

**Using
OSP
with
OpenH323**

Version 1.1

Craig Southeren,
12 January 2005

Table of Contents

1. Introduction.....	1
1.1 Purpose.....	1
1.3 Document history.....	1
1.4 Contact information	1
2. Overview	2
2.1 Introduction to OSP	2
2.2 OpenH323 OSP Interface	3
3. Configuration	3
4. Using OSP with the OpenH323 sample programs.....	5
4.1 Endpoints using an OSP provider directly.....	5
4.2 Endpoints using an OSP-enabled gatekeeper	5
4.3 Using OpenGK with an OSP server.....	5
5. High level OSP API.....	7
5.1 Direct OSP routing.....	7
5.2 OSP gatekeeper token handling	7
6. Low level OSP API	8
6.1 Header file.....	8
6.2 OpalOSP::Provider	8
6.3 OpalOSP::Transaction	9
6.4 Authorising outgoing calls.....	9
6.5 Getting destinations	11
6.6 Validating incoming calls	11
6.7 Ending a call	13
6.8 Example 1 – outgoing call	13
6.9 Example 2 – incoming call	15
7. Helper functions.....	17
8. OSP Sample Program	18

1. Introduction

1.1 Purpose

The Open Settlement Protocol (OSP) provides a vendor neutral mechanism for network elements to exchange routing, billing and authentication information. The OSP Toolkit from Transnexus is the standard library for implementing OSP in applications, and supports both Unix and Windows platforms.

This document describes the OSP support integrated into the OpenH323 stack and sample programs. The API functions in OpenH323 that provide an interface to the Transnexus OSP Toolkit are also described.

This document assumes a basic understanding of H.323 concepts and OpenH323.

1.3 Document history

Ver	Date	By	Description
1.0	20 December 2004	Craig Southeren	Initial version
1.1	12 January 2005	TransNexus	Clarifications on usage reporting

1.4 Contact information

For more information, please contact the author:

Craig Southeren
Post Increment
craigs@postincrement.com
Phone: +61 2 4365 4666
Fax: +61 2 4367 3140

2. Overview

2.1 Introduction to OSP

OSP entities communicate using cryptographic tokens that are encoded into network PDUs as part of call requests. A token is requested from an OSP server or *provider* and remains valid only for the duration of a call. The lifetime of an OSP token is called a *transaction* and a transaction is ended when a call is completed, whether the call is successful or not.

The communication between an endpoint and a provider is authenticated and encrypted using SSL. This requires the endpoint to create private and public keys, and to perform an *enrollment* procedure to register those keys with the provider. It also requires the endpoint to have a copy of the certificate for the certifying authority (CA) that signed the server's SSL certificate.

When an originating endpoint makes an outgoing call, it creates a transaction and sends a request to an OSP provider. If the provider determines that the call can be routed, it will reply with an OSP token that the originator encodes into the outgoing call request for transmission to the destination. In the event that the destination is unreachable, or does not accept the call, the originator can request an alternate route. This process is repeated until the call succeeds, or there are no more routes, at which time the originator closes the transaction. The result of each call attempt, regardless of whether it failed or succeeded, is sent to the provider to allow full network statistics to be gathered.

When a terminating endpoint receives a call request containing an OSP token, it creates a transaction and asks the toolkit to verify the cryptographic signature and validate the token. The OSP toolkit will validate the call if the token was signed by a trusted clearinghouse (the trusted relationship is established during enrollment) and the information in the token matches the call request information, and allows the terminating endpoint to accept the call. When the call completes, the call result is sent to the provider by both ends, and the transaction is closed.

A H.323 network can use OSP in two ways. The recommended method is to integrate the OSP protocol into the H.323 gatekeeper, which sends tokens to the endpoint via `clearToken` fields in the AdmissionConfirm and AdmissionRequest PDUs. Alternatively, H.323 endpoints can communicate directly with an OSP server to resolve E.164 numbers. OpenH323 supports both methodologies.

2.2 OpenH323 OSP Interface

OpenH323 provides two types of interfaces to OSP:

1. Applications that use the `H323Endpoint` class to manage calls are able to enable OSP routing by making a single function call. This is intended for developers of OpenH323-based endpoints and applications who do not need access to the internal details of OSP
2. An API wrapper for the Transnexus OSP Toolkit is provided that integrates OSP functions into the data structures and methodology used by OpenH323. This interface is intended for use by developers of gatekeepers and proxies who need direct access to OSP routing functions.

Support for OSP is integrated into the following OpenH323 sample programs:

- Simph323 – see `openh323/samples/simple`
- Ohphone – see `contrib/ohphone`
- Opengk – see `contrib/opengk`

3. Configuration

The OSP functions require the installation of the Transnexus OSP Toolkit available from <http://www.transnexus.com>.

The OpenH323 configuration process on both Windows and Unix will automatically detect the presence of a correctly compiled and installed Transnexus OSP Toolkit.

On Unix, the toolkit is expected to be in the default location, which is `/usr/local`.

```
$ ./configure
checking for g++... g++
checking for C++ compiler default output... a.out
checking whether the C++ compiler works... yes
..deleted..
configure: Skipping tests for RFC 2190 H.263 support
configure: Skipping tests for VIC H.263 and non-standard H.263
checking osp/osp.h usability... yes
checking osp/osp.h presence... yes
checking for osp/osp.h... yes
checking for OSPPInit in -losp... no
checking for OSPPInit in -losp... yes
checking linux/telephony.h usability... yes
checking linux/telephony.h presence... yes
```

On Windows, the `configure.exe` program will find the OSP install directory automatically, as follows:

```
Configuring Build Options
Opened configure.ac
Located ffmpeg AVCODEC Library at d:\ffmpeg\libavcodec\
Searching C:\
Searching D:\
Located ffmpeg RFC2190 AVCODEC Library at D:\rfc2190avcodec.dll\
Searching E:\
Located Transnexus OSP Toolkit at E:\osptoolkit\
```

An enrollment process is required in order to use any OSP provider. This process will create three cryptographic keys which are needed to access the OSP server. Your OSP service provider will provide information on how to complete the enrollment process for their network.

4. Using OSP with the OpenH323 sample programs

4.1 Endpoints using an OSP provider directly

The sample endpoint program provided with OpenH323 (simph323), and the reference command line endpoint program (ohphone), are capable of routing outgoing calls using a connection directly to an OSP provider. This is enabled by specifying the URL of the OSP server using the `--osp` option as follows:

```
--osp http://osptestserver.transnexus.com:1080/osp
```

The three certificate files obtained during the enrolment process are required for accessing the OSP provider and are assumed to be in the current directory unless an alternate directory is specified with the `--ospdir` option. The names of the files are derived from the hostname of the OSP provider, as follows:

Key file	Filename
Private key	<i>hostname_priv.pem</i>
Public key	<i>hostname_cert.pem</i>
CA cert	<i>Hostname_cacert.pem</i>

An OSP server requires the local endpoint to have an E.164 number as a local alias, so it will be necessary to specify an appropriate number on the command line using the `--u` option.

Both simph323 and ohphone are unable to resolve addresses using simultaneous connections to an OSP provider and a gatekeeper. If you need gatekeeper functionality with OSP, then please read section 4.2 below.

4.2 Endpoints using an OSP-enabled gatekeeper

The simph323 and ohphone programs can operate with OSP enabled gatekeepers to route calls if invoked with the `--osptoken` option. In this case, the endpoints look for OSP tokens in ACF messages received from gatekeepers and include these tokens into outgoing SETUP messages.

No OSP certificates are required for endpoints to use an OSP-enabled gatekeeper as the gatekeeper performs all of the OSP work. In fact, the ability to work with an OSP gatekeeper is available even if the endpoint is compiled without the Transnexus OSP Toolkit. As always when using OSP, the endpoint needs to have an E.164 alias for the OSP server to correctly identify the source endpoint.

4.3 Using OpenGK with an OSP server

OpenGK includes support for routing calls using an OSP server. This function is enabled by configuring OpenGK for OSP using the web page console for OpenGK. This can be found by using a web browser to open the following URL:

<http://hostname:1719/Parameters>

where *hostname* is the IP address or hostname of the computer running OpenGK. The URL for the OSP provider should be entered into the field labeled “OSP Routing URL”. The names of the files containing the three OSP keys are entered into the fields labeled “OSP Private Key”, “OSP Public Key”, and “OSP Server Key”.

If the OSP key fields are left empty (see picture below), OpenGK will construct the names of the key files from the OSP provider hostname and current directory as described in section 4.1.

H501 Interface	<input type="text"/>
OSP Routing URL	<input type="text" value="http://osptestserver.transnexus.com:1080/osp"/>
OSP Private Key	<input type="text"/>
OSP Public Key	<input type="text"/>
OSP Server Key	<input type="text"/>
Gatekeeper Identifier	<input type="text" value="OpenH323 Gatekeeper on i"/>

5. High level OSP API

5.1 Direct OSP routing

Direct OSP routing can be enabled within the `H323Endpoint` class by calling the `SetOSPProvider` function with the URL of the OSP provider, as follows:

```
H323Endpoint ep;  
  
// ...initialize endpoint here...  
  
ep.SetProvider("http://osptestserver.transnexus.com:1080/osp");
```

The names of the certificate files are created from the host name of the OSP server and the current directory as described in section 4.1 above.

OSP routing can be disabled by passing a `NULL` pointer to the form of the `H323Endpoint::SetOSPProvider` function that accepts a `OpalOSP::Provider *` as a parameter, as follows:

```
ep.SetProvider(NULL);
```

Note that the endpoint will not correctly route calls if both gatekeeper support and OSP routing are enabled on the same time.

5.2 OSP gatekeeper token handling

An application that uses an OSP-enabled gatekeeper is required to manage tokens as defined by ETSI TS 101 32 D.3. Full support for OSP token handling can be enabled in OpenH323 by calling the `H323Endpoint::SetGkAccessTokenOID` function with the correct identifier. For OSP, the correct identifier is 0.4.0.1321.1.2 which is available as a constant called `OpalOSP::ETSIXMLTokenOID`. Thus, token handling in the endpoint can be enabled as follows:

```
H323Endpoint ep;  
  
// initialize endpoint here  
  
ep.SetGkAccessTokenOID(OpalOSP::ETSIXMLTokenOID);
```

6. Low level OSP API

6.1 Header file

The interface for all OpenH323 low level OSP functions is contained in the file `opalosp.h`. This file must be included in any application program if direct access to the low level OpenH323 OSP functions is required, as follows:

```
#include <opalosp.h>
```

The API is contained within a C++ namespace named `OpalOSP`, and is implemented by two classes: `OpalOSP::Provider` and `OpalOSP::Transaction`. These encapsulate the `OSPTPROVHANDLE` and `OSPTTRANHANDLE` functions provided by the Transnexus OSP toolkit.

6.2 `OpalOSP::Provider`

The `OpalOSP::Provider` class manages the connection between the network entity and an OSP server. An instance of this class must be created and opened before any OSP functions can be used. This instance is usually associated with the “endpoint” or “manager” class used by the endpoint.

The empty constructor can be used to create an instance of `OpalOSP::Provider`, as follows:

```
OpalOSP::Provider * ospProvider = new OpalOSP::Provider;
```

This instance must be deleted when it is no longer required.

Several versions of the `OpalOSP::Provider::Open` method are provided to establish a connection to an OSP provider. One form requires the URL for the OSP provider as well as the filenames of the private, public and provider CA certificates. These parameters are provided explicitly, as shown below:

```
PString ospName      = "http://osptestserver.transnexus.com:1080/osp";  
PFilePath privKey    = "private_key.pem";  
PFilePath pubKey     = "public_key.pem";  
PFilePath caCertKey  = "ca_cert.pem";  
  
int status = ospProvider->Open(ospname, privKey, pubKey, caCertKey);
```

The return value is 0 if the provider can be opened, else a non-zero error value is returned.

Another version of the `OpalOSP::Provider::Open` method accepts the OSP provider URL as the only parameter. This function constructs the names of the certificate files from the host name of the OSP server and the current directory as described in section 4.1 above.

An instance of `OpalOSP::Provider` can be cast directly to the `OSPTPROVHANDLE` type used by many of the OSP Toolkit functions, as follows:

```
OpalOSP::Provider * provider = new OpalOSP::Provider();  
  
// open and verify provider here..  
  
OSPTPROVHANDLE * ospProvHandle = *provider;
```

6.3 *OpalOSP::Transaction*

The `OpalOSP::Transaction` class holds the information associated with an OSP transaction. An instance of this class must be created and opened before an OSP routing request can be made, and this instance must exist for the duration of a call. This object is usually associated with the “connection” or “call” class used by the endpoint.

The empty constructor can be used to create an instance of `OpalOSP::Transaction`, as follows:

```
OpalOSP::Transaction * ospTransaction = new OpalOSP::Transaction;
```

This instance must be deleted at the end of a call.

The `OpalOSP::Transaction::Open` method is used to open a transaction using a provider. The `Open` function accepts an `OpalOSP::Provider` reference as an argument, as follows:

```
int status = ospTransaction->Open(*ospProvider);
```

The return value is 0 if the transaction was created correctly, else a non-zero error value is returned.

6.4 *Authorising outgoing calls*

There are several `Authorise` member functions that can be used on an opened instance of `OpalOSP::Transaction` to authorize an outgoing call. The recommended version requires the population of an `OpalOSP::Transaction::AuthorisationInfo` structure with all of the information needed by the OSP server. This structure contains the following elements:

```
struct OpalOSP::Transaction::AuthorisationInfo {
    PString ospvSource;
    PString ospvSourceDevice;
    H225_AliasAddress callingNumber;
    H225_AliasAddress callingNumber;
    H225_AliasAddress calledNumber;
    PBYTEArray callID;
    H225_AliasAddress callingNumber;
    H225_AliasAddress calledNumber;
    PBYTEArray callID;
};
```

Each of these fields correspond to parameters associated with an outgoing call and can be extracted from either the ARQ or SETUP PDU that has triggered the OSP request. The `ospvSource` field must contain the IP address of the device which sends the OSP AuthorizationRequest to the OSP server. The `ospvSourceDevice` field must contain the IP address of the device which is the source of the call. If the OSP AuthorizationRequest comes from a gateway, these two fields should have the same IP address since the source of the call and the OSP AuthorizationRequest are the same. If the OSP AuthorizationRequest is sent from a gatekeeper or inter-working proxy the `ospvSource` field must contain the IP address of the gatekeeper or inter-working proxy. The `ospSourceDevice` field must contain the IP address of the device that is the source of the call. With a gatekeeper or inter-working proxy implementation, there are four different cases for `ospvSourceDevice`:

1. IP address of the gateway that sent an ARQ (gatekeeper case).
2. IP address of the gatekeeper that sent an LRQ (gatekeeper case).
3. IP address of the gateway that sent a direct Q.931 call setup without an ARQ or LRQ (proxy case).
4. IP address of the device that sent a direct SIP INVITE (inter-working proxy case).

Note that `ospvSource` and `ospvSourceDevice` fields must contain the source IP address of the originating endpoint in the following form:

`[x.x.x.x] : port`

There are several helper functions that can be used to convert the various OpenH323 address types into this format.

The `OpalOSP::Authorise` method is invoked with as follows:

```
OpalOSP::Transaction::AuthorisationInfo authInfo;

// populate authInfo

unsigned int numberOfDestinations = 1;
int status = ospTransaction->Authorise(authInfo, numberOfDestinations);
```

On entry to the function, the `numberOfDestinations` field specifies the maximum number of alternate routes that the endpoint is interested in. On exit, this field specifies the actual number of destinations that are available.

As always, a return value of 0 indicates that the function succeeded. Any other value indicates that an error has occurred.

6.5 Getting destinations

Once an outgoing call has been authorised, the first destinations is retrieved using the `OpalOSP::Transaction::GetFirstDestination` function. This function accepts a single argument of type `OpalOSP::Transaction::DestinationInfo` that contains the following fields:

```
struct DestinationInfo {
    unsigned timeLimit;
    PByteArray callID;
    H225_AliasAddress calledNumber;
    H323TransportAddress destinationAddress;
    PString destination;
    PByteArray token;
};
```

There are several member functions named `Insert` that can be used to automatically insert the contents of the `OpalOSP::Transaction::DestinationInfo` into a ACF or SETUP PDU. There is also a function called `InsertToken` that will insert the OSP token into a structure of type `H225_ArrayOf_ClearToken`.

If the call to the first destination is not successful, then a call to `GetNextDestination` is used to get subsequent routes. This function is very similar to `GetFirstDestination` with the addition of an `endReason` argument to specify the failure mode for the previous destination. This is repeated until the possible destinations have been exhausted.

6.6 Validating incoming calls

There are several `Validate` member functions that can be used on an opened instance of `OpalOSP::Transaction` to validate an incoming call. The recommended version requires the population of an `OpalOSP::Transaction::ValidationInfo` structure with all of the information needed by the OSP server. This structure contains the following elements:

```
struct OpalOSP::Transaction::ValidationInfo {
    PString ospvSource;
    PString ospvDest;
    PString ospvSourceDevice;
    PString ospvDestDevice;
    H225_AliasAddress callingNumber;
    H225_AliasAddress calledNumber;
    PByteArray token;
};
```

Each of these fields correspond to parameters associated with an incoming call and can be extracted from either the ARQ or SETUP PDU that triggered the validation request. The table below provides a summary of how these fields should be populated.

Field	Description
ospvSource	IP address of the device which sent the call set-up message.
ospvSourceDevice	Not used today by the destination device
ospvDest	The IP address of the device validating the token.
ospvDestDevice	IP address of the gateway which terminated the call. Not used today.

There is a `OpalOSP::Transaction::ValidationInfo::ExtractToken` function that accepts a single argument of type `const H225_ArrayOf_ClearToken &` which can be used to extract the token from the ARQ or SETUP.

The `OpalOSP::Validate` method is invoked as follows:

```
OpalOSP::Transaction::ValidationInfo valInfo;  
  
...populate valInfo...  
  
BOOL authorized;  
unsigned timeLimit;  
int status = ospTransaction->Authorise(valInfo, authorized, timeLimit);
```

On exit from the function, the `authorized` field is set to TRUE if the OSP provider authorised the call, and the `timeLimit` field specifies the maximum call duration (in seconds). If `authorized` field is FALSE then the call requires must be denied.

As always, a return value of 0 indicates that the function succeeded. Any other value indicates that an error has occurred.

6.7 Ending a call

A call can be ended at any time by simply destroying the associated `OpalOSP::Transaction`. However, there are many statistics that an OSP provider can collect on a per-call basis, so it is useful for endpoints to provide this information if possible before the call is closed. There are several member functions on `OpalOSP::Transaction` for this purpose:

```
void CallConnect(const PTime & connectTime)  
Sets the time at which the call answered, which allows the correct call connect time to be reported to the OSP provider when the transaction is destroyed.
```

```
void CallStatistics(  
    unsigned lostSentPackets,  
    signed lostFractionSent,  
    unsigned lostReceivedPackets,  
    signed lostFractionReceived  
);
```

Stores the call statistics for reporting to the OSP provider at call end. This function can be called multiple times as only the last set of statistics will be reported.

```
int CallEnd(unsigned callDuration)
```

Sets the call duration to the specified value, sends all of the call information to the OSP provider, and closes the transaction. There is a second version of this function with no arguments that calculates the call duration from the absolute time reported to the `CallConnect` function.

```
void SetEndReason(int _endReason)
```

This function is used only if the call ends for some reason other than normal call clearing. Calling this function will block the normal reporting of statistics and cause the end reason to be reported as required.

6.8 Example 1 – outgoing call

An outgoing call uses the following sequence of functions:

The following code is performed during the application initialization:

```
// create the OSP provider
OpalOSP::Provider * ospProvider = new OpalOSP::Provider;

// open the provider using default filenames
int status;
if ((status = ospProvider->Open(
    "http://osptestserver.transnexus.com:1080/osp"
)) != 0) {
    cout << "error: cannot open OSP provider - " << status << endl;
    return;
};
```

The following is performed to authorize a new outgoing call:

```
// create a transaction
OpalOSP::Transaction * ospTransaction = new OpalOSP::Transaction;

// open the transaction
int status;
if ((status = ospTransaction->Open(*ospProvider)) != 0) {
    cout << "error: cannot open OSP transaction - " << status << endl;
    delete ospTransaction;
    return;
}

// declare the authorisation information structure
OpalOSP::Transaction::AuthorisationInfo authInfo;

// TODO: populate authInfo

unsigned int numberOfDestinations = 1;
if (status = ospTransaction->Authorise(authInfo,
    numberOfDestinations)) != 0) {
    cout << "error: cannot authorise call- " << status << endl;
    delete ospTransaction;
    return;
}

// check there are destinations
if (numberOfDestinations == 0) {
    cout << "error: no destinations available" << endl;
}
```

```
    delete ospTransaction;
    return;
}

// declare the destination information structure
OpalOSP::Transaction::DestinationInfo destInfo;

// get first (and only) destination
if ((status = ospTransaction->GetFirstDestination(destInfo)) != 0) {
    cout << "error: cannot get destination- " << status << endl;
    delete ospTransaction;
    return;
}

// insert token into SETUP
H225_ArrayOf_ClearToken & clearTokens = setup.m_clearTokens;
destInfo.InsertToken(clearTokens);

// TODO: populate remainder of destInfo fields into SETUP
```

When the call connects, the following is performed:

```
// set the call connect time to now
ospTransaction->CallConnect(PTime());
```

When the call completes, the following is performed:

```
// set the call duration and end the call
ospTransaction->CallEnd();
delete ospTransaction;
```

6.9 Example 2 – incoming call

An incoming call uses the following sequence of functions:

The following code is performed during the application initialization:

```
// create the OSP provider
OpalOSP::Provider * ospProvider = new OpalOSP::Provider;

// open the provider using default filenames
int status;
if ((status = ospProvider->Open(
    "http://osptestserver.transnexus.com:1080/osp"
)) != 0) {
    cout << "error: cannot open OSP provider - " << status << endl;
    return;
};
```

The following is performed to validate a new incoming call:

```
// create a transaction
OpalOSP::Transaction * ospTransaction = new OpalOSP::Transaction;
```

```
// open the transaction
int status;
if ((status = ospTransaction->Open(*ospProvider)) != 0) {
    cout << "error: cannot open OSP transaction - " << status << endl;
    delete ospTransaction;
    return;
}

// declare the validation information structure
OpalOSP::Transaction::ValidationInfo valInfo;

// get the token from the SETUP
const H225_ArrayOf_ClearToken & clearTokens = setup.m_clearTokens;
calInfo.ExtractToken(clearTokens);

// TODO: populate valInfo...

BOOL authorized;
unsigned timeLimit;
if (status = ospTransaction->Validate(valInfo,
                                     authorized,
                                     timeLimitauthInfo,
                                     )) != 0) {
    cout << "error: cannot validate call- " << status << endl;
    delete ospTransaction;
    return;
}

// check call is authorised
if (authorised == 0) {
    cout << "call not authorised" << endl;
    delete ospTransaction;
    return;
};
```

The code for call connection and termination is the same as for an outgoing call

7. Helper functions

OpenH323 provides several help functions that convert from various OpenH323 data types into the string format required by the Transnexus OSP toolkit. These functions are as follows:

```
PString OpalOSP::TransportAddressToOSPString(  
    const H323TransportAddress & taddr // H323 transport address  
);  
  
PString OpalOSP::TransportAddressToOSPString(  
    const H225_TransportAddress & taddr // H323 transport address  
);  
  
BOOL OpalOSP::ConvertAliasToOSPString(  
    const H225_AliasAddress & alias, // alias address to decompose  
    int & format, // OSP format type  
 // (see OSPE_NUMBERING_FORMAT)  
    PString & str // string component  
);  
  
PString OpalOSP::IpAddressToOSPString(  
    const PIPSocket::Address & addr  
)
```

8. OSP Sample Program

A sample program has been created to demonstrate the basic principles of OSP operation, and the OpenH323 OSP interface. This program can be found in the `openh323/samples/ospsample` directory, and provides a simple command line interface for performing common OSP functions.

```
OSP Test Program 1.0alpha
Local address is 10.0.2.13
Command ? help
help
status
localaddr ipaddress
    set local IP address
ospserver url [privkey pubkey cakey]
    set OSP server (i.e. http://osptestserver.transnexus.com:1080/osp)
call srcnum dstnum [callid]
    make a new outgoing call
connect
    set connect time for a call
hangup
    hangup the current call
quit
    exit program
exit
    exit program
Command ? ospserver http://osptestserver.transnexus.com:1080/osp
Transaction::Open returned error code = 0
Command ? call 6124354666 14157773456
Opening new transaction...returned 1 routeDestinations
First destination:
  Dest number: 14157773456
  Device:      [216.162.35.38]
  Destination: (empty)
  Time limit:  14400
  Call ID:
    40 73 ec 48 cd f1 18 10  92 bf  0  d 61 2a 6f 55  @s.H.....a*oU
  Token:
    30 82  4 1b  6  9 2a 86  48 86 f7  d  1  7  2 a0  0.....*.H.....
    82  4  c 30 82  4  8  2   1  1 31  e 30  c  6  8  ...0.....1.0...
...lines deleted...
    d4 18 f8 bd d8  b 98 62  bf 2b  b 94 ba e7 d3 a9  .....b.+.....
    10 79 b3 b2 38 d0 2b b8  48 e6 de 2f 29 60  0  .y..8.+H..)\`.
Command ? connect
Connect time set
Command ? hangup
Call hungup
Command ? exit
Exiting
```