OpenSRS API Integration for XML

Table of Contents

Introduction

Welcome to the OpenSRS API Integration guide for the registration and management of domain names and domain related services.

Using this document, you can provide OpenSRS functionality to your customers by integrating API commands into your website that can send XML requests to OpenSRS over HTTPS Post. You can also use API commands to run queries or automate tasks you would otherwise perform manually using the OpenSRS Reseller Web Interface (RWI).

You can leverage the API to allow registrants to view and update their contact details, nameservers, transfer auth codes, domain locking, whois privacy, and username and password settings. Additionally, the API can be used to perform domain availability lookups, provide domain search suggestions and aftermarket domain lookups, as well as process registrations, transfers, and renewals.

Registrants are set up with a profile in which multiple domains can be stored. A profile is defined by a username, password, and one of the domains in the profile. When requesting information or updating domain information, an authorization cookie needs to be requested first using a domain in the profile and the username and password for the profile. The cookie is then submitted with the request to look up or modify a domain's details, as well as when changing name servers assigned to a registered domain.

Purpose

The method of sending commands to OpenSRS using HTTPS Post supports communication between a client process and the OpenSRS system. This document describes how to formulate the XML commands and how to use HTTPS Post to send the XML commands to OpenSRS.

Using this document as a reference, you can use any programming language to write an implementation that supports this communication. If you are writing your own implementation, please refer to the "Design Considerations" section.

The process for executing commands consists of formulating the command in XML and sending it to OpenSRS via HTTPS Post. This guide outlines the basic method for creating the XML in a format usable by OpenSRS and includes some language-specific methods.

The protocol assumes that the client process that is requesting an action waits for a result from the server process in response to the requested action. The protocol does not support session tracking.

Audience

This document assumes that you are familiar with XML document design and the methods for sending data via HTTPS Post.

Using HTTPS Post to Access OpenSRS

The OpenSRS server supports SSL encryption and handles XML posts from OpenSRS authorized Resellers. The OpenSRS server:

- Applies IP address authentication to all HTTPS Post requests.
- Authenticates the user by verifying the username and MD5 signature of the XML, signed using the Resellers private key (generated in the RWI) in the request header.

OpenSRS uses XML, because it is a more structured and manageable approach than using Name Value Pairs.

Authenticating with the OpenSRS API

To authenticate against the OpenSRS API service, you need your reseller username, and a private key that you can generate from the Reseller Web Interface. (In the **Profile Management** section, click **Generate New Private Key**).

For the Live environment, the IP that connects to the API service must be added to an acceptance list. In the **Profile Management** section of the Reseller Web Interface, click **Add IPs for Script/API Access** and add the IP address to the list of allowed addresses**.**

Testing environment

OpenSRS provides a full test environment to assist with developing and testing API integration. The Horizon test environment duplicates the functionality of the Live environment, and is connected to the test environments of all the registry platforms.

Transactions on Horizon are not real – in other words, if you register a domain using Horizon, it is registered on the test environment of the registry in question, but the domain does not resolve. Your Horizon account is funded with money to simulate payment for transactions you perform. If you required additional testing funds, you can contact Reseller Support.

The Test environment API also requires a private key, but unlike the Live environment, it doesn't require you to authorize your IP address.

Connection Information

Live production environment

Server: rr-n1-tor.opensrs.net/

Port: 55443

OT&E test environment

Server: horizon.opensrs.net/

Port: 55443

MD5 authentication

The MD5 Signature provides the authentication required by OpenSRS. The process involves two steps:

1. Obtain an MD5 signature of the XML Content and the Private Key.
   Note: The XML and the Private Key are concatenated.
2. Perform another MD5 of the signature from Step 1 with the Private Key.
   Note: The MD5 Signature from Step 1 and the Private key are concatenated.

See the section"Writing Your Own Client" for examples of adding an MD5 Signature and creating the XML packet.

Construction of the POST Data

The header for the POST data should have the following format. The items in *italics* should be replaced by the user- and command-specific information.

```
POST / HTTP/1.0

Content-Type: text/xml

X-Username: OpenSRS Username

X-Signature: MD5 Signature

Content-Length: Length of XML Document
```

Following this header should be one blank line followed by the XML document that contains the OpenSRS command data. The header combined with the XML makes up the packet that is sent to OpenSRS to execute your command. This packet is what is sent to the server and port listed above, depending on your environment, to execute the command.

For more information regarding the transmission of data over HTTPS, refer to this document: http://www.ietf.org/rfc/rfc2616.txt

Response Messages

The process returns a response message to a client in answer to an action that was executed on its behalf. Responses contain data that is appropriate for the action that was executed. In some cases, this may be simple strings; in other cases, this may be lists of information. This response is returned in the form of an XML document.

Common fields

There are a few common fields that all responses share, regardless of the action to which they are responding. The following fields comprise a standard response message. Some actions only use standard response messages.

Parameter name	Definition/Value
protocol	The protocol that is being used (XCP).
action	In the case of responses, this is always **REPLY**.
response_code	Response code (meaning is action-specific).
response_text	Response text (meaning is action-specific).
is_success	Indicates whether the command was successful. Returns **0** if not successful and **1** if the action was successful.

Optional fields

In addition, the structure may contain the following fields, depending on the specific action that was requested.

Parameter name	Definition/Value
attributes	A hash that contains any specific parameters or attributes to be sent along with the action request.

Troubleshooting HTTPS Post

401 Authentication Error

- Check that you are using the correct Private Key for the right system – Horizon or Production.
- Check that you have the correct RSP username.
- Check that the IP address of the machine transmitting the data to the OpenSRS server is in your list of allowed IP addresses in the Reseller Web Interface.

If the above checks are correct, the problem is with the MD5:

- Ensure that you have concatenated the XML content and the Private Key.
- Ensure that you have performed an MD5 twice. See the MD5 section for more information.
- Ensure that your HTTP Post implementation is not adding any extra information. Some implementations of HTTP Post add a NULL to the end of the HTTP Request. This is reflected in the MD5 and causes an authentication error.

If you are still not connecting properly, check the result of the MD5 Hash:

- Some MD5 algorithms put the MD5 hash in uppercase. Make sure that the result is in lowercase before sending it to OpenSRS.
- Some MD5 algorithms need to convert the string to bytes before generating the hash. Make sure this is done properly. You can test your script by performing an MD5 on the following text:

  Text: ConnecttoOpenSRSviaSSL

  MD5 Result: e787cc1d1951dfec4827cede7b1a0933

Invalid XML Response

Make sure you are sending the XML. The XML used in the MD5 is only for authentication purposes. You must also send the XML as part of the content header.

OpenSRS protocol

Purpose

The purpose of the OpenSRS Protocol (OPS) is to support the passing of messages between processes in the OpenSRS Architecture.

Design considerations

The OPS uses XML-based messages. By using a meta language to describe the protocol, it can remain programming language-neutral.

Any appropriate transport mechanism can be used to pass OPS protocol messages between processes, provided the message is reassembled precisely on the receiving end. Discussion of transport mechanisms is outside the scope of this document.

Process flow

When a process sends a message to another process, it generates an OPS protocol message that represents the data being sent. The OPS message is then passed to the other process, which can act on it as required and return any information that is needed (again, as a protocol message).

The generation of protocol messages is done with an API that is written for the particular programming language being used (for example, Perl or C++). The API handles the details of encoding and decoding protocol messages from and to appropriate data structures for the language being used.

Protocol message structure

DTD

An OPS protocol message is an XML document, which satisfies the following Document Type Declaration (DTD).

ops.dtd

```
<!--
Opensrs Protocol Suite (OPS) v 0.1
(C) Copyright 2000, Tucows Inc.
All Rights Reserved

OPS Message Definition

-->
<!-- Envelope -->
<!ELEMENT OPS_envelope (header,body)>
<!-- header part -->
<!ELEMENT header (version)>
<!-- body part -->
<!ELEMENT body (data_block)>
<!-- data block -->
<!ELEMENT data_block (dt_assoc | dt_array |
                      dt_scalar| dt_scalarref)>
<!-- data types -->
<!ELEMENT dt_assoc (dt_assoc | dt_array|
                    dt_scalar | dt_scalarref|
                    (item)*)>
<!ELEMENT dt_array (dt_assoc | dt_array |
                    dt_scalar | dt_scalarref |
                    (item)*)>
<!ELEMENT dt_scalar (#PCDATA | dt_assoc | dt_array |
                     dt_scalar | dt_scalarref)*>
<!ELEMENT dt_scalarref (#PCDATA | dt_assoc | dt_array |
                        dt_scalar | dt_scalarref)*>
<!ELEMENT item (#PCDATA |
                   dt_assoc | dt_array |
                   dt_scalar | dt_scalarref)*>
```

```
<!ATTLIST item
       key   CDATA #REQUIRED
       class CDATA #IMPLIED         >
<!ELEMENT version (#PCDATA)>
<!-- document information entities -->
<!ENTITY company 'Tucows'>
<!ENTITY copyright '2000, Tucows'>
```

Description

The preamble of an OPS message must contain encoding and DOCTYPE information specifying the correct DTD (ops.dtd).

```
<?xml version='1.0' encoding='UTF-8' standalone='no' ?>
<!DOCTYPE OPS_envelope SYSTEM 'ops.dtd'>
```

The start and end of an OPS message 'envelope' are delimited by <OPS_envelope> and </OPS_envelope> tags.

```
<OPS_envelope>
  ?
  ?
  ?
</OPS_envelope>
```

A message consists of two basic parts: a header and a body. The header is delimited by <header> and </header> and contains information that is related to protocol transport. Currently the OPS protocol version is supported. It is delimited by <version> and </version> tags.

The body is delimited by <body> and </body> tags. Currently the body can only contain one component, which is the 'data_block'. The data block contains an XML representation of a data structure, which in turn contains the information being passed by the process. The data block is delimited by <data_block> and </data_block> tags.

Example

```
<OPS_envelope>
  <header>
?
```

```
  </header>
  <body>
    <data_block>
?
    </data_block>
  </body>
</OPS_envelope>
```

The data structure within the data block is a represented according to data type. The following data types are supported:

Data type	Definition/Value
associative array (hash)	<dt_assoc></dt_assoc>
array	<dt_array></dt_array>
array or associative array elements	<item key='' [class='']></item> Where key is the name of the key used to access the element. For an array, it is an integer (0...N). For an associative array, it is a string. Example for assoc array: <dt_assoc> <item key='firstname'>Tom</item> <item key='lastname'>Jones</item> </dt_assoc> Example for array: <dt_array> <item key='0'>example1.com</item> <item key='1'>example2.com</item> </dt_array> The class attribute is optional. If specified, it refers to a class name that can be used to reconstruct the data into an object when decoded. The actual details of this are implementation specific. For example, for a Perl implementation the class name would be used to bless the underlying associative array (hash), back into an object. Example for class: <dt_array> <item key='0' class='myClass'>

Data type	Definition/Value
	<dt_assoc> <item key='firstname'>Tom</item> <item key='lastname'>Jones</item> </dt_assoc> </item> </dt_array>
scalars (for example, strings, integers)	<dt_scalar></dt_scalar>
scalar reference	<dt_scalarref></dt_scalarref>

Data types can be arbitrarily nested. For example, you could have an array of an array of associative arrays. Any complex data type can be represented, provided the base data type is an array or an associative array.

OPS message examples

Example 1

This example shows an OPS message that encodes an associative array that contains an array of values:

```
<?xml version='1.0' encoding='UTF-8' standalone='no' ?>
<!DOCTYPE OPS_envelope SYSTEM 'ops.dtd'>
<OPS_envelope>
  <header>
    <version>1.0</version>
  </header>
  <body>
    <data_block>
      <dt_assoc>
        <item key='domain_list'>
          <dt_array>
            <item key='0'>ns1.example.com</item>
            <item key='1'>ns2.example.com</item>
            <item key='2'>ns3.example.com</item>
          </dt_array>
      </dt_assoc>
    </data_block>
  </body>
</OPS_envelope>
```

Example 2

This example shows an OPS message that encodes an associative array that contains other associative arrays:

```
<?xml version='1.0' encoding='UTF-8' standalone='no' ?>
<!DOCTYPE OPS_envelope SYSTEM 'ops.dtd'>
<OPS_envelope>
  <header>
```

```
    <version>1.0</version>
  </header>
  <body>
    <data_block>
      <dt_assoc>
          <dt_assoc>
            <item key='owner'>
              <dt_assoc>
                <item key='first_name'>Tom</item>
                <item key='last_name'>Jones</item>
              </dt_assoc>
            </item>
            <item key='tech'>
              <dt_assoc>
                <item key='first_name'>Anne</item>
                <item key='last_name'>Smith</item>
              </dt_assoc>
            </item>
          </dt_assoc>
        </item>
      </dt_assoc>
    </data_block>
  </body>
</OPS_envelope>
```

Example 3

This example shows an OPS message that encodes a scalar value:

```
<?xml version='1.0' encoding='UTF-8' standalone='no' ?>
<!DOCTYPE OPS_envelope SYSTEM 'ops.dtd'>
<OPS_envelope>
  <header>
    <version>1.0</version>
```

```
  </header>
  <body>
    <data_block>
      <dt_scalar>Tom Jones</dt_scalar>
   </data_block>
  </body>
</OPS_envelope>
```

Semantic validity and API implementation

Any API implementation that encodes/decodes OPS messages should be reflexive. That means that, if a message is encoded and then decoded, the results of that decoding and the original message should be identical. In other words:

```
MSG A := decode (encode (MSG A) )
```

Alternatively,

```
let MSG A' := encode (MSG A)
              then MSG A  := decode (MSG A')
```

This restriction can be relaxed when it comes to the ordering of array elements and associative array elements in the OPS message. As long as the semantic meaning of the elements is preserved (semantic equivalence), the actual textual ordering in the XML stream is not important.

This is because the XML data structure representation uses element numbers (in the case of array) and field names (in the case of associative arrays) to represent the data stream, and this information can be used by the API to reconstruct the XML representation back into an in-memory data structure.

This approach supports languages in which the ordering of elements for in-memory data structures is not deterministic., for example, Perl hashes (associative arrays), where the physical ordering of elements within the hash is not guaranteed.

Example 1

The following two data blocks are considered semantically the same. Notice that the order of elements is not the same, but the key values allow the data blocks to be decoded into the same memory structure (array).

MSG A

```
    <data_block>
```

```
      <dt_assoc>
        <item key='domain_list'>
          <dt_array>
            <item key='0'>ns1.example.com</item>
            <item key='1'>ns2.example.com</item>
            <item key='2'>ns3.example.com</item>
          </dt_array>
      </dt_assoc>
    </data_block>
```

MSG B

```
    <data_block>
      <dt_assoc>
        <item key='domain_list'>
          <dt_array>
            <item key='2'>ns3.example.com</item>
            <item key='1'>ns2.example.com</item>
            <item key='0'>ns1.example.com</item>
          </dt_array>
      </dt_assoc>
    </data_block>
```

Example 2

The following two data blocks are considered semantically the same. Notice that the order of fields is not the same, but the key values allow the data blocks to be decoded into the same memory structure (associative array).

MSG A

```
    <data_block>
      <dt_assoc>
         <item key='first_name'>Tom</item>
         <item key='last_name'>Jones</item>
      </dt_assoc>
    </data_block>
```

MSG B

```
    <data_block>
      <dt_assoc>
         <item key='last_name'>Jones</item>
         <item key='first_name'>Tom</item>
      </dt_assoc>
    </data_block>
```

Writing Your Own Client

This section contains an explanation of the OpenSRS client/server data exchange. This information is useful if you want to write your own client.

XML client protocol fundamentals

In our XML Client Protocol (XCP), the sender of a message (request or reply) must always precede the message with the header 'Content-Length: X', where 'X' is the number of bytes in the actual message (without the header). This header must be followed by a carriage return and line feed combination. Counted bytes only occur on the first non-blank line.

Example

```
Content-length: 55\015\012    # carriage return/line feed
# blank lines are ignored
<?xml version='1.0' encoding='UTF-8' standalone='no' ?>
<!DOCTYPE OPS_envelope SYSTEM 'ops.dtd'>
<OPS_envelope>
  <header>
    <version>0.9</version>
  </header>
  <body>
    <data_block>
[ etc ]
```

XCP allows empty space to be placed in the XML message, as long as the XML is still valid, so empty lines or end-of-line characters may be inserted (though they must be counted in the byte count).

For more information regarding the transmission of data over HTTPS, refer to this document: http://www.ietf.org/rfc/rfc2616.txt

MD5 Examples

The following examples show how to add an MD5 Signature and create an XML packet for the various client languages.

PERL

use Digest::MD5 qw/md5_hex/;

md5_hex(md5_hex($xml, $private_key),$private_key)

PHP

md5(md5($xml.$private_key).$private_key);

VB6

Using the "di_MD5DLL.dll" from DI Management

This free cryptographic software code was written or adapted in Visual Basic and ANSI C by David Ireland.

http://www.di-mgt.com.au/crypto.html#MD5

```
Private Declare Function MakeMD5Digest Lib "di_MD5DLL.dll" _
    (ByVal sData As String, ByVal sDigest As String) As Long
Public Function MD5Encode(ByVal sData As String) As String
    Dim iRet As Long
    Dim sDigest As String
    ' Set sDigest to be 32 chars
    sDigest = String(32, " ")
    iRet = MakeMD5Digest(sData, sDigest)
    MD5Encode = Trim(sDigest)
End Function


MD5Encode(MD5Encode(pXMLDoc.xml & strPrivateKey) & strPrivateKey)
```

VB .NET

```
Public Function cMD5(ByVal str As String) As String
        'Must have Imports System.Web.Security in General Declarations
```

```
        Dim Hash As String =
FormsAuthentication.HashPasswordForStoringInConfigFile(str, "MD5")
        Return Hash.ToLower
    End Function
```

```
cMD5(cMD5(str & PRIVATE_KEY) & PRIVATE_KEY))
```

JAVA

```
protected String md5Sum(String str) {
   String sum = new String();
     try {
       MessageDigest md5 = MessageDigest.getInstance("MD5");
       sum = String.format("%032x", new BigInteger(1,
md5.digest(str.getBytes())));
       } catch (Exception ex) {
       }
     return sum;
}
public String getSignature(String xml) {
     return md5Sum(md5Sum(xml + privateKey) + privateKey);
   }
```

Coding Examples

Perl example

```
#!/usr/bin/perl
use strict;
use warnings;
##########################################################
# Example script that takes registrant credentials in
# from CGI variables and pulls all info about a domain.
# A cookie is requested via a subroutine and is the
```

```
# first of two calls to the API. The second is the query
# for the domain's information.
#########################################################
## Reseller Configuration Variables
my $rspusername = 'username'; # Your OpenSRS Reseller User Name
my $private_key='privatekey'; # Your Private Key Generated In The RWI
## Connection Location:
my $REMOTE_HOST = "horizon.opensrs.net:55443";
## CGI Variable Handling Definitions
use vars qw(%in $cgi);
( %in, $cgi) = ();
# Required Perl Modules
use CGI ':cgi-lib'; # Optional To CGI Handling
use HTTP::Request::Common;
use Digest::MD5 qw/md5_hex/;
use LWP::UserAgent;
use Data::Dumper;
use XML::Simple; # Can Be Replaced With Preferred XML Parser
# NOTE: LWP requires that Crypt::SSLeay is also installed for an HTTPS
call
# Define Data And Connection Handlers And CGI Handling
my $xmltohash = new XML::Simple; # Change To XML Parser Requires Update
Here
my $ua = LWP::UserAgent->new;
# Get, Read, Parse Registrant Credentials From CGI Submitted Form
$cgi = $ENV{SCRIPT_NAME};
%in = ();
ReadParse(\%in);
my $domain = $in{domain}; # Registrant's Domain being queried
my $username = $in{username}; # Registrant's Profile User Name
my $password = $in{password}; # Registrant's Profile Password
# Set Up Output Acceptance For Browser
print "Content-type: text/html\n\n";
```

```
# Subroutine Call To Authenticate Registrant And Generate Cookie
my $cookie = create_cookie($domain,$username,$password);
# XML Call For Domain Info Defined
my $xml = "<?xml version='1.0' encoding='UTF-8' standalone='no' ?>
   <!DOCTYPE OPS_envelope SYSTEM 'ops.dtd'>
   <OPS_envelope>
      <header>
         <version>0.9</version>
      </header>
      <body>
         <data_block>
         <dt_assoc>
            <item key='protocol'>XCP</item>
            <item key='object'>DOMAIN</item>
            <item key='action'>GET</item>
            <item key='cookie'>$cookie</item>
            <item key='attributes'>
               <dt_assoc>;
                  <item key='type'>all_info</item>
               </dt_assoc>
            </item>
            </dt_assoc>
         </data_block>
      </body>
   </OPS_envelope>";
# Connection And Post To OpenSRS API For Domain Info
my $request = POST (
   "https://$REMOTE_HOST",
   'Content-Type' => 'text/xml',
   'X-Username' => $rspusername,
   'X-Signature' => md5_hex(md5_hex($xml, $private_key), $private_key),
   'Content' => $xml);
```

```
my $response = $ua -> request($request);
## Response Handling
#  Output from the API is provide in XML.
#  Two examples of outputing the response from the API follow
# XML Output Of Response
print "XML Output\n\n";
print Dumper($response->{'_content'});
# Conversion Of XML To Perl Hash
# Change To XML Parser Requires Update Here
my $hashresponse = $xmltohash->XMLin($response->{'_content'});
# Perl Hash Output Of Response
print "\n\nHash Output\n\n";
print Dumper($hashresponse);
# Subroutine To Generate Authentication Cookie From Registrant
Credentials
sub create_cookie {
   # Take In Registrant Credentials
   my ($domain, $username, $password) = @_;
   # Define XML For Authentication Cookie Generation
   my $xml = "<?xml version='1.0' encoding='UTF-8' standalone='no' ?>
      <!DOCTYPE OPS_envelope SYSTEM 'ops.dtd'>
      <OPS_envelope>
         <header>
         <version>0.9</version>
         </header>
         <body>
            <data_block>
            <dt_assoc>
               <item key='protocol'>XCP</item>
               <item key='object'>cookie</item>
               <item key='action'>set</item>
               <item key='attributes'>
                  <dt_assoc>
```

```
                    <item key='reg_password'>$password</item>
                    <item key='domain'>$domain</item>
                    <item key='reg_username'>$username</item>
                  </dt_assoc>
               </item>
            </dt_assoc>
            </data_block>
         </body>
      </OPS_envelope>";
   # Connection And Post To OpenSRS API For Authentication Cookie
   my $request = POST (
      "https://$REMOTE_HOST",
      'Content-Type' => 'text/xml',
      'X-Username' => $rspusername,
      'X-Signature' => md5_hex(md5_hex($xml, $private_key),
$private_key),
      'Content' => $xml);
   my $response = $ua -> request($request);
   # Conversion Of XML To Perl Hash
   # Change To XML Parser Requires Update Here
   my $hashresponse = $xmltohash->XMLin($response->{'_content'});
   # Return Of Cookie Value From Hash
   return ($hashresponse->{'body'}->{'data_block'}->{'dt_assoc'}-
>{'item'}->{'attributes'}->{'dt_assoc'}->{'item'}->{'cookie'}-
>{'content'});
}
```

PHP example

```
<html>
<body>
<?php
$username = "RSPUsername";
$private_key = "PrivateKey";
```

```
$xml = '<?xml version=\'1.0\' encoding="UTF-8" standalone="no" ?>
<!DOCTYPE OPS_envelope SYSTEM "ops.dtd">
<OPS_envelope>
 <header>
  <version>0.9</version>
  </header>
 <body>
  <data_block>
   <dt_assoc>
    <item key="object">DOMAIN</item>
    <item key="action">LOOKUP</item>
    <item key="protocol">XCP</item>
    <item key="attributes">
     <dt_assoc>
      <item key="domain">example.com</item>
     </dt_assoc>
    </item>
   </dt_assoc>
  </data_block>
 </body>
</OPS_envelope>';

$signature = md5(md5($xml.$private_key).$private_key);
$host = "horizon.opensrs.net";
$port = 55443;
$url = "/";
$header = "";
$header .= "POST $url HTTP/1.0\r\n";
$header .= "Content-Type: text/xml\r\n";
$header .= "X-Username: " . $username . "\r\n";
$header .= "X-Signature: " . $signature . "\r\n";
$header .= "Content-Length: " . strlen($xml) . "\r\n\r\n";
# ssl:// requires OpenSSL to be installed
```

```
$fp = fsockopen ("ssl://$host", $port, $errno, $errstr, 30);

echo "<pre>";

if (!$fp) {
  print "HTTP ERROR!";
} else {
  # post the data to the server
  fputs ($fp, $header . $xml);
  while (!feof($fp)) {
    $res = fgets ($fp, 1024);
    echo htmlEntities($res);
  }
  fclose ($fp);
}

echo "</pre>";
?>
</body>
```

VB6 example

Thanks to Serguei Seleznev from Softcom Technology for developing this script.

This Script is using di_MD5DLL.dll from DI Management. This is free cryptographic software code that David Ireland has written or adapted in Visual Basic and ANSI C.

http://www.di-mgt.com.au/crypto.html#MD5

```
Dim ErrNumber As Long
Dim ErrDescription As String
Const strUserName = "RSP_USERNAME"
Const strPrivateKey = "Your Private Key Go’s here”
Const strURL = "https://horizon.opensrs.net:55443"
```

```
Private Declare Function MakeMD5Digest Lib "di_MD5DLL.dll" _
    (ByVal sData As String, ByVal sDigest As String) As Long

Sub Main()
' POST XML document using VB6 and MSXML4(has to be installed)

    Dim DocToSend As MSXML2.DOMDocument
    Dim pFileRequest As String
    Dim strXML As String

On Error GoTo err_Main
    pFileRequest = App.Path & "\Sample.XML"
    strXML = ""

    Set DocToSend = New MSXML2.DOMDocument
    If Not ReadXMLDocument(pFileRequest, DocToSend) Then
        MsgBox "Cannot read " & pFileRequest & vbCrLf & _
            ErrNumber & ", " & ErrDescription, vbCritical
        Exit Sub
    End If

    If SendRequestXML(DocToSend, strXML, strURL) Then
        MsgBox "Response has come. " & strXML
        ' here you may save to file or reload in DOMdocument to parse
    Else
        MsgBox "Cannot send " & DocToSend.xml & vbCrLf & _
            ErrNumber & ", " & ErrDescription, vbCritical
        Exit Sub
    End If
    Set DocToSend = Nothing
    Exit Sub
err_Main:
```

```
    ErrNumber = Err.Number
    ErrDescription = "Run-time ERROR in Main. " & Err.Description
    MsgBox "Error " & ErrNumber & ", " & ErrDescription, vbCritical
End Sub

Private Function SendRequestXML( _
    ByRef pXMLDoc As MSXML2.DOMDocument, _
    ByRef pcTmp As String, _
    ByVal pstrURL As String _
    ) As Boolean

    Dim xmlHttp As MSXML2.XMLHTTP40

On Error GoTo err_SendRequestXML
    SendRequestXML = False

    Set xmlHttp = New MSXML2.XMLHTTP40
    xmlHttp.Open "POST", pstrURL, False ' False - syncronous mode
    xmlHttp.setRequestHeader "Content-Type", "text/xml"
    xmlHttp.setRequestHeader "X-Username", strUserName
    xmlHttp.setRequestHeader "X-Signature",
MD5Encode(MD5Encode(pXMLDoc.xml & strPrivateKey) & strPrivateKey)
    xmlHttp.send pXMLDoc.xml

    pcTmp = xmlHttp.responseXML.xml
    Set xmlHttp = Nothing
    SendRequestXML = True

''    'You may try to use asyncronous post
''    xmlHttp.Open "POST", pstrURL, True
''    xmlHttp.setRequestHeader "Content-Type", "text/xml"
''    xmlHttp.setRequestHeader "X-Username", strUserName
```

```
''    xmlHttp.setRequestHeader "X-Signature",
MD5Encode(MD5Encode(pXMLDoc.xml & strPrivateKey) & strPrivateKey)
''    xmlHttp.send pXMLDoc.xml
''    PauseSeconds 1                      'wait
''    pcTmp = ""
''    If xmlHttp.readyState = 4 Then      'we got it
''        If xmlHttp.Status = 200 Then
''            pcTmp = xmlHttp.responseXML.xml
''            SendRequestXML = True
''        End If
''    Else                                ' let's wait for a while
''        PauseSeconds 3
''        If xmlHttp.readyState = 4 Then  ' check again
''            If xmlHttp.Status = 200 Then
''                cTmp = xmlHttp.responseXML.xml
''                SendRequestXML = True
''            End If
''        End If
''    End If

    Exit Function

err_SendRequestXML:
    If IsObject(xmlHttp) Then Set xmlHttp = Nothing

    ErrNumber = Err.Number
    ErrDescription = "Run-time ERROR in SendRequestXML. " &
Err.Description

    Err.Clear
End Function
```

```
Private Function ReadXMLDocument(ByVal pDocName As String, ByRef
pXMLDoc As MSXML2.DOMDocument) As Boolean


    On Error GoTo err_ReadXMLDocument


    ReadXMLDocument = False


    pXMLDoc.async = False

    pXMLDoc.resolveExternals = False    ' otherwise you must have a
refered DTD

    pXMLDoc.validateOnParse = False     ' in a same directory

    pXMLDoc.Load pDocName


    If pXMLDoc.parseError.errorCode = 0 Then

        ReadXMLDocument = True

    Else

        ErrNumber = pXMLDoc.parseError.errorCode

        ErrDescription = "Errors in " & pXMLDoc.parseError.url & ",
line " & pXMLDoc.parseError.Line & ", pos " &
pXMLDoc.parseError.linepos

        ErrDescription = ErrDescription & ". Error #" & ErrNumber & ".
" & pXMLDoc.parseError.reason

    End If


Exit Function

err_ReadXMLDocument:

    ErrNumber = Err.Number

    ErrDescription = "Run-time ERROR in ReadXMLDocument " &
Err.Description & " for " & pDocName

    Err.Clear

End Function


' contents of file Sample.XML
```

```
'<?xml version="1.0" encoding="UTF-8" standalone="no"?>
'<!DOCTYPE OPS_envelope SYSTEM "ops.dtd">
'<OPS_envelope>
'   <header>
'      <version>0.9</version>
'   </header>
'   <body>
'      <data_block>
'         <dt_assoc>
'            <item key="object">DOMAIN</item>
'            <item key="attributes">
'               <dt_assoc>
'                  <item key="domain">MyDomainToLookup.com</item>
'               </dt_assoc>
'            </item>
'            <item key="protocol">XCP</item>
'            <item key="action">LOOKUP</item>
'         </dt_assoc>
'      </data_block>
'   </body>
'</OPS_envelope>

Public Function MD5Encode(ByVal sData As String) As String
    Dim iRet As Long
    Dim sDigest As String
    ' Set sDigest to be 32 chars
    sDigest = String(32, " ")
    iRet = MakeMD5Digest(sData, sDigest)
    MD5Encode = Trim(sDigest)
End Function
```

VB .NET example

```
Dim mypost As New OpenSRS_XMLPOST
        TextBox1.Text = mypost.sendPost(RESPONSE_TEXT)
------------------
Imports System.Web.Security
Public Class OpenSRS_XMLPOST
    Public Const URL_BASE = "https://horizon.opensrs.net:55443"
    Public Const RSP_USERNAME As String = "RSP USERNAME"
    Public Const PRIVATE_KEY = "Enter Private Key"
    Public Function sendPost(ByVal str As String)
        Dim myHttpWebRequest As New System.Net.WebClient
        myHttpWebRequest.Headers.Add("Content-Type", "text/xml")
        myHttpWebRequest.Headers.Add("X-Username", RSP_USERNAME)
        myHttpWebRequest.Headers.Add("X-Signature", cMD5(cMD5(str &
PRIVATE_KEY) & PRIVATE_KEY))
        Dim sendData As Byte() =
System.Text.Encoding.ASCII.GetBytes(str)
        Dim myHttpResponse As Byte() =
myHttpWebRequest.UploadData(URL_BASE, "POST", sendData)
        Return System.Text.Encoding.ASCII.GetString(myHttpResponse)
    End Function
    'Used to convert to MD5
    Public Function cMD5(ByVal str As String) As String
        'Must have Imports System.Web.Security in General Declarations
        Dim Hash As String =
FormsAuthentication.HashPasswordForStoringInConfigFile(str, "MD5")
        Return Hash.ToLower
    End Function
End Class
```

Java example

```
package net.client;
import javax.net.ssl.*;
import javax.net.SocketFactory;
import java.net.*;
```

```
import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;
import java.security.MessageDigest;
import java.util.Hashtable;
import java.math.BigInteger;
import org.apache.commons.httpclient.*;
import org.apache.commons.httpclient.methods.PostMethod;
import org.apache.commons.httpclient.protocol.*;
import com.pureload.task.api.TaskExecuteException;
public class SslClient {
private String privateKey;
private String  host;
private int port;
private String userName;
private Header [] headers = null;
public class MySSLSocketFactory implements SecureProtocolSocketFactory
{
private TrustManager[] getTrustManager() {
TrustManager[] trustAllCerts = new TrustManager[]{
new X509TrustManager() {
public java.security.cert.X509Certificate[] getAcceptedIssuers() {
return null;
}
public void checkClientTrusted(
    java.security.cert.X509Certificate[] certs, String authType) {
}
public void checkServerTrusted(
    java.security.cert.X509Certificate[] certs, String authType) {
}
}
};
```

```
return trustAllCerts;
}

public Socket createSocket(String host, int port) throws IOException,
UnknownHostException {
TrustManager[] trustAllCerts = getTrustManager();
try {
SSLContext sc = SSLContext.getInstance("SSL");
sc.init(null, trustAllCerts, new java.security.SecureRandom());
HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());
SocketFactory socketFactory =
HttpsURLConnection.getDefaultSSLSocketFactory();
return socketFactory.createSocket(host, port);
}
catch (Exception ex) {
throw new UnknownHostException("Problems to connect " + host +
ex.toString());
}
}
public Socket createSocket(Socket socket, String host, int port,
boolean flag) throws IOException, UnknownHostException {
TrustManager[] trustAllCerts = getTrustManager();
try {
SSLContext sc = SSLContext.getInstance("SSL");
sc.init(null, trustAllCerts, new java.security.SecureRandom());
HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());
SocketFactory socketFactory =
HttpsURLConnection.getDefaultSSLSocketFactory();
return socketFactory.createSocket(host, port);
}
catch (Exception ex) {
throw new UnknownHostException("Problems to connect " + host +
ex.toString());
}
```

```
}

public Socket createSocket(String host, int port, InetAddress
clientHost, int clientPort) throws IOException, UnknownHostException {

TrustManager[] trustAllCerts = getTrustManager();

try {

SSLContext sc = SSLContext.getInstance("SSL");

sc.init(null, trustAllCerts, new java.security.SecureRandom());

HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());

SocketFactory socketFactory =
HttpsURLConnection.getDefaultSSLSocketFactory();

return socketFactory.createSocket(host, port, clientHost, clientPort);

}

catch (Exception ex) {

throw new UnknownHostException("Problems to connect " + host +
ex.toString());

}

}

}

  public SslClient(String host, int port, String userName, String
privateKey) {

  this.host=host;

  this.port = port;

  this.userName = userName;

  this.privateKey = privateKey;

    }

protected String md5Sum(String str) {

   String sum = new String();

      try {

       MessageDigest md5 = MessageDigest.getInstance("MD5");

       sum = String.format("%032x", new BigInteger(1,
md5.digest(str.getBytes())));

        } catch (Exception ex) {
```

```
        }
      return sum;
}
public String getSignature(String xml) {
     return md5Sum(md5Sum(xml + privateKey) + privateKey);
   }
public String sendRequest(String xml) throws TaskExecuteException {
HttpClient client = new HttpClient();
client.setConnectionTimeout(60000);
client.setTimeout(60000);
String response = new String();
String portStr = String.valueOf(port);
Protocol.registerProtocol("https", new Protocol("https", new
MySSLSocketFactory(), port));
String signature = getSignature(xml);
String uri = "https://" + host + ":" + portStr + "/";
PostMethod postRequest = new PostMethod(uri);
postRequest.addRequestHeader("Content-Length",
String.valueOf(xml.length()));
postRequest.addRequestHeader("Content-Type", "text/xml");
postRequest.addRequestHeader("X-Signature", signature);
postRequest.addRequestHeader("X-Username", userName);
postRequest.setRequestBody(xml);
        System.out.println("Sending https request....."+postRequest.
toString());
try {
client.executeMethod(postRequest);
}
catch (Exception ex) {
throw new TaskExecuteException("Sending post got exception ", ex);
}
response = postRequest.getResponseBodyAsString();
headers = postRequest.getRequestHeaders();
```

```
return response;
}
public String getPrivateKey() {
return privateKey;
}
public void setPrivateKey(String privateKey) {
this.privateKey = privateKey;
}
public String getHost() {
return host;
}
public void setHost(String host) {
this.host = host;
}
public int getPort() {
return port;
}
public void setPort(int port) {
this.port = port;
}
public String getUserName() {
return userName;
}
public void setUserName(String userName) {
this.userName = userName;
}
public Header[] getHeaders() {
return headers;
}
public void setHeaders(Header[] headers) {
this.headers = headers;
}
```

```
  public static void main(String[] args) {
    String privateKey = "your_private_key";
    String userName = "your_user_name";
    String host="horizon.opensrs.net";
  int port = 55443;
    String xml=
"<?xml version='1.0' encoding='UTF-8' standalone='no' ?>"+
"<!DOCTYPE OPS_envelope SYSTEM 'ops.dtd'>"+
"<OPS_envelope>"+
 "<header>"+
  "<version>0.9</version>"+
  "<msg_id>2.21765911726198</msg_id>"+
  "<msg_type>standard</msg_type>"+
 "</header>"+
 "<body>"+
  "<data_block>"+
   "<dt_assoc>"+
    "<item key='attributes'>"+
     "<dt_assoc>"+
      "<item key='domain'>test-1061911771844.com</item>"+
      "<item key='pre-reg'>0</item>"+
     "</dt_assoc>"+
    "</item>"+
    "<item key='object'>DOMAIN</item>"+
    "<item key='action'>LOOKUP</item>"+
    "<item key='protocol'>XCP</item>"+
```

```
   "</dt_assoc>"+
  "</data_block>"+
 "</body>"+
"</OPS_envelope>";
    SslClient sslclient = new SslClient(host,port,userName,privateKey);
    try {
      String response = sslclient.sendRequest(xml);
      System.out.println("\nResponse is:\n"+response);
    }
    catch (TaskExecuteException e) {
      e.printStackTrace();
    }
  }
}
```

Data exchange

For each line of data passed, you must prepend the data with the length of the string packed in 'network' or big-endian order. In Perl, this is accomplished by:

```
$length = pack('n', length($data));
```

Where $data is the information you are going to send.

For example, assuming you have a socket SERVER already open to the server process, you could send data as follows:

```
print SERVER pack('n', length($data));
print SERVER $data;
```

Since you must always send the length of the string first, it will not work to simply telnet to the OpenSRS server and begin issuing commands.

Authentication handshake

The first step in communicating with the server process is the authentication handshake. This proceeds between the Reseller Client and Reseller Agent (server) as follows:

	Process	Description
1	Reseller Client	Initiates connection with Reseller Agent (server process) on a specific TCP/IP hostname:port. Horizon: horizon.opensrs.net:55000 Live: rr-n1-tor.opensrs.net:55000

2	Reseller Agent	Server sends an XCP 'check version' request.
		Perl Example (hash):
		`{`
		`'protocol' => 'XCP',`
		`'action' => 'check',`
		`'object' => 'version',`
		`'attributes' => {`
		`'sender' => 'OpenSRS SERVER',`
		`'version' => '$VERSION',`
		`'state' => 'ready'`
		`}`
		`}`
		The values of $VERSION could be something such as 'XML:0.1', which indicates the language spoken and the minimum version of the client required by this RSA. At this point, the only value for 'state' is 'ready'. Other states may be added in the future.

3	Reseller Client	Client responds with an XCP 'check version' response (where version is the client's protocol version.)
		Note: This number should not be changed. It allows for API changes and backward compatibility. If you change the version number of the client, results may be unpredictable.
		Perl Example (hash):
		`{`
		`'protocol' => 'XCP',`
		`'action' => 'check',`
		`'object' => 'version',`

	Process	Description

		'attributes' => {
		'sender' => 'OpenSRS CLIENT',
		'version' => 'XML:0.1',
		'state' => 'ready',
		}
		}
		The only difference here is the value of the sender attribute. Again, the only valid state at this point is 'ready'.

4	Reseller Client	Client sends user data for authentication. This is done using the XCP 'authenticate user' request.
		Note: As a Reseller, you have a password and a username. Do NOT send the password in this request, it is not needed. The current XML Perl Client actually sends the username in both the username and password fields. This is because the data packets are not encrypted at this stage of the transmission.
		Perl Example (hash):
		{
		'protocol' => 'XCP',
		'action' => 'authenticate',
		'object' => 'user',
		'attributes' => {
		'crypt_type' => '',
		'username' => '',
		'password' => ''
		}
		}
		The crypt_type can be either 'des' or 'blowfish'.

5	Reseller Agent	If authentication is successful, the Reseller Agent (server side), sends the first challenge, but without XML. The challenge is a random number of random bits.
6	Reseller Client	Reseller Client The client returns the challenge's md5 checksum, encrypted with the Reseller's private key

	Process	Description
		and without XML.

7	Reseller Agent	If the challenge is successful, the Reseller Agent (server) replies with an XCP 'authenticate user' response.
		Perl Example (hash):
		`{`
		`'protocol' => 'XCP',`
		`'action' => 'reply',`
		`'response_code' => '200',`
		`'response_text' => 'Authentication Successful'`
		`}`
		If the Reseller Agent deems that the Reseller Client has failed the challenge, it closes the socket without sending a decline reply because it is assumed that the Reseller Client cannot understand any of the encrypted messages anyway.
		Another possible response code would be code 310, if the Reseller's command rate is exceeded.

8	Reseller Client	If the Reseller Client receives a response code of 200, it can then send its first XCP command. All further communication for the established session is encrypted. The first XCP command the client must send after being authenticated is 'set cookie'. This is required because the cookie is used for all further authenticated commands.

Encryption

Supported ciphers

OpenSRS currently supports the DES and Blowfish encryption algorithms.

The suggested method of using these encryption types is through their respective Perl modules, Crypt::DES and Crypt::Blowfish, which are then accessed through a common interface created by Crypt::CBC. For your convenience, Crypt::CBC is now included in the OpenSRS client distribution.

If you are unable to install Crypt::DES or Crypt::Blowfish there is a third option available: Crypt::Blowfish_PP, which is a module for Blowfish written in Pure Perl (PP). Our initial testing has shown this module to be at least 10 times slower than the standard Crypt::Blowfish, but it may be used as a last resort.

Private key

DES only supports keys of 8 bytes, while Blowfish supports keys of up to 56 bytes for greater security.

Private keys in OpenSRS are 112 characters in length (56 bytes), to provide the maximum security for people using Blowfish. If you are an existing customer and you are using an old key (8 bytes), you will not be able to use the Blowfish cipher until you generate a new key. We recommend that you generate a new private key to ensure the strongest encryption possible. Old keys were 8 bytes in length and new keys are 56 bytes in length.

You can continue using the (old) 8-byte key if you only wish to use DES as your cipher. Even if you have a new key (56 bytes), you can still use it with the DES cipher, since Crypt::CBC only uses the portion of the private key that is needed (in the case of DES, it simply ignores everything after the first 8 bytes).

When creating your encryption cipher, do not use the private key in raw form. Instead, first pack the key into a hexadecimal binary string. In Perl this is accomplished with:

```
$private_key = pack('H*', $private_key);
```

You may then use the private key to create your encryption cipher, authenticate, and begin sending data to the server.

Reseller Agent Return Codes

Code	Response text / Explanation
200	Command Successful. Authentication Successful. Closing Connection. Registration Successful. Modification Successful. Nameserver Update Successful. Nameserver Added.

Code	Response text / Explanation
	Nameserver Created. Nameserver deleted. No nameserver changes necessary. Query Successful. Registration successful.
210	Domain available.
211	Domain taken.
221	Domain taken (a waiting registration exists<?rh-cbt_start condition="Parent_Reseller_Only" ?> in OpenSRS<?rh-cbt_end ?>). Note: This is used for asynchronous registries only.
250	Action submitted successfully for processing to asynchronous registry.
300	Exceeded max command rate. Request deferred (for asynch).
310	Exceeded max simultaneous connections.
350	Number of command per connection exceeded limit. Client must re-authenticate with server. A maximum of 100 commands can be sent through one connection/session. After 100 commands have been submitted, the connection is closed and a new connection must be opened to submit outstanding requests.
400	Internal server error. Invalid Command: xxx. Access denied: invalid IP address. Invalid encryption method; try -des: ... Unable to change nameserver hostname to $new_fqdn. Unable to complete registration. Please retry. Domain already renewed; another renewal cannot be applied until the first request completes at the registry.
404	Internal Server Error.
405	Registry error: domain's nameserver not updated. Registry error, unable to add nameserver to domain. Registry error, unable to remove nameserver. Registry error, nameserver creation failed. Nameserver deletion failed at registry. Unable to modify nameserver record.

Code	Response text / Explanation
410	Reseller authentication error.
415	Registrant (end-user) authentication error. Invalid Cookie Supplied. Invalid Password supplied.
430	Invalid command.
435	Permission denied, subuser permissions. Permission Denied for Modify $type, f_owner=xxx #modify_contact_info. Cannot remove domain owner. Permission Denied: not owner. Request failed validation: Name server 'nameserver.tld' is not found at the registry. Please double check the nameserver and re-submit.
436	This error occurs when the RM does not know what to do with the response code returned by the RA after it talked with the registry. This error usually means that the RM needs to be fixed to understand the new response.
437	Cannot process command because there is already another request waiting on this domain. (Used for asynchronous registries.)
440	Registration Failed: over quota.
445	Nameserver quota exceeded.
447	Sub-user limit exceeded.
460	No Cookie Supplied. No Domain Supplied. Modify type not specified. Required field 'username' not provided. Required field 'password' not provided. Missing required field. Missing required field: fqdn. Missing required field: ip. Info type not specified. Required field 'reg_username' not provided. Required field 'reg_password' not provided.
465	Registration Failed: Invalid data, error= Registration Failed, error=

Code	Response text / Explanation
	Unknown Modify type: xxx Invalid data, error=#modify_contact_info. Subuser cannot be the same as the owner. Invalid syntax for subuser name. Password exceeded maximum length: 20 characters. Invalid domain name syntax. Invalid syntax for registrant username. Password length below minimum: 3 characters. Invalid nameserver syntax (common_rule for nameserver). Invalid syntax for nameserver (2nd level domain). Invalid new nameserver hostname. Invalid domain name syntax. Duplicate nameserver detected for xxx. Duplicate sortorder detected for xxx. Invalid IP address: $ip Unknown Info type: $type' Invalid syntax for username. Invalid registration type: $reg_type Domain does not belong to Registrar
480	Domain xxx not owned by user. Renewals not supported for this TLD. Response text = 'capability is not enabled for domain.ca'. Cookie not found. Subuser not found. Subuser xxx not found. Sub-User not found. User profile for xxx/xxx not found. Profile based on xxx not found. Command failed: unable to verify existence of nameserver xxx. Nameserver doesn't exist. Unable to locate nameserver in local database. Unable to find domain in registry. Nameserver $fqdn not currently mapped to domain. Nameserver not found.

Code	Response text / Explanation
	Non-existent nameserver #(in registry).
485	Domain taken. Nameserver already mapped to domain. Nameserver $fqdn already exists (as result of request in DB or RRP). Nameserver in use. New nameserver hostname already in registry.
486	Entity already exists in a processing state (usually a domain registration) Trying again in a few seconds to a minute should resolve the issue (i.e. show it truly taken or available).
487	Domain not transferable.
541	Domain's current expiration year in registry doesn't match the year provided by user.
552	Domain is less than 60 days old. This can occur if: • domain is not yet 60 days old. • existing registrar has the name locked for either nonpayment or at the end users request - requesting party needs to contact existing registrar to resolve. • domain name is in dispute. • the name has been deleted.
555	Domain has already been successfully renewed, with the current expiration year matching the year provided by the user.
557	Nameserver locked. An attempt has been made to modify or delete a name server that is hosting a TLD in the root zone. Modifications to the root zone can only be made with the approval of the U.S. Department of Commerce and IANA, so if the registrar absolutely needs to modify or delete such a name server; the action needs to be coordinated through the registry operator using an out-of-band communications channel. (Extracted from the RRP spec.)
702	Catch all code for errors related to communication issues.
703	Could not send command.
704	Read empty message on the socket.
705	Timed out, resubmit request. Client timed out waiting for the RA to respond.

Code	Response text / Explanation
	Domain: <sampledomainname> Registration attempt failed: Timeout reading client request. Client must re-authenticate with server.
unnumbered	A connection with the <?rh-cbt_start condition="Parent_Reseller_Only" ?>OpenSRS <?rh-cbt_end ?>server remains open for no more than 60 seconds. If the authentication and sending of a registration request do not complete within 60 seconds, the connection is closed and a new connection must be opened.