/\*

\* smb\_andx\_decode.c

\*

\* Copyright (C) 2004-2006 Sourcefire,Inc

\*

\* This program is free software; you can redistribute it and/or modify

\* it under the terms of the GNU General Public License as published by

\* the Free Software Foundation; either version 2 of the License, or

\* (at your option) any later version.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\* You should have received a copy of the GNU General Public License

\* along with this program; if not, write to the Free Software

\* Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

\*

\* Description:

\*

\* This performs the decoding of SMB AndX commands.

\*

\* NOTES:

\* - 08.12.04: Initial Development. SAS

\*

\*/

#include <stdlib.h>

#include <wchar.h>

#include "debug.h"

#include "snort\_dcerpc.h"

#include "smb\_structs.h"

#include "smb\_andx\_structs.h"

#include "smb\_andx\_decode.h"

#include "dcerpc\_util.h"

#include "dcerpc.h"

#define FIELD\_ACCT\_NAME 0

#define FIELD\_PRIM\_DOMAIN 1

#define SESS\_AUTH\_FIELD(i) ((i == FIELD\_ACCT\_NAME) ? "AccountName" : ((i == FIELD\_PRIM\_DOMAIN) ? "PrimaryDomain" : "Unknown"))

#define FIELD\_NATIVE\_OS 0

#define FIELD\_NATIVE\_LANMAN 1

#define SESS\_NATIVE\_FIELD(i) ((i == FIELD\_NATIVE\_OS) ? "NativeOS" : ((i == FIELD\_NATIVE\_LANMAN) ? "NativeLanMan" : "Unknown"))

/\* Externs \*/

extern DCERPC \*\_dcerpc;

extern SFSnortPacket \*\_dcerpc\_pkt;

extern u\_int8\_t \_disable\_smb\_fragmentation;

extern u\_int16\_t \_max\_frag\_size;

static void ReassembleSMBWriteX(SMB\_WRITEX\_REQ \*writeX, u\_int8\_t \*smb\_data);

static int SMB\_Fragmentation(u\_int8\_t \*smb\_hdr, SMB\_WRITEX\_REQ \*writeX,

u\_int8\_t \*smb\_data, u\_int16\_t data\_size);

**static void ReassembleSMBWriteX(SMB\_WRITEX\_REQ \*writeX, u\_int8\_t \*smb\_data)**

**{**

**SMB\_WRITEX\_REQ temp\_writeX;**

**unsigned int smb\_hdr\_len = (u\_int8\_t \*)writeX - \_dcerpc\_pkt->payload;**

**---------------------------------------------------------------------------------------------------------------------------**

**unsigned int writeX\_len = smb\_data - (u\_int8\_t \*)writeX;**

**-------------------------------------------------------------------------------------------------**

**/\* Make sure we have room to fit into alternate buffer \*/**

**if ( (smb\_hdr\_len + writeX\_len + \_dcerpc->write\_andx\_buf\_len) > \_dpd.altBufferLen )**

**{**

**\_dpd.logMsg("Reassembled SMB packet greater than %d bytes, skipping.",**

**\_dpd.altBufferLen);**

**return;**

**}**

**/\* Mock up header \*/**

**-----------------------------------------------------------------------------------------------------------**

**memcpy(&temp\_writeX, writeX, writeX\_len); -> 1**

**-----------------------------------------------------------------------------------------------------------**

**temp\_writeX.remaining = \_dcerpc->write\_andx\_buf\_len;**

**temp\_writeX.dataLength = \_dcerpc->write\_andx\_buf\_len;**

**/\* Copy headers into buffer \*/**

**/\* SMB Header \*/**

**memcpy(\_dpd.altBuffer, \_dcerpc\_pkt->payload, smb\_hdr\_len);**

**\_dcerpc\_pkt->normalized\_payload\_size = smb\_hdr\_len;**

**---------------------------------------------------------------------------------------------------------------------------------**

**/\* Write AndX header \*/**

**memcpy(\_dpd.altBuffer + \_dcerpc\_pkt->normalized\_payload\_size, &temp\_writeX, writeX\_len);**

**-------------------------------------------------------------------------------------------------------**

**\_dcerpc\_pkt->normalized\_payload\_size += writeX\_len;**

**/\* Copy data into buffer \*/**

**memcpy(\_dpd.altBuffer + \_dcerpc\_pkt->normalized\_payload\_size, \_dcerpc->write\_andx\_buf, \_dcerpc->write\_andx\_buf\_len);**

**\_dcerpc\_pkt->normalized\_payload\_size += \_dcerpc->write\_andx\_buf\_len;**

**\_dcerpc\_pkt->flags |= FLAG\_ALT\_DECODE;**

**ProcessDCERPCMessage(\_dcerpc\_pkt->payload, \_dcerpc->write\_andx\_buf, \_dcerpc->write\_andx\_buf\_len);**

**/\* Get ready for next write \*/**

**DCERPC\_FragFree(\_dcerpc->write\_andx\_buf, \_dcerpc->write\_andx\_buf\_size);**

**\_dcerpc->write\_andx\_buf = NULL;**

**\_dcerpc->write\_andx\_buf\_len = 0;**

**\_dcerpc->write\_andx\_buf\_size = 0;**

**\_dcerpc->fragmentation &= ~SMB\_FRAGMENTATION;**

**\_dcerpc->fragmentation &= ~SUSPEND\_FRAGMENTATION;**

**}**

**int SMB\_Fragmentation(u\_int8\_t \*smb\_hdr, SMB\_WRITEX\_REQ \*writeX, u\_int8\_t \*smb\_data, u\_int16\_t data\_size)**

**{**

**u\_char fragmented = 0;**

**u\_int16\_t writeX\_length;**

**u\_char success = 0;**

**/\* Check for fragmentation \*/**

**if ( \_disable\_smb\_fragmentation )**

**return 0;**

**/\* If not yet reassembling, attempt to parse as full DCE/RPC packet \*/**

**if ( !(\_dcerpc->fragmentation & SMB\_FRAGMENTATION) )**

**{**

**success = ProcessDCERPCMessage(smb\_hdr, smb\_data, data\_size);**

**if ( success )**

**return 0;**

**}**

**/\* Set up writeX buffer to save SMB data. Ignore dataLengthHigh, since we won't**

**handle fragments that big. \*/**

**writeX\_length = writeX->dataLength;**

**/\* Allocate space for buffer**

**For now, ignore offset, since servers seem to \*/**

**if ( !(\_dcerpc->fragmentation & SUSPEND\_FRAGMENTATION) )**

**{**

**if ( \_dcerpc->write\_andx\_buf == NULL )**

**{**

**if ( writeX\_length > \_max\_frag\_size )**

**writeX\_length = \_max\_frag\_size;**

**\_dcerpc->write\_andx\_buf = (u\_int8\_t \*) DCERPC\_FragAlloc(NULL, 0, &writeX\_length);**

**if ( !\_dcerpc->write\_andx\_buf )**

**\_dpd.fatalMsg("Failed to allocate space for SMB Write AndX\n");**

**if ( writeX\_length == 0 )**

**{**

**DEBUG\_WRAP(\_dpd.debugMsg(DEBUG\_DCERPC, "Memcap reached, ignoring SMB fragmentation reassembly.\n"););**

**DCERPC\_FragFree(\_dcerpc->write\_andx\_buf, 0);**

**\_dcerpc->write\_andx\_buf = NULL;**

**\_dcerpc->fragmentation |= SUSPEND\_FRAGMENTATION;**

**return 0;**

**}**

**\_dcerpc->write\_andx\_buf\_size = writeX\_length;**

**\_dcerpc->write\_andx\_buf\_len = 0;**

**}**

**else**

**{**

**u\_int16\_t new\_size;**

**new\_size = \_dcerpc->write\_andx\_buf\_size + writeX->dataLength;**

**\_dcerpc->write\_andx\_buf = (u\_int8\_t \*) DCERPC\_FragAlloc(\_dcerpc->write\_andx\_buf,**

**\_dcerpc->write\_andx\_buf\_size, &new\_size);**

**if ( !\_dcerpc->write\_andx\_buf )**

**\_dpd.fatalMsg("Failed to allocate space for SMB Write AndX\n");**

**if ( new\_size == \_dcerpc->write\_andx\_buf\_size )**

**{**

**DEBUG\_WRAP(\_dpd.debugMsg(DEBUG\_DCERPC, "Memcap reached, suspending SMB fragmentation reassembly.\n"););**

**\_dcerpc->fragmentation |= SUSPEND\_FRAGMENTATION;**

**DCERPC\_FragFree(\_dcerpc->write\_andx\_buf, \_dcerpc->write\_andx\_buf\_size);**

**\_dcerpc->write\_andx\_buf = NULL;**

**\_dcerpc->write\_andx\_buf\_len = 0;**

**\_dcerpc->write\_andx\_buf\_size = 0;**

**return 0;**

**}**

**\_dcerpc->write\_andx\_buf\_size = new\_size;**

**}**

**}**

**/\* SMB frag \*/**

**if ( writeX\_length > (\_dcerpc->write\_andx\_buf\_size - \_dcerpc->write\_andx\_buf\_len) )**

**{**

**writeX\_length = \_dcerpc->write\_andx\_buf\_size - \_dcerpc->write\_andx\_buf\_len;**

**}**

**memcpy(\_dcerpc->write\_andx\_buf + \_dcerpc->write\_andx\_buf\_len, smb\_data, writeX\_length);**

**\_dcerpc->write\_andx\_buf\_len += writeX\_length;**

**\_dcerpc->fragmentation |= SMB\_FRAGMENTATION;**

**if ( IsCompleteDCERPCMessage(\_dcerpc->write\_andx\_buf, \_dcerpc->write\_andx\_buf\_len) )**

**{**

**ReassembleSMBWriteX(writeX, smb\_data);**

**\_dcerpc->fragmentation &= ~SMB\_FRAGMENTATION;**

**}**

**return 0;**

**}**

static int IsIPC(u\_int8\_t \*s, u\_int16\_t len, u\_int32\_t isUnicode)

{

u\_int16\_t i;

u\_int8\_t unicode\_ipc[] = { 'I', '\0', 'P', '\0', 'C', '\0', '$', '\0' };

if ( isUnicode )

{

if ( len < 8 )

return 0;

for ( i = 0; i < (len - 8)/2; i++ )

{

if ( memcmp(s+(i\*2), unicode\_ipc, 8) == 0 )

return 1;

}

}

else

{

if ( len < 5 )

return 0;

for ( i = 0; i < (len - 5); i++ )

{

if ( memcmp(s+i, "\\IPC$", 5) == 0 )

return 1;

}

}

return 0;

}

int SkipBytes(u\_int8\_t \*data, u\_int16\_t size)

{

u\_int16\_t i = 0;

while ( \*data != 0 && i < size )

{

data++;

i++;

}

return i;

}

int SkipBytesWide(u\_int8\_t \*data, u\_int16\_t size)

{

u\_int16\_t i = 0;

while ( \*data != 0 && i < size )

{

data += 2;

i += 2;

}

return i;

}

int ProcessSMBTreeConnXReq(SMB\_HDR \*smbHdr, u\_int8\_t \*data, u\_int16\_t size, u\_int16\_t total\_size)

{

SMB\_TREE\_CONNECTX\_REQ \*treeConnX = (SMB\_TREE\_CONNECTX\_REQ \*)data;

u\_int16\_t byteCount = smb\_ntohs(treeConnX->byteCount);

u\_int16\_t passwdLen = smb\_ntohs(treeConnX->passwdLen);

unsigned char \*smb\_data = data + sizeof(SMB\_TREE\_CONNECTX\_REQ);

int skipBytes = 1;

int isIPC = 0;

size -= sizeof(SMB\_TREE\_CONNECTX\_REQ);

/\* Sanity check \*/

if ( byteCount > size )

return 0;

/\* Password data \*/

if (passwdLen > 0 && byteCount > 0)

{

/\* This passwd will always be ASCII -- equiv of

\* CaseInsensitivePasswd field from SessSetupAndX message \*/

#ifdef DEBUG\_DCERPC\_PRINT

printf("Password: %.\*s\n", passwdLen, smb\_data);

#endif

/\* Skip past the password -- no terminating NULL \*/

if ( passwdLen > size )

return 0;

smb\_data += passwdLen;

if ( byteCount < passwdLen )

return 0;

byteCount -= (passwdLen);

}

/\* Get path \*/

if (HAS\_UNICODE\_STRINGS(smbHdr)) /\* Service field is ALWAYS ascii \*/

{

if (\*smb\_data != '\0')

{

#ifdef DEBUG\_DCERPC\_PRINT

wprintf(L"Path: %s\n", smb\_data);

#endif

skipBytes = SkipBytesWide(smb\_data, byteCount) + 2;

}

isIPC = IsIPC(smb\_data, byteCount, 1L);

}

else

{

if (\*smb\_data != '\0')

{

#ifdef DEBUG\_DCERPC\_PRINT

printf("Path: %s\n", smb\_data);

#endif

skipBytes = SkipBytes(smb\_data, size) + 1;

}

isIPC = IsIPC(smb\_data, byteCount, 0L);

}

smb\_data += skipBytes;

byteCount -= skipBytes;

if ( isIPC && \_dcerpc->smb\_state == STATE\_START )

\_dcerpc->smb\_state = STATE\_GOT\_TREE\_CONNECT;

/\* Print out service field \*/

#ifdef DEBUG\_DCERPC\_PRINT

if (\*smb\_data != '\0')

{

printf("Service: %s\n", smb\_data);

}

#endif

/\* Handle next andX command in this packet \*/

if (treeConnX->andXCommand != SMB\_NONE)

{

u\_int16\_t data\_size;

u\_int16\_t andXOffset = smb\_ntohs(treeConnX->andXOffset);

if ( andXOffset > total\_size )

return 0;

/\* Make sure we don't backtrack or look at the same data again \*/

if ( andXOffset <= (data - (u\_int8\_t \*)smbHdr) )

return 0;

/\* Skip header, get size of remaining data \*/

data\_size = total\_size - andXOffset;

/\* Next block is at smbHdr + smb\_ntohs(sess\_setupx\_req->andXOffset) \*/

return ProcessNextSMBCommand(treeConnX->andXCommand, smbHdr,

(u\_int8\_t \*)smbHdr + andXOffset, data\_size, total\_size);

}

return 0;

}

int ProcessSMBNTCreateX(SMB\_HDR \*smbHdr, u\_int8\_t \*data, u\_int16\_t size, u\_int16\_t total\_size)

{

SMB\_NTCREATEX\_REQ \*ntCreateX = (SMB\_NTCREATEX\_REQ \*)data;

#ifdef DEBUG\_DCERPC\_PRINT

int byteCount = smb\_ntohs(ntCreateX->byteCount);

unsigned char \*smb\_data = data + sizeof(SMB\_NTCREATEX\_REQ);

/\* Appears to be a pad in there to word-align if unicode \*/

if (HAS\_UNICODE\_STRINGS(smbHdr))

{

smb\_data++;

byteCount--;

}

if (byteCount > 0)

{

int i=0;

printf("Create/Open: ");

for (i=0;i<byteCount;)

{

if (HAS\_UNICODE\_STRINGS(smbHdr))

{

wprintf(L"%c", smb\_data[i]);

i+=2;

}

else

{

printf("%c", smb\_data[i]);

i++;

}

}

printf("\n");

}

#endif

if ( \_dcerpc->smb\_state == STATE\_GOT\_TREE\_CONNECT )

\_dcerpc->smb\_state = STATE\_GOT\_NTCREATE;

/\* Handle next andX command in this packet \*/

if (ntCreateX->andXCommand != SMB\_NONE)

{

u\_int16\_t data\_size;

u\_int16\_t andXOffset = smb\_ntohs(ntCreateX->andXOffset);

if ( andXOffset >= total\_size )

return 0;

/\* Make sure we don't backtrack or look at the same data again \*/

if ( andXOffset <= (data - (u\_int8\_t \*)smbHdr) )

return 0;

/\* Skip header, get size of remaining data \*/

data\_size = total\_size - andXOffset;

/\* Next block is at smbHdr + smb\_ntohs(sess\_setupx\_req->andXOffset) \*/

return ProcessNextSMBCommand(ntCreateX->andXCommand, smbHdr,

(u\_int8\_t \*)smbHdr + andXOffset, data\_size, total\_size);

}

return 0;

}

**int ProcessSMBWriteX(SMB\_HDR \*smbHdr, u\_int8\_t \*data, u\_int16\_t size, u\_int16\_t total\_size)**

**{**

**SMB\_WRITEX\_REQ \*writeX = (SMB\_WRITEX\_REQ \*)data;**

**u\_int8\_t \*dce\_stub\_data;**

**u\_int16\_t data\_size;**

**/\* Only process WriteAndX packet if it is part of a DCE/RPC session \*/**

**if ( \_dcerpc->smb\_state != STATE\_GOT\_NTCREATE )**

**{**

**return 0;**

**}**

**if ( writeX->dataOffset >= total\_size )**

**{**

**return 0;**

**}**

**------------------------------------------------------------------------------------------------------------------------**

**dce\_stub\_data = (u\_int8\_t \*)smbHdr + writeX->dataOffset;**

**-----------------------------------------------------------------------------------------------------------------------**

**#ifdef DEBUG\_DCERPC\_PRINT**

**if (writeX->dataLength > 0)**

**{**

**int i=0;**

**printf("Write: ");**

**for (i=0;i<writeX->dataLength;i++)**

**{**

**printf("%c", dce\_stub\_data[i]);**

**}**

**printf("\n");**

**}**

**#endif**

**/\* Get size of actual remaining SMB data in packet \*/**

**data\_size = total\_size - (data - (u\_int8\_t \*) smbHdr) - sizeof(SMB\_HDR);**

**SMB\_Fragmentation((u\_int8\_t \*) smbHdr, writeX, dce\_stub\_data, data\_size);**

**/\* Handle next andX command in this packet \*/**

**if (writeX->andXCommand != SMB\_NONE)**

**{**

**u\_int16\_t andXOffset = smb\_ntohs(writeX->andXOffset);**

**if ( andXOffset >= total\_size )**

**return 0;**

**/\* Make sure we don't backtrack or look at the same data again \*/**

**if ( andXOffset <= (data - (u\_int8\_t \*)smbHdr) )**

**return 0;**

**/\* Skip WriteX header, get size of remaining data \*/**

**data\_size = total\_size - andXOffset;**

**/\* Next block is at smbHdr + smb\_ntohs(sess\_setupx\_req->andXOffset) \*/**

**return ProcessNextSMBCommand(writeX->andXCommand, smbHdr,**

**(u\_int8\_t \*)smbHdr + andXOffset, data\_size, total\_size);**

**}**

**return 0;**

**}**

int ProcessSMBTransaction(SMB\_HDR \*smbHdr, u\_int8\_t \*data, u\_int16\_t size, u\_int16\_t total\_size)

{

SMB\_TRANS\_REQ \*trans = (SMB\_TRANS\_REQ \*)data;

u\_int8\_t \*dce\_stub\_data;

u\_int16\_t data\_size;

/\* Only process Trans packet if we think it is part of a DCE/RPC session

NTCREATE state is when we get the bind packet

IS\_DCERPC is when we get a request packet

\*/

if ( \_dcerpc->smb\_state != STATE\_GOT\_NTCREATE )

{

return 0;

}

if ( trans->dataOffset >= total\_size )

{

return 0;

}

/\* We got a Tree Connect followed by a NTCreate, followed by Trans.

Assume DCE/RPC \*/

\_dcerpc->state = STATE\_IS\_DCERPC;

/\* This should be start of the DCE/RPC stub data \*/

dce\_stub\_data = (u\_int8\_t \*)smbHdr + trans->dataOffset;

/\* Get size of actual SMB data in packet \*/

data\_size = total\_size - (data - (u\_int8\_t \*) smbHdr) - sizeof(SMB\_HDR);

ProcessDCERPCMessage((u\_char \*)smbHdr, dce\_stub\_data, data\_size);

#ifdef DEBUG\_DCERPC\_PRINT

if (trans->totalDataCount > 0)

{

int i=0;

printf("Write: ");

for (i=0;i<trans->totalDataCount;i++)

{

printf("%c", dce\_stub\_data[i]);

}

printf("\n");

}

#endif

return 0;

}

int ProcessSMBReadX(SMB\_HDR \*smbHdr, u\_int8\_t \*data, u\_int16\_t size, u\_int16\_t total\_size)

{

SMB\_READX\_REQ \*readX = (SMB\_READX\_REQ \*)data;

u\_int16\_t data\_size;

/\* Handle next andX command in this packet \*/

if (readX->andXCommand != SMB\_NONE)

{

u\_int16\_t andXOffset = smb\_ntohs(readX->andXOffset);

if ( andXOffset >= total\_size )

return 0;

/\* Make sure we don't backtrack or look at the same data again \*/

if ( andXOffset <= (data - (u\_int8\_t \*)smbHdr) )

return 0;

/\* Skip ReadX header, get size of remaining data \*/

data\_size = total\_size - andXOffset;

/\* Next block is at smbHdr + smb\_ntohs(sess\_setupx\_req->andXOffset) \*/

return ProcessNextSMBCommand(readX->andXCommand, smbHdr,

(u\_int8\_t \*)smbHdr + smb\_ntohs(readX->andXOffset), data\_size, total\_size);

}

return 0;

}

#ifdef UNUSED\_SMB\_COMMAND

int ProcessSMBSetupXReq(SMB\_HDR \*smbHdr, u\_int8\_t \*data, u\_int16\_t size, u\_int16\_t total\_size)

{

int extraIndex = 0;

SMB\_SESS\_SETUPX\_REQ\_HDR \*sess\_setupx\_req\_hdr = (SMB\_SESS\_SETUPX\_REQ\_HDR \*)data;

/\* Ptr to first null terminated data element \*/

unsigned char wordCount = smb\_ntohs(sess\_setupx\_req\_hdr->wordCount);

/\* Skip the common header portion, wordCount byte + parameter bytes \* 2 \*/

unsigned char \*smb\_data;

short byteCount = 0, extraBytes = 0;

int skipBytes = 1;

int passwdLen = 0;

char unicodePasswd = 0;

switch (wordCount)

{

case 10:

{

/\* Old session setup andx \*/

SMB\_SESS\_SETUPX\_REQ\_AUTH\_OLD \*sess\_setupx\_auth =

(SMB\_SESS\_SETUPX\_REQ\_AUTH\_OLD \*)

(data + sizeof(SMB\_SESS\_SETUPX\_REQ\_HDR));

passwdLen = smb\_ntohs(sess\_setupx\_auth->passwdLen);

byteCount = extraBytes = smb\_ntohs(sess\_setupx\_auth->byteCount);

smb\_data = data + sizeof(SMB\_SESS\_SETUPX\_REQ\_HDR) +

sizeof(SMB\_SESS\_SETUPX\_REQ\_AUTH\_OLD);

}

break;

case 12:

{

/\* Extended Security session setup andx \*/

SMB\_SESS\_SETUPX\_REQ\_AUTH\_NTLM12 \*sess\_setupx\_auth =

(SMB\_SESS\_SETUPX\_REQ\_AUTH\_NTLM12 \*)

(data + sizeof(SMB\_SESS\_SETUPX\_REQ\_HDR));

passwdLen = 0; /\* Its a blob \*/

byteCount = extraBytes = smb\_ntohs(sess\_setupx\_auth->byteCount);

skipBytes = smb\_ntohs(sess\_setupx\_auth->secBlobLength);

smb\_data = data + sizeof(SMB\_SESS\_SETUPX\_REQ\_HDR) +

sizeof(SMB\_SESS\_SETUPX\_REQ\_AUTH\_NTLM12);

}

break;

case 13:

{

/\* Non-Extended Security session setup andx \*/

SMB\_SESS\_SETUPX\_REQ\_AUTH\_NTLM12\_NOEXT \*sess\_setupx\_auth =

(SMB\_SESS\_SETUPX\_REQ\_AUTH\_NTLM12\_NOEXT \*)

(data + sizeof(SMB\_SESS\_SETUPX\_REQ\_HDR));

if (sess\_setupx\_auth->passwdLen)

{

passwdLen = smb\_ntohs(sess\_setupx\_auth->passwdLen);

unicodePasswd = 1;

}

else if (sess\_setupx\_auth->iPasswdLen)

{

passwdLen = smb\_ntohs(sess\_setupx\_auth->iPasswdLen);

}

byteCount = extraBytes = smb\_ntohs(sess\_setupx\_auth->byteCount);

smb\_data = data + sizeof(SMB\_SESS\_SETUPX\_REQ\_HDR) +

sizeof(SMB\_SESS\_SETUPX\_REQ\_AUTH\_NTLM12\_NOEXT);

}

break;

default:

return -1;

break;

}

size -= sizeof(SMB\_SESS\_SETUPX\_REQ\_HDR);

/\* Password data \*/

if (passwdLen)

{

int i=0;

if ( unicodePasswd )

{

#ifdef DEBUG\_DCERPC\_PRINT

/\* UNICODE Password \*/

wprintf(L"Case Sensitive Password: %.\*s\n", passwdLen, smb\_data);

#endif

/\* Skip past the password -- no terminating NULL \*/

smb\_data += passwdLen;

extraBytes -= passwdLen;

/\* Jump past the pad that re-aligns the next fields \*/

if (HAS\_UNICODE\_STRINGS(smbHdr))

{

smb\_data += 1;

extraBytes -= 1;

}

}

else

{

#ifdef DEBUG\_DCERPC\_PRINT

/\* ASCII Password \*/

printf("Case Insensitive Password: %.\*s\n", passwdLen, smb\_data);

#endif

/\* Skip past the password -- no terminating NULL \*/

smb\_data += passwdLen;

extraBytes -= passwdLen;

/\* Jump past the pad that re-aligns the next fields -- pad

\* is present when ascii password is an even # of bytes. \*/

if (HAS\_UNICODE\_STRINGS(smbHdr) &&

(passwdLen %2 == 0))

{

smb\_data += 1;

extraBytes -= 1;

}

}

for (i=0;i<2;i++)

{

skipBytes = 1;

if (HAS\_UNICODE\_STRINGS(smbHdr))

{

if (\*smb\_data != '\0')

{

#ifdef DEBUG\_DCERPC\_PRINT

printf("%s: ", SESS\_AUTH\_FIELD(extraIndex));

wprintf(L"%s\n", smb\_data);

#endif

skipBytes = SkipBytesWide(smb\_data, size) + 2;

}

}

else

{

if (\*smb\_data != '\0')

{

#ifdef DEBUG\_DCERPC\_PRINT

printf("%s: %s\n", SESS\_AUTH\_FIELD(extraIndex), smb\_data);

#endif

skipBytes = SkipBytes(smb\_data, size) + 1;

}

}

extraIndex++;

smb\_data += skipBytes;

extraBytes -= skipBytes;

}

}

else

{

#ifdef DEBUG\_DCERPC\_PRINT

/\* The security blob... \*/

int i;

printf("Security blob... ");

for (i=0;i<skipBytes;i++)

{

if ( isprint(smb\_data[i]) )

printf("%c ", smb\_data[i]);

else

printf("%.2x ", smb\_data[i]);

}

printf("\n");

#endif

smb\_data += skipBytes;

extraBytes -= skipBytes;

/\* Jump past the NULL Pad (ie fields following are word aligned) \*/

if (skipBytes%2 == 0)

{

smb\_data += 1;

extraBytes -= 1;

}

}

extraIndex = 0;

/\* Some extra data \*/

while (extraBytes > 0)

{

skipBytes = 1;

if (HAS\_UNICODE\_STRINGS(smbHdr))

{

if (\*smb\_data != '\0')

{

#ifdef DEBUG\_DCERPC\_PRINT

printf("%s: ", SESS\_NATIVE\_FIELD(extraIndex));

wprintf(L"%s\n", smb\_data);

#endif

skipBytes = wcslen(smb\_data) + 1;

}

skipBytes \*= 2;

}

else

{

if (\*smb\_data != '\0')

{

#ifdef DEBUG\_DCERPC\_PRINT

printf("%s: %s\n", SESS\_NATIVE\_FIELD(extraIndex), smb\_data);

#endif

skipBytes = strlen(smb\_data) + 1;

}

}

extraIndex++;

smb\_data += skipBytes;

extraBytes -= skipBytes;

}

/\* Handle next andX command in this packet \*/

if (sess\_setupx\_req\_hdr->andXCommand != SMB\_NONE)

{

u\_int16\_t data\_size;

u\_int16\_t andXOffset = smb\_ntohs(sess\_setupx\_req\_hdr->andXOffset);

if ( andXOffset >= total\_size )

return 0;

/\* Make sure we don't backtrack or look at the same data again \*/

if ( andXOffset <= (data - (u\_int8\_t \*)smbHdr) )

return 0;

/\* Skip header, get size of remaining data \*/

data\_size = total\_size - andXOffset;

/\* Next block is at smbHdr + smb\_ntohs(sess\_setupx\_req->andXOffset) \*/

return ProcessNextSMBCommand(sess\_setupx\_req\_hdr->andXCommand, smbHdr,

(u\_int8\_t \*)smbHdr + smb\_ntohs(sess\_setupx\_req\_hdr->andXOffset), data\_size, total\_size);

}

return 0;

}

int ProcessSMBLogoffXReq(SMB\_HDR \*smbHdr, u\_int8\_t \*data, u\_int16\_t size, u\_int16\_t total\_size)

{

SMB\_LOGOFFX\_REQ \*logoffX = (SMB\_LOGOFFX\_REQ \*)data;

int byteCount = smb\_ntohs(logoffX->byteCount);

if (byteCount > 0)

{

return -1;

}

/\* Handle next andX command in this packet \*/

if (logoffX->andXCommand != SMB\_NONE)

{

u\_int16\_t data\_size;

u\_int16\_t andXOffset = smb\_ntohs(logoffX->andXOffset);

if ( andXOffset >= total\_size )

return 0;

/\* Make sure we don't backtrack or look at the same data again \*/

if ( andXOffset <= (data - (u\_int8\_t \*)smbHdr) )

return 0;

/\* Skip header, get size of remaining data \*/

data\_size = total\_size - andXOffset;

/\* Next block is at smbHdr + smb\_ntohs(sess\_setupx\_req->andXOffset) \*/

return ProcessNextSMBCommand(logoffX->andXCommand, smbHdr,

(u\_int8\_t \*)smbHdr + smb\_ntohs(logoffX->andXOffset), data\_size, total\_size);

}

return 0;

}

int ProcessSMBLockingX(SMB\_HDR \*smbHdr, u\_int8\_t \*data, u\_int16\_t size, u\_int16\_t total\_size)

{

SMB\_LOCKINGX\_REQ \*lockingX = (SMB\_LOCKINGX\_REQ \*)data;

unsigned char \*smb\_data = data + sizeof(SMB\_LOCKINGX\_REQ);

u\_int16\_t numUnlocks = smb\_ntohs(lockingX->numUnlocks);

u\_int16\_t numLocks = smb\_ntohs(lockingX->numLocks);

int lockRangeSize;

if (lockingX->lockType & LOCKINGX\_LARGE\_FILES)

{

lockRangeSize = sizeof(SMB\_LARGEFILE\_LOCKINGX\_RANGE);

#ifdef DEBUG\_DCERPC\_PRINT

if (numUnlocks > 0)

{

int i;

printf("Unlocking PIDs: ");

for (i=0;i<numUnlocks;i++)

{

SMB\_LARGEFILE\_LOCKINGX\_RANGE \*lock =

(SMB\_LARGEFILE\_LOCKINGX\_RANGE \*)(smb\_data +

lockRangeSize \* i);

printf("%d ", lock->pid);

}

printf("\n");

}

if (numLocks > 0)

{

int i;

printf("Locking PIDs: ");

for (i=0;i<numLocks;i++)

{

SMB\_LARGEFILE\_LOCKINGX\_RANGE \*lock =

(SMB\_LARGEFILE\_LOCKINGX\_RANGE \*)(smb\_data +

lockRangeSize \* numUnlocks+

lockRangeSize \* i);

printf("%d ", lock->pid);

}

printf("\n");

}

#endif

}

else

{

lockRangeSize = sizeof(SMB\_LOCKINGX\_RANGE);

#ifdef DEBUG\_DCERPC\_PRINT

if (numUnlocks > 0)

{

printf("Unlocking PIDs: ");

for (i=0;i<numUnlocks;i++)

{

SMB\_LOCKINGX\_RANGE \*lock =

(SMB\_LOCKINGX\_RANGE \*)(smb\_data +

lockRangeSize \* i);

printf("%d ", lock->pid);

}

printf("\n");

}

if (numLocks > 0)

{

printf("Locking PIDs: ");

for (i=0;i<numLocks;i++)

{

SMB\_LOCKINGX\_RANGE \*lock =

(SMB\_LOCKINGX\_RANGE \*)(smb\_data +

lockRangeSize \* numUnlocks+

lockRangeSize \* i);

printf("%d ", lock->pid);

}

printf("\n");

}

#endif

}

/\* Handle next andX command in this packet \*/

if (lockingX->andXCommand != SMB\_NONE)

{

u\_int16\_t data\_size;

u\_int16\_t andXOffset = smb\_ntohs(lockingX->andXOffset);

if ( andXOffset >= total\_size )

return 0;

/\* Make sure we don't backtrack or look at the same data again \*/

if ( andXOffset <= (data - (u\_int8\_t \*)smbHdr) )

return 0;

/\* Skip header, get size of remaining data \*/

data\_size = total\_size - andXOffset;

/\* Next block is at smbHdr + smb\_ntohs(sess\_setupx\_req->andXOffset) \*/

return ProcessNextSMBCommand(lockingX->andXCommand, smbHdr,

(u\_int8\_t \*)smbHdr + smb\_ntohs(lockingX->andXOffset), data\_size, total\_size);

}

return 0;

}

#endif /\* UNUSED\_SMB\_COMMAND \*/