APM

POSEIDON HOUSE • CASTLE PARK • CAMBRIDGE CB3 0RD UNITED KINGDOM
+44 1223 515010 • Fax +44 1223 359779 • Email: apm@ansa.co.uk • URL: http://www.ansa.co.uk

E2S

STAR Security Engineering

Dave Otway

Abstract

This document describes a design for an initial implementation of secure sessions between remote clients and STAR servers.

Doc No. 1979.00.03 **Draft** 16 May 1997

Distribution:
Supersedes:
Superseded by:

TABLE OF CONTENTS

1 OVERVIEW

This document describes a design for an initial implementation of secure sessions between remote clients and STAR servers as described in [Madsen]. Due to the short timescales it does not implement the key management described in [Herbert] but concentrates on transferring the basic functionality developed in [Otway].

1.1 Objectives

The objectives of the STAR security engineering described in this document are to:

- authenticate (only) legitimate STAR clients and servers
- set up secure sessions (only) between legitimate clients and servers
- enable client and server keys to be stored on a physically secure authentication server which never reveals them to any party
- minimise the use of client and server keys by using random keys for encryption during a secure session between an authenticated client and server
- ensure the integrity of messages in a secure session
- ensure the confidentiality of messages in a secure session
- ensure the timeliness of messages in a secure session

This engineering is not responsible for:

- key distribution
- key management
- key storage

1.2 Constraints

This design has been prepared under the following constraints:

- the client must run over OrbixWeb
- source code for remote invocations must be CORBA compliant
- the first version should be ready as soon as possible

1.3 Hooks

OrbixWeb has three hooks for transparently processing the arguments and results of remote invocations; transformers, per-process filters and per-object filters.

1.3.1 Transformers

These are the only way to gain access to the byte buffer in order to encipher and decipher requests and replies.

```
public class IT_reqTransformer
{
    public boolean transform (_sequence_Octet data, String
host,
                                boolean is_send) ;
    public String transform_error() ;
}
```

There is a minimum of information on which to base a decision; the name of the remote host and whether the message is being sent or received.

1.3.2 Per-Object Filters

A particular object can have a pre and/or a post filter associated with it. But these filters can only be added in the server, there are no equivalent per-object client filters. Therefore, per-object filters are uninteresting as a security mechanism.

1.3.3 Per-Process Filters

These can be attached at any of eight insertion points during a remote procedure call. A client process can attach filters for:

- outRequestPreMarshal, outRequestPostMarshal, inReplyPreMarshal, inReplyPostMarshal

A server process can attach filters for:

- inRequestPreMarshal, inRequestPostMarshal, outReplyPreMarshal, outReplyPostMarshal

Each filter is passed a Request as its argument. Out-PreMarshal and in-PostMarshal filters are called before the parameters have been added to, or after they have been removed from, the request. In-PreMarshal and out-PostMarshal filters are called after the parameters have been removed from, or before they have been added to, the request.

A filter can find out the target object from the request, so it can do per-object filtering as well.

The arguments are only extractable from the request as specified data types, so it is difficult to digest them.

1.4 Approach

The encryption can only be done in the transformers. The only information they have available is the name of the remote host and the direction of the request. The remote host name can be used to index a local list of active sessions. If an entry is found then the session key will be used to encipher or decipher the request depending on the direction. To avoid any potential problems with multi-threaded clients (including multiple machines hidden behind a firewall), a random initialisation vector will be prepended to each request.

If no active session is found for the remote host, the transformer will pass on the request in clear. This is required for the authentication of a new active session without making the client's key available to all servers, but it is also a potential weakness.

The authentication of a new active session for a client will be done using the Recursive Symmetric Authentication protocol [Bull-Otway]. In this protocol, a digest request is constructed using the client's key. This will be passed to the server as an argument to an invocation of the server's security manager object which will nest the client's digest request in one of its own and pass them on to an authentication server. The authentication server, which knows both the client's and server's keys, will authenticate both digests and generate a random session key which it returns enciphered in a nested digest response. The server will extract its copy of the session key and add a new entry to its active session list. It returns the inner digest response to the client which extracts its copy of the session key and adds a new entry to its active session list. Any future invocations from the client's host machine to a server on the same machine as the security manager will be enciphered/deciphered by the transformers.

The digest request and response classes will be designed so that they can be used directly by an application programmer or hidden in per-process filters.

This leaves two problems:

- An attacker could simply invoke a server in clear.
- If the client's host machine is hidden behind a firewall that does address translation then if a second client host machine behind the same firewall invoked the same security manager as the first client, the client's transformer would transmit the request in clear but the server's transformer would decipher it.

To solve the first problem, the security manager (or a per-process filter acting on its behalf) needs to know whether the request was deciphered by the transformer. The transformer can't add anything to the request as this could be spoofed. The filter could check whether the host name in the object

reference it retrieves from the request has an entry in the list of active sessions, but without access to the OrbixWeb sources it is impossible to be sure that this can't be spoofed (but even this would be difficult).

A more easily verified option would be to add a keyed digest to each message in per-process filters. This does not need to be a digest of the arguments, because a valid request will be protected by encryption. The purpose of the digest is to prove that the message is part of an active authenticated session and has therefore been enciphered/deciphered by the transformers, so a digest of the session ID and a non-repeating nonce keyed with the session key would do the job.

The second problem can be solved by marking plaintext requests. Enciphered requests have a random initialisation vector prepended to them and plaintext requests can be marked with an invalid value such as all zeros. The receiving transformer would not attempt to decipher plaintext requests even if an active session existed for the client's host. The client's transformer can detect requests that should be sent in plaintext because it won't be able to find an active session for the server's host, but the server's transformer can't always detect plaintext replies because it may already have an active session for another client host behind the same address translating firewall. If the server's outgoing per-process reply filter inserted a boolean[1] at the front of the argument list, the transformer could find this in a (reasonably) fixed place in an outgoing octet sequence. A false value would signal a plaintext reply and a true value a ciphertext reply.

[1] Alternately, the session ID can be passed in a global object keyed by thread ID.

2 DETAILS

2.1 Sessions

Session objects store security data for a secure session of client/server invocations. Sessions come in pairs, one in a client referring to a server and a matching one in the server referring to the client. The session class contains a static list of all currently active sessions. A session can be looked up using either the name of the remote host with the matching session or its ID. The ID is big enough to include date and time information so that out of date session IDs can never clash with current ones.

```
final class Session
{
    static long new_ID () ;
    static byte[] new_key () ;
    static Session lookup (String remoteHost) ;
    static Session lookup (long id) ;

    public Session (String otherParty, String remoteHost,
                    byte[] key, long id) ;
    void addParty (String otherParty) ;
    public void close (String remoteHost) ;

    String otherParty () ;
    String remoteHost () ;
    long id () ;
    byte[] key () ;
    byte[] digest (long nonce) ;
    public String toString () ;
}
```

A server creates a session with a client's name, client's host name, a key and a session ID. A client creates a matching session with the server's name and server's host name plus the key and ID it received from the server in a DigestResponse [see below].

The access methods otherParty(), remoteHost(), id() and key() return the corresponding data from the session. The addParty() method is used by a server to add other clients from the same host to a session and the close() method counts down the clients using the session and finally removes the session from the static lookup tables when there are no more clients.

The digest method constructs a keyed digest of :

session key, padding, session ID, nonce, session key

which is used authenticate an incoming request to the servers inRequest filter after it has been deciphered by its transformer. The digest is secure as long as the nonce is non-repeating (in a session). The padding extends the first session key up to the internal block size of the SHA algorithm. For a discussion of this and the double inclusion of the key see [Schneier, Page 458].

2.2 DigestRequest

A client authenticates itself to a server using a digest request. When a new digest for a particular server is constructed, a new non-repeating nonce is generated and a keyed digest is constructed for the message:

client key, padding, nonce, client, server, client key

The digest request can then be marshalled into a byte array so it can be passed to the server. This array contains byte representations of:

nonce, client, server, digest

The server that receives this byte array can unmarshal it back into a digest request using the byte[] constructor.

```
public final class DigestRequest
{
    public DigestRequest (String source, String target)
                        throws KeyNotFound ;
    public DigestRequest (String source, String target,
                        long sessionID, byte[] sessionKey,
                        DigestRequest nested)
                        throws KeyNotFound ;
    public DigestRequest (byte[] marshalled) ;
    public byte[] marshal () ;
    public long nonce () ;
    public String source () ;
    public String target () ;
    public DigestRequest nested () ;
    public long sessionId () ;
    public byte[] sessionKey() ;
    public boolean check_digest () throws KeyNotFound ;
    public String toString () ;
}
```

The access methods nonce(), source() and target() return the plaintext fields of the request. If a server's KeyManager has access to the client key it can invoke the check_digest method itself to authenticate the client. But if the client keys are held on a third party authenticator, it must construct a nested

digest from its own name, the authenticator's name and the client's digest. This generates a new nonce and the digest of the message:

server key, padding, nonce, server, authenticator, client request, server key

which marshals as a byte representation of:

nonce, server, authenticator, client request, digest

When the authenticator has unmarshalled the server's request it can use the nested() method to extract the nested client request and then check the digests of both requests.

If a session already exits for another client on the same host then the server creates a nested request with the same id and key so that the authenticator can extract them and encode them in the responses. Otherwise the server creates a nested session with an ID of zero and a null key.

2.3 DigestResponse

The digest response is used to return an encoded session key from a server back to a successfully authenticated client. A nested digest response returns from an authenticator to a server the same session key encoded for both server and client.

```
public final class DigestResponse
{
    public DigestResponse (DigestRequest request,
                      long sessionID, byte[] sessionKey)
                      throws KeyNotFound ;
    public DigestResponse (DigestRequest request,
                           DigestResponse nested)
                      throws KeyNotFound ;
    public DigestResponse (byte[] marshalled) ;
    public byte[] marshal () ;
    public byte[] session_key () throws KeyNotFound ;
    public long session_ID () ;
    public long nonce () ;
    public String source () ;
    public String target () ;
    public DigestResponse nested () ;
    public boolean check_digest () throws KeyNotFound ;
    public String toString () ;
}
```

A single (or inner) digest response is constructed from the client (or inner) request. A nested digest response is constructed from the server (outer) request and the client (inner) response,

The marshal method for a nested response returns a byte array representation of:

server nonce, client, server, sessionID, session key ⊕ digest1,
marshalled inner response, digest2

Where: ⊕ = "exclusive or",

digest1 is:

server key, padding, server nonce, client, server, sessionID, server key

and digest2 is a digest of the whole response excluding itself keyed with the session key.

In the marshalled inner response, the server key = client key, server nonce = client nonce and the inner response is null.

The server and client unmarshal their digest responses using the (byte[]) constructor. The server can extract the client's (inner) response using the nested() method. The plaintext fields can be accessed with the nonce(), source() and target() fields.

For an incoming response the digest can be checked with the check_digest() method and the session key and ID can be extracted using session_key() and session_ID().

2.4 KeyManager

The KeyManager interface is implemented by the local class that stores primary keys. Primary keys are retrieved by name using the lookup method.

```
interface KeyManager
{
    byte[] lookup (String name) throws KeyNotFound ;
}
```

A client has a simple KeyManager which retrieves keys from a disc file protected by being XORed with a digest of the name and a secret pass phrase. A server's key manager may have more sophisticated key management capabilities.

2.5 Crypto

The Crypto class hides all primary keys and most of the details of the cryptographic algorithms used. All its methods and fields are static. In the short term it avoids the need for primary keys to be stored in application classes and hides as much as possible of the details of the cryptographic algorithms being used. In the longer term it enables all primary keys and cryptographic algorithms to be moved onto smartcards.

The DIGEST_LENGTH is the number of bytes in the digest produced by a MessageDigest or KeyedDigest. The KEY_LENGTH is the number of bytes in a key used by the cipher method. The BLOCK_LENGTH is the number of bytes in the internal block of the cryptographic algorithm used by the cipher method; in particular it is the length of the initialisation vector (iv).

```
final class Crypto
{
    static final int DIGEST_LENGTH ;
    static final int KEY_LENGTH ;
    static final int BLOCK_LENGTH ;
    static void key_manager (KeyManager manager) ;
    static long new_nonce () ;
    static byte[] new_iv () ;
    static byte[] new_key () ;
    static MessageDigest message_digest() ;
    static KeyedDigest keyed_digest (String name) ;
    static KeyedDigest keyed_digest (byte[] key) ;
    static void cipher (byte[] buffer, int offset,
                        int length, byte[] iv, byte[] key,
                        boolean encipher);
}
```

The key_manager method sets the interface to the object which access the primary keys. The new_nonce, new_iv and new_key methods return new instances with the right properties. Nonces and IVs are non-repeating and keys are random.

The message_digest returns a new SHA MessageDigest [from the Cryptix library]. The keyed_digest methods returns a subclass of an SHA MessageDigest which has been initialised with the key plus padding and has remembered the key for inclusion at the end of the message when the digest is completed. The method with the String argument is for primary keys looked up by name in the KeyManager. The method with the byte[] argument is for use with session keys.

The cipher method enciphers or deciphers (depending on the final argument) a block length bytes long stating at the index offset in the buffer, using the supplied initialisation vector and session key.

2.6 Security Manager

The security manager is an object in the server process that authenticates clients and opens secure sessions. It may have multiple open methods for different servers, but each method has a marshalled digest request as an in parameter, a marshalled digest response as an out parameter, the client's host name as an in parameter and returns an object reference to the service if the authentication is valid. Otherwise it throws an exception.

```
public interface SecurityManager
{
```

```
    Service openService (byte[] request, byte[] response,
                         String host)
        throws AuthenticationFailed ;
}
```

A new instance of the server object should be created for each client and the client name (extract from the request) set as the marker of the server's object reference by calling:

```
super(client);
```

as the first statement of the server's implementation class constructor.

The security manager can authenticate the client itself, or construct a nested digest request and invoke a third party authenticator.

2.7 Authenticator

A third party authenticator checks the digests in a nested digest request and if both are valid returns a new session key to both parties in a nested digest response.

```
public interface Authenticator
{
    void authenticate (byte[] request, byte response,
                       String host)
        throws AuthenticationFailed ;
}
```

2.8 Invocation path

The invocation path from a client object for the filters and transformers described below is:

```
client outRequestPreMarshal
    client transformer outgoing
        server transformer incoming
              server inRequestPreMarshal
                   server object
              server outReplyPreMarshal
        server transformer outgoing
    client transformer incoming
client inReplyPreMarshal
```

2.9 Per-process Filters

All filters have the same signature, only the name changes.

```
public boolean FilterName (Request r)
    throws SystemException ;
```

2.9.1 Client outRequestPreMarshal

This filter retrieves the host name from object reference in the request then looks up its session. If a session is found then the session ID, new nonce and digest of the session ID and nonce keyed with the session key are inserted into the request's argument list. If no session is found then just a zero session ID is inserted.

2.9.2 Server inRequestPreMarshal

This filter extracts the session ID from the request. If the ID is zero then the invocation is only allowed to proceed if the object reference in the request is for the security manager. If the ID is non-zero, then the nonce and digest are extracted and the invocation is only allowed to proceed if the digest is correct and the client name in the session matches the marker in the object reference.

2.9.3 Server outReplyPreMarshal

This filter retrieves the object reference from the request. If it is for the security manager it inserts a boolean value of false in the argument list to signal to the transformer not to encipher the request, otherwise it inserts true.

2.9.4 Client inReplyPreMarshal

This filter extracts the cipher boolean from the request and discards it.

2.10 Transformers

The client and server transformers are responsible for encryption. They must inherit from IT_reqTransformer and override the transform method.

```
public class IT_reqTransformer
{
    public boolean transform (_sequence_Octet data,
                                  String host,
                                boolean is_send) ;
    public String transform_error() ;
}
```

The client and server processes both have a single transformer which handles both incoming and outgoing messages, which it distinguishes by its is_send argument.

If the server invokes other servers then its transformer must be careful not to interfere with such requests. To enable it to do this, the server must maintain a list of other server hosts it is using.

Note that the server does not have any outRequest or inReply filters.

2.10.1 Client outgoing

The host name is used to look up a session. If one is found, the key is retrieved, a random (non-zero) 8 byte initialisation vector inserted at the start of the octet sequence and the rest of the sequence is enciphered. Otherwise, 8 zero bytes are inserted at the start of the sequence and the request is transmitted in clear.

2.10.2 Server incoming

The host name is looked up in the list of other servers. If found the request is passed up without any processing. Otherwise, the 8 byte initialisation vector is extracted from the octet sequence and if it is non-zero then the host name is used to lookup the session, the key retrieved and the rest of the sequence is deciphered.

2.10.3 Server outgoing

The host name is looked up in the list of other servers. If found the request is passed down without any processing. Otherwise, the cipher boolean is read from the octet sequence. If it is false then 8 zero bytes are inserted at the start of the sequence, otherwise, the session is looked up with the host name, the key retrieved, a random (non-zero) 8 byte initialisation vector inserted at the start of the octet sequence and the rest of the sequence is enciphered.

2.10.4 Client incoming

The 8 byte initialisation vector is extracted from the octet sequence. If it is non-zero then the host name is used to lookup the session, the key retrieved and the rest of the sequence is deciphered.

3 REFERENCES

[Bull-Otway] The Authentication Protocol
John Bull, Dave Otway
DRA/CIS3/PROJ/CORBA/SC/1/CSM436-04/0.3,
APM Ltd 97/02/25

[Herbert] SBCW Demonstrator
Andrew Herbert
APM Ltd. 97/02/07

[Madsen] Securing STAR for the Internet
Mark Madsen
APM Ltd. 97/03/07

[Otway] Design for Secure Remote Method Invocation
Dave Otway
DRA/CIS3/PROJ/CORBA/SC/1/CSM436-07/0.1,
APM Ltd 97/03/17

[Schneier] Applied Cryptography, Second Edition
Bruce Schneier,
Wiley, ISBN 0-471-11709-9, 1996