WebDAVA: An Administrator-Free Approach To Web File-Sharing

Alexander Levine
Alexander.Levine@drexel.edu
Drexel University

Vassilis Prevelakis
vp@drexel.edu
Drexel University

John Ioannidis
ji@research.att.com
AT&T Labs – Research

Sotiris Ioannidis
sotiris@dsl.cis.upenn.edu
University of Pennsylvania

Angelos D. Keromytis
angelos@cs.columbia.edu
Columbia University

Abstract

Collaboration over the Internet depends on the ability of the members of a group to exchange data in a secure yet unobtrusive manner. WebDAVA is a system that allows users to define their own access-control policies to network resources that they control, enabling secure data sharing within an enterprise. Our design allows users to selectively *give fine-grain access to their resources without involving their system administrators. We accomplish this by using authorization credentials that define the users' privileges.*

Our prototype implements a file-sharing service, where users maintain sensitive-information folders and can allow others to access parts of these. Clients interact with the server over HTTP via a Java applet that transparently handles credential management. This mechanism allows users to share information with users not a priori known to the system, enabling administrator-free management.

This work was supported by DARPA and NSF under Contracts F39502-99-1-0512-MOD P0001 and CCR-TC-0208972 respectively.

1 Introduction

The Web has become the standard mechanism by which most users access networked information resources today. Despite the high sophistication of available Web services, some common usage cases remain unnecessarily complex. A typical example of such a case is when a user, Alice, wants to share some files, *e.g.,* drafts of a legal agreement under preparation, with another user, Bob, who is not part of the same enterprise or division (and thus does not have access to the Alice's file server). Bob's only way of accessing these documents is "over the Web." Alice would typically place the files in a directory on her web server where she has local access (meaning, she is an authenticated user of the system), and then give Bob a URL. If she does not want anyone else to see the drafts, Alice has to set up a way for the server to authenticate Bob before granting him access. Depending on how this is done, Bob may or may not be able to give the URL to some of *his* colleagues (that is, *delegate* access). If Alice wants to give access to different subsets of her work items to different users, she has to resort to some rather complicated hacks, or create individual directories with copies of different subsets of the documents. A similar example would be that of a small number of users from different organizations collaborating on a joint project (*e.g.,* researchers from various institutions co-authoring a paper). Traditional access-control mechanisms are ill-suited for a world-wide network of millions of computers each with the ability to act as a resource server to disparate clients who cannot all be authenticated with a uniform mechanism.

We examine the dynamics of use of a credential-based capability mechanism that provides file-access services to an extended user base. In the traditional approach, Bob must register with Alice's server and inform her of his new user-id, so that she can give him access to the files. Thus, the system must perform two separate operations in order to allow Bob access to Alice's files. First, it must authenticate Bob (this is why advance registration is needed) and then determine whether Bob is authorized to access the files.

Significant effort has gone into optimizing this process. Kerberos allows a collection of servers to use a centralized database of users for authentication purposes. X.509 certificates were designed with a similar objective in mind. Certificates allow a trusted third party, called the Certification Authority (CA), to vouch for the binding of the owner's name to a key. This is done by having the CA digitally sign the certificate. These arrangements appear to work satisfactorily as long as we constrain ourselves to a single administrative domain. If we try to combine different domains via cross-certification, we are likely to "turn the hierarchy of trust into the spaghetti of doubt" [3]. The importance of allowing geographically-dispersed users to access distributed resources is identified in [8]. To address the problem, they employ an access-control mechanism that uses digitally-signed certificates to enforce their access policy. However, the servers that provide the resources remain burdened with locating attribute and condition certificates.

Our system uses credentials that directly grant access privileges. The credentials are based on the KeyNote trust-management system [2], allowing us to use the flexibility of the policy definition language within the credentials to specify additional access constraints (*e.g.,* access only on weekends, outside office hours, *etc.*). Group membership is implemented by issuing separate credentials to different members of the group. Furthermore, a single credential can authorize access to collections of files, easing credential management. Perhaps the most significant benefit of using KeyNote credentials is delegation: rather than require Bob to register with the site before accessing Alice's files, the system allows Alice to delegate a subset of her access rights (*e.g.,* read-only access) to Bob by using her private key to sign a credential granting Bob's public key the privileges.

The elegance of this approach is that the credential itself contains information that is meaningless outside the server: the credential need only contain the file handle, Alice's and Bob's public keys and the digital signature. As long as Alice is satisfied that the public key she received actually does belong to Bob, there is no need for third parties (*e.g.,* Certification Authorities) to be involved in the transaction. Acquiring the credential does not give a third party access, since the request for the file will have to be signed by Bob.

2 WebDAVA Design

We now consider our requirements for a system where an authorized user can grant access to external users. First, the system must be able to handle large numbers of files or services, and an even larger number of users accessing these. Second, the system must maintain as little additional state as possible, apart from the actual data stored. Third, administrator involvement in privilege management should be minimal. Local users must be able to directly authorize access to files by external users, rather than having to create local accounts. Furthermore, the file access conditions must be flexible and extensible. Additionally, the administrator must be able to specify the default access policies for the entire site. This policy must be obeyed by the system, regardless of individual users' settings. Finally, the authorization mechanism must allow use of different authentication and traffic-protection mechanisms, *e.g.,* TLS or IPsec.

Existing systems have several shortcomings when used for information-sharing tasks. First, traditional user authentication implies that the user is known to the system, before file requests can be processed. Second, file and directory permissions are concepts inherited from multi-user operating systems. Sharing is achieved by either account sharing (which is ill-advised, as it defeats accountability) or through the use of group access permissions. Such permissions lack flexibility and fine granularity, and, perhaps most importantly, extensibility: there is no way of adding new permission mechanisms if the existing ones prove inadequate.

The leading design constraint was for the system to be usable from most commonly available hardware/software platforms. This led naturally to our two major design decisions: to use HTTP as the transfer protocol, and to use a Java applet as the client interface. A user using any popular browser on any platform can use our system.

2.1 The KeyNote Trust-Management System

To express access rights and the diverse conditions under which these are granted, we need some form of policy definition language. We use the KeyNote trust management system for this purpose, because of our familiarity with it. The basic service provided by KeyNote is *compliance checking;* that is, checking whether a proposed action conforms to policy. Actions are specified as a set of name-value pairs, called the *action attribute set.* Policies are written in the KeyNote assertion language and either accept or reject action attribute sets presented to the policy engine (non-binary results are also possible, but we do not consider them here). Policies can be broken up and distributed as credentials, which are signed assertions that can be sent over a network and to which a local policy can refer when making a decision. The credential mechanism allows for arbitrarily complex graphs of trust, in which credentials signed by several entities are considered when authorizing actions. Group-based access is handled by issuing the appropriate credentials to all the group members. Furthermore, subgroups can be created at any level in the hierarchy.

```
Authorizer: ADMINISTRATOR'S_PUBLIC_KEY
Licensees: ALICE'S_PUBLIC_KEY
Conditions: (AppDomain == "WebServer") &&
         (File_UID == "666240") -> "RWX";
Comment:  Owner (Alice) can do anything!
          UID:666240
signature: SIGNED_BY_ADMIN'S_PRIVATE_KEY
```

Figure 1. Credential by the administrator giving Alice access to a file. Public keys and signatures are replaced with symbolic names.

Figures 1 and 2 show two credentials forming part of a delegation chain in our system. User Alice is granted access to a particular file through a credential issued by the administrator. User Bob is then granted read-only access (by

```
Authorizer: ALICE'S_PUBLIC_KEY
Licensees: BOB'S_PUBLIC_KEY
Conditions: (AppDomain == "WebServer") &&
        (localtime >= "20021115000001") &&
        (localtime <= "20021115235959") &&
        (method == "GET") &&
        (File_UID == "666240") -> "RWX";
Comment:  UID:666240
Signature: SIGNED_BY_ALICE'S_PRIVATE_KEY
```

Figure 2. Credential from Alice delegating read-only access to a file to Bob, for one day. Again, public keys and signatures are replaced with symbolic names.

Alice) to access the same file during November 15, 2002. KeyNote will allow this delegation because the authority granted to Bob is a subset of the authority granted to Alice. Notice that users are identified only by their public keys.

The advantage of using this system is that we no longer need to have *a priori* knowledge of the user base. Thus, the system does not need to store information about every person or entity that may need to retrieve a file. We also provide our users with the ability to propagate access to the files by passing on (delegating) their rights to other users. In this way, users pass credentials rather than passwords, thus allowing the system to associate access requests with keys and also to be able to reconstruct the authorization path from the administrator to the user making the request (and thus grant access). The system may not know that Bob is trying to get at a file, but it can log that key B (Bob's key) was used and that key A (Alice's key) authorized the operation. Logging such information creates a clear audit trail for any access request, which can be used by the administrator to validate that the appropriate usage policy is being followed. Note that a user can at most pass on the privileges she holds (there is no *rights amplification*); furthermore, the user can delegate only a subset of her privileges (*e.g.,* access to only a specific file in the user's directory).

2.2 Saving Files

The HTTP PUT method is used to upload a file to the server. When a client needs to upload a file to the server, a PUT request must be sent. The PUT header includes the length of the file, the URI of the file, and the user's public key. If the URI refers to a file that is already on the server, then the server requests a valid credential from the user that includes proper access rights (this situation is discussed in Section 2.4). If all of the information in the PUT request is valid, then the server returns a `100 Continue` response. Upon receiving this message, the client can upload the file to the server. If the information is invalid, then the server returns a `401 Unauthorized` response to the client, and the client must either try again or abort the transfer.

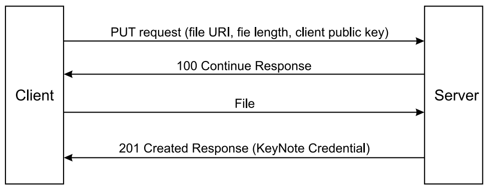
PUT request (file URI, fie length, client public key)
100 Continue Response
Client
File
Server
201 Created Response (KeyNote Credential)

Figure 3. Uploading a file to the server.

After a file has been successfully uploaded to the server, the server creates a KeyNote credential granting full-access privileges for that file to the client. A unique identifier (UID) for that file is associated with the file; this UID is also included in the credential. The server then returns a `201 Created` message to the client. This response includes the credential in the *ETag* portion of the message. While the *ETag* header is generally a caching-related header, it is used here as it is the only way to send back information about a resource in the response header. The receipt of the `201 Created` response completes the transaction. The software on the client must then extract the credential and store it. This credential will be required for future access to the file. Figures 3 and 4 show successful and unsuccessful file upload exchanges between a browser and a server.

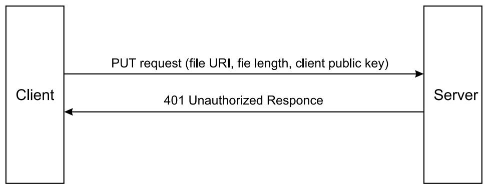
PUT request (file URI, fie length, client public key)
Client
401 Unauthorized Responce
Server

Figure 4. Failed attempt to upload a file.

2.3 The Challenge-Response Scheme

The system employs a challenge-response scheme to protect against replay attacks. When a user attempts to access a secure file on the WebDAVA server, the server sends back a `401 Unauthorized` response containing a challenge. Included in the *WWW-Authenticate* header is the name of the authentication method being used, a nonce, and the server's public key:

```
WWW-Authenticate: Keynote nonce=<nonce> \\
    server_key=<server_key>\r\n
```

The client then prepares a response that includes the nonce from the challenge message. The client first creates a new *nonce credential* delegating trust to the server key for the desired action (GET, PUT, *etc.*). The nonce credential is signed with the user's private key. The Authorization header of the response includes the user's public key, the original file credential, and the newly created nonce credential:

```
Authorization: client_key=<client_key> \\
   credential=<original_credential>\n\n \\
   <nonce_credential>\r\n
```

When the server receives the message, it verifies that all the necessary information is included. If not, a `401 Unauthorized` message is sent. If all the necessary information is there, the server passes all supplied credentials to the KeyNote policy engine, along with the action name/value pairs of (AppDomain, "*WebServer*"), (method, "<*method*>"), (file, "<*File_UID*>"), and (nonce, "<*nonce*>"), as well as the server's local policy, and the user's public key as the *action authorizer.* <*File_UID*> refers to the UID of the requested file and <*method*> to the HTTP method being called (GET, PUT, DELETE). If KeyNote approves the request, the file is sent to the client. Otherwise, a `403 Forbidden` message is returned. Figures 5 and 6 show successful and unsuccessful challenge-response message exchanges for a GET request.

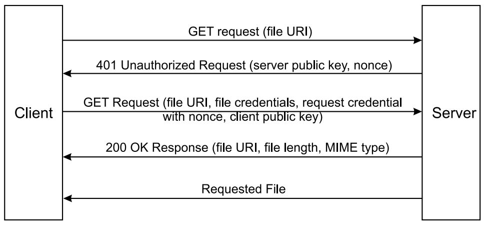
Client
Server
GET request (file URI)
401 Unauthorized Request (server public key, nonce)
GET Request (file URI, file credentials, request credential with nonce, client public key)
200 OK Response (file URI, file length, MIME type)
Requested File

Figure 5. Message exchange in a successful GET request.

2.4 Downloading, Editing, and Deleting Files

Downloading a file from the web server is done via the HTTP GET method. Files on the server are identified via the UID assigned when they are first uploaded to the server. First, the client sends a GET request for the file using the UID as the file URI. The server then attempts to validate the user's request via the challenge-response scheme mentioned previously. If the credential verification is successful, a `200 OK` response is generated and the file is transmitted.

Modifying a stored file is done by overwriting it using the HTTP PUT method, as described previously in Section 2.2. Deleting a file is done by saving an empty file; the server notices that the file is empty and removes it. If the user really wants an empty file, she must create one.

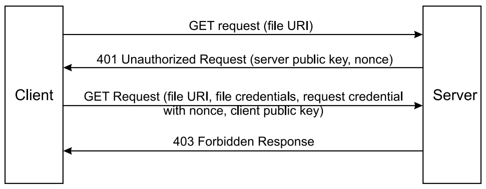
Client
Server
GET request (file URI)
401 Unauthorized Request (server public key, nonce)
GET Request (file URI, file credentials, request credential with nonce, client public key)
403 Forbidden Response

Figure 6. Message exchange in a failed GET request.

3 Prototype

Our prototype implementation of WebDAVA allows users to create the delegation credentials and send them to other users. Moreover, the software allows users to accept and integrate within their workspace credentials sent to them by other users. The server, written in C++, implements the initial uploading of files and their subsequent downloading, based on receipt of proper credentials.

The server puts all uploaded files in a directory called "storage". When the server receives a file, it creates a UID to represent it using the *uuid_generate* API that is part of the Linux *E2fsprogs* ext2 filesystem library. The file is then stored on the server using this UID. A client trying to retrieve a file must sends a special file access block (FAB) that contains the file UID along with additional information (note that the FAB in Figure 7 is that of the file owner, as it does not contain any delegation credentials).

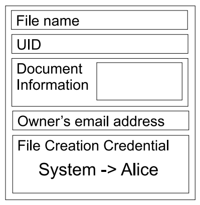

Figure 7. FAB with a single credential.

For portability, the client was implemented as a signed Java applet. Clients can upload a file to the server, for which they will receive a file access credential in return (contained inside the file access block). They can then use this credential to retrieve the file from the server, or they can pass it, along with a delegation credential they create, to another user with a valid set of keys. When Alice wants to download a file from the server, she sends the credential representing the file. If the credential is valid, the server then sends the file to the client. The applet can search for the appropriate credential using the UID as a key in its credential database.

When Alice wants to allow another user, Charlie, to access a file stored on the WebDAVA server, she needs to retrieve Charlie's public key, construct the credential delegating access to Charlie's key, and then send this credential along with her own access credentials to Charlie. Charlie must import these credentials and use them to access the file. While Alice may use any mechanism to get Charlie's key, we provide a key-server that stores the keys of the various users to simplify credential management.

To transfer a credential to Charlie, Alice simply selects the credential and enters Charlie's email address. When users run the Java applet for the first time, their email addresses and public keys are passed to the key-server. The key-server stores a mapping of email address/public key pairs. If Charlie is not registered, Alice must obtain his key through other means. If Charlie is registered with the key-server, Alice creates a credential transferring authority to Charlie's key and appends it to a copy of the credentials she already has for that file. The client then creates an email message using these credentials and sends it to Charlie. Charlie can use these credentials to download the file from the server. Figure 8 shows Charlie's FAB.

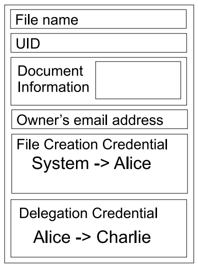

Figure 8. FAB with the file access credential and one delegation credential.

4 Evaluation

Performance Analysis The prototype implementation was used to determine the costs of the initial message exchange and credential verification. We used a Pentium 866 Mhz machine running Red Hat Linux 7.2 (server) and an IBM X.21 laptop running Windows 2000 (client). We measured the average elapsed time for repeatedly processing a GET request for the same file. We performed this operation with the access-control mechanism disabled, using only the original credential delegated from the server to the client who had initially uploaded the file, and with an increasing number of delegation credentials. Figures 9 and 10 show the results of using a local (on the same machine as the server) and a remote (over the network) client respectively.

The time was measured in system-clock ticks, with a res-

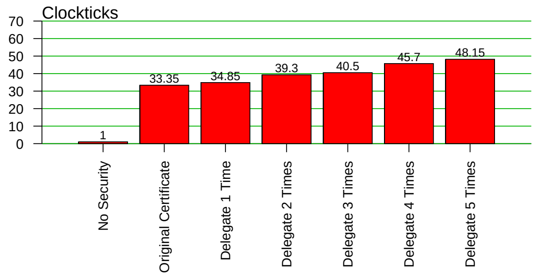

Figure 9. Time to handle a GET request, client and server on the same host.

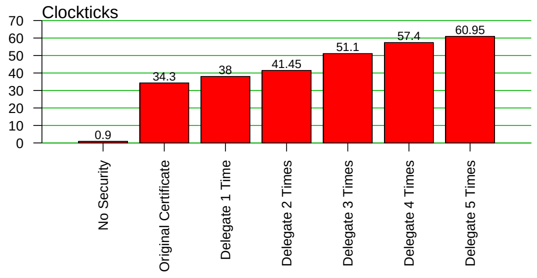

Figure 10. Time to handle a GET request, client connecting over the network.

olution of 10^{-6} seconds. Increasing the number of delegations slightly increases request-processing time. Additional tests showed that the policy evaluation itself required between 1 and 2 clock ticks, even after 5 delegations. The cost of retrieving the file from disk is minimal since, after the first request, it resides entirely in the operating system's cache. Thus, the increase in latency can be directly attributed to the overhead of the security mechanism and the network-stack (present even in the local-access case).

Security Analysis As the performance analysis indicates, the challenge-response phase incurs a significant performance penalty. This cost, however, must be weighed against the benefits gained from a security perspective. The design primarily addresses replay attacks, whereby an eavesdropper records an interaction between a user and the server, and sends the same packets at a later time. The existence of the nonce in the server's challenge ensures that each exchange is unique (assuming a good source of entropy).

Revocation One of the major problems in capability-based systems is revocation. In our system, each file has an associated file that stores information about revoked credentials. Before the KeyNote compliance checker is consulted, the file-access credential is checked against the corresponding revocation file to make sure it is still valid. If it is found in the revocation file, the compliance checker does not consider it in the policy evaluation. In this way, the revocation file acts as a Certificate Revocation List. The fact that the file is located in one server makes this simple approach acceptable. The only remaining problem, of how revoked credentials are added to the file, is solved by having the issuer of the credential to be revoked attempt to upload the credential in the revocation file (using the PUT method), using our protocol to authenticate. The server intercepts the PUT method, verifies that the key used to authenticate the request is the same as the key that signed the credential, and adds the hash of the credential to the revocation file.

Other Services Web servers are increasingly used as front-ends to databases and other services. Clients issue parameterized POST requests that are passed to programs spawned by the web server in response (*e.g.,* CGI scripts). This model has proved very successful, and various standardization efforts use it for remote method invocation and as an inter-process communication mechanism. Authentication using simple public key certificates does not allow for fine-grain access control at the parameter level. Since KeyNote allows for arbitrarily complex conditions to be expressed as part of the policy, we can pass the inputs from a POST request to the policy engine, as part of the access-control decision-making process. Access to services can then be arbitrarily fine-grained using the WebDAVA model.

5 Related Work

Although network file systems such as NFS [6] and AFS [4] are the most popular mechanisms for sharing files in tightly-administered domains, crossing administrative boundaries creates numerous problems. Similar issues arise with SFS [5]. Although its *self-certifying pathnames* (file names that effectively contain the appropriate remote server's public key) allows clients to contact previously-unknown servers, clients must still use X.509 certificates or a shared-secret protocol to authenticate to the server.

WebFS [9, 1] implements a network file system using user-level HTTP servers to transfer data, along with a kernel module that implements the file system. It uses X.509 certificates to identify users and transfer privileges. Each file is associated with an ACL that enumerates which users have read, write, or execute permission on the file. By avoiding traditional ACLs altogether, we remove the need for maintaining, and controlling access to, a separate administrative interface for managing the ACLs.

Capafs [7] provides a weak security system for file access by encoding the access capabilities in the file name. Apart from producing complex file names, knowledge of the file name allows access to the file, making it a password.

6 Conclusions

Our motivation for this work was the inappropriateness of traditional access control mechanisms for use on the web. Their main failing is that policy is embedded in the operating or file system (*e.g.,* directory permissions), so that producing variations to suit changing needs is difficult. Therefore, while it quite possible to implement strategies like ours using traditional mechanisms, the result would be inflexible and non-intuitive, thus alienating users. Examples abound where user apathy or incomprehension have caused serious security breaches. By creating a portable application that allows any user to utilize the WebDAVA secure web server we have also shown that simple solutions employing novel techniques are often more effective than the older monolithic mechanisms. Although we have concentrated on the Web model, nothing prevents the same techniques from being used in a distributed file system or in a resource allocation mechanism as part of an operating system. We will continue our research along these lines and apply the WebDAVA model in other domains.

References

[1] E. Belani, A. Vahdat, T. Anderson, and M. Dahlin. The CRISIS Wide Area Security Architecture. In *Proceedings of the USENIX Security Symposium*, pages 15–30, August 1998.

[2] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The KeyNote Trust Management System Version 2. RFC 2704, September 1999.

[3] P. Gutmann. PKI: It's Not Dead, Just Resting. *IEEE Computer Magazine*, 35(8):41–49, August 2002.

[4] J. H. Howard et al. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[5] D. Mazieres, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 124–139, 1999.

[6] R. Sandberg et al. Design and Implementation of the Sun Network File System. In *Proceedings of the Summer USENIX Conference*, June 1985.

[7] J. Regan and C. Jensen. Capability File Names: Separating Authorization from User Management in an Internet File System. In *Proceedings of the USENIX Security Symposium*, pages 211–233, August 2001.

[8] M. Thompson, W. Johnston, S. Mudumbai, G. Hoo, K. Jackson, and A. Essiari. Certificate-based access control for widely distributed resources. In *Proceedings of the USENIX Security Symposium*, pages 215–228, August 1999.

[9] A. Vahdat. *Operating System Services for Wide-Area Applications*. PhD thesis, University of California, Berkeley, December 1998.