementing the SKMS has the use of a PKI for this purpose.

With this, the client application is now ready to make requests for symmetric key services from the SKMS.

3.5 Security

Given that a centralized key-management system would be the ultimate treasure trove to attackers, it is incumbent that the architecture consider known threats and address them with appropriate counter-measures. The design of the SKMS addresses a number of threats by using "message-level" security in the messages that were in transit or during rest.

Message-level security addresses the following vulnerabilities in the following manner:

- **Authenticity of requests and responses**: Every message arriving at the client or server is assumed to be from an untrusted source. As such, the architecture requires that every message be digitally signed and verified against a trusted certificate hierarchy known to clients and servers within the SKMS. This requires every SKMS client and server be provisioned with an X.509/PKIX-compliant digital certificate for signing requests and responses. This ensures that clients and servers are dealing with only properly validated messages;
- **Integrity of requests and responses**: To ensure that the messages sent and received by SKMS clients and servers do not lose their integrity either accidentally or through a deliberate attack, all SKSML messages use digital signatures to verify the integrity of the messages. These are the same signatures used for authenticity checking;
- **Confidentiality**: Since an SKMS deals with the most sensitive data within an IT infrastructure, it is incumbent on the architecture to protect the payload to the fullest extent possible. **For transmission** of the symmetric key to clients, the design uses public-key encryption, using the digital certificate of the recipient. Since the client is expected to be the only one having access to the private key corresponding to the targeted digital certificate, the payload can only be accessible to authorized recipients. **For storage and recovery** of the symmetric key within the databases, the SKMS architecture uses the public-key in the digital certificate of the SKS server to protect the payload. The architecture also permits the creation of "global" SKS servers, and Security Officers, with their own digital certificates, so that the symmetric key is encrypted with their public-keys too. This allows the symmetric key to be recovered, through an appropriate process, by entities other than the SKS server that generated the symmetric key.

- **Database access**: The SKMS architecture uses an RDBMS to store its data. Traditionally, Database Administrators (DBA) have privileged access to such databases over application administrators and security officers. To ensure that the SKMS database contents are protected from unauthorized manipulation, the SKMS design protects database objects with digital signatures. For example, when SKMS objects are stored in the database, a new digital signature is computed and stored with the object by the SKS server. When reading the object back from the database, the SKS server verifies its digital signature to ensure that the integrity of the message has not been compromised.
- **Key protection**: The astute reader will have recognized that the SKMS architecture relies on the cryptographic key-pair of the clients and servers to preserve the integrity of the infrastructure. This is correct. To ensure these keys are protected, the SKMS uses well-established interfaces to external cryptographic tokens – such as PKCS#11, CAPI and JCE – to take advantage of FIPS 140-2 validated devices, as necessary. By controlling access to the private keys through mechanisms provided in these tokens, an SKMS implementation can extend the security of its infrastructure beyond the boundary of the SKMS client or server itself.

4. SKSML PROTOCOL

The heart of the SKMS is in its protocol – the Symmetric Key Services Markup Language (SKSML). SKSML consists of the following types of messages:

- A request for a single new symmetric key;
- A request for multiple new symmetric keys;
- A request for a single existing symmetric key;
- A request for multiple existing symmetric keys;
- A request for a key-caching policy object;
- A response with a single symmetric key;
- A response with multiple symmetric keys;
- A response with a key-caching policy object;
- A response with a fault message;

Each of these SKMS messages is wrapped in a Web Services Security (WSS) header, which provides the digital signature and encryption capabilities at the message layer using SOAP for object-encapsulation. The use of the WSS standard allowed the SKMS to focus on application functionality rather than on the mechanical details of security messages.

All SKSML messages are in XML, and use the XML Schema Definition (XSD) language to provide the semantic rules for how messages must be constructed.

(*Note: This protocol is currently working its way through the standards process at the Organization for the Advancement of Structured Information Standards (OASIS), and has not been finalized yet. It is, however, targeting for anticipated standards status by the summer of 2008*).

4.1 Request for a single new symmetric key

This request forms the most basic request – for a single new symmetric key without specifying a key-class. (*Note: Key-classes are designations that allow clients to request keys conforming to specific business requirements*.) The abbreviated request (without the WSS header) is as follows:

```
<ekmi:SymkeyRequest
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:ekmi='http://docs.oasis-open.org/ekmi/2008/01>
  <ekmi:GKID>0-0-0</ekmi:GKID>
</ekmi:SymkeyRequest>
```

The highlighted line - **`<ekmi:GKID>0-0-0</ekmi:GKID>`** - essentially specifies a GKID where the DomainID, ServerID and KeyID are all zeros. This special value is an indicator to the server that the request is for a new symmetric key (since all existing keys would have non-zero GKIDs).

The fact that the request does not specify a key-class does not mean that the returned symmetric key does not belong to a specific key-class; the SKS server will simply generate and return a symmetric key of the "Default" key-class for the specific requester.

A request for a single symmetric key of a specific key-class (without the WSS header), is as follows:

```
<ekmi:SymkeyRequest
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:ekmi='http://docs.oasis-open.org/ekmi/2008/01>
  <ekmi:GKID>0-0-0</ekmi:GKID>
  <ekmi:KeyClasses>
      <ekmi:KeyClass>HR-Class</ekmi:KeyClass>
  </ekmi:KeyClasses>
</ekmi:SymkeyRequest>
```

In this case, the client has requested a single symmetric key of the "HR-Class" key-class. The meaning of "HR-Class" is application-defined. It is assumed that the creators of security policies within the SKMS have defined "HR-Class" to mean the issuance of a specific type of symmetric key with specific permissions (to be discussed later).

4.2 Request for a multiple new symmetric keys

While the request for a single symmetric key could dispense with specifying the key-class in the request, a request for more than a single symmetric key must specify the key-classes of the requested keys in the symmetric key request, as follows:

```
<ekmi:SymkeyRequest
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:ekmi='http://docs.oasis-open.org/ekmi/2008/01>
  <ekmi:GKID>0-0-0</ekmi:GKID>
  <ekmi:KeyClasses>
     <ekmi:KeyClass>EHR-CDC</ekmi:KeyClass>
     <ekmi:KeyClass>EHR-CRO</ekmi:KeyClass>
     <ekmi:KeyClass>EHR-DEF</ekmi:KeyClass>
     <ekmi:KeyClass>EHR-EMT</ekmi:KeyClass>
     <ekmi:KeyClass>EHR-HOS</ekmi:KeyClass>
     <ekmi:KeyClass>EHR-INS</ekmi:KeyClass>
     <ekmi:KeyClass>EHR-NUR</ekmi:KeyClass>
     <ekmi:KeyClass>EHR-PAT</ekmi:KeyClass>
     <ekmi:KeyClass>EHR-PHY</ekmi:KeyClass>
  </ekmi:KeyClasses>
</ekmi:SymkeyRequest>
```

In this request, the client application (assumed to be some Electronic Health Record application) is requesting nine (9) new symmetric keys, each corresponding to the named key-class. It is assumed that the EHR application is choosing to encrypt different parts of this new health record with different symmetric keys, so that target users of this EHR will need to request only a subset of the nine symmetric keys to view data meaningful to their business process.

For example, when this patient's health record is stored, encrypted with nine symmetric keys, applications used by nurses will be authorized to request only keys that conform to the EHR-DEF (Default) and the EHR-NUR (Nurse) key-classes. This allows them to perform their work without compromising the security of other parts of this EHR.

It is possible for an application to request multiple new symmetric keys of the same class with the following request – for three new keys of the "ATM-Class" key-class:

```
<ekmi:SymkeyRequest
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:ekmi='http://docs.oasis-open.org/ekmi/2008/01>
  <ekmi:GKID>0-0-0</ekmi:GKID>
  <ekmi:KeyClasses>
     <ekmi:KeyClass>ATM-Class</ekmi:KeyClass>
     <ekmi:KeyClass>ATM-Class</ekmi:KeyClass>
     <ekmi:KeyClass>ATM-Class</ekmi:KeyClass>
  </ekmi:KeyClasses>
</ekmi:SymkeyRequest>
```

4.3 Request for a single existing symmetric key

The following shows a request for a single symmetric key, that was previously generated and escrowed, and is now being requested by the application to decrypt ciphertext. It is assumed that the requester has the authorization to receive this key, otherwise it would be pointless to make the request. In the following example, the client application is requesting a symmetric key with a GKID of 10514-1-23 (the DomainID is 10514, ServerID is 1 and the KeyID is 23):

```
<ekmi:SymkeyRequest
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:ekmi='http://docs.oasis-open.org/ekmi/2008/01>
  <ekmi:GKID>10514-1-23</ekmi:GKID>
</ekmi:SymkeyRequest>
```

4.4 Request for a multiple existing symmetric keys

These requests resemble the following:

```
<ekmi:SymkeyRequest
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:ekmi='http://docs.oasis-open.org/ekmi/2008/01>
  <ekmi:GKID>10514-1-2783</ekmi:GKID>
  <ekmi:GKID>10514-3-532</ekmi:GKID>
  <ekmi:GKID>10514-2-1423</ekmi:GKID>
  <ekmi:GKID>10514-6-243</ekmi:GKID>
</ekmi:SymkeyRequest>
```

Not only is each GKID individually named, but there is no need to specify a key-class, since the returned key is going to belong to whatever class it belonged to, at the time of key-generation.

4.5 Request for a key-caching policy object

The only request type that does not request a symmetric key, is when a client needs to know its key-caching policy (KCP). This is normal in three situations:

i. When the client has been connected to the SKMS for the first time and is making its first request for a symmetric key and must know its caching policy before it can request a key;
ii. The current key-caching policy on the client machine has expired and the client must refresh it to get new policy information;
iii. The key-caching policy object on the client is either corrupted, or the integrity of the object cannot be verified;

In all cases, the client machine sends the following KCP request (shown without the WSS header):

```
<ekmi:KCPRequest
  xmlns:ekmi="http://docs.oasis-open.org/ekmi/2008/01"/>
```

As one can notice, the KCP request is empty.

However, the WSS header that carries the digital signature of the requesting client is all that the SKS server needs, to determine the authenticity of the request as well as the identity of the requester. Based on this deduced & verified information, the SKS server is able to determine the precise key-caching policy that applies to this requester and respond accordingly.

4.6 Response with a single symmetric key

In response to a request for a single symmetric key, whether new or existing, the SKS server responds with the following SKSML (shown without the WSS header):

```
<ekmi:SymkeyResponse
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:ekmi='http://docs.oasis-open.org/ekmi/2008/01'
  xmlns:xenc='http://www.w3.org/2001/04/xmlenc#'
  xsi:schemaLocation=
      'http://docs.oasis-open.org/ekmi/2008/01
  symkeyResponse.xsd'>

  <ekmi:Symkey>
    <ekmi:GKID>10514-1-235</ekmi:GKID>
    <ekmi:KeyUsePolicy>
      <ekmi:KUPID>10514-4</ekmi:KUPID>
      <ekmi:PolicyName>DES-EDE Policy</ekmi:PolicyName>
      <ekmi:KeyClass>HR-Class</ekmi:KeyClass>
      <ekmi:KeyAlgorithm>
        http://www.w3.org/2001/04/xmlenc#tripledes-cbc
      </ekmi:KeyAlgorithm>
      <ekmi:KeySize>192</ekmi:KeySize>
      <ekmi:Status>Active</ekmi:Status>
      <ekmi:Permissions>
        <ekmi:PermittedApplications>
          <ekmi:PermittedApplication>
            <ekmi:ID>10514-23</ekmi:ID>
            <ekmi:ApplicationName>
                 Payroll Application
            </ekmi:ApplicationName>
            <ekmi:Version>1.0</ekmi:Version>
            <ekmi:DigestAlgorithm>
                 http://www.w3.org/2000/09/xmldsig#sha1
            </ekmi:DigestAlgorithm>
            <ekmi:DigestValue>
                 NIG4bKkt4cziEqFFuOoBTM81efU=
            </ekmi:DigestValue>
          </ekmi:PermittedApplication>
        </ekmi:PermittedApplications>
        <ekmi:PermittedDates>
          <ekmi:PermittedDate>
            <ekmi:StartDate>2007-01-01</ekmi:StartDate>
            <ekmi:EndDate>2007-12-31</ekmi:EndDate>
          </ekmi:PermittedDate>
        </ekmi:PermittedDates>
        <ekmi:PermittedTimes>
          <ekmi:PermittedTime>
            <ekmi:StartTime>07:00:00</ekmi:StartTime>
            <ekmi:EndTime>19:00:00</ekmi:EndTime>
          </ekmi:PermittedTime>
        </ekmi:PermittedTimes>
      </ekmi:Permissions>
    </ekmi:KeyUsePolicy>
    <ekmi:EncryptionMethod Algorithm=
         "http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
    <xenc:CipherData>
      <xenc:CipherValue>
         E9zWB/y93hVSzeTLiDcQoDxmlNxTuxSffMNwCJmt1dIqzQH
         BnpdQ81g6DKdkCFjJMhQhywCx9sfYjv9h5FDqUiQXGOca8E
         U871zBoXBjDxjfg1pU8tGFbpWZcd/ATpJD/UJow/qimxi8+
         huUYJMtaGHtXuLlWtx27STRcRpIsY=
      </xenc:CipherValue>
    </xenc:CipherData>
  </ekmi:Symkey>
</ekmi:SymkeyResponse>
```

A successful symmetric-key response (SymkeyResponse) contains one or more Symkey elements, each of which contains the encrypted symmetric key and an associated **KeyUsePolicy (KUP)**. The symmetric key is encrypted with the recipients' public key so that only the targeted recipient of the response message may decrypt it.

The KUP object provides detailed information on how the SKCL may use the associated symmetric key. Other than some meta-data, the heart of the KUP is in the **Permissions** element. SKSML allows SKMS implementations to restrict the use of symmetric keys by specifying a complex permissions-model that permits the use of the symmetric key when the conditions in the permissions-model are satisfied.

The Permissions element in SKSML allows SKMS implementations to restrict the use of the symmetric key based on one or more of the following categories:

- Permitted Applications;
- Permitted Dates;
- Permitted Durations – i.e., between any two date-times;
- Permitted Levels – for multi-level security (MLS) aware systems;
- Permitted Locations – that can be based on GPS coordinates;
- Permitted Times;
- Permitted Transactions – i.e., the actual number of encrypted transactions permitted with a key; and
- Permitted Uses;

If a permission appears for a specific category in the KUP, the SKCL enforces the use of the key according to that permission. If a permission does not appear for a specific category, the key *can be used* within that category without restrictions. For example, if two (2) applications are explicitly listed within *PermittedApplications*, then only the listed applications are authorized to use the symmetric key. However, if the *PermittedApplications* category is missing from the KUP, then *all* applications are allowed the use of the symmetric key within that client.

While this is an atypical method of granting use, it permits the KUP to be minimal when granting broad access to keys. Otherwise, KUPs might tend to become overly verbose and complex.

4.7 Response with multiple symmetric keys

In response to a request for multiple symmetric keys - new or existing - the SKS server responds with the following SKSML (shown without the WSS header):

```
<ekmi:SymkeyResponse
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:ekmi='http://docs.oasis-open.org/ekmi/2008/01'
  xmlns:xenc='http://www.w3.org/2001/04/xmlenc#'
  xsi:schemaLocation=
      'http://docs.oasis-open.org/ekmi/2008/01
  symkeyResponse.xsd'>

  <ekmi:Symkey>
     <ekmi:GKID>10514-4-1235</ekmi:GKID>
     (Content removed for conciseness).
  </ekmi:Symkey>
  <ekmi:Symkey>
     <ekmi:GKID>10514-1-2385</ekmi:GKID>
     (Content removed for conciseness).
  </ekmi:Symkey>
  <ekmi:Symkey>
     <ekmi:GKID>10514-3-1237</ekmi:GKID>
     (Content removed for conciseness).
```

```
  </ekmi:Symkey>
  <ekmi:Symkey>
     <ekmi:GKID>10514-4-1238</ekmi:GKID>
     (Content removed for conciseness).
  </ekmi:Symkey>
</ekmi:SymkeyResponse>
```

The content inside each Symkey element is similar to the response presented in the earlier section (Response with a single symmetric key). The SKCL parses through each Symkey and processes it as if it were a single-key response.

4.8 Response with a key-caching object

In response to a request for a **KeyCachingPolicy (KCP)** the SKS server responds with the following SKSML (shown without the WSS header):

```
<ekmi:KeyCachePolicy
   xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
   xmlns:ekmi='http://docs.oasis-open.org/ekmi/2008/01'
   xsi:schemaLocation=
         'http://docs.oasis-open.org/ekmi/2008/01
   EKMICoreLibrary.xsd'>

   <ekmi:KCPID>10514-17</ekmi:KCPID>
   <ekmi:PolicyName>
        Corporate Laptop Symmetric Key Caching Policy
   </ekmi:PolicyName>
   <ekmi:Description>
      This policy defines how company-issued laptops
      will manage symmetric keys used for file/disk
      encryption in their local cache.  This policy must
      be used by all laptops that use the company EKMI.
   </ekmi:Description>
   <ekmi:StartDate>2008-01-01T00:00:01</ekmi:StartDate>
   <ekmi:EndDate>2008-12-31T24:00:00</ekmi:EndDate>
   <ekmi:PolicyCheckInterval>
      86400
   </ekmi:PolicyCheckInterval>
   <ekmi:Status>Active</ekmi:Status>
   <ekmi:NewKeysCacheDetail>
     <ekmi:MaximumKeys>3</ekmi:MaximumKeys>
     <ekmi:MaximumDuration>86400</ekmi:MaximumDuration>
   </ekmi:NewKeysCacheDetail>
   <ekmi:UsedKeysCacheDetail>
     <ekmi:MaximumKeys>3</ekmi:MaximumKeys>
     <ekmi:MaximumDuration>86400</ekmi:MaximumDuration>
   </ekmi:UsedKeysCacheDetail>
</ekmi:KeyCachePolicy>
```

The KCP essentially tells the client machine how many new and used symmetric keys (used for at least one encryption transaction) it may cache and for how long. This policy is defined centrally for individual, group and/or all clients requesting key-management services. The *PolicyCheckInterval* tells the client how frequently it must check back with the SKS server for updates to the KCP. This is to ensure that client machines' caching policies can be changed centrally with a small notice period.

4.9 Response with a fault message

In the event the SKS server cannot respond to a request for one or more symmetric keys successfully, it sends back the following SKSML (shown without the WSS header):

```
<ekmi:SymkeyResponse
   xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
   xmlns:ekmi='http://docs.oasis-open.org/ekmi/2008/01'
   xsi:schemaLocation=
      'http://docs.oasis-open.org/ekmi/2008/01
   symkeyResponse.xsd'>

   <ekmi:SymkeyError>
      <ekmi:RequestedGKID>0-0-0</ekmi:RequestedGKID>
      <ekmi:RequestedKeyClass>
         HR-Class
      </ekmi:RequestedKeyClass>
      <ekmi:ErrorCode>9025</ekmi:ErrorCode>
      <ekmi:ErrorMessage>
         A KeyUsePolicy to issue a symmetric key with
         the requested key-class does not exist for
         this request.  Please contact your Security
         Officer if you have any questions.  Provide
         them the following information if asked:
         SRID: 10514-2-8643
      </ekmi:ErrorMessage>
   </ekmi:SymkeyError>
</ekmi:SymkeyResponse>
```

The response provides a reference to the requested GKID and key-class for the client application to correlate the error message with its request. It is up to the application to determine how to process the *ErrorCode* and *ErrorMessage* elements.

A response for multiple symmetric keys that results in partial success will return the following SKSML that includes both symmetric keys and SymkeyError's:

```
<ekmi:SymkeyResponse
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:ekmi='http://docs.oasis-open.org/ekmi/2008/01'
  xmlns:xenc='http://www.w3.org/2001/04/xmlenc#'
  xsi:schemaLocation=
      'http://docs.oasis-open.org/ekmi/2008/01
  symkeyResponse.xsd'>

  <ekmi:Symkey></ekmi:Symkey>
  <ekmi:Symkey></ekmi:Symkey>
  <ekmi:Symkey></ekmi:Symkey>
  <ekmi:SymkeyError></ekmi:SymkeyError>
  <ekmi:SymkeyError></ekmi:SymkeyError>

</ekmi:SymkeyResponse>
```

Applications are expected to keep track of which requests received successful responses, which ones did not, and how to deal with the mixed result.

5. IMPLEMENTATION EXPERIENCE

Based on the design and protocol described in earlier sections, an open-source software implementation of this architecture[14] was released on the Internet in 2006.

The SKMS consisted of two centralized SKS servers – a primary and a disaster recovery server – and any number of clients using the **Symmetric Key Client Library (SKCL)** to request services from the SKS servers. (While they are referred to as clients, the client software may themselves be database servers, web-servers, application-servers and/or any business application).

The SKSML protocol implemented in this software is based on a DRAFT 1.0 version of the protocol (which is a little different from the DRAFT 3.0 protocol described earlier in the paper). This protocol is currently going through a standardization process at OASIS[15].

Each implemented SKS server consisted of:

- a server-class computer running an operating system – typically Linux, UNIX or Windows - that had a compliant Java Virtual Machine (JVM) available for it;
- a relational database to serve as the storehouse for the symmetric encryption keys;
- a J2EE-compliant application server to host the application that would respond to requests over the network, serving as the workhorse of the SKMS;
- a JCE-compliant cryptographic provider to perform the cryptographic operations of key-generation, key-protection, digital signing, verification, etc.;
- an optional, **but strongly recommended,** Hardware Security Module (HSM) or Trusted Platform Module (TPM) for securely storing the cryptographic keys that protect the database's contents;
- the SKS server software itself, consisting of an Enterprise Archive (EAR) and a Web Archive (WAR) file for the administration console, along with ancillary utilities;

Each SKCL client platform consisted of:

- a client machine running an operating system – once again, typically, Linux, UNIX or Windows, but included the OS/400 - that had a compliant Java Virtual Machine (JVM) available for it;
- a JCE-compliant cryptographic provider to perform the cryptographic operations of encryption, decryption, digital signing, verification, etc.;
- an optional, but highly recommended, Trusted Platform Module (TPM), smartcard or other USB-based cryptographic token for securely storing the cryptographic keys that protect the clients' authentication credentials;
- the SKCL software itself, consisting of an API callable by Java applications for communicating with the SKS server and performing cryptographic functions (non-Java applications used a Java Native Interface (JNI) library to call the SKCL);

To exercise the protocol a client utility called "xenc" was created that would encrypt files, directories and data in relational database tables. Using xenc and the implemented architecture, we were successfully able to demonstrate the request of symmetric encryption keys from dissimilar client platforms and receive symmetric keys based on predefined policies at the server. The encrypted data was stored in the W3C XML Encryption standard for compatibility and transferred to other platform machines, where another successful call by xenc retrieved the required symmetric key from the SKS server and decrypted the data successfully.

Key-caching was tested by first getting the key-caching policy from the SKS server. This led to the SKCL requesting and receiving symmetric keys from the SKS server to conform to the KCP. Once the keys were cached on the client, the client was disconnected from the network and xenc was used to successfully encrypt and decrypt files on the local client.

6. SECURITY

Given the sensitivity of the information managed within the SKMS implementation, the infrastructure was predicated on an extraordinary level of security. (*As with any security architecture, the controls and procedures in place at the implementation determined the degree of vulnerability the SKMS had against attacks; we still continued to configure the firewall and other operating system controls to secure the machine.*).

The implemented SKMS incorporated the following security features:

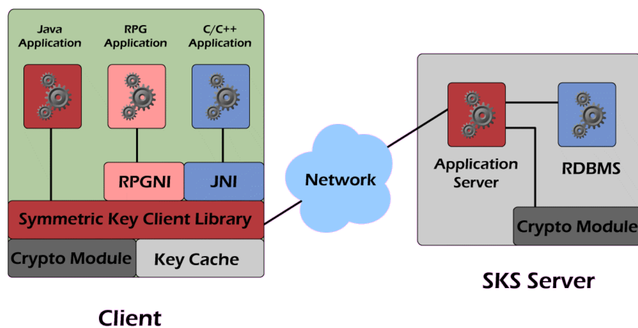
- all symmetric keys were generated using multiple compliant cryptographic providers – some hardware and others in software;
- all symmetric encryption keys were themselves, encrypted using multiple RSA asymmetric keys – one belonging to the SKS server, one to the GSKS and one ot the Security Officer;
- all database records on the SKS server were digitally signed before storage, and verified upon retrieval to ensure their integrity hadn't been compromised;
- all administrative operations through the console were digitally signed and maintained in a history log for audit purposes, and verified upon retrieval;
- all administrative operations through the console required SSL/TLS-based client-authentication;
- only digitally signed client-requests were accepted by the SKS server from SKCL clients;
- only digitally signed responses from the SKS server were accepted by SKCL clients;
- all symmetric keys were transported, encrypted for the specific client making the request;
- all cached-keys on the client were digitally signed and encrypted on storage, decrypted and verified upon retrieval to ensure their integrity;

To have this level of security enabled within the SKMS, and to ensure that this security could scale to internet levels, the architects of the open-source SKMS software predicated the use of a PKI to secure the SKMS.

The PKI allowed the implementers to manage large numbers of digital certificates much more easily than managing raw asymmetric cryptographic keys. With the use of a PKI, every SKMS client and server was issued a digital certificate. Not only was the security level maintained, but once the digital certificates were issued, the provisioning of symmetric key-management services was completely automated thus providing the internet-level scalability required for enterprise operations.

6.1 Operation

The following diagram explains how the implementation works.



When a client – be it a laptop, a DB application or an e-commerce web-server - needs a symmetric key to encrypt some information, it makes a request for a new symmetric key to the linked in SKCL.

The SKCL checks its key-cache to determine if it has any cached symmetric keys that are valid for use. If so, it retrieves the key, decrypts it, verifies its integrity, checks its key-use-policy (every symmetric key object has an encryption policy embedded in it, previously defined by the site Security Officer) and then hands the requesting application the symmetric key for use.

If any of the local checks result in no valid symmetric key being available for use, the SKCL creates a new symmetric-key request, digitally signs it with its authentication credentials, and sends the request to one of its pre-configured SKS servers as an OASIS Web Services Security (WSS)-compliant SOAP request. (*Note: It is noteworthy to mention here, that since all requests and responses between the SKCL and the SKS servers were secured (digitally signed and encrypted) at the message-level, transport-level security (SSL/TLS or IPSec) was not required for the operations of the SKMS; plain old HTTP was sufficient. Administration console communications, however, did rely on mutually-authenticated SSL/TLS)*.

The SKS server, upon receiving such a request, verifies the authenticity and integrity of the request, determines the authorization and the symmetric-key policy in force for the requester (or the default policy), generates a new symmetric key based on this policy, assigns it a **Global Key-ID (GKID)**, escrows the key (which includes encrypting it with multiple RSA keys), encrypts the key with the requester's transport digital certificate, logs the transaction details (which includes digitally signing the transaction) and responds to the client with a WSS-compliant SOAP response.

The SKCL client, upon receiving the response, verifies the authenticity and integrity of the request, caches the secured object if so configured, decrypts the symmetric key and the embedded key-use-policy and returns it to the calling application. The calling application at this time may choose to have the SKCL perform the actual encryption or perform it, itself.

A similar process is repeated when a client application needs to decrypt a previously-encrypted object such as a file, directory of files, database record, etc. The application determines the GKID of the symmetric key it needs (which was previously stored with the encrypted ciphertext in the XML Encryption format for files, and in a corresponding column for an RDBMS) and makes a request for this key to the SKCL. The SKCL checks to see if the requested key is in the key-cache. If it is, it goes through the standard security-checks and returns the symmetric key to the application; if not, it makes a request to the SKS server for this symmetric key. Upon receiving the request and after the standard security-checks, the SKS server responds with the symmetric key to the client. If the key does not exist for any reason, or the client is not authorized to receive the key, or for other error conditions, the SKS server returns a SOAP Fault to the requesting client.

It is noteworthy to mention, that given this operational infrastructure, it was feasible to use a unique symmetric key to encrypt every record in a database. With such an encryption policy, the breach of any key reduces the exposure of the database down to just a single record. This is in stark contrast to existing designs, where a single key typically encrypts an entire database or dataset, thus magnifying the loss associated with the loss of that single key.

7. BUILDING AN SKMS

The construction of an SKMS began with the creation of a PKI – or procurement of PKI services - to manage the issuance of digital certificates to every client. *The architecture deliberately eschewed the use of User-ID/Password for authentication because of their inability to prevent attacks against single-factor credentials.* The clients and servers in an SKMS use digital certificates for authentication, and secure storage & transport of symmetric keys within the infrastructure. (Notwithstanding the use of digi-

tal certificates, the administration console allows an Operations/Security officer to “deactivate” any client or server on the network without revoking the digital certificate of the affected entity).

Simultaneously, the application that will use the SKCL was created/modified to integrate the SKCL's API, accommodate the encrypted data (ciphertext) and the GKID in its database.
Multiple SKS servers were deployed and encryption policies configured on the servers while digital certificates were issued to clients that communicate with the servers. The applications were now ready to start requesting key-management services from the SKS servers. The SKMS transitioned to Production status at this point, and traditional operational activities took over (backup, configuration management, DR, etc.).

8. CONCLUSION

While symmetric encryption has been in use for decades within general computing, we have reached a confluence of inflection points in technology, the Internet and in regulatory affairs, that require IT organizations to implement Symmetric Key Management Systems (SKMS) as independent infrastructures. Using the soon-to-come Symmetric Key Services Markup Language (SKSML) standard from OASIS and the architecture defined in this paper, , IT organizations have another – and perhaps, one of the most effective – defense weapon in their arsenal against an increasingly hostile Internet.

REFERENCES

[1] History of cryptography - http://en.wikipedia.org/wiki/History_of_cryptography

[2] A chronology of data breaches - http://www.privacyrights.org/ar/ChronDataBreaches.htm

[3] California's Senate Bill 1386 - http://www.strongauth.com/regulations/sb1386/sb1386Index.html

[4] Breach at UCLA exposes data on 800,000 - http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9005925

[5] Retailer TJX reports massive data breach - http://www.infoworld.com/article/07/01/17/HNtjxbreach_1.html

[6] TJX Breach was twice as big as admitted, banks say - http://www.theregister.co.uk/2007/10/24/tjx_breach_estimate_grows

[7] TJX Form 10Q - http://www.theregister.co.uk/2007/10/24/tjx_breach_estimate_grows

[8] PCI Security Standards Council - https://www.pcisecuritystandards.org/index.htm

[9] Health Insurance Portability and Accountability Act of 1996 (HIPAA) - http://aspe.hhs.gov/admnsimp/pl104191.htm

[10] The Financial Modernization Act of 1999, aka “Gramm-Leach-Bliley Act (GLBA) - http://www.ftc.gov/privacy/privacyinitiatives/glbact.html

[11] Personal Information Protection and Electronic Documents Act (PIPEDA) - http://www.privcom.gc.ca/legislation/02_06_01_01_e.asp

[12] Directive 95/46/EC of the European Parliament aka EU Directive - http://www.cdt.org/privacy/eudirective/EU_Directive_.html

[13] IANA Private Enterprise Numbers (PEN) as used by RFC2578 - http://www.iana.org/assignments/enterprise-numbers

[14] StrongKey - http://www.strongkey.org

[15] OASIS Enterprise Key Management Infrastructure Technical Committee - http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ekmi