Jonah: Experience Implementing PKIX Reference Freeware

Mary Ellen Zurko[1], John Wray[1], Ian Morrison[2], Mike Shanzer[1], Mike Crane[2], Pat Booth[3], Ellen McDermott[2], Warren Macek[1], Ann Graham[2], Jim Wade[2], and Tom Sandlin[2]

[1]*Iris Associates*
5 Technology Park Drive
Westford, MA 01886

[2] *IBM*
[3] *Lotus*
Contact email: mzurko@iris.com

Abstract

IBM made a business decision to promote an open, secure, interoperable and integrateable environment by embracing the IETF's Public Key Infrastructure (X.509) standards, developing a reference implementation of the core set, and giving it away as freeware. We discuss the architecture of this reference code, emphasizing lessons learned from implementing the standards in an integrated PKI system.

1 Introduction

The explosion of the Internet and the Web has forced the distribution of information and resources. Various types of trust relationships must span these resources and the applications that use them. Application and middleware developers have responded to the challenge by creating their own security solutions, which are unfortunately non-interoperable. Thus, organizations of every kind are finding the management of security to be difficult and costly and interoperability of their applications impossible without major intervention. From a business perspective, with projected growth of business to business Internet commerce going from $8 billion in 1997 to $327 billion by 2002, it was seen by IBM as crucial that the infrastructure be in place to provide an open, secure, interoperable, and integrateable environment. Unfortunately, the trend so far has been for vendors to go their own way, trying to follow standards as they emerge, and otherwise interoperate in a pairwise fashion as the market dictates.

Customer feedback and internal technical advice caused us to focus on creating a common Public Key Infrastructure (PKI). IBM decided that the best strategy to fulfill its business requirements was to find the most mature standard for PKI available and throw its support behind that standard, by developing and making available a freeware reference implementation, and integrating that standard into its products. A freely available reference implementation allows potential integrators to try it out without a large investment, and provide a baseline for interoperability testing. In addition, feedback from the development experience enhances the quality of the specifications. By providing leadership within the industry, IBM believes it advances Internet computing and ultimately drives faster acceptance/implementation of electronic business. IBM's goal is to help the industry converge on one security infrastructure for the web, thus allowing faster deployment of a secure interoperable underpinning for electronic business. We have learned over the years that disparate infrastructures slow evolution and acceptance of technology; the true market is in applications that exploit common infrastructures.

After surveying the body of work already done in this area, as well as the realities of the commercial world, we chose the Internet Engineering Task Force's (IETF) Public Key Infrastructure (PKIX [PKIX]) drafts as our architectural basis. These standards were the beneficiaries of much hard work on the part of many companies, and included the X.509v3 certificate format, which had emerged as a de facto standard. They cover the certificate life-cycle: enrollment and initial certification, key-pair update, certificate update, certificate and Certificate Revocation List (CRL) publication, and certificate revocation. Jonah, our freeware project, currently implements the following PKIX standards: Internet X.509 Public Key Infrastructure Certificate Management Protocols [RFC2510], Certificate Request Message Format [RFC2511], Internet Public Key Infrastructure X.509 Certificate and CRL Profile [RFC2459], and Internet X.509 Public Key Infrastructure LDAPv2 Schema [LDAP98]. Besides implementing PKIX standards, Jonah includes code for Common Data Security Architecture (CDSA) [CDSA97] from Intel, which has been accepted as a standard by The Open Group. CDSA is a framework that can support multiple security providers. The code is a freeware version of IBM's Keyworks product.

IBM then put together its first Iris/Lotus/IBM cross company team. The team consisted mostly of a group of engineers, representing different security and technical specialties, who had different ideas about PKI, yet were able to create a reference implementation using the IETF RFCs that were available at the time.

This team takes advantage of Iris' Domino PKI experience and the pool of talent in PKIs in the Massachusetts area, Lotus' expertise in security product management, and IBM's long history of cryptography and security standards experience, as well as their broad pool of internal expertise. The next section discusses how the architecture was designed and implemented, in spite of moving specifications that had gaps. With Jonah one can run an EE, an RA and a CA. The architectural discussion covers our design for ASN.1 support, implementation of the transaction protocol and extensions to support RA to CA communication, architectural support for portability, use of CDSA as an abstraction layer to cryptography and data storage, our server API and persistent storage support. We then discuss trust policy, also accessed via CDSA, including our support for chain building (which is not specified in the PKIX standards). The discussion of smart card usage includes our layering of PKCS#11 and CDSA. The challenges of implementing a freeware Graphical User Interface (GUI) are outlined in the next section. After that, the section on porting issues discusses how we are porting our NT code to UNIX, and the section on testing emphasizes the issues that arose with testing freeware. We conclude with a look at current status and issues.

2 Description and Evolution of the Jonah Server Architecture

In this section we describe the Jonah architecture as it evolved over time. The architecture was affected by both high- and low-level considerations. The PKIX architecture [RFC2459] describes three PKI components that are involved during enrollment: the End Entity (EE), the Registration Authority (RA), and the Certification Authority (CA). The End Entity acts as the agent of the end-user, participating in the enrollment and other protocols with the RA on the user's behalf. The RA is responsible for enforcing most policy decisions (for example, choice of a distinguished name for the user). The CA is responsible for taking final actions that can occur offline (such as actually creating and signing a certificate).

Since certificates and all PKIX protocols are specified using ASN.1 [ASN.1], a component was needed to perform ASN.1 parsing and generation. A commercial ASN.1 compiler could not be used for freeware. Instead, we opted to construct a C++ class library to handle these tasks. The class library directly implements basic ASN.1 types (e.g., character string) and provides constructs for building up more complicated objects (e.g., sequences). The primary design goal of the class library was that the resulting C++ structures should be programmer-friendly. They should manage their own memory, provide straightforward access to the values within an ASN.1 stream, and hide from the programmer as much as possible ASN.1 concepts (like optional and default values) that relate more to the encoding of values than to their use. The class library enables the construction of high-level objects (like certificates) that know how to encode themselves as an ASN.1 Distinguished Encoding Rules (DER) stream and set their values from an ASN.1 Basic Encoding Rules (BER) stream [ISO8825]. This allows us to treat certificates and protocol messages as straightforward C++ objects, eliminating the need for a specialized "syntax" layer that converts between BER-encoded external messages and internal C++ objects. This decision had a significant impact on the design and implementation of the rest of Jonah. Having a single consistent internal representation of all the high-level PKI constructs as first-class objects greatly simplifies the software, and allows the developers to easily understand one another's code. The library's ease of use is demonstrated by the fact that we chose to use it for encoding much of our on-disk storage (see Section 2.1.2), in spite of there being no requirement to use ASN.1 encoding for purely local storage. In addition, groups within IBM are freely integrating the Jonah ASN.1 objects into their products.

2.1 Certificate Management Protocol Transaction Support

2.1.1 Certificate Management Protocols

PKIX defines protocols by which an EE can communicate with an RA. At the time that the Jonah project commenced, only Certificate Management Protocols (CMP) was at a stage suitable for implementation. CMP is derived from the Entrust PKI management protocol. It targets management functions for the entire certificate/key life. CMP defines message formats in the Certificate Request Message Format (CRMF), and describes various transports and how message protection should be done independent of transport. CMP is the only registration protocol that Jonah currently implements.

The other PKIX protocol is Certificate Management Protocol using CMS (CMC [CMC98]). It is co-authored by representatives of Netscape and Microsoft (among others). Its purpose is to be compatible with PKCS #7 [PKCS7] wrapped responses and PKCS #10 [PKCS10] certificate requests which web browsers currently use. The authors of Certificate Management Messages over CMS (CMC) have also added support for CRMF. CMC attempts to finish all transactions in a single round trip, an important consideration for the Internet where bandwidth and lagtime can be important. At time CMP has become an RFC, while CMC is still a internet-draft. Also, CMP includes its own message protection, so it does not rely on a secure transport for message security. The CMP specification includes a description of TCP, SMTP, or HTTP as transport mechanisms. We chose to implement TCP as the primary transport for CMP, but want to allow additional transport mechanisms to be added reasonably easily. Thus we created an abstraction layer over sockets that could support additional transports. Jonah plans to implement the following CMP messages: Certificate Request/Reply, Revocation Request/Reply, CRL Announce, Proof of Possession Request/Reply, Cross Certification Request/Response, CA Key Update, Confirmation, and our extended General Messages. The EE only talks to the RA. Since the CA can be offline at any given time, or located on an isolated network, direct communications to the CA are kept to a minimum. (For the same reason, all LDAP interactions happen on the RA.)

After we decided to use CMP for EE communications, we needed to decide how the RA and the CA should share information. Nothing in PKIX specified such a protocol. We had the choice of specifying a new protocol or extending an existing one. Since we were implementing CMP and it seemed reasonably full featured and extensible, we decided to use CMP as the RA to CA protocol. In addition, since this is a standard published protocol it would be possible for others to write RAs and CAs that interoperated with ours. So far, we have only had to extend CMP in two areas, CRL requests and RA enrollment (RA to CA association). While CMP supports RA initiated certificate revocation, there is no message defined to allow an RA to request an on-demand CRL. Secondly, while CMP defines the RA and CA, it does not describe how they are introduced. For RA enrollment, we use the standard certificate request message, extending its control information to indicate that it is a request to become an RA for the target CA. We also defined a certificate extension that indicates the certificate is for one of the issuer's RAs. We expect this extension to be useful when an end-entity is looking for RAs. We added new general messages for an RA request a CA to generate and publish a new CRL, and for announcing that an RA is de-enrolling from a CA. We would like to standardize the extensions necessary for extending CMP to RA/CA communications.

2.1.2 Object-Store

Since the enrollment process may require human intervention at any of the three servers, a round-trip enrollment can take a significant amount of time. The system had to be designed to withstand any of its servers being stopped and restarted, without loss of transaction data. For this, each server has a disk-based *object-store* where short-term state data is stored. For example, an enrollment request is constructed at the End Entity by creating an empty request in its object-store, setting various fields in the request (e.g. SubjectName), then submitting the completed request to the RA. When the request is submitted, the object in the EE's object-store is marked as a *surrogate*, indicating that the real object is active on another server. Since CMP is a polling protocol when operating over TCP, the surrogate object stores the data needed to poll the RA for a response. When the enrollment succeeds, the resulting certificate is retrieved and the surrogate object becomes active once again.

Jonah views the object-store as a series of storage locations, each of which can hold an object. The objects stored in the object-store are constructed using the ASN.1 class library discussed earlier in Section 2. They are a sequence of a CMP message and additional control information such as sender and recipient names or addresses. To minimize the ASN.1 parsing overhead, Jonah uses a layer above the object-store that implements a write-through cache of object-store objects. Generally, this means that an object-store object need be parsed only the first time it is referenced by the Jonah code after a server restart. This object-cache layer also provides an additional per-object storage area that is not backed by a disk file. This is used for storing transient, security-related information (for example, the password under which a CMP preregistration record is protected [PKCS5]). An additional feature provided by the object-cache is locking of object-store objects, protecting against simultaneous access by multiple threads.

2.1.3 CMP Issues

One of the goals of doing a reference implementation is to verify the standards against an operational environment. CMP's most outstanding shortcoming is its lack of support for the current de facto standard of PKCS #10 and PKCS #7, as discussed in Section 2.1. Beyond that we have found several small problems with implementing CMP. We discovered places where CMP is vague. The most notable was discussion of the support for multiple certificate requests and revocation requests in a single message. The specification supports them, but handling them is underspecified. More information is needed on mapping multiple replies to the requests. In addition, a compliant service may not support multiples. However, its response to a request with multiple entries is undefined. CMP also mandates many out-of-band steps when initiating a relationship with a new entity (for instance, an EE's first certificate request). An RA administrator may be easy to identify and find in a corporate environment. The approach is unlikely to scale well to the Internet.

We have several issues around CMP's use of time. The CMP TCP protocol uses an absolute time as a polling time instead of a relative time (see Section 2.5 for a description of polling). This makes the assumption that all machine clocks are fairly well synchronized. The polling is represented by a 32-bit integer. CMP does not specify whether it is signed or unsigned, which is an interoperability issue. This integer stores when the next poll should happen based on the number of seconds from January 1, 1970 GMT. If it is signed, it will run out of space in the year 2038. If it is unsigned, it does not run out of space until 2106. If this field was a relative time, the longest amount of time you could have between polling events would be around 136 years, give or take a few months.

The specification should make transaction IDs and polling references more homogenous and consistent. A message transaction is labeled with a transaction ID, an octet string that must be included with every message. It is used to track the transaction from server to server. The polling reference in the protocol is a 32-bit integer value set by the recipient used for associating the recipient's delayed response with the initial request. It would be much easier to track a request from EE to RA to CA and back if these two fields shared the same value. For example, our test team would like to drive many transactions per minute, and track them easily with a value they control. In addition, we use transaction ID to associate a certificate request with the password for it specified via out of band means to the RA. CMP leaves open the exact value used for this purpose.

The PKIX LDAPv2 Schema segregates the list of revoked CAs into Authority Revocation Lists (ARLs), while other revoked certificates are listed in CRLs. Both CMP and the CRL profile are silent about ARLs. We use the Issuing Distribution Point extension to indicate whether a particular CRL contains only user certificates (CRL) or only CA certificates (ARL). We would like to see this method of defining an ARL explicitly called out as such, to minimize the chance of future interoperability problems.

Finally, as is often the case with policy-based decisions, there is some useful information not included in certificate requests. The most obvious missing feature is that there is no algorithm-independent way of determining the key length of the private key associated with a public key supplied in the certificate request.

2.2 Portability Concerns: Threads, Messages and Configuration

Although the primary development platform for Jonah is Windows NT, portability to various UNIX platforms is essential, both for Jonah to succeed as a reference implementation, and also for internal consumption within IBM. The next stage of architectural refinement after protocol definition was to abstract three areas that can be significant sources of portability problems: threading, message catalogs, and configuration. The *threading* primitives used by Jonah are based on Draft 10 of the POSIX 1003.1c threading standard [ISO9945]. The only significant differences are that the **lock** primitive for mutexes has been extended with an optional *timeout* parameter (which allows a thread that has been waiting on a mutex for a significant amount of time to perform another task before re-trying to acquire the mutex, as well as permitting deadlock recovery) and that once-blocks are not supported. The Jonah primitives are expected to be implemented over whatever native thread services are available on the target platform. Implementing them on Windows NT was fairly straightforward.

Jonah uses a *message catalog*, based on the XPG.4 [UNIX] standard, for all its status codes, both internal and exposed. It is intended that a port to a platform with a real XPG.4 system would use the native catalog, rather than the lightweight implementation provided as part of the freeware. One unexpected complication on Windows NT was that the Microsoft C Run Time Library implementation of `printf()` does not support the positional parameters required for handling internationalized messages. The Jonah freeware includes an implementation of `printf()` that supports this feature.

Configuration is handled in Jonah via a Windows 3.1-style initialization file, encapsulated in a C++ class. A productized version of Jonah would most likely replace this implementation with a platform-specific configuration data repository (e.g., the system registry on Windows NT) and GUI access. One of the major uses of this file is to configure policy information indicating how the CA handles certificate requests and revocations, and how any application uses the trust policy code to determine the validity of a certificate chain (Section 3 discusses the latter use).

The RA does all the policy verification of the EE for the CA. The CA trusts the RA for policy decisions like identity verification. The CA also checks all requests against its policy, and may involve an administrator in the final decision. Certificate policy in the initialization file indicates which optional fields the CA supports and which fields the RA must specify the value for. Currently, Jonah CAs can specify whether key usage is supported (and whether it's required), and whether a Policy is required. Policy configuration also indicates whether the RA (or the EE) may specify the certificate validity start time, whether Policy is marked critical in certificates from a CA (and is therefore used to constrain chains of certificates), and which signature algorithms the CA supports. Certificate revocation policy at the RA specifies whether an EE may request the revocation of any certificate, only one of their own certificates, or no certificates. CRL policy at the CA specifies how often CRLs are issued, how long each is valid, and which algorithm should be used to sign them (if the CA maintains more than one type of valid CRL signing key).

Other policy configuration information includes the server's name and the names of its RAs or CAs, and the URLs used to communicate with them. These URLs are of the form pkix://hostname:port. The initialization file is also used for LDAP configuration information at the RA and for text to object identifier (OID) translation. The latter use allows for extensions to the processing of Distinguished Names and algorithms. General configuration information specifies which signing and encryption algorithms a CA supports. The administrator must be careful to specify only algorithms that are supported in software. This is one of several examples of configuration information that motivates the need for a GUI in the future. A GUI could query the CDSA interface (see Section 2.3) to determine what algorithms are supported in software. Another issue with the use of the initialization file is the need to keep CA policy values in synch between CAs and their supporting RAs. Managing configuration information dynamically would allow CAs to notify their RAs of policy changes.

2.3 CDSA

Jonah uses Intel's Common Data Security Architecture (CDSA [CDSA97]) for access to cryptographic and data storage services. CDSA is a framework that supports dynamic loading of modular, pluggable low-level service providers. The Common Security Services Manager (CSSM) layer provides a consistent API to the underlying service provider modules, provides management services for loading and unloading providers, and determines their capabilities. By adopting CDSA, we were able to use existing code for virtual smart card and directory access, as well as allowing easy switching between our two supported cryptographic libraries: The Cylink Foundation Suite and RSADSI's BSAFE [BSAFE]. In Jonah, we decided not to use the Certificate Library (CL) component, since the ASN.1 class library is both more powerful and easier to use than the CDSA CL API. The Jonah reference implementation includes a stripped-down version of IBM's KeyWorks Toolkit, which is a product implementation of CDSA 1.2. The reference implementation removes KeyWorks' value-add features such as mutual authentication of framework and providers, and key recovery features.

Although the CDSA API is powerful, it is not particularly easy to program. Each routine takes parameters that specify the particular provider(s) to use. In Jonah, we preferred to be able to set providers at startup based on information in the configuration file. In addition, the objects that are passed across the CDSA API are CDSA-defined data structures with associated memory management semantics, whereas the natural structures within Jonah are ASN.1 objects. Another drawback to CDSA, which persists with CDSA 2.0, is that applications generally need to know the details of which providers they're dealing with, e.g., the supported interfaces and data structure conventions. Several groups within IBM have tackled this problem by creating an object-oriented layer on top of CDSA that protects the application from these details and presents a uniform interface to CDSA. Supporting a new provider requires only an update to the object-oriented shim, not the calling application. To address all these problems, we wrote a set of wrapper APIs that encapsulate the CDSA Data Storage Library (DL) and Cryptographic Service Provider (CSP) APIs. (Our CDSA Trust Policy support is discussed in Section 3.) They are relatively lightweight shims that provide a great deal of functionality by leveraging the Jonah ASN C++ classes and existing CSP and DL providers. The Jonah Krypto Library (JKL) provides useful domain-related APIs like *JKL_SignCertificate*, *JKL_VerifyCRL*, and *JKL_GenerateRandomData*. It deals with application domain objects like certificates, CRLs, and public keys in Jonah ASN C++ classes. The Jonah Directory Library (JDL) also takes ASN objects as parameters. It can manipulate all of the PKIX LDAP schema object classes and attributes through the CDSA DL interface.

We have extended the LDAP schema to include communications information (URLs) for contacting a CA. This is used for RA enrollment.

Although the architecture provides a framework for pluggable service providers, generally an application needs to know specifically which add-in providers it is dealing with. For example, while the Cylink and BSAFE CSPs present the same interfaces through CSSM, the data formats for each differ. The BSAFE CSP returns keys in Distinguished Encoding Rules (DER) format; the Cylink CSP returns the same objects in flattened uint32 length/data buffers. The calling application must be aware of the data formats expected for common cryptographic data like public keys, private keys, signed data, hashes, and so on.

run-time. The only difference between the three servers is the GUI that starts them (see Section 5). The GUI calls the API to configure the server as either an EE, an RA or a CA. This call is an obvious place to perform initialization functions in a layered fashion. In addition, initialization that is local to a single module is handled by declaring a static object in that module and performing the necessary initialization in the constructor of that object. We planned on per-module finalization in the corresponding destructors. However, a bug in Microsoft Visual C++ (present up to at least Version 5) causes such destructors not to run for modules in a DLL, so we were forced to eliminate any finalization code.

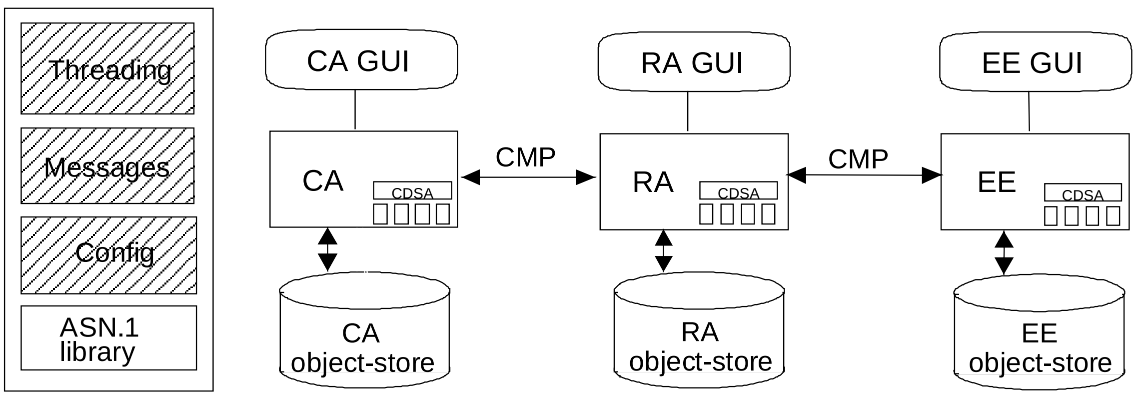

Figure 1. Jonah Architecture

Applications that use CDSA generally need to develop an interface layer that provides easy-to-use, domain specific interfaces that encapsulate the low-level details of CSSM. The result is an unfortunate loss of generality (one of the goals of CSSM) and less plug-and-play functionality. The Jonah CDSA-wrapper layers are the types of middle-tier interfaces needed on top of CDSA to make it useful to a wide range of users.

2.4 Unified Server Architecture and API

Figure 1 illustrates the three Jonah PKIX servers. The EE is a lightweight service that does not require much of the storage and authorization support that exists in the RA and CA. Future versions can integrate stripped down EE functionality into a variety of applications. The RA is the heavyweight process that approves and forwards transactions and interacts with LDAP. The CA is separate to enable offline processing to protect its key, and to enable separation of duty between the RA and CA administrators. This three server model is supported by PKIX transactions.

There is overlap in the infrastructural services used by each server, so Jonah is built as a single logical "back-end", which can be configured as EE, RA or CA at

We decided to expose the CDSA Trust Policy (TP) API as the *end application* API for Jonah (the API to evaluate chains of certificates, see Section 3), limiting the Jonah-specific API to certificate high-level life-cycle management functions (JNH API). Given the *unified-server architecture*, the entire JNH API is available on all servers. Services that are not appropriate at all servers (for example, the EE or RA cannot sign certificates) will return a "Wrong Server" status if invoked on an inappropriate server.

The majority of the API consists of *accessor* functions. These are routines that extract or modify individual fields of an object in the object-cache. Before invoking an accessor function, the API caller must first reserve the target object, which both locks the object against access by other threads and, if necessary, reads it into the object-cache from the object-store. After a series of accessor calls, the object may either be stored back in the object-store, or discarded, and unlocked. This mechanism allows for some degree of transactional behavior when updating object-store objects. The caller makes a series of changes to an object, then saves all the changes back to the object-store in a single operation. Most of the other API functions transfer an object from one server to another, by forwarding, fulfilling or rejecting a request. For example, the CA invokes a JNH_Create_Certificate routine to create a signed certificate. The routine creates a certificate and places it back in the object-store, marking the object-store entry complete, so that the certificate will be returned on the next poll from the RA.

The transaction model means that objects will sometimes appear and disappear in a server's object-store independent of any explicit user action. Any Jonah GUI should be able to present a dynamic view of the contents of the server's object-store, so that certificate and revocation requests can be directly manipulated by the user. This leads naturally to an event-notification mechanism across the JNH API, whereby the back-end sends events to the GUI whenever an object-store object changes state. That state consists of several distinct pieces of information, encoded in a 32-bit integer value: an actual state value, a field that indicates whether an error has been reported or whether a password or PIN is required, a flag that says whether the GUI or the back-end currently "owns" the object, and a flag that indicates whether the object is active or a surrogate. On startup, the GUI makes a JNH call to request that the back-end send a series of events describing the current state of all object-store objects, allowing it to initially populate its view of the object-store.

2.5 Timers and Persistent Storage

CMP over TCP is a polling protocol, which requires a timer component at the RA and EE (the CA never initiates any communication that requires a delayed response). Whenever a message is sent for which a delayed response is expected, the sending server (RA or EE) will record the "next poll time" in the appropriate object-store entry, and queue a *timer* event to occur at that time for the object. On startup, a server walks through its object-store, and requeues timer events for any surrogate objects it finds, according to their next poll time. When the event fires, the corresponding object-store entry is used to poll the CA or RA to see if the delayed response is ready If not, another timer event is queued. In addition to polling, the CA must create CRLs on a regular basis, and both the RA and CA must handle key rollover. The *scheduler* saves queued events on disk, so that they will automatically be delivered on time if the server is running, or otherwise the first time the server runs after the scheduled time.

Creation of CRLs also requires that the CA maintain a secured on-line history of the certificates that it has issued. In Jonah, this history is stored in the *Issued Certificate List*, or ICL. The ICL contains a copy of each certificate issued, indexed by serial number. Since the index is implemented as a general-purpose skip-list [Pugh90], additional indices may be added reasonably easily. Since the ICL is mostly write-only (certificates are never changed, and can only be removed from the ICL by a pruning operation that deletes contiguous blocks of expired certificates from the start of the list), maintenance of these additional indices is relatively straightforward. In the future, we might want to provide a way for verifiers to ask the CA directly about the current status of a given certificate (for example, to support the PKIX Online Certificate Status Protocol).

On the CA, we need to store counters to maintain the serial numbers that will be used for certificates and CRLs. In addition, the CA and RA need a place to store their own certificates, and certificates for other RAs and CAs they trust. To satisfy these needs, we created a binary bin, a catch-all place where any persistent binary data is placed. The BinBin has a simple structure. The three serial numbers (for certificates, end-user CRLs and authority CRLs) appear first, followed by a BER-encoded sequence of certificates. The file is read in on server startup and cached in memory. Any modifications to serial numbers or certificates are written back to the file immediately.

3 Trust Policy (TP)

The Trust Policy (TP) module is implemented as a CDSA add-in (see Section 2.3). The TP interfaces are intended to be called directly by PKIX-based applications that build and validate certificates chains. Both of these activities are expected to be common functions of PKIX certificate using systems, like SSL [FKK96] and S/MIME [S/MIME]. The Jonah TP can be used for both the chain building and chain validation operations, via the CDSA CSSM interfaces *TP_CertGroupConstruct* and *TP_CertGroupVerify*, respectively. The other CDSA TP interfaces are not implemented. The TP performs minimal caching of data retrieved from the directory. Because of time constraints, the freeware TP does not cache directory data. For high performance applications like SSL which cannot afford to perform LDAP queries for each session, this is an unacceptable limitation. Further work is needed to create a trusted cache of certificates and CRLs already retrieved from LDAP and verified. The cache needs to age-out objects from the cache as they expire. Also needed would be options for applications to disable or flush the cache on demand, as well as set the age-out parameter (e.g., "all objects in the cache older than 1 day should be flushed, regardless of expiration time"). Some applications may prefer no cache, and trust only the latest data from the directory. The Jonah cache is not persistent across API invocations nor shared by running threads.

3.1 Validating Certificate Chains

TP_CertGroupVerify is used to validate PKIX certificate chains and uses the algorithm suggested in Section 6 of [RFC2459]. One exception is the validation algorithm for policy constraints and policy mapping, which was taken from [X.50997]. In our opinion, the [X50997] algorithm for validating the complex interaction among the certificate policies, policy mapping, policy constraints extension is more clearly explained and more rigorous that the algorithm defined in [RFC2459]. The X.509 algorithm also allows the user finer control over the initial validation state (via the initial-explicit-policy indicator, initial-policy-mapping-inhibit indicator, and initial-policy-set variables), unlike the [RFC1459] algorithm. Therefore, the Jonah TP validation algorithm is a faithful implementation of Section 6 of [RFC2459], with the exception of policy validation, which follows Section 12 of [X50997]. The inputs to chain validation, passed via parameters, are:

- A list of certificates ordered from an anchor certificate to the end-entity in question. How the chain is obtained is left unspecified in [RFC2459]. For example, the chain may have been sent as part of some application-level protocol flow. Alternatively, the caller may have previously constructed the chain using the TP interface, *TP_CertGroupConstruct*.
- The validation time for which the chain is to be evaluated. Usually, this is the current time, but one could compute the validity of a chain at some point in the past (with an extension to find necessary archival CRLs).
- The *initial-policy-set*, zero or more certificate policyIdentifiers, acceptable to the caller (e.g., a caller may choose to only accept paths consistent with some high-assurance certificate policy identifier).
- Two boolean flags, *initial-explicit-policy* and *initial-policy-mapping-inhibit*. These flags are identical in meaning to the fields in the PolicyConstraints extension [RFC2459]. The effect is to allow the caller to set the policy constraints state immediately, without waiting for the extension to appear in the path. For example, the user can set *inhibit-policy-mapping* to *true* to make sure the identifiers in the *initial-policy-set* are not mapped by a CA in the chain. The user can set *initial-explicit-policy* to *true* to ensure the critical certificate policies in the chain are consistent with the *initial-policy-set*.
- An optional pointer to an LDAP directory server. This pointer will be used, if necessary (and allowed by TP initialization parameters (see Section 3.3)), to retrieve CRLs to check certificate revocation status. The TP uses the JDL interfaces discussed in section 2.3.
- A pointer to a CSP to perform signature verification. The TP uses the JKL interfaces discussed in Section 2.3.

It is the caller's responsibility to verify the signature and revocation status of the top anchor certificate. It is assumed that the calling application maintains a list of trusted anchor certificates, which are deemed acceptable terminal points for certificate chains. Other implementations are possible (e.g., the TP could be configured with its own internal list of anchor certificates or trusted public keys). The application of the PKIX validation rules is straightforward. The Jonah TP performs the following checks:

- Name constraints: full support for permitted and excluded subtrees applied to Distinguished Names and alternative names for strings types rfc822Name, dNSName, directoryName, and uniformResourceIdentifier. Support for the legacy PKCS-9 emailAddress attribute in DNs and name constraint enforcement. The ASN C++ classes provide strong support for such character comparison.
- Policies: full support for certificate policies, policy constraints, and policy mapping.
- Revocation checking: for each CA in the chain, except the first, check issuer's *authorityRevocationList* (ARL) directory attribute for revocation status. For the end-entity certificate, check issuer's *certificationRevocationList* (CRL) directory attribute for revocation status. Other revocation checking methods, like OCSP, delta-CRLs, and CRLDistributionPoints, are not supported in the freeware TP.

As suggested by [RFC2459] and [X.509], the TP returns a list of evidence, or proof, collected as part of the validation process. The proof contains details of any policy mapping that occurred, a list of CRLs/ARLs used, and the list of certificate policyIdentifiers agreed upon by the CAs in the chain as acceptable. As an example, if validation fails because of revocation, the calling application might present the CRL in a GUI display. If validation succeeds, the caller still might decide to reject the certificate based on the policyIdentifiers returned.

3.2 Building Certificate Chains

Consider an application X, with a set of trusted anchor certificates, that needs to authenticate user Y. Authentication is possible if X can discover a certification path from Y to at least one of its trusted anchor certificates. This is expected to be a common operation needed by PKIX-enabled applications and is supported by the TP interface *TP_CertGroupConstruct*.

The PKIX LDAP schema [LDAP98] defines the *certificatePair* directory attribute, which stores all the cross-certificates issued to and by the CA. It holds both forward-certificates (certificates issued to the CA), and optionally, reverse-certificates (certificates issued by the CA). Since Jonah builds chains from the bottom up, only the forward element of the *certificatePair* is used. The Jonah TP chain-building algorithm is as follows:

1. For certificate Y, check if the issuer’s name matches the subject name on one of the trusted anchor certificates T. If so, check that T in fact created certificate Y (by checking Y's signature). If so, stop, and return certification path {Y, T}. Otherwise, continue to step 2.
2. For certificate Y, retrieve all of issuer’s forward certificates, F. Since Y’s issuer may have multiple key pairs and multiple certificates for the same key, perform the following filtering steps to avoid following unpromising paths: (A) Discard any forward certificates whose subject name does not match Y’s issuer name. This protects against bogus certificates in the directory and ensures proper DN name chaining. (B) If Y contains an *AuthorityKeyIdentifier* extension with the *keyIdentifier* field, discard any forward certificates that contain a non-matching *SubjectKeyIdentifier* extension. This makes sure F’s certified public key is the same one used to sign certificate Y.
3. From the remaining list of filtered forward certificates F, check if any were issued by an anchor certificate T. If so, check that T in fact created F’s signature. If so, stop, and return certification path {Y, F, T}. Otherwise, for each forward certificate F, recursively apply step 2.

The algorithm stops when (1) a certification path is found from Y to one of the anchor certificates T, (2) the search fails to find a path connecting certificate Y to one of trusted anchor certificate T, or (3) the search depth in the directory exceeds the INI defined *MaximumChainSearchDepth* (default value is 15). This value sets a reasonable limit on how much effort to expend building a certification path, and provides some protection against a directory seeded with bogus certificates.

Several important points were glossed over in the discussion above. Jonah implements a breadth-first search of the directory. As a result, the shortest chain is returned to the caller. Another possible algorithm is a depth-first search of the directory, which might return a different chain than a breadth-first search. For performance reasons, a depth-first search was avoided because of concerns over lengthy, recursive searches through the directory. Other algorithms were not tried; therefore our comments about one algorithm being to slow are speculative. We really did not have enough sample data to properly weight the merits of each approach. We expect to tune the TP caching model and directory search method based on real world scenarios.

In addition, there is no guarantee that the first chain found is in fact valid. Any of the intermediate certificates in the chain could be revoked, expired, invalid, or incorrectly signed. For performance reasons these checks are not done when building the chain. In fact, some of them cannot be done since extensions like NameConstraints and PolicyConstraints can only be checked from the top down. If the shortest chain happens to be invalid, returning it is of little value. We considered returning all chains (or all chains at a given depth) or returning only those chains that can be verified by *TP_CertGroupVerify*. These options require a search of the directory for all possible paths and were avoided for performance reasons. They also force the caller to decide which chain is the best. We decided to return the first chain that can be verified by *TP_CertGroupVerify* as the best compromise between accuracy and performance. Note that Jonah assigns each chain equal weight; it has no concept of intra-domain or inter-domain paths. All certificates in the *certificatePair* attribute are treated equally. Product versions of Jonah may apply additional rules that rank chains based on some criteria (like closeness to local name hierarchy).

3.3 TP Initialization File Parameters

The Trust Policy has configuration parameters that allow applications and users to set more lenient policy on verifying chains of certificates. The default behavior of the TP is to check the revocation of every certificate in a path. An option allows you to disable CRL checking. This may be useful in environments where a directory service is not used, or CRLs are not issued, or the revocation status is tracked external to the TP. When checking revocation, the TP finds the most relevant CRL available (usually the one with the highest CRLNumber). Two flags allow CRLs with *nextUpdate* dates in the past or with *thisUpdate* dates in the future to be considered valid. The default behavior of the TP is to reject them. Because of possible clock skew between different computer systems, CRLs issued on one system may not be active on another system until a few seconds or minutes in the future. Another flag allows a zero search result for a CA's CRL/ARL to be considered non-fatal. Every CA should have at least one CRL and ARL, even if they are empty. Since PKIX requires every CA to issue at least an empty CRL, enabling this is not recommended. The default TP behavior is to reject chains where CRLs cannot be located. This flag might be useful in cases where a CA publishes certificates to the directory, but for some reason, does not publish CRLs.

Flags allow certificates with *notAfter* and validity dates in the past or *notBefore* validity dates in the future to be considered valid. The default TP behavior is to reject them. The reasons for allowing them are the same as for future CRLs. Turning on these flags turns off certificate validity checking, which is not PKIX compliant. Another flag allows name constraints to be applied only to the end-entity certificate. During validation, name constraints are enforced for every certificate in the chain. Some practitioners believe that, in practice, the two algorithms should give the same results for any reasonable hierarchy of CAs, and that any differences would not be important.

4 Smart Card Usage in Jonah

Smart cards are portable cryptographic devices that are suitable for storing certificates and keys, as well as performing cryptographic operations with the keys without releasing the private key off the card (specifically signing). Jonah contains a virtual smart card (one implemented totally in software with private keys and certificates stored in a file) which allows experimental use of Jonah in environments without smart card hardware. The EE, RA and CA servers each have a smart card configured. The CA and RA each initialize their smart cards to have a private key and corresponding certificate that are self-signed. This allows RAs to sign requests and CAs to sign CRLs without exposing their private keys. PKCS#11 [PKCS11] provides the interface to a smart card.

Smart card support performs the following functions: smart card initialization, changing a smart card password, storing and retrieving a certificate, storing a private key, retrieving information about a private key, returning information about the certificates associated with a private key, generating a key pair, signing, and verification. CAs and RAs use it as a storage mechanism if they need to distribute their certificate to other entities. EEs also store certificates for verifiers with whom they exchange data, although they are limited by the available storage on the smart card. Typically, a smart card will include its own software package used to administer the smart card for functions such as initialization and password management. Administration of a PKCS#11 smart card was somewhat problematic. This is because while the PKCS#11 standard has administrative functions, most smart cards provide their own non-standard administration interfaces. While we provide functions to administer the virtual smart card through PKCS#11, administration of other smart cards will certainly be specific to each smart card.

The Jonah smart card interface provides an API for smart card operations to the rest of Jonah. It translates Jonah ASN.1 data structures and semantics into CDSA data structures and semantics, then calls CDSA to access the smart card. Use of the CDSA framework in our virtual smart card support provides consistency across Jonah for access to our cryptographic providers. We anticipate that CDSA will make it easier for Jonah to use PKCS #11 compliant hardware smart cards. The cryptographic operations provided by the virtual smart card go back through the CDSA interfaces to CSPs that are configured to work with Jonah. Since smart cards devices typically work through a serial connection at speeds around 9600 baud, and smart card cryptographic functions are fairly slow, we do not think this layering will cause any notable performance problems.

The smart card CDSA support translates calls into PKCS#11 calls. In the Jonah reference implementation, a PKCS#11 virtual smart card is used. A smart card must support PKCS#11 functions for key pair generation, data storage, and signing to be fully functional for the Jonah reference implementation. Private keys are stored on a PKCS#11 smart card by splitting the private keys into their base parts, including a modulus. When keys are generated on the smart card, a key identifier is created for indexing keys on the smart card. In addition, the public key is returned to Jonah for use in a certificate request. When using the smart card for key pair generation, the private key will usually never leave the smart card. This is the safest way to generate and store private keys. If archival of keys is required, the keys can be generated in software and archived, then stored on the smart card. We have struggled with issues around protecting private keys. The best design for storing and protecting keys for archival purposes seems to be to use a secret key to protect the private keys while stored in a PKCS#12 [PKCS12] file. Hardware smart cards may not be willing to export private keys they generate. There may be later interoperability problems between such smart cards and applications that insist on having private keys in their own local keyring. We also need to protect virtual smart card data stored in a file. We settled on utilizing hashes of the security officer and user pins, as well as combinations of random and secret keys.

We discovered several issues while integrating virtual smart card support with the Jonah public key infrastructure. We found several places needing translation layers between what Jonah needs and what smart cards traditionally provide. As discussed above, we needed several translation layers to allow access to the smart card via CDSA. In addition, the general practice with smart cards is to use the index generated by the card when referencing items stored on the card. Jonah needs to index and manage multiple certificates that correspond to a single private key. For example, a CA may have different certificates for certificate signing and for CRL signing, but both may reference the same private key.

5 Jonah's Graphical User Interface

The Jonah GUI, like the rest of Jonah, must run on multiple platforms including Windows NT and various platforms of UNIX (specifically Solaris and AIX). As freeware, the GUI is downloaded and built on all of the above platforms. Staffing issues and the initial snapshot schedule meant that from concept to initial release, the GUI had to handle certificate requests and communicate with the back-end within 3 months. This section describes how we met those goals, as well as issues encountered so far.

While the portability layer (see Section 2.2) allowed us to write the back-end in a portable, platform-neutral fashion in plain C++, this approach is not suitable for GUI implementation. We decided to build the GUIs in Java [Java], taking advantage of its platform independence, language safety, and object-oriented classes. The Java Development Kit (JDK [Borl98a]) 1.1, with its compiler and run-time virtual machine, is free and is supported on all of our target platforms. JDK 1.2 was not considered because of lack of support for UNIX platforms like AIX. The GUI code invokes the JNH API via a Java Native Interface (JNI [Cay98]) wrapper layer. We used the Java Foundation Classes (JFC)/Swing classes [Nels98], and associated freeware widgets, instead of the Java Active Window Toolkit (AWT [Flan97]) to save development time. We chose Borland JBuilder 2.0 as our Java development environment because it produces 100% pure cross-platform code, has a fully supported implementation of the JFC, has an upgradeable path to future releases of JDK 1.2 and beyond, shares its Interface Developer's Environment with Delphi (a mature environment for building Pascal programs), and our team members have had excellent experience with Borland products in the past. We used JBuilder's ability to create individual freely releasable widgets as Java Beans for Jonah-specific inputs like a spin button for dates and X500 distinguished name input. The Jonah GUI consists of three Jonah GUI executables for the CA, RA and EE. This Java byte code is zipped up as JAR [Borl98b] files into four packages, CA, RA, EE and Base. The Base class, like the back-end, is used for the common functionality shared by the servers. This code sharing has already eased debugging and support. Use of inheritance within the OO paradigm also paid off in reusing code. The layering provided by the JNI wrapper made it very easy to divide up the work and to detect which layer crashes were in.

The event-notification mechanism required some effort to integrate with a Java GUI. Java threads are not necessarily implemented using the platform's native threading mechanism. A Java Virtual Machine (JVM [Deit97]) may multiplex a single native thread to service all Java threads [Davi96]. Therefore, a native thread does not necessarily have sufficient thread context data to be able to invoke Java code. Java threads may invoke native routines, but the JNI does not permit calls in the reverse direction. This restriction required that the Java GUI implement an event collection thread, whose sole purpose is to check for events and pass them back to the main Java thread. The JNI wrapper layer implements a message-queue to pass event data from the native notification call-back routine to the event collection thread. This, in turn, required that the Jonah locking primitives be exposed at the JNH layer, so that the event notification callback routine and the Java event collection thread could use them to synchronize access to the message queue. Since the JVM might be implemented as a single native thread, we could not block the collection thread while it waited for an event, since this might block the entire JVM. Therefore, if the collection routine finds no events waiting, it calls back into Java and performs a Java sleep() operation before checking the message queue again. Thus, apart from very brief periods when the collection thread is holding a Jonah mutex to inspect the contents of the message queue, it is using pure Java synchronization mechanisms, which allow the JVM to continue to run regardless of how it implements Java threads. We added a second similar notification mechanism to allow the back-end to send text messages for display to the user, either on the GUI status-bar, as pop-up dialog boxes, or in a scrolling log window. The last was extremely useful during debugging. The current GUI has three application threads: the main thread that maintains the GUI itself, and two event collection threads for events and text messages.

One difficulty was providing a front-end that was user-friendly and intuitive to the PKIX standard back-end, supporting only standard data types for displaying information. The PKIX standard does not always provide the information one would choose while designing a front-end. An example is validity dates. The PKIX standard uses start and end dates for certificate validity. An end-user or RA administrator might more reasonably enter a validity duration to be applied against the time the CA signs the certificate or a start time and a duration. Extra code and checks are required in the GUI to set up the data as PKIX standard. Extra code keeps the user's notion of duration aligned with the beginning and ending validity dates. Distinguished Name (DN) support is also a challenge. The GUI is coded to understand the full standard format of DNs. It contains a full encoding of the attribute types and their ordering for the X.500 useful object classes, and can lead the user through the creation of a DN by presenting the next valid attribute type as a field for editing. This Java bean is fully table driven and can therefore be easily extended to support additional schema. Although the entire team has had several discussions of how best to support DN input and editing, we are not thoroughly satisfied with our current solution. In addition, expertise in GUI development and in security and PKIX standards is split across team members. Every new GUI feature generates a great deal of discussion on both sides of the split. While this is a substantial investment in time, the fresh ideas on both sides have produced many breakthroughs. Having a single contractor develop the entire GUI reduced coordination and code overhead, and expertise also guaranteed that the front-end followed standard Windows and Motif behavior, although it necessitated many long hours of work.

6 Porting Jonah

This section discusses the build environment, Standard Template Libraries (STL); Java portability; issues between NT and UNIX; and CDSA, message, and threads porting issues. The back-end, C++ portion of Jonah was originally built using the MKS make program [MKS97] with batch file wrappers to simulate the OSF Development Environment (ODE [ODE]). Both the NT batch files and the MKS Makefiles were not portable, since the MKS syntax is similar to but not identical to systems provided by the AT&T UNIX operating system. Our Java build environment, JBuilder, is also an NT-specific tool. In order to port all the build procedures to UNIX platforms, we used ODE. ODE has been ported to many platforms including UNIX, NT and Windows. We have not found a software compilation environment that supports as many platforms as the OSF ODE environment supports. It supports both C++ and Java.

One area that turned out to be a problem in porting Jonah from NT to AIX was the Standard Template Libraries. The Standard Template Library [Step94] provides a set of C++ container classes and generic algorithms. It is based on research in generic programming and generic software libraries. When the template is instantiated or invoked, types are supplied as parameters and methods are created for those types. The Standard Template Library has been adopted as part of the February 1998 ANSI/ISO C++ standard. Microsoft Visual C++ conforms to this version of the standard. Unfortunately, most compiler venders have not had time to come out with conforming implementations. The C++ compiler that we use on AIX is based on a 1992 version of the ANSI/ISO C++ standard. To allow programmers to use the STL on platforms that do not have a compiler that conforms to the February 1998 version of C++ standard, STL has been heavily parameterized with conditional compilation flags that indicate whether a compiler supports various template features. For example, our AIX compiler does not support default arguments for templates, so we had to back out of using that feature. It takes some investment of time to determine what template features a compiler supports and how to correctly set the corresponding flags. We may have similar problems for any new platform Jonah is ported to. In addition, the AIX compiler also tends to be very strict about what it allows. For instance, it requires that if a template class is defined then each method and operator used by the class must be defined, even if the application does not invoke it. The Microsoft Visual C++ 5.0 compiler only requires the methods and operators be defined if they are actually instantiated.

There was very little work to port the Java GUI to AIX. We wrote some build rules for the ODE build environment that compile the Java source and create a JAR for each of the GUIs, and wrote a simple script that invoked the GUIs under the JDK. No changes to the Java source code were needed. By adhering to the JDK's JNI standard, the Java code is able to call the native C++ code in a portable manner. In some cases, native data types were incorrectly declared (for instance, as "long" instead of "jlong"). This generated some mapping bugs. The Java code was easier to port than the C++ code, though based on efficiency of generated code, we believe coding the back-end in C++ was the right decision.

We ran into some problems with general differences between NT and UNIX. Since NT is case insensitive, our developers were not careful when applying case to new file names and then maintaining case sensitive #include statements. Since the UNIX operating system is case sensitive, there were several places where we had to either change the name of the include file or the #include directive in the code. Also, the initialization files contain information about where to store various Jonah files in the file system. On NT, you include the drive specifier along with the path, while on UNIX systems there is not a drive specifier. These directives need to be modified to conform to the file system structure of each operating system. Our CDSA code contains routines for storing information about where the plugins should be found and loaded. On the NT platform, this information is stored in the NT registry as the default database to store OS-specific information. On AIX, the code stores the data in the AIX Object Data Management (ODM) database, which performs an equivalent function. The code to find plugins needs to be changed for each operating system to use a data store similar to the NT registry or the AIX ODM. One possible option is to use a DB44 database [Slee] if the OS does not have a common database.

Jonah attempts to segregate all of its platform-specific code to a single layer, called the OSSRV layer. Exceptions to this are found in the CDSA layer and a few places where small changes are conditionally compiled depending on the platform. The OSSRV layer consists of support for the portability library (see Section 2.2). We had one porting decision to make around message files, and found one porting problem in threads. Instead of converting the error messages to the format that the native AIX message tools require, and creating calls to AIX messaging routines, we decided to port the message catalog routines used on NT to AIX. The Jonah code supplies an XPG4 like set of routines and a message catalog generating program (gencat). Since the Jonah code only opens one catalog file and most gencat routines expect to open a separate catalog for each .msg file, we ported the gencat and XPG4 routines provided with the Jonah delivery. In addition, we ran into a problem where the default stack size on AIX for each thread (100K) was not large enough for the C++ code. This problem caused Jonah to crash in a number of random ways. We were surprised to find that we needed to increase the default stack size of the threads created by Jonah to 256K.

7 Testing

The final aspect of Jonah development is testing. As discussed by Marick [Mari97], the role of a test team is to find bugs. However, there is a distinction to be drawn between a test designed to find a large volume of bugs with minimal significance and one designed to find bugs that will have a major impact on the core use of the code. The goal of the Jonah test team is the latter. This was defined as those bugs that users encounter during basic use of the code, including usability problems impacting the user's ability to understand how to use the code. The focus of the test is, therefore, on exercising the main paths of the Jonah code (certificate creation, certificate revocation, and issuing CRLs). While many of the challenges with testing Jonah are familiar territory, some are exacerbated by, or unique to, its status as a freeware package. This section discusses the issues involved in testing the Jonah freeware code. It describes differences between freeware testing and regular product testing, setting priorities, and the challenges the test team faced.

Since Jonah is a freeware package that uses PKIX drafts as requirements and specifications documents, the test team believed that test processes needed to be modified. In the usual process, testing is divided into Functional Verification Testing (FVT) and System Verification Testing (SVT) [IBM98]. FVT verifies that the code functions correctly with respect to product specifications. FVT exercises the external and internal interfaces, functions, error handling capabilities, and the maintainability and serviceability of the product. SVT verifies that the code works correctly as a *system*. The goal of SVT is to test products in a customer-like environment (at times, utilizing industry customer scenarios) thus ensuring delivery of high-quality solutions that meet or exceed customer expectations. SVT exercises scenarios that test not only product requirements, but also load and stress; interoperability and coexistence; installation, migration, configuration and connectivity; reliability, system-level error recovery, and internationalization. For Jonah, with a goal of finding the most important defects, the test team executed a mixture of traditional FVT and SVT scenarios. We exercised the external interfaces (using the GUI and some APIs) and functions using an integrated system. We attempted to exercise the product requirements (using the PKIX drafts as the specifications), as well as product installation using minimal configuration scenarios.

Another aspect of testing freeware code is that limited resources (hardware, people, and time) are allocated to the effort. As a result, the test team is much smaller than usual. Thus, we had to set priorities. As previously discussed, the test team chose to focus mainly on the mainline paths through the code, using defaults instead of making major modification to different parameters. Additionally, since we understood that the first potential customers (IBM product groups) would use the APIs to exercise the Jonah code, the test team chose to spend time using the API set. This also allowed us to begin building our test suites, which could be used for later product testing. For the API testing, we chose the C interface since we had both example code and a skill set in place to code C applications. In respect to LDAP testing, we limited the test to one LDAP server, choosing the IBM LDAP 2.0 server [IBMLDAP], since this was the easiest server for us to install. We also developed a process to document the important bugs requiring fixing.

The biggest testing challenge was the test team's lack of knowledge about the technology. Because of organizational changes, within days of receiving the Jonah test assignment, management expected a positive contribution from the test team. Learning the technology involved reading the Internet drafts, talking with the developers, and practicing with the code at every opportunity, and asking numerous questions. Contributing to this challenge was the fact that the Jonah code was itself lacking documentation and continually referenced the PKIX drafts. Finally, the test team came from a very structured test environment and was used to testing mature products. Jonah required us to adjust our thinking from a strict process driven environment (including test plan execution plan, daily status, and war room meetings) to a less structured environment that focused on delivering the code as soon as possible. Our test team has worked hard to make the adjustment and they have contributed substantially to the quality of our code.

8 Status and Conclusions

The Jonah code is being hosted by the Massachusetts Institute of Technology (MIT), who assume responsibility for code distribution and change control. It is downloadable from the web site at MIT at http://web.mit.edu/pfl. MIT agreed to host the code because they have hosted security code with export control issues in the past (such as Kerberos), and because Jeff Schiller is both IETF Security Area Director and is responsible for the MIT network. The Internet Mail Consortium hosts the Jonah discussion list at http://www.imc.org/imc-pfl/. Also, interest in PKIX reference implementations is increasing. NIST [NIST] and DSTC [Oscar] have also announced reference implementations.

Interoperability is important to our goal of being a reference implementation. We are currently involved in a sequence of interoperability testing events with Entrust, NIST, Xeti, and Baltimore Technologies. We have exchanged certificate requests and responses to test interoperability. This has uncovered both mistakes in the different implementations and deficiencies in the PKIX specifications. The majority of the problems discovered in the first round of testing were with the ASN.1 encoding rules. The CMP specification uses ASN.1 explicit tagging by default. The CRMF specification uses ASN.1 implicit tagging by default. ASN.1 also has rules about when you must use explicit tagging. For instance, explicit tagging is required within a *choice*, which overrides the default of implicit tagging in the CRMF specification. Another related problem is that some of the ASN.1 structures have changed from earlier drafts to the final RFC. The lack of change control in versions of these RFCs means that small changes are often not discovered until interoperability testing. In addition, the specification is vague about how to associate a certificate request with the password for that request shared with the RA. The purpose of the interoperability events is to identify problem areas and feed then into the next version of the protocol. Participants will be reporting on the results at the July IETF meeting.

One of the largest challenges we have faced implementing freeware reference code has been organizational. Our team crosses organizational units, company cultures, geographical sites, and time zones. We have worked hard to work together, and have been recognized for it with two IBM teamwork awards. Many of the organizational difficulties we had were exacerbated when working across expertise or geography. Even though our business case had been made, we had continuing difficulty getting and keeping resources in the face of pressure in nearby revenue-making groups who also needed resources. With a small team, some of whom were juggling other commitments, we adopted a heads-down, lets-code attitude without design documents or a detailed schedule. This has produced the desired aggressively timed freeware snapshots, but also placed an added burden on areas such as GUI development, testing, porting, and management. For example, testing has not always been told just what was in a code drop, which made it difficult for them to determine where to concentrate their efforts. They are also located in Texas, while the team leads work out of Massachusetts, so special effort to communicate was required. As we transition internally from freeware to internal product support, we are producing more documents and communicating more effectively.

We are not aware of many other papers on the topic of developing and distributing freeware reference implementations. Kerberos [SNS88] is a well documented standard backed by freeware code. However, most discussion of freeware implementations occurs on email lists or informally at IETF sessions. We hope others will document their systems and experience implementing freeware reference implementations.

Acknowledgments

Jonah would not have been possible without the contributions of Les Kendall, Rina Rivera, Shiu Fun Poon, Maryann Hondo, Dave Dyer, Charlie Kaufman, Mark Davis, and Stacey Powers. This paper has benefited from the work of our editor, Lynda Balthorp, the comments from our anonymous reviewers, and the direction of our shepherd, Carl Ellison.

Bibliography

[ASN.1]"Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation", ISO.IEC 824-1.

[Borl98a] “Borland JBuilder2. Developers Guide.” Borland International Inc., Scotts Valley, CA 95067-0001, 1998.

[Borl98b] “Borland JBuilder2. Quick Start.” Borland International Inc., Scotts Valley, CA 95067-0001, 1998.

[BSAFE] “RSA BSAFE CryptoC.” http://www.rsa.com/rsa/products/cryptoc/index.html.

[Cay98] Cay S. Horstmann, Gary Cornell, “Core Java Volume II- Advanced features.” Sun Microsystems Press, 901 San Antonio Road, Palo Alto, California, 1998.

[CDSA97] "Common Data Security Architecture." Intel, http://developer.intel.com/ial/security/specifications.htm.

[CMC98] Michael Myers, Xiaoyi Liu, Barbara Fox, and Jeff Weinstein. “Certificate Management Messages over CMS.” November 11, 1998, http://www.ietf.org/internet-drafts/draft-ietf-pkix-cmc-04.txt.

[Davi96] Stephen R. Davis, “Learn Java Now.” Microsoft Press, 1996.

[Deit97] H. M. Deitel, P.J. Deitel, “Java How To Program.” Prentice Hall Inc., Upper Saddle River, New Jersey, 1997.

[FKK96] A. Frier, P. Karlton, and P. Kocher, "The SSL 3.0 Protocol." Netscape Communications Corp., Nov 18, 1996.

[Flan97] David Flanagan, “Java In A Nutshell.” O’Reilley & Associates Inc., 101 Morris Street, Sebastopol, CA 95472, May 1997.

[IBM98] IBM, “IBM Development Guidelines (DevGuide) Integrated Product Development (IPD) Guide Release 1.7”, 1998, http://w3.enterlib.ibm.com/cgi-bin/bookmgr/books/zdgipd.

[IBMLDAP] IBM, "Lightweight Directory Access Protocol." http://www.software.ibm.com/network/directory.

[ISO8825] "Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER)." ISO/IEC 8825-1.

[ISO9945] "Information technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language]." ISO/IEC 9945-1:1996.

[Java] “Java Platform Documentation.” http://java.sun.com/docs/.

[LDAP98] Sharon Boeyen, Tim Howes, and Pat Richard, “Internet X.509 Public Key Infrastructure LDAPv2 Schema.” September 1998, http://www.ietf.org/internet-drafts/draft-ietf-pkix-ldapv2-schema-02.txt.

[Mari97] Marick, Brian, “Classic Testing Mistakes.” Testing Foundations, 1997. http://www.stlabs.com/marick/Classic/mistakes.html.

[MKS97] “MKS Toolkit User Guide.” Mortice Kern Systems Inc., 35 King Street North, Waterloo, Ontario N2J 2W9, Canada, 1997, http://www.mks.com/solution/tk/.

[Nels98] Mathew T. Nelson, “Java Foundation Classes.” McGraw-Hill Inc., 11 West 19th Street, New York, NY 10011, 1998.

[NIST] “MISPC CD-ROM Request Form.” http://csrc.nist.gov/pki/mispc/refimp/cdj2.htm.

[ODE] “ODE Home Page.” http://www.ede.com/ode/.

[Oscar] “Oscar: DSTC’s Public Key Infrastructure Project.” http://oscar.dstc.qut.edu.au/.

[PKCS5] “Password Based Cryptography Standard.” RSA LABS, 3rd draft, 2/2/1999. ftp://ftp.rsa.com/pub/pkcs/pkcs-5v2/pkcs-5v2draft3.pdf.

[PKCS7] “PKCS #7: Cryptographic Message Syntax Standard.” RSA Laboratories Technical Note, Version 1.5, Revised November 1, 1993, ftp://ftp.rsa.com/pub/pkcs/ascii/pkcs-7.asc.

[PKCS10] “PKCS #10: Certification Request Syntax Standard.” RSA Laboratories Technical Note, Version 1.0, November 1, 1993, ftp://ftp.rsa.com/pub/pkcs/ascii/pkcs-10.asc.

[PKCS11] “PKCS#11: Cryptographic Token Interface Standard.” RSA Laboratories, Redwood City, CA. http://www.rsa.com/rsalabs/pubs/PKCS/html/pkcs-11.html.

[PKCS12] “PKCS #12: Personal Information Exchange Syntax Standard.” RSA Laboratories Technical Note, Version 1.0, April 30, 1997, ftp://ftp.rsa.com/pub/pkcs/pkcs-12/PKCS12.PDF.

[PKIX] “Public Key Infrastructure (X.509) Charter.” http://www.ietf.org/html.charters/pkix-charter.html.

[Pugh90] William Push, “Skip lists: a probabilistic alternative to balanced trees.” Communications of the ACM, Vol. 33, No. 6 (June 1990).

[RFC2459] R. Housley, W. Ford, W. Polk, and D. Solo, "Internet X.509 Public Key Infrastructure Certificate and CRL Profile." January, 1999, ftp://ftp.isi.edu/in-notes/rfc2459.txt.

[RFC2510] C. Adams, S. Farrell, “Internet X.509 Public Key Infrastructure Certificate Management Protocols.” March 1999, ftp://ftp.isi.edu/in-notes/rfc2510.txt.

[RFC2511] M. Myers, C. Adams, D. Solo, and D. Kemp, “Internet X.509 Certificate Request Message Format.” March 1999, ftp://ftp.isi.edu/in-notes/rfc2511.txt.

[Slee] “The Sleepycat Software Home page.” http://www.sleepycat.com/.

[S/MIME] Blake Ramsdell, Editor, "S/MIME Version 3 Certificate Handling." http://www.imc.org/draft-ietf-smime-cert, Expires June 14, 1999.

[SNS88] J.G. Steiner, B.C. Newman, and J. I. Schiller, “Kerberos: An Authentication Service for Open Network Systems.” Proc. USENIX Conference, February 1988.

[Step94] Alexander Stepanov, Meng Lee, “The Standard Template Library.” HP Labs Technical Report HPL-94-34 (R. 1), August 1994.

[UNIX] "The Single UNIX Specification - 5 Vol Set for UNIX 95." The Open Group, T910.

[X.50997] "ITU-T Recommendation X.509: The Directory Authentication Framework." 1997.