CVE-2004-0719: Microsoft SSL PCT vulnerability

Kyle C. Quest
kquest@toplayer.com

Abstract

This paper describes some of my research findings. It's an attempt to answer the question I had just before I started my research.

It's always frustrating to hear about a newly discovered vulnerability and not know what the actual problem is. This is true especially if you are responsible for figuring out how to stop network attacks exploiting the vulnerability.

Juliano Rizzo already provided a great description for the actual buffer overflow and the exploits that are out there right now shedding the light on the mystery behind the SSL message they use. I'll build upon his work answering additional question people might have.

Note:

I used Windows 2000 with no service packs for my research. The diagrams I provide may not be accurate for latter versions of Windows.

1. Overview

The vulnerability lies in the *Pct1SrvHandleUniHello* function (*schannel.dll*) where it tries to modify the SSL 2 challenge data. It can be exploited by a ClientHello SSL 2 message (or a message falsely believed to be SSL 2) that signals the use of Microsoft PCT 1 extensions to SSL 2 by means of a special *Cipher Spec* that starts with 0x8f. The message would have to contain a *Challenge* field more than 16 bytes long and less than or equal to 32 bytes.

2. SSL 2 Protocol

Attacks exploiting this vulnerability will use SSL 2 Client Hello messages that must have a special Microsoft Cipher Spec as the first Cipher Spec in the list. It looks like Microsoft created a private cipher spec early on when they were designing their PCT 1 protocol. PCT 1 protocol doesn't use SSL 2 Client Hello packets anymore, but the functionality to handle them is still there. If you capture PCT 1 handshake messages generated by Internet Explorer, you'll see that it uses native PCT 1 Client Hello message format.

SSL 2 Client Hello Format:

char MSG-LENGTH[2 or 3]
char MSG-CLIENT-HELLO(Value:1)
char CLIENT-VERSION[2]
char CIPHER-SPECS-LENGTH[2]
char SESSION-ID-LENGTH[2]
char CHALLENGE-LENGTH[2]
char CIPHER-SPECS-DATA[LENGTH]
char SESSION-ID-DATA[LENGTH]
char CHALLENGE-DATA[LENGTH]

The last 3 fields are optional and may not be present if their corresponding length fields are set to 0.

Note that the CIPHER-SPECS-LENGTH field contains the total size of the CIPHER-SPECS-DATA section in bytes (it's not the number of Cipher Specs)!

The SSL 2 Cipher Specs themselves are 3 bytes long with the following format:

Byte 1: Cipher Spec ID
Byte 2: Cipher Spec Sub ID
Byte 3: Key Size (in bits)

The Cipher Spec ID is usually a unique number with the Cipher Spec Sub ID set to 0. However, when Cipher Specs differ only by the hash algorithm they use, their Cipher Spec IDs match. In this case the Cipher Spec Sub ID field is used to uniquely distinguish them.

Microsoft overloaded the meaning of Cipher Specs by creating a special one with Cipher Spec

ID set to 0x86. They also overloaded the meaning of bytes 2 and 3 using them to provide the PCT protocol version.

3. Chasing the Vulnerability

There are many different attack vectors for this vulnerability. I chose IIS web server as an example. Refer to Appendix A for a diagram showing the path SSL2/PCT1 Client Hello packet takes.

IIS uses an extensible web server architecture where new processing functionality can be provided using filters. This is actually how IIS implements URLScan and, more importantly, this is how it implements secure communication protocols (sspifilt.dll). These filters register for the processing events they are interested in. The SSPIFILT filter registers for a number of events including the SF_NOTIFY_READ_RAW_DATA, which instructs IIS to send raw data to SSPIFILT for preprocessing. Any filter can register for this event (you can wrap OpenSSL or some other SSL code in one of those filters and replace Microsoft's SSL code if you wish).

When a Client Hello packet arrives, it causes the IIS server to create a new HTTP_REQUEST object that reads data in a loop in the *DoWork()* function. This function reads the Client Hello packet and passes it to the SSPIFILT filter where it processes the Client Hello sending a Server Hello in reply, which is done by calling the *AcceptSecurityContext()* function. This is actually the same function you'd have to call if you wanted to create an SSL server using MS SSL libraries (refer to Microsoft documentation for the detailed function description).

The call to *AcceptSecurityContext()* eventually ends up in schannel DLL, which is where all of the secure protocols are implemented. To make it all the way there, the Client Hello packet had to go through another DLL, secur32.dll, which made a Local Procedure Call to the LSASS.EXE, the process that's responsible for all kinds of security requests.

Once in LSASS.EXE, the Client Hello message is delivered to schannel.dll where the security protocol is determined for the message passing it to the appropriate protocol handler.

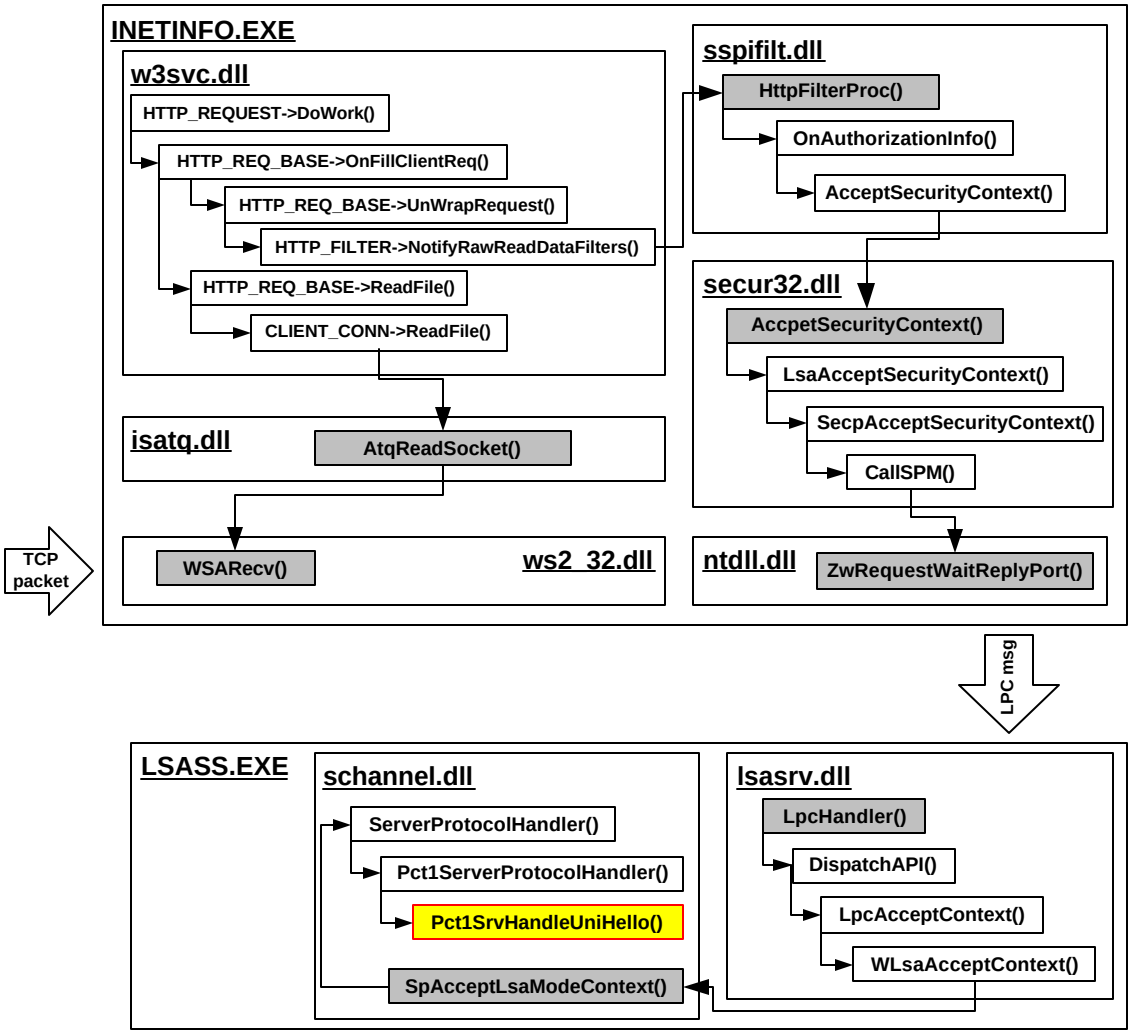
Our PCT 1 Client Hello message is passed to *Pct1ServerProtocolHandler()*.

A malicious PCT 1 Client Hello message must come in an SSL 2 formatted message described above for the vulnerability to be triggered (Native PCT 1 Client Hello messages will not get to the bug).

The *Pct1ServerProtocolHandler()* function calls *Pct1SrvHandleUniHello()* after decoding the Client Hello message filling in some of the response message fields including the Challenge Data field.

The rest is history…

Appendix A: SSL PCT 1 Client Hello Processing Trace.



Appendix B: SpAcceptLsaModeContext() function.

```
NTSTATUS
SpAcceptLsaModeContext(LSA_SEC_HANDLE  CredHandle,
                       LSA_SEC_HANDLE  ContextHandle,   /*[ebp+0x0c]*/
                       SecBufferDesc*  InputBuffer,     /*[ebp+0x10]*/
                       ULONG  ContextRequirements,      /*[ebp+0x14]*/
                       ULONG           TargetDataRep,
                       LSA_SEC_HANDLE* NewContextHandle,/*[ebp+0x1c] */
                       SecBufferDesc*  OutputBuffer,    /*[ebp+0x20] */
                       ULONG*          ContextAttributes,/*[ebp+0x24]*/
                       TimeStamp*      ExpirationTime,
                       BOOLEAN*        MappedContext,
                       SecBuffer*      ContextData
                                               )
{
      SecBuffer*      local1 = NULL;
      SecBuffer*      local4 = NULL;
      DWORD           local5 = 0x1001C;
      ULONG           ctxAttrs; /* edx */
      SecBuffer*      inTokenBuf = NULL;
      SecBuffer*      outBuf = NULL;
      SecBuffer*      curBuf;
      DWORD           bufCount;
      DWORD           bufType;
      LSA_SEC_HANDLE* context;
      SecBuffer       in;
      SecBuffer       out;

      // test dword ptr [ebp+0x14],0x20001
      if(ContextRequirements & (ASC_REQ_DELEGATE|ASC_REQ_INTEGRITY))
            return SEC_E_UNSUPPORTED_FUNCTION;
      if((ContextRequirements & 0xff000000) == 0x02000000)
            local3 = 0x100;

      // Process the input buffers (looking for token buffers)
      for(bufCount=InputBuffer->cBuffers,
          curBuf = InputBuffer->pBuffers;
            bufCount > 0;
            bufCount--,curBuf += sizeof(SecBuffer))
      {
            bufType = curBuf->BufferType;
            if(bufType &= 0x0fffffff) // mask off the buffer attributes
            {
                  if(bufType == SECBUFFER_TOKEN)
                  {
                        local1     = curBuf;
                        inTokenBuf = curBuf;
                  }
                  else
                  {
                        // cmp edx,0x80000002
                        //it can never be true because
                        //the MSB is masked off???
                        if(bufType ==
                          (SECBUFFER_READONLY|SECBUFFER_TOKEN))
```

```
                        {
                                local1     = curBuf;
                                inTokenBuf = curBuf;
                        }
                }
        }
        else
        {
                //this buf is SECBUFFER_EMPTY
                if(inTokenBuf == NULL)
                {
                        local1     = curBuf;
                        inTokenBuf = curBuf;
                }
                else
                {
                        if(local4 == NULL)
                                local4 = curBuf;
                }
        }
}
//Process the output buffers (looking for empty buffers)
for(bufCount=OutputBuffer->cBuffers,
    curBuf = OutputBuffer->pBuffers;
      bufCount > 0;
      bufCount--,curBuf += sizeof(SecBuffer))
{
      bufType = curBuf->BufferType;
      if((bufType &= 0x0fffffff) == 0)
      {
            //It's SECBUFFER_EMPTY
            if(outVar == NULL)
            {
                  local2 = curBuf;
                  outBuf = curBuf;
            }
      }
      else
      {
            //See if it's SECBUFFER_TOKEN
            bufType -= 2;
            if(bufType == 0)
            {
                  local2 = curBuf;
                  outBuf = curBuf;
            }
      }
}
if(outBuf == NULL)
      return SEC_E_INVALID_TOKEN;

outBuf->BufferType = SECBUFFER_TOKEN;

//omitted...

if(ContextRequirements & 0x00ff0000) == ASC_REQ_STREAM)
{
```

```
            ctxAttrs /*edx*/ = local5;
      }
      else
      {
            ctxAttrs          = 0x1011C;
            outBuf->pvBuffer  = 0;
            outBuf->ulVersion = 0;
      }

      out.pvBuffer  = outBuf->pvBuffer;
      out.ulVersion = outBuf->ulVersion;
      out.BufferType = 0;

      if(ContextRequirements & ASC_REQ_EXTENDED_ERROR)
      {
            ctxAttrs |= ASC_RET_EXTENDED_ERROR;
            local3 |= ASC_RET_INTEGRITY;
      }
      if(ContextRequirements & ASC_REQ_CONNECTION)
      {
            ctxAttrs |= ASC_RET_CONNECTION;
            //or byte ptr [ebp-0Bh],10h
      }
      if(ContextAttributes != NULL)
            *ContextAttributes = ctxAttrs;

      //we need to create the context handle
      //when the function is called for the first time
      if(ContextHandle == NULL)
      {
            context = SPContextCreate(ecx);
            if(context == NULL)
                  return SEC_E_INSUFFICIENT_MEMORY;
      }
      else
            context = ContextHandle;

      //omitted...
      SPContextSetCredentials(context,CredHandle);
      //omitted ...
      context->ServerProtocolHandler(context,&in,&out,var);
      return 0;
}
```

Appendix C: ServerProtocolHandler() function.

```
#define FIRST_CIPHER_SPEC 0xb

NTSTATUS
ServerProtocolHandler(LSA_SEC_HANDLE context,
                      SecBuffer*     in,
                      SecBuffer*     out,
                      int            var)
{
      char* packet = in->pvBuffer;
      WORD  ptype;
      WORD  cipher_specs_len;

      if(packet[0] != 0x16)
      {
            memcpy((void*)&type,(packet + 3),2);
            if(ptype >= 0x8001)
                  //process packet as PCT 1 ...
            if(ptype < 2)
                  return 0x80000003;
            //omitted...
            memcpy((void*)&cipher_spec_len,(packet + 5),2);
            if(cipher_specs_len < 1)
                  return 0x80000003;

            if(ptype < 0x301)
            {
                  if(ptype < 0x300)
                  {
                        if(packet[FIRST_CIPHER_SPEC] != 0x86)
                        {
                              //grab the protocol version
                              //from this special Cipher Spec
                              memcpy((void*)&ptype,
                                     (packet+FIRST_CIPHER_SPEC+1),2);
                              if(ptype < 0x8001)
                                    //process packet as SSL 2 ...
                              else
                              {
                                    if(!(context->field_offset8 & 1))
                                          //process packet as SSL 2
                                    else
                                    {
                                          //process packet as PCT 1
                                          obj->field_offset4 = 0xfe;
                                          obj->ProtoHandlerFn =
                                          Pct1ServerProtocolHandler;
                                          obj->UnknownFn  = 0x7817a535;

                                          Pct1ServerProtocol
                                          Handler(obj,in,out,var);
                                    }

                              }
                        }
```

```
                        else
                              //process packet as SSL 2 ...
                  }
                  else
                        //process packet as SSL 3 ...
            }
            else
                  //process packet as TCL 1 ...

      }
      else
            //omitted...

      return 0;
}
```

Appendix D: Pct1ServerProtocolHandler() function.

```
NTSTATUS
Pct1ServerProtocolHandler(obj,SecBuffer* in,
                          SecBuffer* out,int var)
{
      char* packet;
      //omitted ...
      switch(obj->field_offset4)
      {
      case 0xffff:
            //...
            break;
      case 0x6:
            //...
            break;
      case 0xfffe:
            //...
            break;
      case 0xfffd:
            //...
            break;
      default:
            packet = in->pvBuffer;
            switch(packet[2]) //switch on message type
            {
                  //omitted ...
                  default:
                  //parse the Client Hello message
                  //and fill in some of the response fields
                  Ssl2UnpackClientHello(in,&out);

                  //process Client Hello
                  Pct1SrvHandleUniHello(obj,in,
                                        out->pvBuffer,argx);
            }
      }
      return 0;
}
```

Appendix E: Pct1SrvHandleUniHello() function.

```
NTSTATUS
Pct1SrvHandleUniHello(obj,
                      SecBuffer* in,
                      char*      outData,
                      int        var)
{
      char challengeBuffer[32];
      char* buffer;
      unsigned int max;
      unsigned int idx;
      //omitted...
      max = outData[ChallangeLengthOffset];
      //skip to the challenge data
      //that's already in the output buffer
      buffer = (outData + 0x30);
      memcpy(challengeBuffer,buffer,max);

      //FINAL DESTINATION: THE BUFFER OVERFLOW
      for(idx=0,idx < max,idx++)
            challengeBuffer[idx + max] =
                        ~challengeBuffer[idx];

      return 0;
}
```