13

Protocol Negotiation

I don't have an accent.
—Oh yes you do.

CIFS is a very rich and varied protocol suite, a fact that is evident in the number of SMB dialects that exist. Five are listed in the X/Open SMB protocol specification, and the SNIA doc — published ten years later — lists eleven. That's a bigbunch, and they probably missed a few. Each new dialect may add new SMBs, deprecate old ones, or extend existing ones. As if that were not enough, implementations introduce subtle variations within dialects.

All that in mind, our goal in this section will be to provide an overview of the available dialects, cover the workings of the `NEGOTIATE PROTOCOL` SMB exchange, and take a preliminary peek at some of the concepts that we have yet to consider (things like virtual circuits and authentication). For the most part, the examples and discussion will be based on the "NT LM 0.12" dialect. The majority of the servers currently available support some variation of NT LM 0.12, and at least one client implementation (jCIFS) has managed to get by without supporting any others. Server writers should be warned, however, that there really are a lot of clients still around that use older calls. Even new clients will use older calls, simply because of the difficulty of acquiring reliable documentation on the newer stuff.

13.1 A Smattering of SMB Dialects

In keeping with tradition, the list of dialects is presented as a table with the dialect name in the left-hand column and a short description in the right, ordered from oldest to newest. Most of the references to these dialects seem to do it this way. Our list is not quite as complete as you might find elsewhere. The aim here is to highlight some of the better-known examples in order to provide a bit of context for the examination of the `SMB_COM_NEGOTIATE` message.

Where relevant, important differences between dialects will be noted. It would be very difficult, however, to try to document all of the features of each dialect and all of the changes between them. If you really, really need to know more (which is likely, if you are working on server code) see the SNIA doc, the X/Open doc, the expired IETF drafts, and the other old Microsoft documentation that is still freely available from their FTP server.[1]

SMB dialects

Dialect Identifier	Notes
`PC NETWORK PROGRAM 1.0`	Also known as the **Core Protocol**. This is the original stuff, as documented in `COREP.TXT`. According to ancient lore, this dialect is sometimes also identified by the string "`PCLAN1.0`".
`MICROSOFT NETWORKS 1.03`	This is the **Core Plus Protocol**. It extends a few Core Protocol SMB commands, and adds a few new ones.
`MICROSOFT NETWORKS 3.0`	Known as the **Extended 1.0 Protocol** or **LAN Manager 1.0**. This dialect was created when IBM and Microsoft were working together on OS/2. This particular variant was designed for DOS clients, which understood a narrower set of error codes than OS/2.
`LANMAN1.0`	Identical to the `MICROSOFT NETWORKS 3.0` dialect except that it was intended for use with OS/2 clients, so a larger set of error codes was available. OS/2 and DOS both expect that the `STATUS` field will be in the DOS-style `ErrorClass` / `ErrorCode` format. Again, this dialect is also known as **LAN Manager 1.0** or as the **Extended 1.0 Protocol**.

1. ...or was, last time I checked. Once again, that URL is: `ftp://ftp.microsoft.com/developr/drg/CIFS/`. See the References section for links to specific documents.

SMB dialects

Dialect Identifier	Notes
`LM1.2X002`	Called the **Extended 2.0 Protocol**; also known as **LAN Manager 2.0**. This dialect represents OS/2 LANMAN version 2.0, and it introduces a few new SMBs. The identifier for the DOS version of this dialect is "`DOS LM1.2X002`". As before, the key difference between the DOS and OS/2 dialects is simply that the OS/2 version provides a larger set of error codes.
`LANMAN2.1`	Called the **LAN Manager 2.1** dialect (no surprise there), this version is documented in a paper titled *Microsoft Networks SMB File Sharing Protocol Extensions, Document Version 3.4*. You can find it by searching the web for a file named "`SMB-LM21.DOC`". You will likely need a conversion tool of some sort in order to read the file, as it is encoded in an outdated form of a proprietary Microsoft format (it's a word-processing file). The cool thing about the `SMB-LM21.DOC` document is that instead of explaining how LANMAN2.1 works it describes how LANMAN2.1 differs from its predecessor, LANMAN2.0. That's useful for people who want to know how the protocol has evolved.
`Samba`	You may see this dialect listed in the protocol negotiation request coming from a Samba-based client such as `smbclient`, KDE Konqueror (which uses Samba's `libsmbclient` library), or the Linux `SMBFS` implementation. No one from the Samba Team seems to remember when, or why, this was added. It doesn't appear to be used any more (if, indeed, it ever was).
`NT LM 0.12`	This dialect, sometimes called NT LANMAN, was developed for use with Windows NT. All of the Windows 9x clients also claim to speak it, as do Windows 2000 and XP. As mentioned above, this is currently the most widely supported dialect. It is, quite possibly, also the sloppiest with all sorts of variations and differing implementations.
`CIFS`	Following the release of the IETF CIFS protocol drafts, many people thought that Microsoft would produce a "CIFS" dialect, and many documents refer to it. No such beast has actually materialized, however. Maybe that's a good thing.

Section 3.16 of the SNIA *CIFS Technical Reference, V1.0* provides a list of of SMB message types categorized by the dialect in which they were introduced. There is also a slightly more complete list of dialects in Section 5.4 of the SNIA doc.

13.2 Greetings: The NEGOTIATE PROTOCOL REQUEST

We have already provided a detailed breakdown of a NEGOTIATE PROTOCOL REQUEST SMB (back in Section 11.3 on page 186), so we don't need to go to the trouble of fully dissecting it again. The interesting part of the request is the data section (the parameter section is empty). If we were to write a client that supported all of the dialects in our chart, the NEGOTIATE_PROTOCOL_REQUEST.SMB_DATA field would break out something like this:

```
SMB_DATA
  {
  ByteCount = 131
  Bytes
    {
    Dialect[0] = "\x02PC NETWORK PROGRAM 1.0"
    Dialect[1] = "\x02MICROSOFT NETWORKS 1.03"
    Dialect[2] = "\x02MICROSOFT NETWORKS 3.0"
    Dialect[3] = "\x02LANMAN1.0"
    Dialect[4] = "\x02LM1.2X002"
    Dialect[5] = "\x02LANMAN2.1"
    Dialect[6] = "\x02Samba"
    Dialect[7] = "\x02NT LM 0.12"
    Dialect[8] = "\x02CIFS"
    }
  }
```

Each dialect string is preceded by a byte containing the value 0x02. This, perhaps, was originally intended to make it easier to parse the buffer. In addition to the 0x02 prefix the dialect strings are nul-terminated, so if you go to the trouble of counting up the bytes to see if the ByteCount value is correct in this example don't forget to add 1 to each string length.

Listing 13.1 provides code for creating a NEGOTIATE PROTOCOL REQUEST message. It also takes care of writing an NBT Session Message header for us — something we must not forget to do.

Listing 13.1: Negotiate Protocol Request

```
/* Define the SMB message command code.
 */
#define SMB_COM_NEGOTIATE 0x72

int nbt_SessionHeader( uchar *bufr, ulong size )
  /* ------------------------------------------------- **
   * This function writes the NBT Session Service header.
   * Note that we use NBT byte order, not SMB.
   * ------------------------------------------------- **
   */
  {
  if( size > 0x0001FFFF ) /* That's the NBT maximum. */
    return( -1 );
  bufr[0] = 0;
  bufr[1] = (size >> 16) & 0xFF;
  bufr[2] = (size >>  8) & 0xFF;
  bufr[3] = size & 0xFF;
  return( (int)size );
  } /* nbt_SessionHeader */

int smb_NegProtRequest( uchar  *bufr,
                        int     bsize,
                        int     namec,
                        uchar **namev )
  /* ------------------------------------------------- **
   * Build a Negotiate Protocol Request message.
   * ------------------------------------------------- **
   */
  {
  uchar *smb_bufr;
  int    i;
  int    length;
  int    offset;
  ushort bytecount;
  uchar  flags;
  ushort flags2;

  /* Set aside four bytes for the session header. */
  bsize    = bsize - 4;
  smb_bufr = bufr + 4;
```

```
/* Make sure we have enough room for the header,
 * the WORDCOUNT field, and the BYTECOUNT field.
 * That's the absolute minimum (with no dialects).
 */
if( bsize < (SMB_HDR_SIZE + 3) )
  return( -1 );

/* Initialize the SMB header.
 * This zero-fills all header fields except for
 * the Protocol field ("\ffSMB").
 * We have already tested the buffer size so
 * we can void the return value.
 */
(void)smb_hdrInit( smb_bufr, bsize );

/* Hard-coded flags values...
 */
flags  = SMB_FLAGS_CANONICAL_PATHNAMES;
flags |= SMB_FLAGS_CASELESS_PATHNAMES;
flags2 = SMB_FLAGS2_KNOWS_LONG_NAMES;

/* Fill in the header.
 */
smb_hdrSetCmd(    smb_bufr, SMB_COM_NEGOTIATE );
smb_hdrSetFlags(  smb_bufr, flags );
smb_hdrSetFlags2( smb_bufr, flags2 );

/* Fill in the (empty) parameter block.
 */
smb_bufr[SMB_HDR_SIZE] = 0;

/* Copy the dialect names into the message.
 * Set offset to indicate the start of the
 * BYTES field, skipping BYTECOUNT.  We will
 * fill in BYTECOUNT later.
 */
offset = SMB_HDR_SIZE + 3;
for( bytecount = i = 0; i < namec; i++ )
  {
  length = strlen(namev[i]) + 1;       /* includes nul  */
  if( bsize < (offset + 1 + length) )  /* includes 0x02 */
    return( -1 );
  smb_bufr[offset++] = '\x02';
  (void)memcpy( &smb_bufr[offset], namev[i], length );
  offset += length;
  bytecount += length + 1;
  }
```

```
/* The offset is now the total size of the SMB message.
 */
if( nbt_SessionHeader( bufr, (ulong)offset ) < offset )
  return( -1 );

/* The BYTECOUNT field starts one byte beyond the end
 * of the header (one byte for the WORDCOUNT field).
 */
smb_SetShort( smb_bufr, (SMB_HDR_SIZE + 1), bytecount );

/* Return the total size of the packet.
 */
return( offset + 4 );
} /* smb_NegProtRequest */
```

13.3 Gesundheit: The NEGOTIATE PROTOCOL RESPONSE

The NEGOTIATE PROTOCOL RESPONSE SMB is more complex than the request. In addition to the dialect selection, it also contains a variety of other parameters that let the client know the capabilities, limitations, and expectations of the server. Most of these values are stuffed into the SMB_PARAMETERS block, but there are a few fields defined in the SMB_DATA block as well.

13.3.1 *NegProt Response Parameters*

The NEGOTIATE_PROTOCOL_RESPONSE.SMB_PARAMETERS.Words block for the NT LM 0.12 dialect is 17 words (34 bytes) in size, and is structured as shown below. Earlier dialects use a different structure and, of course, the server should always match the reply to the dialect it selects.

```
typedef struct
  {
  uchar WordCount;              /* Always 17 for this struct */
  struct
    {
    ushort DialectIndex;        /* Selected dialect index    */
    uchar  SecurityMode;        /* Server security flags     */
    ushort MaxMpxCount;         /* Maximum Multiplex Count   */
    ushort MaxNumberVCs;        /* Maximum Virtual Circuits  */
    ulong  MaxBufferSize;       /* Maximum SMB message size  */
    ulong  MaxRawSize;          /* Obsolete                  */
    ulong  SessionKey;          /* Unique session ID         */
    ulong  Capabilities;        /* Server capabilities flags */
    ulong  SystemTimeLow;       /* Server time; low bytes    */
    ulong  SystemTimeHigh;      /* Server time; high bytes   */
    short  ServerTimeZone;      /* Minutes from UTC; signed  */
    uchar  EncryptionKeyLength; /* 0 or 8                    */
    } Words;
  } smb_NegProt_Rsp_Params;
```

That requires a lot of discussion. Let's tear it up and take a close look at the tiny pieces.

`DialectIndex`

Things start off fairly simply. The `DialectIndex` field contains the index of the dialect string that the server has selected, which will be the highest-level dialect that the server understands. The dialect strings are numbered starting with zero, so to choose "NT LM 0.12" from the list in the example request the server would return 7 in the `DialectIndex` field.

`SecurityMode`

`SecurityMode` is a bitfield that provides some information about the authentication sub-protocol that the server is expecting. Four flag bits are defined; they are described below. Challenge/Response and **M**essage **A**uthentication **C**ode (MAC) message signing will be explained later (this is becoming our mantra), when we cover authentication. It will take a little while to get there, but keep your eyes open for additional clues along the way.

SecurityMode

Bit	Name / Bitmask / Values	Description
7–4	0xF0	**<Reserved> (must be zero)**
3	NEGOTIATE_SECURITY_SIGNATURES_REQUIRED 0x08 0: Message signing is optional 1: Message signing is required	If set, this bit indicates that the server is requiring the use of a **M**essage **A**uthentication **C**ode (MAC) in each packet. If the bit is clear then message signing is optional. This bit should be zero if the next bit (mask 0x04) is zero.
2	NEGOTIATE_SECURITY_SIGNATURES_ENABLED 0x04 0: Message signing not supported 1: Server can perform message signing	If set, the server is indicating that it is capable of performing **M**essage **A**uthentication **C**ode (MAC) message signing. This bit should be zero if the next bit (mask 0x02) is zero.
1	NEGOTIATE_SECURITY_CHALLENGE_RESPONSE 0x02 0: Plaintext Passwords 1: Challenge/Response	This bit indicates whether or not the server supports Challenge/Response authentication (which will be covered further on). If the bit is clear, then plaintext passwords must be used. If set, the server may (optionally) reject plaintext authentication. If this bit is clear and the client rejects the use of plaintext, then there is no way to perform the logon and the client will be unable to connect to the server.

SecurityMode

Bit	Name / Bitmask / Values	Description
0	NEGOTIATE_SECURITY_USER_LEVEL 0x01 0: Share Level 1: User Level	Ah! Finally something we've already covered! This bit indicates whether the server, as a whole, is operating under Share Level or User Level security. Share and User Level security were explained along with the TID and UID header fields, back in Section 12.4 on page 209.

MaxMpxCount

Remember the PID and MID fields in the header? They could be used to multiplex several sessions over a single TCP/IP connection. The thing is, the server might not be able to handle more than a fixed number of total outstanding requests.

The MaxMpxCount field lets the server tell the client how many requests, in total, it can handle concurrently. It is the client's responsibility to ensure that there are no more than MaxMpxCount outstanding requests in the pipe at any time. That may mean that client processes will block, waiting for their turn to send an SMB.

MaxNumberVCs

The MaxNumberVCs field specifies the maximum number of **V**irtual **C**ircuits (VCs) that the server is able to accommodate. VCs are yet another mechanism by which multiple SMB sessions could, in theory, be multiplexed over a single transport-layer session. Note the use of the phrase "in theory." The dichotomy between theory and practice is a recurring theme in the study of CIFS.

MaxBufferSize

MaxBufferSize is the size (in bytes) of the largest message that the server can receive.

Keep in mind that the transport layer will fragment and defragment packets as necessary. It is, therefore, possible to send very large SMBs and let the lower layers worry about ensuring safe, fast, reliable delivery.

How big can an SMB message be?

In the NT LM 0.12 dialect, the `MaxBufferSize` field is an unsigned longword. As described much earlier on, however, the `Length` field in the NBT `SESSION MESSAGE` is 17 bits wide and the naked transport header has a 24-bit `Length` field. So the session headers place slightly more reasonable limits on the maximum size of a single SMB message.

`MaxRawSize`

This is the maximum size of a raw data buffer.

The X/Open doc describes the `READ RAW` and `WRITE RAW` SMBs, which were introduced with the Extended 1.0 version of SMB (the `MICROSOFT NETWORKS 3.0` and `LANMAN1.0` dialects). These were a speed hack. For a large read or write operation, the first message would be a proper SMB, but subsequent messages would be sent in "raw" mode, with no SMB or session header. The raw blocks could be as large as `MaxRawSize` bytes in length. Once again, the transport layer was expected to take care of fragmentation/defragmentation as well as resending of any lost packets.

Raw mode is not used much any more. Among other things, it conflicts with message signing because the raw messages have no header in which to put the MAC Signature. Thus, the `MaxRawSize` field is considered obsolete.[2]

`SessionKey`

The `SessionKey` is supposed to be used to identify the session in which a VC has been opened. Documentation on the use of this field is very

2. There may be a further problem with raw mode. Microsoft has made some obtuse references to obscure patents which may or may not be related to `READ RAW` and `WRITE RAW`. The patents in question have been around for quite some time, and were not mentioned in any of the SMB/CIFS documentation that Microsoft released up until March of 2002. Still, the best bet is to avoid `READ RAW` and `WRITE RAW` (since they are not particularly useful anyway) and/or check with a patent lawyer. The Samba Team released a statement regarding this issue, see `http://us1.samba.org/samba/ms_license.html`.

poor, however, and the commentary in various mailing list archives shows that there is not much agreement about what to do with it.

In theory, the `SessionKey` value should be echoed back to the server whenever the client sends a `SESSION SETUP` request. Samba's `smbclient` does this, but some versions of jCIFS always reply with zero, and they don't seem to have any trouble with it. In testing, it also appears that Windows 2000 servers do not generate a session key. They send zero in `NEGOTIATE PROTOCOL RESPONSE` messages. Hmmm...

It would seem that the use of this field was never clearly defined — anywhere by anyone — and that most servers really don't care what goes there. It is probably safest if the client echoes back the value sent by the server.

Capabilities

This is a grab-bag bitfield, similar in style to the `FLAGS` and `FLAGS2` fields in the header except, of course, that it is not included in every message. The bits of the `Capabilities` field indicate specific server features of which the client may choose to take advantage.

We are already building up a backlog of unexplained features. We will also postpone the discussion of the `Capabilities` field until we get some of the other stuff out of the way.

SystemTimeLow and SystemTimeHigh

The `SystemTime` fields are shown as two unsigned longs in the SNIA doc. We might write it as:

```
typedef struct
  {
  ulong timeLow;
  ulong timeHigh;
  } smb_Time;
```

Keeping byte order in mind, the completed time value should be read as two little-endian 32-bit integers. The result, however, should be handled as a 64-bit *signed* value representing the number of tenths of a microsecond since January 1, 1601, 00:00:00.0 UTC.

WHAT?!?!

Yes, you read that right folks. The time value is based on that unwieldy little formula. Read it again five times and see if you don't get a headache. Looks as though we need to get out the protractor, the astrolabe,

and the didgeridoo and try a little calculating. Let's start with some complex scientific equations:

- 1 microsecond = 10^{-6} seconds,
- 1/10 microsecond = 10^{-7} seconds.

In other words, the server time is given in units of 10^{-7} seconds.[3] Many CIFS implementations handle these units by converting them into Unix-style measurements. Unix, of course, bases its time measurements on an equally obscure date: January 1, 1970, 00:00:00.0 UTC.[4] Converting between the two schemes requires knowing the difference (in seconds) between the two base times.

Email

```
       From: Andrew Narver
In-Reply-To: A message from Mike Allen sent to Microsoft's CIFS
             mailing list and the Samba-Technical mailing list.

> (what's the number of seconds between 1601 and 1970 again?)

Between Jan 1, 1601 and Jan 1, 1970, you have 369 complete
years, of which 89 are leap years (1700, 1800, and 1900 were
not leap years). That gives you a total of 134774 days or
11644473600 seconds.
```

So, if you want to convert the `SystemTime` to a Unix `time_t` value, you need to do something like this:

```
unix_time = (time_t)(((smb_time)/10000000) - 11644473600);
```

which gives you the server's system time in seconds since January 1, 1970, 00:00:00.0 UTC.

3. There is no name for 10^{-7} seconds. Other fractions of seconds have names with prefixes like deci, centi, milli, micro, nano, pico, even zepto, but there is no prefix that applies to 10^{-7}. In honor of the fact that this rare measure of time is used in the CIFS protocol suite, I propose that it be called a **bozo**second.

4. January 1, 1970, 00:00:00.0 UTC, known as "the Epoch," is sometimes excused as being the approximate birthdate of Unix.

ServerTimeZone

ServerTimeZone, of course, is the timezone in which the server believes it resides. It is represented as an offset relative to UTC, in minutes. Minutes, that is. Multiply by 60 in order to get seconds, or 600,000,000 to get tenths of a microsecond.

The available documentation (the SNIA doc and the Leach/Naik IETF draft) states that this field is an *un*signed short integer. They're wrong. The field is a signed value which is subtracted from the SystemTime to give local time.

If, for example, your server is located in the beautiful city of Saint Paul, Minnesota, it would be in the US Central timezone[5] which is six hours west of UTC. The value in the ServerTimeZone field would, therefore be 360 minutes. (Except, of course, during the summer when Daylight Savings Time is in effect, in which case it would be 300 minutes.) On the other hand, if your server is in Moscow in the winter, the ServerTimeZone value would be –180.

The basic rule of thumb:

```
LocalTime = SystemTime - ( ServerTimeZone × 600000000 )
```

...which returns local time in units of 10^{-7} seconds, based on January 1601 as described above.

If you found all of that to be complicated, you will be relieved to know that this is only one of many different time formats used in SMB. Time And Date Encoding is covered in Section 3.7 of the SNIA doc.

EncryptionKeyLength

This is the last field in the NEGOTIATE_PROTOCOL_RESPONSE.SMB_PARAMETERS block. It provides the length, in bytes, of the Challenge used in Challenge/Response authentication. SMB Challenges, if present, are always 8 bytes long, so the EncryptionKeyLength will have a value of either 8 or 0 — the latter if Challenge/Response authentication is not in use.

---

5. This is probably because Saint Paul is at the center of the universe. The biomagnetic center of the universe used to be located across the river in Minneapolis until they closed it down. It was a little out of whack in the same way that the magnetic poles are not quite where they should be. The magnetic north pole, for instance, is on or near an island in northern Canada instead of at the center of the Arctic Ocean where it belongs.

The name of this field is probably a hold-over from some previous enhancement to the protocol — still in use for "historical reasons."

Wow... a lot of stuff there. No time to sit and chat about it right now, though. We still need to finish out the examination of the `NEGOTIATE_PROTOCOL_RESPONSE.SMB_DATA` block.

13.3.2 *NegProt Response Data*

`SMB_DATA`, of course, is handed to us as an array of bytes with the length provided in the `ByteCount` field. The parsing of those bytes depends upon the values in the `SMB_PARAMETER` block that we just examined. The structure is completely different depending upon whether Extended Security has been negotiated.

Here is what it looks like, more or less, in the NT LM 0.12 dialect:

```
typedef struct
  {
  ushort ByteCount;          /* Number of bytes to follow  */
  union
    {
    struct
      {
      uchar GUID[16];        /* 16-byte Globally Unique ID */
      uchar SecurityBlob[];  /* Auth-system dependent      */
      } ext_sec;             /* Extended Security          */
    struct
      {
      uchar EncryptionKey[]; /* 0 or 8 bytes long          */
      uchar DomainName[];    /* nul-terminated string      */
      } non_ext_sec;         /* Non-Extended Security      */
    } Bytes;
  } smb_NegProt_Rsp_Data;
```

The first thing to note is that this `SMB_DATA.Bytes` block structure is the union of two smaller structures:

- `ext_sec` is used if Extended Security has been negotiated,
- `non_ext_sec` is used otherwise.

The second thing to note is that this is pseudo-code, not valid C code. Some of the array lengths are unspecified because we don't know the byte-length of the fields ahead of time. In real code, you will probably need to use

pointers or some other mechanism to extract the variable-length data from the buffer.

Okay, let's chop that structure into little bits...

`GUID`

`GUID` stands for **G**lobally **U**nique **ID**entifier. The `GUID` field is always 16 bytes long.

As of this writing, research by Samba Team members shows that this value is probably the same as the `GUID` identifier used by Active Directory to keep track of servers in the database. Standalone servers (which are not listed in any Active Directory) also generate and use a `GUID`. Go figure.

Though this field is only present when Extended Security is enabled, it is not, strictly speaking, a security field. The value is well known and easily forged. It is not clear (yet) why this field is even sent to the client. In testing, a Samba server was configured to fill the `GUID` field with its own 16-byte Server Service NetBIOS name... and that worked just fine.

`SecurityBlob`

The `SecurityBlob` is — as the name says — a blob of security information. In other words, it is a block of data that contains authentication information particular to the Extended Security mechanism being used. Obviously, this field will need to be covered in the Authentication section.

The `SecurityBlob` is variable in length. Fortunately, the `GUID` field is always 16 bytes, so the length of the `SecurityBlob` is `(ByteCount - 16)` bytes.

`EncryptionKey`

This field should be called `Challenge` because that's what it actually contains — the Challenge used in Challenge/Response authentication. The SMB Challenge, if present, is always eight bytes long. If plaintext passwords are in use then there is no Challenge, the `EncryptionKey` will be empty, and the `SMB_PARAMETERS.EncryptionKeyLength` field will contain `0`.

`DomainName`

This field sometimes contains the NetBIOS name of the Workgroup or NT Domain to which the server belongs. (We have talked a bit, in previous sections, about Workgroups and NT Domains so the terms should

be somewhat familiar.) In testing, Samba servers always provided a name in the `DomainName` field; Windows systems less reliably so. Windows 98, for example, would sometimes provide a value and sometimes not.[6]

The SNIA doc calls this field the `OEMDomainName` and claims that the characters will be eight-bit values using the OEM character set of the server (that's the 7-bit ASCII character set augmented by an extended DOS code page which defines characters for the upper 128 octet values). In fact, this field may contain either a string of 8-bit OEM characters or a Unicode string with 16-bit characters. The value of `SMB_HEADER.FLAGS2.SMB_FLAGS2_UNICODE_STRINGS` will let you know how to read the `DomainName` field.

13.4 Are We There Yet?

Okay, let's be honest... Ripping apart that `NEGOTIATE PROTOCOL RESPONSE` SMB was about as exciting as the epic saga of undercooked toast. It doesn't get any better than that, though, and there's a lot more of it. Implementing SMB is a game of patience and persistence. It also helps if you get a cheap thrill from fiddly little details. (Just don't go parsing your packets in public or people will look at you funny.)

It seems, too, that our overview of the SMB Header and the `NEGOTIATE PROTOCOL` exchange has left a bit of a mess on the floor. We have pulled a lot of concepts off of the shelves and out of the closets, and we will need to do some sorting and organizing before we can put them back. Let's see what we've got:

- Opportunistic Locks (OpLocks), which were taking up space in the `SMB_HEADER.FLAGS` field,
- Virtual Circuits (we found these in the box labeled `MaxNumberVCs`),

6. A lot of time was wasted trying to figure out which configuration options would change the behavior. The results were inconclusive. At first it seemed as though the DomainName was included if the Windows 98 system was running in User Level security mode and passing logins through to an NT Domain Controller. Further testing, however, showed that this was not a hard-and-fast rule. It should also be mentioned that if the systems are running naked transport there may not be an NT Domain or Workgroup name. SMB can be mightily inconsistent — but not all the time.

- The `Capabilities` bits (and pieces),
- Distributed File System (DFS), which spilled out when `FLAGS2` fell open,
- Character Encoding — which seems to get into everything, sort of like cat hair and dust,
- Extended vs. DOS Attributes,
- Long vs. short names, and...
- Authentication, including plaintext passwords, Challenge/Response, Extended Security, and Packet Signing.

The only way to approach all of these topics is one-at-a-time. ...but first, take another break. Every now and then, it is a good idea to stop and think about what has been covered so far. This is one of those times. We have finished tearing apart SMB headers and the body of a `NEGOTIATE PROTOCOL` message. That should provide some familiarity with the overall structure of SMBs. Try doing some packet captures, or skim through the SNIA *CIFS Technical Reference*. It should all begin to make a little more sense now than it did when we started.