Practical Security in Large Scale Object Oriented Databases

Andrew Hanushevsky[1]

[1] Stanford Linear Accelerator Center, USA

Abstract

The BaBar experiment at the Stanford Linear Accelerator Center is designed to perform a high precision investigation of the decays of the B-meson produced from electron-positron interactions. The experiment, started in April 1999, will generate approximately 200TB/year of data for 10 years. All of the data will reside in Objectivity databases accessible via the Advanced Mult-threaded Server (AMS). While most of the data is accessed via on-site computers, some is remotely available to scientists at other installations. In either case, all of the data must be protected from unauthorized modification. The sheer quantity of data together with distributed data access requirements necessitates an extended security infrastructure not commonly found in object oriented databases. The SLAC-designed Generic Authentication Protocol (GAP) provides this infrastructure and has been incorporated by Objectivity into their database product. This paper describes the design of the security protocol used in Objectivity/DB to authenticate users, the mechanism used to actually provide for proper authorization, and how the protocol can handle various authentication models such as Kerberos and PGP

keywords access control, authentication, authorization, capabilities, Objectivity/DB, security

1. Introduction

Few people would deny the necessity for security in most database systems. By security, we mean the combination of authentication (i.e., who are you), authorization (i.e., what can you do), and enforcement (i.e., appropriately limiting your actions). Close inspection of most database systems, even when one ignores non-commercial ones, reveals very few ruggedly secure systems, and fewer still that seamlessly integrate into existing security infrastructures. The reasons are historical, technical, and ultimately economic. Many database systems were developed at a time when security was synonymous with system login. The system provided access control to some critical resource (e.g., file access) based on the user's system supplied login credentials. That upbringing shows in current database offerings. So, as security models matured and became network-based, the number of variations multiplied. Incorporating a wide range of security models for architectures developed at a simpler time just became too costly. Hence, today's serious lack of rugged manageable security in most database systems.

2. Security in Objectivity/DB

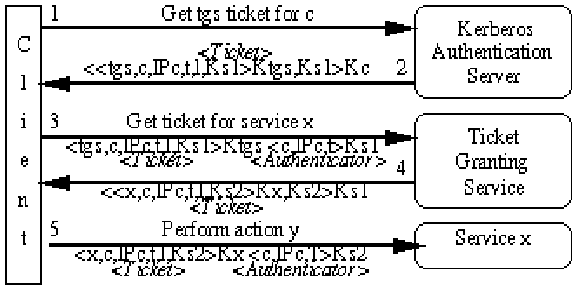
Objectivity/DB had a similar upbringing; not only did the database system have little security, but also putting in security without disrupting a large installed user-base was daunting. We set ourselves four goals:

- The system should handle at least private and public key authentication models,
- It should integrate with any installed security infrastructure,
- authorization should be independent of authentication, and

Produced under contract DE-AC03-76SF00515 between Stanford University and the Department of Energy

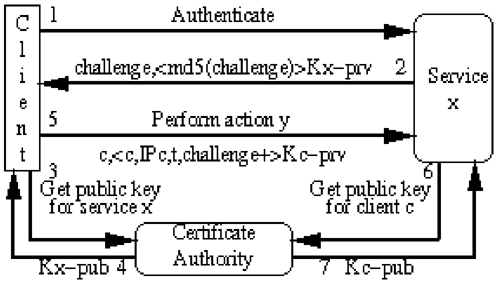
- The model should be extensible (i.e., new security protocols could be incorporated without changing the database server protocol).

In order to appreciate the complexity of the task; consider the differences between Kerberos (figure 1), a private key authentication system and DASS (figure 2), a typical public key authentication system.



Client
1 Get tgs ticket for c
Kerberos Authentication Server
<Ticket>
<<tgs,c,IPc,t,l,Ks1>Ktgs,Ks1>Kc 2
3 Get ticket for service x
Ticket Granting Service
<tgs,c,IPc,t,l,Ks1>Ktgs <c,IPc,t>Ks1
<Ticket> <Authenticator> 4
<<x,c,IPc,t,l,Ks2>Kx,Ks2>Ks1
<Ticket>
5 Perform action y
Service x
<x,c,IPc,t,l,Ks2>Kx <c,IPc,t>Ks2
<Ticket> <Authenticator>

Figure 1: Private KeyAuthentication



Client
1 Authenticate
Service x
challenge,<md5(challenge)>Kx–prv 2
5 Perform action y
c,<c,IPc,t,challenge+>Kc–prv
3 Get public key for service x
6 Get public key for client c
Certificate Authority
Kx–pub 4
7 Kc–pub

Figure 2: Public Key Authentication

In Kerberos, a client authenticates with a specialized authentication server, KAS, by asking the KAS for a “ticket” to use the ticket granting service, TGS. The KAS responds with a packet containing the TGS name, client’s name, current time, ticket lifetime, and a random session key, Ks1, all encrypted with a key only known by the KAS and TGS. The same session key, Ks1, is appended and the whole data stream is encrypted with a key, Kc, known only by the client and the KAS. Therefore, by decrypting the ticket, the client proves its authenticity. The client can then construct an authenticator containing the client’s name, IP address, and current time all encrypted with Ks1. To authenticate to an arbitrary service, the client asks the TGS for a ticket to the service and sends the ticket obtained from the KAS and the authenticator. The TGS can prove the client’s authenticity because it can decrypt the ticket, extract out the session key, Ks1; and use it to decrypt the authenticator. The client is authentic if the information in the authenticator matches the same information in the ticket, proving that the client knew the original private key, Kc. If the client passes this test, the TGS issues a service ticket which functions identically as the TGS ticket but only for the requested service. Thus, whenever the client requests service from a server, the client sends the server’s ticket along with an authenticator to prove the client’s authenticity.

In a public key system, the client first contacts the required service and asks for authentication information. The service responds by issuing a challenge to the client. The challenge is typically signed with the service’s private key, Kx-prv, so that the client may verify the authenticity of the service. Once the client receives the challenge, it obtains the service’s public key from a trusted Certificate Authority, CA, and authenticates the service. It then modifies the challenge in some appropriate way (i.e., typically the adding one to its numeric value) and signs or encrypts it with its private key, Kc-prv. On the subsequent request, the client also sends the modified challenge. The server authenticates the client by obtaining the client’s public key from a trusted CA, decrypts the challenge, and checks if it has been correctly modified. This protocol is followed for each server request.

Multiple variations exist of these two different protocols. In the private key arena Kerberos V4, AFS Kerberos (a version 4 variant), Kerberos V5, DCE Kerberos (a version 5 variant), and Microsoft Kerberos (a DCE variant) are popular. In the public key arena DASS, PGP, RSA, SSL, and TSL are used.

It was clear to use that accommodating such a wide array of protocols would require a radically different approach. We decided to use a tunneling single challenge protocol with optional arbitration. Such a protocol can handle virtually all private and public key authentication variants in existence today. The tunneling aspect of the protocol means that the authentication protocol is transparently carried across the database protocol. This makes the security protocol independent of the database protocol, and prevents changes in either one from impacting the other. External shared libraries using a well-defined object-oriented interface handle the actual implementation of the protocol. While the native authentication implementation supports Kerberos, any other protocol can be implemented by merely replacing the security library. Furthermore, the scheme allows for support of multiple protocols – a boon for heterogeneous environments. A typical client server interaction is shown in figure 3.

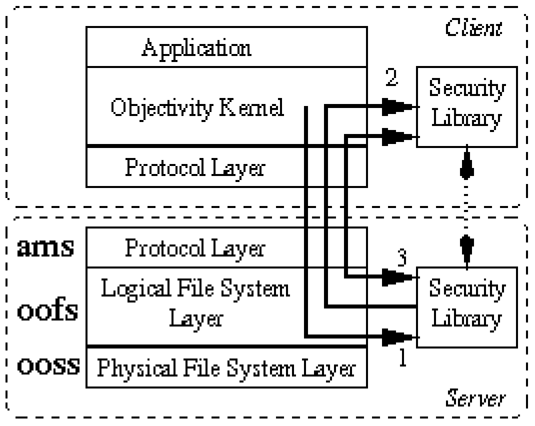


Figure 3: Objectivity Generic Authentication Protocol

In Objectivity/DB Version 5.2, the client-side kernel has been re-engineered to automatically call appropriate security functions during communications with the AMS in order to implement the Generic Authentication Protocol. On initial contact with an AMS, the client's Objectivity Kernel asks the AMS for a security token (i.e., one-time challenge). The server may respond with something like:

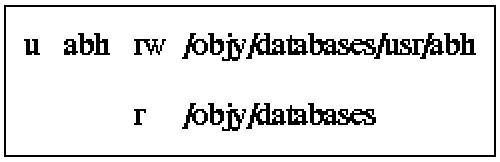
&P=KRB4,amsserv@slac.stanford.edu,0f00&P=PKP3,ams01:3333,0f00,0fce1100

The AMS is willing to use one of two protocols: Kerberos Version 4 and PKP Version 3. For Kerberos, the client must get a ticket for the service associated with amsserv in the slac.stanford.edu realm. When generating credentials, a request mask of 0f00 should be used. For PKP, protocol arbitration should be done by connecting to host ams01 at port 3333 in the same domain and using token 0fce1100 to start the negotiation phase. The credentials request mask is 0f00.

The token is passed to a method in the security library that creates a security object. That object is used to obtain future credentials. Whenever the Objectivity kernel makes a request to the AMS, it first obtains authentication credentials. The credentials, regardless of the method used, are wrapped in a self-describing envelope and tunneled through the database request stream. On the AMS side, the credentials are unwrapped and passed to the oofs layer along with the particular request (e.g., open, read, write, etc.). The oofs layer uses complimentary security routines to decode the credentials and establish the caller's authenticity. The same layer also invokes user authorization and enforces any access limits.

Another replaceable object class in the security library handles the actual authorization process. Consequently, any authorization model may be easily instantiated. However, for the native implementation, we considered the difficulty of handling authorization for hundreds of users relative to millions of databases. We chose a capability model. where each user is associated with database permissions as opposed to an access control model where each database carries with it a list of allowed users. In an environment where there are far fewer user objects than database objects, a capability model is generally easier to maintain and manage. To further ease administration, the implementation supports Unix groups, user groups, and capability templates.

Figure 4 shows a simple capability entry. Here, user abh has read access to any databases that starts on the path /objy/databases and read-write access to only databases on the path /objy/databases/usr/abh. We chose a path-prefixing scheme because it is simple to understand and effective in protecting large volumes of data. Also, more complicated schemes would require that the AMS have undue knowledge of database internals.



```
u  abh  rw  /objy/databases/usr/abh

         r  /objy/databases
```

Figure 4: Sample Authorization Entry

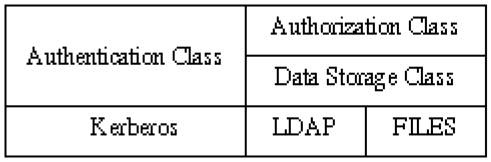


Figure 5: Security Class Relationships

Finally, the authorization data must be stored in a permanent place. We decided to divorce the data storage method from its processing. This allows for any number of storage methods to be used by simply replacing the data storage class. The two supported methods allow for storage in simple flat files or in an LDAP interfaced database. The former was chosen as the simplest method for installations with very small numbers of users while the latter for installation wishing to integrate existing user directory information. Figure 5 shows the relationship of the classes used by the security library.

References

1 J. G. Steiner, C Neuman, J. I. Schiller, "Kerberos: An Authentication Service for Open Networks", M.I.T. Project Athena, Cambridge, Massachusetts, March 30, 1988.
2. C Kaufman, "Internet rfc1507", September 1993.
3. A. Hanushevsky, "AMS Extensions Version 5.0", SLAC, October 26, 1999