A Secure Service Discovery Service

Steven Czerwinski and Ben Zhao
{czerwin, ravenben}@cs.berkeley.edu
University of California, Berkeley

December 14, 1998

Abstract

We have designed and implemented a security framework for the Service Discovery Service that provides service descriptions to clients on the Ninja framework. The new security model provides authentication, data integrity, and access control. The security framework utilizes public key encryption for authentication, shared key encryption for communication, and an access control list / capability hybrid model to provide flexible and scalable access control.

1 Introduction

Two major trends, that of increasing pervasiveness of ubiquitous networks and the increasing numbers of networked devices, are shaping the future of systems research. As the network reaches further into every aspect of our lives, it offers the myriad of network services to a variety of computing devices. The key for clients to successfully leverage network services lies in searching for and locating those useful services. As a vital part of the network, the directory service must be depended on to provide correct and secure information to clients.

We've implemented a prototype of the Service Discovery Service (SDS) on top of the Ninja active network framework. Our objective in this project was to ensure that service data are correctly and securely communicated according to flexible access policies.

In this paper, we will first discuss in Section 2 Ninja, the proactive network that the SDS runs on, and the motivation for integrating a security framework into the SDS. Then in Section 3, we will examine some of the related work done in directory services, and how their security models compare to those of the SDS. We will then discuss the specific design goals of this project in Section 4 and follow that with a detailed discussion of the secure SDS architecture. Next, we will discuss our implementation efforts, and some initial results on performance analysis. Finally, we will mention our points for future work, and take a look at how our design and implementation have met our initial goals.

2 Background

2.1 Ninja and the SDS

Our research has been conducted on the Ninja active networks framework. Ninja exemplifies the kind of active network that will be flourishing in the future, embedding computational resources inside the network. The Ninja model is built around the idea that relays and routers in the network should possess computational power, and use it to provide novel functionality to Ninja clients. Ninja exemplifies the type of network that demands a directory service, because a wide array of Ninja services are location independent, and could reside on any node on the network.

The Service Discovery Service addresses this need by providing a directory service for the potentially millions of services that may reside on a wide area Ninja network. The design of the SDS calls for a hierarchical network of SDS servers sharing the network coverage both in terms of service descriptions and service to clients. The main components of the SDS infrastructure include the SDS server, the client, and the services. In the previous non-secure SDS model, services provide their service descriptions encoded in XML to the SDS server, which forwards them as requested to the appropriate SDS clients.

2.2 Motivation for the Secure SDS Project

The simple prototype of the SDS does not address any security concerns. As we prepare the SDS for deployment into the Ninja network, we realize that security is an integral part of the discovery service, and a failure to address the security concerns could lead to catastrophic consequences for the users and the network as a whole.

It's easy to imagine some of the results of deploying an unprotected location discovery service into a network such as Ninja. Attackers can easily set up malicious services on the network and distribute misleading service descriptions through out the network of SDS servers. Trusting clients will then divulge sensitive information to the attackers. Another scenario could have unauthorized clients making use of costly services provided by other organizations. We give a more detailed discussion of the possible attacks against the SDS server in Section 4.

3 Related Work

There has been a lot of work in the area of location or discovery and directory services. Numerous projects have tried to perfect the discovery service. Some notable examples include Service Location Protocol (SLP), Domain Naming Service (DNS), and Lightweight Directory Access Protocol (LDAP)

Not all of the directory service efforts have viewed security as a desired goal. Depending on the type of data being searched for and the desired performance, security may or may not be worth the performance degradation. One notable example is DNS. The focus in DNS is for performance, and since DNS servers refresh their cache over time, invalid entries will be flushed of its own accord. The high requirement on throughput and minimum latency make a fully secure DNS difficult to design.

The Service Location Protocol does have a notion of built in security, through the use of local scopes [5]. The security model is all or nothing, however, and is not extensible to the wide area. Also, the lack of validation for User Agents reject the possibility of access control on even the scope level.

Commercial implementations of directory services such as Microsoft's Active Directory and Novell's Novell Directory Service include full implementations of access control mechanisms, but are too heavy weight, and are generally dependent on their specific platform for authentication [3, 4, 1]. LDAP also includes access control, but like the commercial implementations, lack the encryption mechanism necessary to service privacy [2].

4 Design Goals

Before we examine the details of the project design, we should outline the main goals of the project. There are several possible attacks against a directory service such as the SDS. We will explain our primary goals with respect to these attacks, and discuss how each attack is or is not affected by our project design.

4.1 Primary Goals

In the context of the Ninja network, the SDS is responsible for delivering descriptions of mobile services available on the Ninja network to Ninja clients. In this respect, its functionality is very similar to those of DNS and SLP. As is with the other projects, the main security concern is the correctness of the data offered. To this end, the main focus of the secure SDS project is to guarantee that the SDS does not refer the client to a malicious service. A secondary goal of the secure SDS effort is to impose a strict access control model on the service descriptions. This goal follows from the fact that the Ninja network is meant to be deployed in the wide area, making user-based access control to resources a high priority. Restricting information about these services is one step towards accomplishing that goal.

With that in mind, we also consciously make the decision that we will not attempt to circumvent denial of service attacks on the SDS server or the SDS infrastructure. Compared to the consequences of mistakenly passing sensitive information to a malicious server or users accessing sensitive information from the SDS, a denial of service attack is much less severe, and would require a level of complexity we consider to be unwarranted.

4.2 Threat Model

The SDS provides several points of attack for the malicious attacker. Attacks can be launched against the identities of the clients, identities of the servers, authenticity of the service data, reliability of the data transport, and unauthorized knowledge and usage of services.

4.2.1 Attacking Identity

Possibly the easiest attack to orchestrate on the SDS framework and consequently the Ninja network is to spoof the identity of either an SDS server or an authorized client. Once the false identity is assumed, the attacker can wreak havoc on the SDS system as well as the Ninja network.

A malevolent SDS server may intercept requests, and return responses that guide a user to a corrupt service. The evil SDS may also allow unauthorized users access to private information, or act irrationally to cause confusion in the SDS hierarchy. An attacker who assumes the identity of an authorized user receives all of the users privileges. Depending on whether the compromised identity is an administrator, the attacker could provide access to private services to any number of outside users, or simply take advantage of the resources himself.

As stated before, the secure SDS project's intent was to focus on attacks that result in real damage, namely, the loss of sensitive service information to malicious services and unauthorized clients.

Our solution to this problem is to use public key certificates and cryptographic signatures to establish authentication of users and servers.

4.2.2 Attacking the Service Data

The correct operation of the SDS revolves around the transferral of correct service description data. In a scenario where that is compromised by attackers broadcasting invalid or misleading service descriptions, the functionality of the SDS is completely defeated. More importantly, the unknowing SDS server would lead trusting users to malevolent services.

To solve this problem, we extend the public key authentication system to include service descriptions, so that each service description is accompanied by its certificate and an authorized signature. The combination can be verified to establish a true association between the service and a trusted principal.

4.2.3 Attacking the Data Transport

Even though clients, servers and service descriptions can all be authenticated with their own private key signatures, the integrity of that data is not ensured without a secure transport mechanism. An attacker can intercept messages between the various components of the system, in order to gain access to knowledge of private services or to tamper with messages in order to circumvent the authentication mechanisms.

As a result, we include in the design encrypted versions of Java RMI and unreliable IP broadcasts. The result is a communication framework that is immune to eavesdropping and tampering at the transport level.

4.2.4 Unauthorized Usage and Knowledge

One of the motivations in the initial design of the secure SDS was that we wanted to protect users who wish to restrict not only the usage of their services, but also the knowledge of those services. So in responses to client requests, the SDS should only return service descriptions that the client has authorization to access.

To satisfy this design goal, we introduce the notion of user capabilities into the SDS framework. Each user owns a list of capabilities which define the set of security domains the user is authorized to access.

4.3 Denial of Service Attacks

While protecting against denial of service attacks is not one of our design goals, it is useful to discuss some of these attacks and what consequences they will have on the normal operations of the SDS system.

4.3.1 Flooding

The simplest of the denial of service attacks is a derivative of the standard flooding attack. The malicious party can flood the SDS server with requests or alternatively flooding the network. The result is each case is to make it difficult for the SDS server to handle requests from legitimate clients. This can be defended against using currently known techniques, such as selectively dropping packets and address filtering.

4.3.2 Misleading Service Advertisements

As we will show later, our authentication mechanism prevents several attacks, including misleading clients to malicious services. Our authentication mechanism, however, depends on DSA signature verification, which is a computationally non-trivial operation. As a result, malicious parties can slow down the server by injecting misleading descriptions with invalid signatures into the network. There's no simple defense against this attack, but we might devise filters to discard advertisements and requests from addresses with repeated authentication failures.

4.3.3 Masquerading Attempts

Because our security depends largely on authentication through certificates and digital signatures, we open up an avenue for denial of service through signature and certificate verification. Similar to misleader service advertisements, the malicious party might request other principals' certificates or create invalid signatures, all in an effort to degrade the performance of the SDS. These attacks can be handled the same way as we do misleading service advertisements, through filters that discard requests from clients who repeatedly fail to authenticate themselves.

5 Architecture

To realize our design goals, we built our system using a wide range of cryptographic programming concepts: from public key certificates to message digests, the secure SDS has it all. In this section, we will describe some of these tools, and how they fit into the individual components of the system. Where appropriate, we will explain our motivation for choosing the tools we used, and the tradeoffs in using alternative methods.

5.1 Overview

Figure 1 depicts all the components that make up the Service Discovery Service, and how they communicate with one another. The method of communication between two components is labeled on the path between them. Also, dashed arrows correspond to periodic broadcasts, where as solid lines correspond to one-time RMI connections. We should point out that this figure is meant as a high-level overview of the design of the system, not as representative of the actual classes that implement the SDS. There are a few minor differences in the implementation, as discussed in section 6.

The communication methods between the components probably represent most of our design work. After all, protecting the information going over the untrusted network was one of our most important design goals. We had to guard against intruders from both listening in on sensitive data, and from inserting or modifying packets to propagate inaccurate information. Section 5.2 will discuss the methods of we used to accomplish this.

Of course, the communication mechanism is only part of the system. Sections 5.4 through 5.7 describe the various components of the SDS, focusing on their security roles. For now, we will just point out that every component is identified and authenticated by a public key certificate. The certificates are signed by well known certificate authorities, and it is up to the individual components to set the policy on which certificates they trust.

5.2 Broadcasts

Broadcasts are an integral part of how information is propagated in the SDS, and therefore, the first subject of redesign. As figure 1 shows, broadcasts are used to announce a SDS server's existence to the world, and to announce services to the SDS server. These broadcasts act as a form of heartbeat – they continuously remind the recipient that the server is available. These constant advertisements also make it easier for components to rebuild soft state after experiencing a failure, since they can just listen for a short period and relearn all the service information.

5.2.1 Authenticated Server Broadcasts

SDS server broadcasts were easy to secure because the design goals only called for the broadcasts to be authenticated. We do not care if outsiders know where an SDS server is running; instead, we would like to prevent others from making false SDS server advertisements. In order to implement this, the server broadcast packet simply contains the certificate name used by the SDS server and a signature of all the clear text information in the packet. The packet information also includes a time-stamp to prevent replays. Upon receipt, the user can verify the certificate, verify the signature, and decide if he trusts the certificate principal.

In terms of performance, this design was reasonable. We set a policy that the same signature could be used for broadcasts for up to five minutes if the information being advertised was unchanged . Thus, the SDS server only has to perform a public key signing every

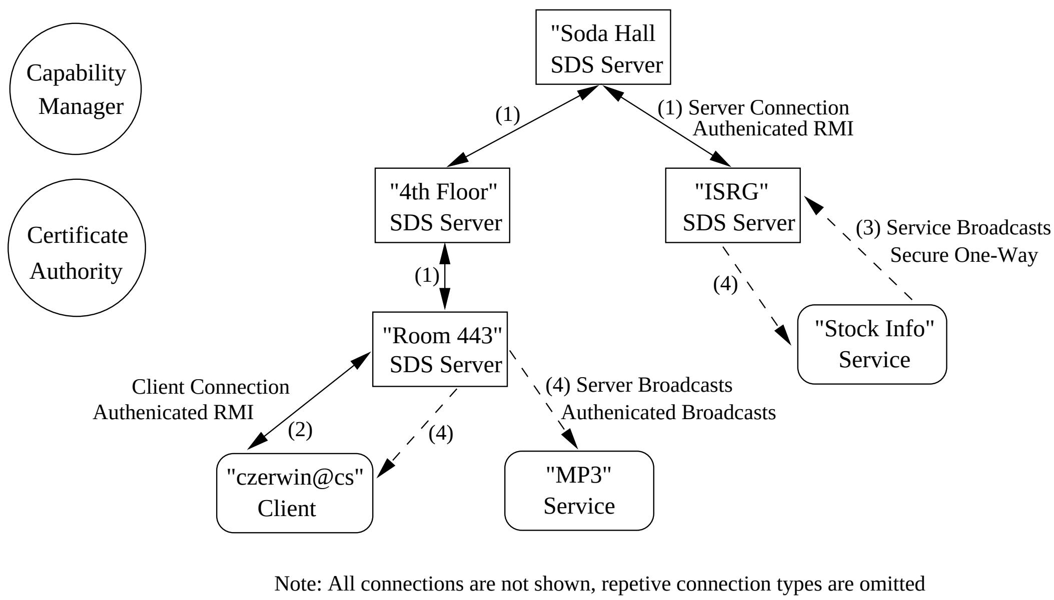

Figure 1: Design Overview

five minutes. Correspondingly, the broadcast listeners only has to re-verify the broadcast every five minutes, which is reasonable, especially since verification is cheaper than the signing.

5.2.2 Secure One-way Service Broadcasts

The service broadcasts represented a more difficult problem to solve. These broadcasts are used to advertise the existence of a particular service to the SDS server responsible for their domain. They are sent every few seconds, so that SDS servers will quickly notice if a particular service has failed. Our design called for these service broadcasts to be encrypted, since we did not want information regarding sensitive services to be exposed. Additionally, we wanted the information being advertised to be authenticated, so that the SDS server could map a service and its description to a particular certificate.

Early in the design, we realized we could not use public key cryptography to encrypt the service announcements, since we wanted a single SDS server to handle thousands of service advertisements per minute. However, we could not really use symmetric key cryptography either, since that requires the service and server to share some sort of secret, which violates principles of soft state. The SDS server should always be able to throw away its state, and completely rebuild all the service descriptions by listening to the broadcasts. This allows the SDS servers to handle faults much more easily.

To solve this problem, we created a hybrid public key / symmetric key system that allowed the services to transmit a single packet describing their service securely, while allowing servers to decrypt the payload using a symmetric key. Figure 2 shows the packet format. In essence, the beginning portion of the packet, which we refer to as the *ciphered secret*, contains a symmetric key (S_K) which is encrypted using the destination server's public encryption key (E_K). This symmetric key is then used to encrypt the data payload of the rest of the packet. The server's encryption key is published in the certificate authority, and is changed every day or so, allowing the individual services to cache it. Since only the server's private key can decrypt the ciphered secret, only the server will be uncover the symmetric key. To authenticate the service to the server, the plaintext of the ciphered secret is signed using the service's certificate (C_P).

With this design, the SDS server does not need to perform a private key decryption every time it receives an announcement from a service. This is because the ciphered secret is cache-able: once a SDS server sees a particular ciphered secret from a service, it can cache the decryption result for later reuse. Basically, the server retains the original ciphertext along with the symmetric key and expiration, and when it sees the same ciphertext from the service, it can skip the public key decryption and get the symmetric key from its cache.

ID	Ciphered Secret	Payload
Sender Name	$\{$Sender, Destination, Expire, S_K, Sign(C_P)$\}_{E_K}$	$\{$Data, Time, MAC$\}_{S_K}$

Figure 2: Format of Secure One-Way Broadcasts

Because the symmetric keys are fairly long lived (on the order of hours), the cache key will be good for awhile. Thus, the server only has to do one public key decryption per couple hours per service, instead of the one per second per service, as in the old scheme. When a SDS server is serving thousands of services, this is a significant saving.

This design works well for the soft-state model of the SDS server: each announcement includes all the information the SDS server needs to decrypt it. The caching of the cipher secret is just an optimization, allowing the SDS server to only have to perform a few public key operations in the common case.

5.3 Authenticated RMI

The new broadcast protocols only solved part of the problem. For the communication between individual SDS servers and between client applications and SDS servers, we needed a protocol that was two-way and authenticated both endpoints. To this end, we used authenticated Remote Method Invocation (RMI). This is basically a protocol layer that allows applications to invoke methods on remote objects in an authenticated, encrypted fashion. We did not implement a version of authenticated RMI for our project, but rather used an implementation made for the Ninja project by Ian Goldberg and David Wagner.

Authenticated RMI has many features that made it ideal for our project. First, RMI worked well into our basic object-oriented design. We created the project in Java after all, so it made the usage of the protocol a lot easier. Second, the end-to-end authentication is based on public-key certificates, which allow us to integrate it easily into our design. Basically, each endpoint has a certificate, and when establishing a connection, the certificates are exchanged. Also, in the implementation we used, application writers could specify a policy to decide which certificates to accept for the connection. So an client using an SDS server could specify a policy that made sure they accessed remote SDS servers that had an "sds-server" certificate. Finally, authenticated RMI does provide encryption on the information being transfered, thus providing the secrecy we desired. The performance of the encryption shouldn't be too bad, since it is a symmetric encryption method: after the initial public-key handshake on establishing a connection, the two endpoints decide on a symmetric key for encryption. We will talk about the actual performance of the authenticated RMI in Section 6.

5.4 Certificate Authority

Because we base the majority of the security system on public key cryptography, it's essential to have a secure Certificate Authority to distribute certificates. Certificates authenticate a binding between a principal and its public key, and are the basis of trust for verifying digital signatures in establishing identities of SDS components.

As discussed in Section 5.2.2, we also make use of encryption keys to encrypt broadcast messages so that only the destined receiver will be able to decrypt them. This design decision calls for the existence of an encryption key distribution service, much like the certificate distribution service. In actuality, the encryption keys are simply public keys which are securely bound to the primary certificate of the principal. In an effort to eliminate complexity and network traffic, we decided to leverage off their similarities, and integrate the two services into a single certificate distribution service.

5.4.1 Certificate Authority

The certificate distribution portion of this distribution service is very simple. For security reasons, we want to keep the creation and signing of certificates offline. The online portion of the Certificate Authority only serves as a distributor of certificates which have been generated offline.

The certificate authority is a very simple structure. The validity of a certificate is based on the signature of another principal, testifying to its validity. On verification, the certificate is checked for the existence of the signature, and then the signature can be verified with the signer's certificate. This recursive verification method can be carried out for as many levels as desired, although we do place a constant boundary on the number of recursions in order to detect signing loops. So certificates are based on a hierarchical web of trust, where ultimately, the trust emanates from one or more trusted signers at the top of the certificate tree. As a result, certificates can be verified using the data they embed, and authentication of certificate requests is not necessary.

5.4.2 Encryption Key Certificates

Encryption keys are used in the security framework to encrypt broadcasts between the SDS server and the services, ensuring that only the addressee and decrypt and read the contents. In order to provide more security redundancy, we allow for multiple encryption keys to be associated with a single principal. The idea there is that if one single encryption key is compromised, the damage would be minimized while all parties using the compromised key are informed, and a new encryption key is chosen. Another reason for using multiple encryption keys is to allow for the existence of different communication channels for the same principal. In the event that the same principal is responsible for authenticating multiple services, then we would like to have a different encryption key for each service, instead of having different services being able to read each other's messages. This follows our design decision that the trust is based in the authenticator of a service, not the service itself. As such, we expect the same authenticator or principal to apply to different services, which leads to a need for multiple encryption channels per principal.

Operation-wise, the encryption key distribution mechanism works just like the certificate authority. For each principal, a set of encryption key certificates are associated using an encryption key index. It's the job of the encryption key distributor to take in a principal name, an encryption key index, and return the correct encryption key certificate. The encryption keys are embedded in their own certificates, providing an authenticated binding between the principal name, encryption key index name, and the public key of the encryption key pair. Because the encryption public key can only be used to encrypt messages addressed to the principal or one of its instances, there is no need to authenticate requests for the certificates. Internally, the encryption key distributor uses a simple hashtable to store the certificates.

5.5 Capability Manager

An important goal of the SDS server is to provide real time access control on the service descriptions that the SDS servers provide. We implement this as part of the Capability Manager (CM). This follows from the idea that we want users to have total control over their own resources, including not only who can access them, but who can see their existence. For instance, we may have a set of private printers in our local security domain, and we want to ensure that not only do external principals not have access to our printing services, but that they do not even know of its existence. It should be noted that we found little value in providing a model where users can know of a service but have no access to it, so we designed a system whether knowledge implied access, and vice versa.

5.5.1 Hybrid Model

To satisfy this need, we needed a capability based access control system to orchestrate who has access to what services. It's long been known that there is a problem with capability based systems, that of capability revocation. Access control lists (ACLs), in contrast, are much easier to administer and revoke privileges on. But ACLs have their own problems, namely verification, where a traversal of the access structure has to occur in order for a privilege to be verified. As a result, ACL based access verification is tied to the server, and not scalable at all. Our conclusion was that in order to provide a flexible system with reasonable administration and revocation, we would need a hybrid between capabilities and access control lists. The access control lists would be the base structure for storing privileges, but capabilities are the instantiation of those privileges in a form easily verified by a third party. By making capabilities expire within a certain time period, we can do coarse grain revocation on the size of the time constant chosen.

5.5.2 Access Control List

At the base of our privilege structure, we have a large table of permission objects, each responsible for representing the privileges of a certain principal on a certain resource or service. The actual ACL is implemented using a two-dimensional indexed hashtable, which allows for fast access for its entries, as well as providing a model that does not waste storage.

Associated with each principal and service pair is a permission object, which essentially store two boolean values, *admin* and *grant*. While *grant* simply denotes whether the principal has access to know of and use this service, *admin* represents whether the principal is the authenticator and hence the administrator of the service. As the authenticator of a service, the principal's certificate becomes the service certificate, imparting the principal's trust onto the service. As the administrator, the principal is responsible for making changes to the access control list. Such modifications require authentication of the principal's identity before succeeding.

5.5.3 Capabilities

Capabilities are the low level structures that represent access to a service. A principal has a set of capabilities that has been assigned to it, and can request those capabilities at any given time. Capabilities are meant as short term tickets to prove access to a service, and are renewed on a regularly basis as they expire.

Internally, capabilities are just secure bindings between a principal's name, the service in question, and the level of permission granted. Also embedded inside the certificate is an expiration time. The security is provided in the form of a digital signature by the capability manager, and can be verified using the manager's certificate. The signature is based on the entire contents of the capability, ensure that the verification would fail in the event the capability was tampered with. And since capabilities encode the principal and service names, they are useless to anyone but the principal, and can be distributed without authentication.

One performance issue that we are concerned about is the creation of capabilities, since we have shown that DSA signatures are computational non-trivial. So what we've done is to essentially emulate the effect of an offline signer, by providing a background certificate generator that routinely cycles through the ACL and generates any expired capabilities that are not currently stored in the cache. This allows the signing to be low priority, as long as the creations of the capabilities takes less time than the total expiration time of a capability.

5.6 SDS servers

As mentioned earlier, the SDS servers are responsible for collecting information about the available services in their domain, and to execute client queries on the service information. In order to fulfill this, the SDS server performs several actions. First, it advertises its existence along with the CM's and CA's in an authenticated manner so clients and services know how to contact them. It then listens for service descriptions, which are transmitted in a secure manner. Finally, it accepts authenticated RMI connections from clients, over which the clients invoke the search routines on the SDS server, passing in the clients capabilities. In this section, we will describe the main security roles the SDS is trusted to perform.

The most important security role the SDS server performs is to guard the sensitive service information it holds. The SDS server has access to the clear text descriptions of all the services in its domain, including those of private services. These services trust the SDS server not to distribute their information to non-authorized users.

To do so, the SDS server must accurately authenticate all clients who connect to it over RMI. Also, it must make sure that the user passes in valid capabilities when he wishes to do a search on service information. These capabilities describe what information rights the user has, thus determining what services the SDS server can return. We should point out that the capabilities do not provide any authentication information: they simply map a certificate name to a set of privileges. The SDS server must make sure that the certificate used to authenticate the client to the server through RMI is the same certificate specified in the capabilities.

Another role of the SDS server is to update its publicly known encryption key regularly and make sure that the encryption key is difficult to break. This encryption key is used by the services to transmit their sensitive information, and if it is compromised, an adversary could eavesdrop on the information being transmitted. The adversary could even use it to obtain the symmetric key, and therefore forge future service broadcasts. Of course, services could perform checks on the encryption key and refuse to use it, but that would be more difficult.

One role the SDS server does not perform is ensuring services correctly implement whatever service they advertise themselves to be. The SDS server simply guarantees that the service description they return to clients was signed by the certificate specified. In fact, the SDS servers return the certificate and the signature as part of the requests, so the user can verify it for themselves. However, the user must decide based on the signing certificate if they trust that service or not.

Finally, we have to trust the SDS server to accurately return search results. This means that the SDS should not return old service announcements that are clearly out of date. Also, the SDS server cannot hold back information about available services. There is no easy way to prevent false negatives, so the clients need to trust their SDS servers.

5.7 Services

The services have limited security roles in the SDS system. First, they must transmit their service descriptions securely to a valid SDS server. In order to be sure the SDS server is valid, they must decode the SDS server broadcasts, and decide whether or not they trust the signing certificate. After that, they use the secure one-way service broadcast mechanism described in Section 5.2.2. To do this correctly, they need to always encode their information using the latest SDS server encryption key, and to change the symmetric key often. And since they have to sign their service descriptions online, they need to protect their private key which is kept in memory.

The individual services are also responsible for contacting the Capability Manager in a secure fashion and properly setting the permissions for individual users. If the service does not do this, no one will be able to gain any information about the service.

6 Implementation and Status

We have implemented our current design as a set of services in the Ninja infrastructure. Each component of the SDS is written in Java and is an iSpace service. We used many of the Ninja components, especially the `ninja.rmi.auth` and `ninja.ispace.auth` packages, which provide the framework for certificates and authenticated RMI.

For the additional cryptography we needed, we relied on a publicly available Java library supplied by Cryptix. Their package included routines to perform such operations as RSA encryption/decryption, Triple DES encryption/decryption, and an assortment of message digest operations. We also used the security extensions supplied by Sun in the `java.security` package. This package provide the base framework for dealing with certificates and keys.

The structure of the implementation, shown in Figure 3, follows mostly from the structure of the design, with a few minor exceptions. In the current version, the CA, CM and SDS server are all iSpace services, and are required to run in the same iSpace. There is no critical reason why they must in the same iSpace, and in future versions, we will change this. All services are also implemented as iSpace services, and therefore must also run in an iSpace. Client applications, on the other hand, do not need to run in iSpaces, which cuts down on the overhead needed in order to use the system from a user point of view.

The only component in Figure 3 that was not discussed in the design section is the iSpaceForwarder. This component discovers the local SDS server through its broadcasts, generates service descriptions for all services running in an iSpace, and forwards the descriptions to the SDS server. In our design, we had said each service would perform this action itself, but later we decided this would make service implementation too hard. Rather, we provided the iSpaceForwarder as a general tool. Since the iSpaceForwarder automatically generates descriptions for the services and signs them using the service's certificate, it must be trusted. The iSpaceForwarder is part of the infrastructure, however, which means it's part of the trusted code base regardless.

Table 1 shows the cryptographic algorithms used for the various components and their associated parameters. Our design does not rely on any of the algorithm choices. We simply chose the algorithms based on what was available, and set what we thought were reasonable parameters. Our implementation is fairly modular, so we could change any parameters and algorithms easily.

The only choice that we were unhappy with at first was the use of RSA with an ECB blocking scheme for the secure one-way service broadcasts. We would have rather used a CBC blocking scheme, but were constrained by the Cryptix library, since their RSA does not work with CBC. In the future, we would like to remedy this. Also, as we found out later, Triple DES might be too heavy weight to use for encryption in the service broadcasts. We will discuss this more in Section 7.

Our current implementation also has a rather simplistic strategy for accepting certificates. Currently, as long as the certificate is signed by the central certificate authority (i.e., signed by Ian), it is accepted and whatever action is being performed proceeds. This is fine for users using the SDS, but might not be appropriate for authenticating components such as the

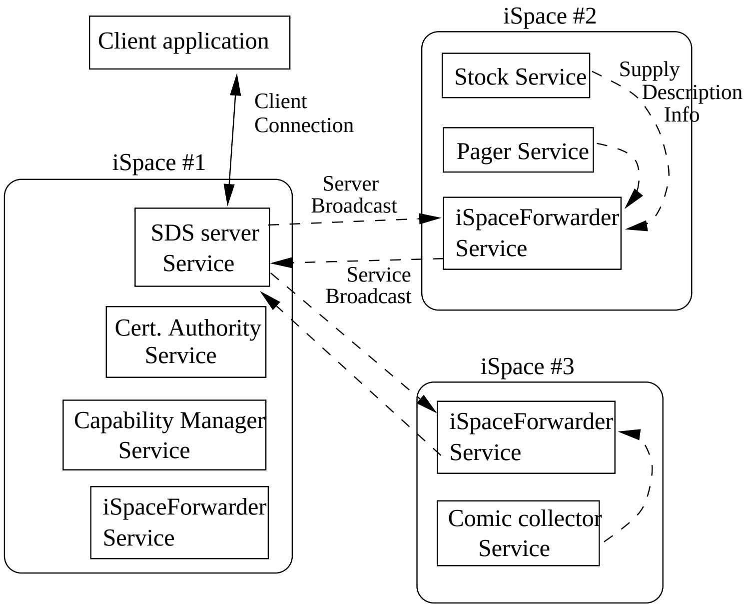

Figure 3: Implementation Structure

Component	*Public Key*		*Symmetric Key*	
Name	*Algorithm*	*Key Size (bits)*	*Algorithm*	*Key Size (bits)*
Certificates	DSA	1024	-	-
Authenticated RMI	Diffie Hellman	-	Triple DES/OFB	192
Secure Service Broadcasts	RSA/ECB	512	Triple DES/CBC	192
Auth. Server Broadcasts	DSA	1024	-	-

Table 1: Cryptographic Algorithm Choices

SDS server or services. In the future, we would like to make an SDS server certificate and a service certificate, so that we can better differentiate service and server certificates from user certificates.

The current status of the implementation is that it works. We have created services, securely advertised their descriptions to the SDS server and specified what capabilities are needed to learn about them. A client application is able to make requests on the SDS in a secure fashion using authenticate RMI, passing in the user's capabilities. The SDS searches for appropriate services, and returns only the descriptions that for which the user has capabilities.

Only a few modifications are necessary to make the implementation really usable. First, we have to implement a new interface for iSpace services, to provide hooks through which the service can specify whether or not it wants to be advertised to the SDS, and how to get a signing object for the service. These features are currently hard-coded, and need to be generalized. Also, we need improved hierarchical SDS server support. The general framework for the SDS still has not been changed over to the hierarchical approach, and the security side of the hierarchy has not been completed. Currently, we only have SDS hierarchical support through hard-coding.

7 Performance

Part of the goal of this project was to determine the effect additional security had on performance. Ideally, we would have liked to test out several implementations of the SDS, with incrementally degrees of security features installed.

Due to timing constraints, we did not get a chance to test the incremental degrees of security implementations. Instead, we did two tests. First, we dissected several of the key components of the security overhead, and ran timings on them to determine their individual effects on the system. Then, we took the system as a whole and tested overall performance, before and after the security modifications.

7.1 Component Tests

There are several easily profiled components of the overhead. They include the use of DSA signatures, both signing and verifying, and the use of RSA (EBC) encryption and decryption in the encrypted broadcasts. We measured the execution times for these routines and placed the results in Table 2. We should point out we used a 1 KB input size for encryption/decryption for both RSA and 3DES. Also, these times were gathered on a 350 Mhz Pentium II running a standard JDK 1.1.6 (no JIT) for Linux.

Name	*Time*	*Std. Dev*
DSA Signature	49.68 ms	3.4
DSA Verfication	51.98 ms	0.95
RSA Encryption	11.77 ms	-
RSA Decryption	120.58 ms	-
3DES Encryption	36.75 ms	-
3DES Decryption	35.94 ms	-

Table 2: Timings of cryptographic routines

There are some interesting numbers in this table. To start, we would have guessed that the DSA signature algorithm would be significantly faster in verification phase relative to the signing phase. But that was not the case. Another surprise came in the form of RSA packet encryption and decryption times. While both RSA and Triple DES encryption and decryption scaled with the size of the input, it's interesting to see the relative times taken in each algorithm for the 2 phases. We would have thought that Triple DES encryption/decryption would be significantly faster than both RSA encryption and decryption, but this was not the case. Triple DES is slower than RSA encryption and only three times faster than the decryption. Also, we were surprised by the cost of Triple DES in general. Obviously, this protocol is too expensive to be used to decrypt thousands of service advertisements quickly. We plan to move to a weaker algorithm, along with investigating faster Java tools including JITs.

7.2 Overall Performance

After the single component tests, we decided to directly compare the per request latency for the simple SDS versus the new secure SDS, and to see where that additional overhead is incurred. These measurements are shown in Table 3. We first measured the raw overhead of receiving an encrypted service broadcast packet. That is the additional latency imposed on the SDS server while it listens to service description broadcasts from the iSpaceForwarder/Service. We then measured per request latencies for null queries and document queries, in both the old and new SDS systems.

Again, the performance numbers are quite interesting. The overhead of receiving encrypted broad-

Name	*Time/Req.*	*Std.Dev.*
Decoding Service Annc.	43.41 ms	-
Null Query SDS	6.75 ms	2
Null Query Secure SDS	215.02 ms	0.7
Query SDS	540 ms	49.8
Query Secure SDS	2848.78 ms	2.2

Table 3: Timings of System

casts is rather high, which was expected, considering the results of the 3DES decryption times. The null queries show the raw increase in latency that is due to the newly implemented security features. While the increase is likely an additive constant rather than a multiplicative one, it is still quite significant, dwarfing the actual query overhead. What is really shocking, however, are the times from doing full queries on the SDS compared to the null queries. Comparing the null and full queries, we can see that the XML matcher overhead for searching the list of documents is dominant over all other overhead times, accounting for roughly 89% of the total time in a query. The difference is even more significant in the new SDS, where the XML overhead is now over 93% of the total latency. We believe the increase in the XML's overhead proportional to the total request time in the new SDS is due to two additional XMLmatcher operations per query in the new SDS, required to check the capabilities of every service. Until now, we had not realized the relative size of the XMLmatcher operations compared to the rest of the SDS. With such an overwhelming difference, it's certainly worth the time to optimize the XMLmatcher to its fullest in order to get the maximum throughput from the SDS. We do note, however, that this version of the XMLmatcher uses simple document to document comparison, and uses the worst case algorithm during matches. Hopefully, the new XML search engine performance will be much improved. Knowing where the latency is going, we can plan our future work to optimize accordingly.

8 Issues and Future Work

There are still several issues that we feel are unresolved in our current implementation of the secure SDS. To get the first version done, we went ahead with a quick design and implementation. In retrospect, we believe the following need to be examined more thoroughly:

- Online versus offline capability signing
- Self-administrated capabilities
- Optimization for local domain
- Use of long-lived authenticated RMI connections

The capability manager represents the largest component we need to reexamine. In order to accomplish its tasks, it has to be highly trusted. The CM does everything from distributing capabilities, maintaining the access control lists, and signing the capabilities. This may be too much trusted code, and could use a reorganization.

One of the major issues is that the capability signing happens online. When a service updates its ACL, the CM regenerates the necessary capabilities, signing them as it generates them. This means the same program that handles users requests has the CM's private key, which is a vulnerability. Instead, we would like to break the CM up into two parts, one which distributes the capabilities, and one that works offline to sign capabilities according to the ACL. This would create a waiting period between the time the ACL is modified and when a capability is available, but would be more secure.

Another issue with the capability manager is that it is centralized system: whenever a service wants to modify its own ACL, it must contact the domain's single CM and ask it to update the list. We would rather move to a self-administered model if possible. Instead of having one central trusted CM, we could set up a system so that the individual service's could efficiently control their own ACL. We had trouble designing such a system, so implemented the simple approach.

The current design of the secure SDS is optimized for the local area case, which is not the desired model. Currently, the SDS relies heavily on the CA and CM, entities that are not yet scalable to large distributed systems. For example, MIT's certificate authority will probably be managed separately from Berkeley's CA. Hence, the SDS creates islands of security, rather than a well connected system of secure components. One solution to solve this is to design a scalable hierarchy of certificate authorities.

One final issue is the whether or not authenticated RMI is appropriate for client to server communication. Clearly, we do need the end to end authentication and encryption authenticated RMI provides, but we believe that the current implementation performs poorly for the sporadic client requests. The poor performance would be due to the huge latency incurred every time a request goes through and a public key handshake is made. Also, as we found in Section 7, Triple DES might be too computationally intensive for Java to do quickly. We would like to investigate more efficient forms of authenticated RMI for cases of infrequent requests.

9 Conclusion

In this project, we designed and implemented a security system for the SDS. This involved investigating what the various security threats were and which ones were possible to protect against. We found that considerable effort had to be put in guarding the communication channels between the components, since this was the easy method of attack. Therefore, we used various public key and symmetric key encryption algorithms to encrypt and validate all the communications. In addition to the standard cipher algorithms, we had to create our own protocols to deal with broadcasts. To support these algorithms, we had to increase the number of SDS components in order to include such things as certificate authorities and capability managers.

We also decided it was necessary to provide authentication on all the components of the SDS system, including the individual service descriptions that are returned to the user. This design enables each component to decide which principals it trusts, and act accordingly. It also pushes policy designs all the way to the end user: the user can determine for himself if he trusts a particular service description since the SDS server pushes the description, the certificate, and the signature all the way to the user.

The authentication built into the system also allows us to provide access control on the dissemination of the service information. We have built a hybrid certificate/ACL system through which individual services can specify which users are allowed to access their information.

Our current implement does work, but certainly needs to be reexamined. As we analyzed its performance, we found that some of the encryption routines were larger bottlenecks than we thought. We need to revisit these parts of the program and change the encryption routines to get faster speeds. We should also investigate what the affect of using JITs and native methods. Currently, our system does not scale as well as it should, but with a little work on the implementation, we believe we can fix this.

In the future, we will also be investigating some of the larger issues, such as redesigning the capability manager and providing better wide area support for the SDS server. The current design is a good starting point for providing a secure directory service, but later versions will build on it through experiences gained from this implementation, enabling even more advanced directory services.

References

[1] Microsoft Corporation. Active directory technical summary. In *Strategic White Paper, Professional Developers Conference*, San Diego, CA, September 1997.

[2] Timothy A. Howes. The lightweight directory access protocol: X.500 lite. Technical Report 95-8, Center for Information Technology Integration, U. Mich., July 1995.

[3] Novell. NDS for NT Technical White Paper. Technical report, Novell, September 1998. http://www.novell.com/whitepapers/nds4nt/twhitepaper.html.

[4] Novell. NDS White Paper. Technical report, Novell, September 1998. http://www.novell.com/whitepapers/nds/whitepaper.html.

[5] Charles Perkins. SLP White Paper. http://playground.sun.com/srvloc/slp_white_paper.html, June 1998.