Secure Access to Data Over the Internet

Eric Bina
Rob McCool
Natl. Center for Supercomputing Applns.
Univ. of Illinois, Champaign, IL 61820
{ebina,robm}@ncsa.uiuc.edu

Vicki Jones
Marianne Winslett
Department of Computer Science
Univ. of Illinois, Champaign, IL 61820
{vjonwinslett}@cs.uiuc.edu

Abstract

Security issues are the major roadblock currently preventing greater access to databases over the Internet. In this paper we propose a framework for secure access to databases by a wide audience over the Internet, using World Wide Web information servers, a modified version of the NCSA Mosaic Internet browser, and role wrappers (software modules that assign roles to incoming database requests, based on the credentials submitted with the request). We also describe an application embodying this framework.

1 Introduction

Traditionally, databases have relied largely on the underlying operating systems and physical environments for security. This approach works fairly well when the set of users authorized to access a particular database is relatively small and carefully controlled. As databases become accessible over the network, however, security becomes a daunting concern. With today's technology one can easily encapsulate a database and make it available over the Internet, using NCSA Mosaic, the Web, and back-end databases. However, relatively few databases are accessible in this manner, due to lack of facilities to authenticate incoming requests and lack of privacy for requests and responses. Currently, a server has no way of knowing exactly who has sent it a request, and no help in deciding what database authorizations (what role) the request should be given.

We acknowledge support from NSF under a PYI grant and from ARPA under the POB program. Our thanks also to Roy Campbell for his idea of a personal security agent.

2 Issues in Internet data access

Perhaps the most obvious approach to connecting a database to the Internet is to have the DBMS server listen on a network port for requests. However, potential new clients will have difficulty learning about the new service and using it, and all security needs must be met by the DBMS and clients themselves.

A second approach is to add a database-specific front-end program to encapsulate the database and handle requests. The encapsulating script (script for short) can compensate for the poor security facilities available within the DBMS by providing a fixed set of services to clients. However, publicity and accessibility are still problems; secure scripts are very difficult to write correctly; and usability will suffer if users have to learn a different way to interact with each database.

By putting the script on or in a general-purpose information server, such as a Web server, the database can be made widely known and easily accessible. A user can request an HTML (HyperText Markup Language) document containing a form from a Web server, fill out the form (implicitly specifying a query), and return it. The server can then invoke a form-specific script which processes the form, encapsulating and accessing back-end databases as needed. The output of the script will be displayed to the user.

Standard browsers for the Web, such as the very popular NCSA Mosaic [1], guarantee that users can access many different databases and other resources using a single consistent form-based interface. Due to Mosaic's and the Web's wide acceptance and universality, and their good support for form-based access, we believe that they are currently the best candidates for interfaces to widely-available databases on the Internet.

1Many of the discussions in this paper are severely abbreviated due to space constraints, and we refer the interested reader to [5] for a full presentation.
2Since two of us are Mosaic developers, we may be somewhat biased on this point.
ternet. Still, Mosaic and the Web currently have some significant drawbacks for database use; see [5] for details.

As the first step in choosing a security protocol for Internet communication (e.g., between a Web server and a Mosaic client), one must determine the security goals of the system: what information should be protected, and what type of protection is needed—privacy, authentication, or both? For request, response, or both? If privacy is a concern but not authentication, then a protocol which provides encryption but does not authenticate users is appropriate. If authentication is required, then a protocol with digital signatures and asymmetric key encryption is probably needed. Other issues: Are there guarantees (e.g., formal analyses) that the protocol meets the security requirements? Is the overhead of the protocol acceptable for the application? Will it mesh well with the protocol currently used between users and servers? Is the protocol available on all users' platforms? Can it handle a large number of users at far-flung locations? How will clients and servers get the needed software and keys to run the protocol? Will import or export limitations eliminate potential users?

Given an authenticated request, how can a script determine the appropriate authorizations for a client, possibly one not previously known to the server? Under role-based security, a DBMS request is assigned one or more roles, which determine the access authorizations for the request. We anticipate that DBMSes will offer more powerful support for roles in the future than they do today.

For some applications, the user's identity may be all that is needed to assign a request to a role. For others, a declaration of identity may be insufficient. For example, suppose Mary Smith falls ill on vacation, and the attending physician wishes to access her medical records over the Internet. A declaration of the doctor's identity should be insufficient to convince Mary's home-town clinic to release her records; instead, at the minimum a proof that the requester is a medical doctor is in order.

To obtain such proof, the server-side software might query a trusted credential server. More likely, however, the server would prefer to place the time-consuming burden of proof on the client, demanding that the client produce some particular credentials that it will use to assign the current request to a role. As an advantage, the client can choose which of its credentials it is willing to divulge to the server.

On the server side, who should be responsible for processing credentials and selecting roles? A custom implementation of a Web server could have the necessary software built into it. If an off-the-shelf Web server is used, then the role resolution software must either be built into the script or reside in a separate module that we call a role wrapper. The Web server decides whether the user is authorized to access this server, while the role wrapper determines what role(s) the user is authorized to play within a particular requested database application.

Role wrappers extract credentials from a request for service, determine which are useful for the request, assign a role or roles to the request, and hand it to a script for further processing. The credentials presented by a user may not exactly match a unique role, and the role wrapper must mediate between the credentials offered by the requester and those needed by the service. Given a sufficiently powerful DBMS, the role information will allow the rest of the processing to be handled by ordinary code with no special attention given to security matters.

3 An Architecture for Database Access Over the Internet

Figure 1 presents our draft architecture using Web servers to encapsulate a database on the Internet. A secure version of the HTTP protocol used by all Web servers [2, 4] is used for client-server communication, with a 'personal security agent' managing the user's credentials, determining the protocols required for services, and handling security concerns for Mosaic.

The secure HTTP protocol allows the Web server to determine what security protocol is being used for each request. The server passes the request to a protocol-
specific module for decryption and authentication. If the authentication is successful, the request and the 'security identity' of the requester (the name by which it is known to the rest of the world) go to the role wrapper for that request. The wrapper uses the requester's identity and any additional credentials supplied with the request to determine the appropriate role(s) for the request, then passes the request to the appropriate script. The script's output goes back to the Web server, which repackages it by calling the appropriate protocol module, and ships the resulting message to the requester. The personal security agent then extracts the response using the appropriate protocol, and presents the results to the user. In the remainder of this section, we describe how we implemented each of the components of the architecture.

We modified the HTTP protocol, HTML form language, and Mosaic to allow the use of authentication protocols, and the new versions were released as version 2.3 of Mosaic for X. The new HTTP includes an optional tag called 'encytype' on each message sent across the network. The tag allows a message sender to name the authentication protocol that should be used for that message.

The new HTML allows a form to declare that it should be returned to its server using a particular security protocol, and to declare the server's public key and security identity. This prevents the form from accidentally being shipped unencrypted, e.g., leading to theft of credit card numbers.3

The new Mosaic adds some support for personal security agent fundamentals, by allowing the user to select an authentication protocol before sending a request, thus protecting requests other than fill-out forms (see Figure 2). However, the network security protocol implementations available today encrypt the request using the identity and public key of the server, which may not be known to the client.4

Mosaic 2.3 knows two implementations of security protocols: RIPEM and PGP. Both are implementations of the PEM protocol and use a combination of symmetric and asymmetric encryption to authenticate messages and guarantee privacy.

The installation procedure for Mosaic 2.3 is relatively complex if its security features are to be used. Suppose that Mary Smith wishes to use the new Mosaic features. She obtains the binary from ftp.ncsa.uiuc.edu, and procures an executable for RIPEM and/or PGP. She chooses a security identity to be included in requests to secure Web servers. She generates a public/private key pair for that identity, and somehow informs the secure servers of her public key, so that they will be able to read messages that she sends to them. (Our architecture does not address issues of key distribution.) Her copy of Mosaic must be told where RIPEM and/or PGP reside, and be given the parameters necessary to invoke them.

We modified the Web server code (now version 1.1 at ftp.ncsa.uiuc.edu) so that selected directories on a server can require an authentication protocol for access. The server must remember the security identity of the clients authorized to access each such 'protected' directory. The server must also have a security identity and have access to the selected protocol(s). If a Web client requests a protected document without using the appropriate protocol, the server asks the client to use that protocol. The Mosaic 2.3 client will then reissue the request using the appropriate protocol, the public key of the server, and the user's private key. If the request authenticates, and the sender is authorized for access, the requested document is processed using the appropriate protocol and returned.

Unfortunately, RIPEM and PGP will not pass arbitrary credentials back to the Web server, even though the PEM protocol allows their inclusion in a message. Thus the user's security identity is the only credential our role wrappers can currently use. Since the roles in our application are so simple ('student' and 'instructor') and our database (ONTOS) does not support roles, our initial implementation merged the role wrapper and script into a single software module. In this experimental role wrapper, we used the user's authenticated identity to control the flow of execution of the database functions.

4 StudentAliases: An Example

Our testbed secure application is the selection of student aliases for posting grades. (To preserve anonymity, our university does not allow posting of student grades with identifying information.) Essentially all course information for the database course in spring 1994 was on a Web server. We placed student information in an object-oriented database (ONTOS), created an alias-changing script, and used RIPEM as an authentication program (see Figure 2) both for requests and responses.

3 Note the importance of having a secure mechanism for distribution of public keys (not addressed by our architecture): a con artist could collect the credit card numbers of unsuspecting individuals by duplicating the order form but specifying his or her own Web server.

4 RIPEM resolves this by examining the addressee's fingerprint, a deplorably insecure arrangement.
Figure 2: Architecture for Secure Student Database Access

We needed authenticated private requests for alias changes. We created an HTML form to request an alias change (see http://sparc30.cs.uiuc.edu:8004/winalett/jones/aliasnote.html), and placed its script in a protected directory on the server. We told the server the security identities of the the instructors and the thirty-five students in the course. We distributed public and private keys to students using ordinary email—secure, but satisfactory for our application. The students installed their keys and set up their environment so that the secure version of Mosaic would work correctly. All went well.

We collected performance figures for access to protected and ordinary documents on the course server, and found that authentication incurred a significant penalty.\(^5\) If the client requests the 800-byte alias-changing form from an unprotected directory, the response time is 0.4 sec. (includes all but display time). If the same document is protected, response time is 4 sec. Unfortunately, response time grows linearly with the message lengths; for example, a request for a 780 KB document required 0.5 sec. if unprotected and 32 sec. if protected. While these response times are acceptable for the small documents in our application, they will not do for all applications.

5 Conclusions

We proposed an architecture for connecting databases to the Internet and providing easy access to a world-wide population of potential clients, along with easy advertising of the existence of the service. The architecture also includes facilities to ensure privacy of requests and responses, to authenticate clients and servers, and for a ‘personal security agent’ to manage security-related information on the client. We embodied our architecture (minus the personal security agent) in modified versions of Mosaic and Web servers, and tested it in a student grade information database application with approximately 40 users. While the application was a success, many major open issues remain: (1) The personal security agent should play an important role in every the simplest secure database applications. The agent must find server identities and public keys given a URL, determine the protocol for the URL, and invoke it correctly. (2) Credential servers are needed to serve as on-line repositories for unforgeable credentials associating security identities with public keys. Agents can access these servers to find Web server public keys. (3) More formal analysis of proposed security protocols is needed. For example, RIPEM has a known weakness that a clever adversary might exploit. (4) The interfaces between the user, the personal security agent, and Mosaic need refinement to make it easy for users to specify which credentials should be included with a particular request. Also, a simple mechanism is needed to determine whether a particular request should use any authentication protocol at all; a special type of URL has been included in the new ‘SHHTTP’ proposal as a solution.

References


[5] The full version of this paper is available at URL http://bunny.cs.uiuc.edu/CDR/WinalettGroup/pubs/secureDBAccess.ps

---

\(^5\) Our measurements were taken on a lightly loaded Sparc 10 with 32 MB of memory and a local disk as the client, and a Sparc IPX with 32 MB of memory and a remote disk as the server.