XML Encryption Strategy

Updated Wednesday, July 13th, 2005

Introduction

The Department of Public Safety (DPS) will implement the following strategy for encryption within XML documents transmitted to and from the DPS XMLConduit web service. These rules will allow any specific block of PCData to be encrypted. Client or server parsing routines can dynamically discover the encrypted element and programmatically generate a cipher key from an MD5 digest of a dynamically determined seed string.

To facilitate this encryption strategy, DPS will provide a Java class and a Microsoft Windows COM object to agencies submitting XML web service requests. Source code will be controlled and maintained by DPS to promote confidentiality.

Rules for encryption

1] Encryption is limited to PCData contained in Element nodes, i.e., no dynamic encryption support is provided for data in attribute values using DPS interfaces although any collaborating systems can certainly implement a strategy.

2] The encryption algorithm standardizes using AES (rijndaal), ECB mode, no padding.

3] Keys are generated from a 128 bit MD5 hash of a dynamically determined seed string (no static seed values from document to document).

4] Seed strings are comprised of XML drawn from the context document and noted with an attribute, "source", on the element containing the PCData to be encrypted/decrypted, e.g., source="udps:DocumentDescriptor"

5] The seed string shall contain the designated element and all contained XML (child elements, etc.) if any, plus a statically appended string written within the DPS algorithms.

6] The fact that the PCData is encrypted shall be noted with an attribute, "encrypted", on the element containing the encrypted pcdata, e.g., encrypted="True".

7] Encrypted data shall always be base64 encoded.

On the next page is a complete example of a request XML document that has the PCData of the <udps:UDPSLogon> and <udps:UDPSPassword> elements encrypted and specifies the seed string node as <udps:TransactionParameters>. A server component need only detect the encrypted="True" attribute, and then call the decryptString() function—passing the XML, the name of the source element, and the base64 encoded PCData. For guaranteed predictability in generating the seed string, the XML sent into the encrypt() and decryptString() functions should be void of any structural whitespace characters between elements (whitespace within PCData is ok). For example, the incoming xml should look something like the following (notice that there are no carriage returns or tabs between elements):

```
<udps:UDPSXML><udps:DocumentDescriptor type="XID" class="SI" authenticator="DPS" routingCode="L"/></udps:UDPSXML>
```

For example, do not use documents that have structural whitespace like this:

```
<udps:UDPSXML>
    <udps:DocumentDescriptor type="XID" class="SI" authenticator="DPS" routingCode="L"/>
</udps:UDPSXML>
```

Different parsers handle whitespace with slight variations so it is best to remove it so that the sending and receiving systems are sure to generate the same seed string from the designated element node.

This approach creates flexibility around the elements that need encrypted PCData for transport purposes, and it allows systems to dynamically choose the key material text to be used for creating MD5 message digests. This way the hash can be dynamically set and determined by programs—somewhat similar to key pairs encryption without the overhead or round trip key exchange. In the example below, the seed string is not the string "**udps:TransactionParameters"** but all the xml contained by the <udps:TransactionParameters> element, or in the example's case, the string:

<udps:TransactionParameters><j:Subject><j:PersonAssignedIDDetails IDIssuingAuthorityText=""UT""><j:PersonDriverLicenseID><j:ID>149051162</j:ID><j:IDJurisdictionCode>UT</j:IDJurisdictionCode></j:PersonDriverLicenseID></j:PersonAssignedIDDetails></j:Subject><udps:PhotoRequestedIndicator allPhotosIndicator=""false"">false</udps:PhotoRequestedIndicator></udps:TransactionParameters>, or in other words, the source element and all the XML it contains. (Note: wrapping in the above XML occurs because of word processing, in actuality, the string would constitute one line of characters).

Note on Whitespace

Whitespace is not stripped from the seed string within the encrypt() or decryptString() functions so it is best to strip all structural whitespace from the XML document before calling the functions.

Caveats

You cannot specify a source element that contains the encrypted data or the decrypting routine will never get the same key material that the encrypting system used -- the encrypted data is added after extracting the seed data on the encrypting side, and the decrypting side will extract a seed string that contains the encrypted data and its accompanying element, and the two sides will therefore generate unequal cipher keys. Nor should you specify an element that is duplicated within the document because the decrypting system will always pull the first instance of the element (and its children) as the seed source. These rules are enforced somewhat in the attached DPS algorithms.

At this time, the udps:UDPSLogon and udps:UDPSPassword elements must be encrypted. As new transactions are created, they will need to be evaluated for encryption requirements.

An example XML document

(copy and save the text off to a file (e.g. sample.xml) and load it in Internet Explorer to view it as a navigable tree. Note: the data here is for example purposes and will not decrypt as explained. (Note: wrapping in the XML below occurs because of word processing, in actuality, the string would constitute one line of characters.)

<?xml version="1.0" encoding="UTF-8"?><udps:UDPSXML xmlns:j="http://www.it.ojp.gov/jxdm/3.0" xmlns:udps="http://webservices.ps.state.ut.us/udpsxml/release/1.0"><udps:DocumentDescriptor type="XID" class="SI" authenticator="UCJIS" routingCode="L"/><udps:Header><udps:Version>3.0</udps:Version><udps:System><j:ID>TEST1svr</j:ID></udps:System><udps:TransactionID><j:ID>123459</j:ID></udps:TransactionID><udps:Submitter><udps:UDPSAgency><j:ID>DPSMIS</j:ID></udps:UDPSAgency><udps:UDPSAuthentication><udps:**UDPSLogon** encrypted="true" source="udps:TransactionParameters">**PnwZ2auzvavt4XYLR+QUBQ==**</udps:UDPSLogon><udps:**UDPSPassword** encrypted="true" source="udps:TransactionParameters">**TsR87OpPhGEYDE4kZ7UNQA==**</udps:UDPSPassword></udps:UDPSAuthentication></udps:Submitter><udps:InitiatingAgency><j:Agency><j:OrganizationORIID><j:ID>UTTESTORI</j:ID></j:OrganizationORIID></j:Agency></udps:InitiatingAgency><udps:Destinations><udps:Destination><j:Agency><j:OrganizationID><j:ID>UT</j:ID></j:OrganizationID></j:Agency></udps:Destination></udps:Destinations></udps:Header><udps:TransactionParameters><j:Subject><j:PersonAssignedIDDetails IDIssuingAuthorityText=""UT""><j:PersonDriverLicenseID><j:ID>149051162</j:ID><j:IDJurisdictionCode>UT</j:IDJurisdictionCode></j:PersonDriverLicenseID></j:PersonAssignedIDDetails></j:Subject><udps:PhotoRequestedIndicator allPhotosIndicator=""false"">false</udps:PhotoRequestedIndicator></udps:TransactionParameters></udps:UDPSXML>

The DPS Encryption Utility Interfaces

DPS will provide three interfaces for facilitating uniform encryption and decryption between disparate systems. Two are contained in a DLL and the other is a java class. The provided DLL (ucjissec.DLL) contains a com object: ucjissec.UCJISCipher that will encrypt and decrypt base64 strings created from the java AES class and vice versa. The DLL also exports the Encrypt and DecryptString functions for direct access if desired (bypassing the COM interface). Therefore, the DLL encapsulates both the COM object's interface and a direct access interface (see the C++ example).

Note: the DLL must be registered (using regsvr32) before attempting to use the COM interface. The methods are essentially the same on all three interfaces:

The COM Interface

```
IUCJISCipher = interface(IUnknown)
  function DecryptString(const pXML: WideString; const pCipherText: WideString;
                         const pSourceElement: WideString): WideString; stdcall;
  function Encrypt(const pXML: WideString; const pPlainText: WideString;
                   const pSourceElement: WideString; pBase64: WordBool): WideString; stdcall;
end;
```

The DLL Interface

```
  HRESULT __cdecl DecryptString(char* pXML, char* pCipherText, char* pSourceElement,
                                char* pPlainText, int pPlainTextBufSize );
  HRESULT __cdecl Encrypt(char* pXML, char* pCipherText, char* pSourceElement,
                                char* pPlainText, int pPlainTextBufSize, int pBase64);
```

The Java Interface

```
public class UCJISCipher {
  public static String decryptString( final String pXML, final String pEncodedCipherText, final String pSourceElement );
  public static String encrypt( final String pXML, final String pPlainText, final String pSourceElement  );
}
```

Usage Instructions

The decryptString() method always takes a base64 encoded string as the second parameter because the return string from encrypt() is always a base64 encoded string.

The COM object must be registered before it can be used. This can be done at any command prompt (e.g., C:\ ). To register the COM object, enter the following command: regsvr32 [path]ucjissec.dll [enter]. The UCJISCipher.class is not contained within a package so it can be deployed in any classpath; however, it does require J2SDK 1.4.2+. It also contains a main() method that may be executed to ensure the proper JDK version and configuration.

Sample Code

Java Example

```
   public static void main( String[] args ) throws Exception {
      String xml = new String("<udps:TransactionParameters><j:Subject><j:PersonAssignedIDDetails
IDIssuingAuthorityText=\"UT\"><j:PersonDriverLicenseID><j:ID>149051162</j:ID><j:IDJurisdictionCode>UT</j:IDJurisdictionCode></j:PersonDrive
rLicenseID></j:PersonAssignedIDDetails></j:Subject><udps:PhotoRequestedIndicator
allPhotosIndicator=\"false\">false</udps:PhotoRequestedIndicator></udps:TransactionParameters>");

      try {
         System.out.println( "XML String: " + xml );
         String data = "changeme";
         String resultStr = UCJISCipher.encrypt( xml, data, "udps:TransactionParameters" );
         System.out.println( "Encrypted and Encoded: " + resultStr );

         resultStr = UCJISCipher.decryptString( xml, resultStr, "udps:TransactionParameters" );
         System.out.println( "Decrypted: " + resultStr );
      } catch ( Throwable t ) {
         t.printStackTrace();
```

```
    }
  }
```

MS Access VBA Example

(after registering the COM Object's Interface—regsvr32 ucjissec.dll):

```
Public Sub Command0_Click()
  Dim O As New UCJISCipher
  Dim xml As String
  Dim source As String
  Dim plaintext As String
  Dim ciphertext As String

  xml = "<udps:TransactionParameters><j:Subject><j:PersonAssignedIDDetails
IDIssuingAuthorityText=""UT""><j:PersonDriverLicenseID><j:ID>149051162</j:ID><j:IDJurisdictionCode>UT</j:IDJurisdictionCode></j:PersonDrive
rLicenseID></j:PersonAssignedIDDetails></j:Subject><udps:PhotoRequestedIndicator
allPhotosIndicator=""false"">false</udps:PhotoRequestedIndicator></udps:TransactionParameters>"
  source = "udps:TransactionParameters"
  plaintext = "changeme"
  ciphertext = O.Encrypt(xml, plaintext, source, False)
  'now insert the ciphertext in the XML

  'test the decryption if desired
  plaintext = ""
  plaintext = O.DecryptString(xml, ciphertext, source)

  'if this was to encrypt the password, the resulting UDPSPassword element would look like this:
  ' <udps:UDPSPassword encrypted="True" source="udps:TransactionParameters">Lf3oU/Zaz9szLvOZURm7XA==</udps:UDPSPassword>

End Sub
```

Delphi Example Unit

(after installing the COM Object's Interface—see import type library in Delphi's help):

```
unit TestFormU;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
Forms,   Dialogs, StdCtrls, OleServer, ucjissec_TLB;

type
  TForm1 = class(TForm)
    Encrypt: TButton;
    PlainTextEdit: TEdit;
    CipherTextEdit: TEdit;
    Decrypt: TButton;
    PlainText: TLabel;
    CipherText: TLabel;
    XML: TMemo;
    NodeXML: TMemo;
    NodeXMLStr: TEdit;
    Source: TLabel;
    UCJISCipher1: TUCJISCipher;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure EncryptClick(Sender: TObject);
    procedure DecryptClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  UCJISCipher1.Connect;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  UCJISCipher1.Disconnect;
end;

procedure TForm1.EncryptClick(Sender: TObject);
begin
  CipherTextEdit.Text := UCJISCipher1.Encrypt( XML.Text,
PlainTextEdit.Text, NodeXMLStr.Text, False);
end;

procedure TForm1.DecryptClick(Sender: TObject);
begin
  PlainTextEdit.Text := UCJISCipher1.DecryptString(XML.Text,
CipherTextEdit.Text, NodeXMLStr.Text);
end;

end.
```

Visual C++ Example

(accessing the DLL directly):

```
#include "stdafx.h"
#include <stdio.h>
#include <windows.h>

typedef int (*EncryptProc)(LPTSTR, LPTSTR, LPTSTR, LPTSTR, int, int);
typedef int (*DecryptStringProc)(LPTSTR, LPTSTR, LPTSTR, LPTSTR, int );

int main(int argc, char* argv[]) {

   HINSTANCE hinstLib;
   EncryptProc EncryptProcAddr;
   DecryptStringProc DecryptProcAddr;
   LPTSTR pXML = "<udps:TransactionParameters><j:Subject><j:PersonAssignedIDDetails
IDIssuingAuthorityText=\"UT\"><j:PersonDriverLicenseID><j:ID>149051162</j:ID><j:IDJurisdictionCode>UT</j:IDJurisdictionCode></j:PersonDriverLicenseID>
</j:PersonAssignedIDDetails></j:Subject><udps:PhotoRequestedIndicator
allPhotosIndicator=\"false\">false</udps:PhotoRequestedIndicator></udps:TransactionParameters>";
   LPTSTR pPlainText = "changeme";
   LPTSTR pSourceElement = "udps:TransactionParameters";
   char pCipherText[256];
   char pPlainTextBuf[256];

   BOOL fFreeResult, fRunTimeLinkSuccess = FALSE;

   // Get a handle to the DLL module.

   hinstLib = LoadLibrary(TEXT("ucjissec"));

   // If the handle is valid, try to get the function address.

   if (hinstLib != NULL) {
      EncryptProcAddr = (EncryptProc) GetProcAddress(hinstLib, TEXT("Encrypt"));

      // If the function address is valid, call the function.

      if (NULL != EncryptProcAddr) {
         fRunTimeLinkSuccess = TRUE;

         //encrypt example
         int result = (EncryptProcAddr) (pXML, pPlainText, pSourceElement, pCipherText, 256, 0);
         switch( result ) {
            case 0:
               printf( pCipherText );
               printf( "\n" );
               break;
            case -1:
               printf( "XML param was NULL\n" );
               break;
            case -2 :
               printf( "CipherText Buffer was NULL\n" );
               break;
            case -3 :
               printf( "SourceElement param was NULL\n" );
               break;
            case -4 :
               printf( "PlainText param was NULL\n" );
               break;
            case -5 :
               printf( "XML param was an empty string\n" );
               break;
            case -6 :
               printf( "PlainText param was an empty string\n" );
               break;
            case -7 :
               printf( "SourceElement param was an empty string\n" );
               break;
            case -8 :
               printf( "Return Buffer too small for CipherText\n" );
               break;
         }
```

```
    }

    DecryptProcAddr = (DecryptStringProc) GetProcAddress(hinstLib, TEXT("DecryptString"));

    // If the function address is valid, call the function.

    if (NULL != DecryptProcAddr) {
       //decryptString example
       int result = (DecryptProcAddr) (pXML, pCipherText, pSourceElement, pPlainTextBuf, 256);
       switch( result ) {
          case 0:
             printf( pPlainTextBuf );
             printf( "\n" );
             break;
          case -1:
             printf( "XML param was NULL\n" );
             break;
          case -2 :
             printf( "CipherText Buffer was NULL\n" );
             break;
          case -3 :
             printf( "SourceElement param was NULL\n" );
             break;
          case -4 :
             printf( "PlainText param was NULL\n" );
             break;
          case -5 :
             printf( "XML param was an empty string\n" );
             break;
          case -6 :
             printf( "CipherText param was an empty string\n" );
             break;
          case -7 :
             printf( "SourceElement param was an empty string\n" );
             break;
          case -8 :
             printf( "Return Buffer too small for CipherText\n" );
             break;
       }
    }

    // Free the DLL module.

    fFreeResult = FreeLibrary(hinstLib);
  }

  // If unable to call the DLL function, use an alternative.

  if (! fRunTimeLinkSuccess)
    printf("Did not link to dll methods\n");

  return 0;
}
```