15

Authentication

Car locks are there
to keep the honest people honest.

— Something my brother Robert
once told me. (He sells cars.)

Now for the big one...

If you are familiar with authentication schemes, then this section should be comfortable for you. If not, then perhaps it's time for a fresh pot of tea. Some people find their first experience with the innards of password security to be a bit intimidating, possibly because the encryption formulae are sometimes made to look a lot like mathematics. Authentication itself isn't really that complex, though. The basic idea is that the would-be users need to prove that they are who they say they are in order to get what they want. The proof is usually in the form of something private or secret — something that only the user has or knows.

Consider, for example, the key to an automobile (something you have). With the key in hand, you are able to unlock the door, turn the ignition switch, and start the engine. As far as the car is concerned, you have proven that you have the right to drive. Likewise with the password you use to access your computer (something you know). If you enter a valid username/password pair at the login prompt, then you can access the system. Unfortunately passwords,

like keys, can be stolen or forged or copied. Just as locks can be picked, so passwords can be cracked.[1]

In the early days of SMB, when the LANs were small and sheltered, there was very little concern for the safety of the password itself. It was sent in plaintext (unencrypted) over the wire from the client to the server. Eventually, though, corporate networks got bigger, modems were installed to provide access from home and on the road, the "disgruntled employee" boogeyman learned how to use a keyboard, and everything got connected to the Internet. These were hard times for plaintext passwords, so a series of schemes was developed to keep the passwords safe — each more complex than its predecessor.

For SMB, the initial attempt was called LAN Manager Challenge/Response authentication, often simply abbreviated "LM." The LM scheme turned out to be too simple and too easy to crack, and was replaced with something stronger called Windows NT Challenge/Response (known as "NTLM"). NTLM was superseded by NTLMv2 which has, in turn, been replaced with a modified version of MIT's Kerberos system.

Got that?

We'll go through them all in various degrees of detail. The LM algorithm is fairly simple, so we can provide a thorough description. At the other extreme, Kerberos is an entire system unto itself and anything more than an overview would be overkill.

15.1 Anonymous and Guest Login

Gather and study piles of SMB packet captures and you will notice that some `SESSION SETUP` requests contain no username and password at all. These are *anonymous* logins, and they are used to access special-purpose SMB shares such as the hidden "`IPC$`" share (the **I**nter-**P**rocess **C**ommunications share).

---

1. In addition to "something you have" and "something you know" there is another class of access tokens sometimes described as "something you are." This latter class, also known as "biometrics," includes such things as your fingerprints, your DNA pattern, your brainwaves, and your karmic aura. Some folks have argued that these features are simply "something you have" that is a little harder (or more painful) to steal. There was great hope that biometrics would offer improvements over the other authentication tokens, but it seems that they may be just as easy to crack. For example, a group of researchers in Japan was able to fool fingerprint scanners using fake fingertips created from gelatin and other common ingredients.

You can learn more about `IPC$` in Part III on page 335. Put simply, though, this share allows one system to query another using RAP function calls.

Anonymous login may be a design artifact; something created in the days of Share Level security when it seemed safe to leave a share unprotected, and still with us today because it cannot easily be removed. Maybe not. One guess is as good as another.

"GUEST" account logons are also often sent sans password. The guest login is sometimes used in the same way as the anonymous login, but there are additional permissions which a guest account may have. Guest accounts are maintained like other "normal" accounts, so they can be a security problem and are commonly disabled. When SMB is doing its housekeeping, the anonymous login is generally preferred over the guest login.

15.2 Plaintext Passwords

This is the easiest SMB authentication mechanism to implement — and the least secure. It's roughly equivalent to leaving your keys in the door lock after you've parked the car. Sure, the car is locked, but...

Plaintext passwords may still be sufficient for use in small, isolated networks, such as home networks or small office environments (assuming no disgruntled employees and a well-configured firewall on the uplink — or no Internet connection at all). Plaintext passwords also provide us with a nice opportunity to get our feet wet in the mired pool of authentication. We can look at the packets and clearly see what is happening on the wire. Note, however, that many newer clients are configured to prevent the use of plaintext. Windows clients have registry entries that must be twiddled in order to permit plaintext passwords, and jCIFS did not support them at all until version 0.7.

In order to set up a workable test environment you will need a server that does not expect encrypted passwords, and a client that doesn't mind sending the passwords in the clear. That is *not* an easy combination to come by. Most contemporary SMB clients and servers disable plaintext by default. It is easy, however, to configure Samba so that it requests unencrypted passwords. Just change the `encrypt passwords` parameter to `no` in the `smb.conf` file, like so:

```
; Disable encrypted passwords.
encrypt passwords = no
```

Don't forget to signal `smbd` to reload the configuration file after making this change.

On the client side we will, once again, use the jCIFS `Exists` utility in our examples. If you would rather use a Windows client for your own tests, you can find a collection of helpful registry settings in the `docs/Registry/` subdirectory of the Samba distribution. You will probably need to change the registry settings to permit the Windows client to send plaintext passwords. Another option as a testing tool is Samba's `smbclient` utility, which does not seem to argue if the server tells it not to encrypt the passwords.

This is what our updated `Exists` test looks like:

shell

```
$ java -DdisablePlainTextPasswords=false Exists \
> smb://pat:p%40ssw0rd@smedley/home
smb://pat:p%40ssw0rd@smedley/home exists
$
```

A few things to note:

- The `-DdisablePlainTextPasswords=false` command-line option tells jCIFS that it should permit the use of plaintext.
- The username and password are passed to jCIFS via the SMB URL. The syntax is fairly common for URLs.[2] Basically, it looks like this:

  ```
  smb://[[user[:password]@]host[:port]]
  ```

  The username in our example is `pat`.
- The password in our example is `p@ssw0rd`, but the '@' in the password conflicts with the '@' used to separate the `userinfo` field from the `hostport` field.[3] To resolve the conflict we encode the '@' in `p@ssw0rd` using the URL escape sequence "`%40`", which gives us `p%40ssw0rd`.

---

2. ...sort of. Support for inclusion of a password within a URL is considered very dangerous. The recommendation from the authors of RFC 2396 is that new applications should not recognize the password field and that the application should instead prompt for both the username and password.

3. Yet again we seek the wisdom of the RFCs. See Appendix A of RFC 2396 for the full generic syntax of URLs, and RFC 2732 for the IPv6 update.

- If at all possible, applications should be written to request the password in a more secure fashion, and to hide it once it has been given. The `[:password]` syntax is not part of the general URL syntax definition, and its use is highly discouraged. Having the password display on the screen is as naughty as sending it across the wire in plaintext.

15.2.1 *User Level Security with Plaintext Passwords*

User and Share Level security were described back in Section 12.4 on page 209, along with the `TID` and `[V]UID` header fields. The `SecurityMode` field of the `NEGOTIATE PROTOCOL RESPONSE` SMB will indicate the authentication expectations of the server. For User Level plaintext passwords, the value of the `SecurityMode` field will be `0x01`.

Below is an example `SESSION_SETUP_ANDX.SMB_DATA` block such as would be generated by the jCIFS `Exists` tool. Note, once again, that the discussion is focused on the NT LM 0.12 dialect.

```
SMB_DATA
 {
 ByteCount = 27
 Bytes
   {
   CaseInsensitivePassword = "p@ssw0rd"
   CaseSensitivePassword   = <NULL>
   Pad                     = <NULL>
   AccountName             = "PAT"
   PrimaryDomain           = "?"
   NativeOS                = "Linux"
   NativeLanMan            = "jCIFS"
   }
 }
```

There are always fiddly little details to consider when working with SMB. In this case, we need to talk about upper- and lowercase. (bLeCH.) The example above shows that the `AccountName` field has been converted to uppercase. This is common practice, but it is not really necessary and some implementations don't bother. It is a holdover from the early days of SMB when lots of things (filenames, passwords, share names, NetBIOS names, bagels, and pop singers) were converted to uppercase as a matter of course. Some older servers (pre-NT LM 0.12) may require uppercase usernames, but newer servers

shouldn't care. Converting to uppercase is probably the safest option, just in case...

Although the `AccountName` in the example is uppercase, the `CaseInsensitivePassword` is not. Hmmm... Odd, eh? The situation here is that some server operating systems (e.g. most Unixy OSes) use case-sensitive password verification algorithms. If the password is sent all uppercase it probably won't match what the OS expects, resulting in a login failure even though the user entered the correct password. The field may be labeled case-insensitive (and that really is what it is *intended* to be) but some server OSes prefer to have the original password, case preserved, just as the user entered it.

This is a sticky problem, though, because some clients *insist* on converting passwords to uppercase before sending them to the server. Windows 95 and '98 may do this, for example. As you might have come to expect by now, the reason for this odd behavior is backward compatibility. There are older (pre-NT LM 0.12) servers still running that will reject passwords that are not all uppercase. Windows 9x systems solve the problem by forcing all passwords to uppercase even when the NT LM 0.12 dialect has been selected. Samba's `smbd` server, which generally runs on case-sensitive platforms, must go through a variety of contortions to get uppercase plaintext passwords to be accepted.[4]

Another annoyance is that Windows 98 will pad the plaintext password string to 24 bytes, filling the empty space with semi-random garbage. This behavior was noted in testing, but there wasn't time to investigate the problem in-depth so it may or may not be wide-spread. Still, it's the odd case that will break things. Server implementors should be careful to both check the field length *and* look for the first terminating nul byte when reading the plaintext password.

In short, client-side handling of the plaintext `CaseInsensitive-Password` is inconsistent and problematic — and the server has to compensate. That's why you need piles of SMB packet captures and lots of different clients to test against when writing a server implementation. It *can* be done, but it takes a bit of perseverance. When writing a new client, ensure that the client sends the password as the user intended. If that fails, and the dialect is pre-NT LM 0.12, then convert to uppercase and try again. Believe it or not, the use of challenge/response authentication bypasses much of this trouble.

---

4. See the discussion of the `password level` parameter in Samba's `smb.conf(5)` documentation for more information about these problems.

...but that's only half the story. In addition to the `CaseInsensitivePassword` field there is also a `CaseSensitivePassword` field in the data block, and we haven't even touched on that yet. This latter field is only used if Unicode has been negotiated, and it is rare that both Unicode and plaintext will be used simultaneously. It can happen, though. As mentioned earlier, Samba can be easily configured to provide support for Unicode plaintext passwords.[5] In theory, this should be a simple switch from ASCII to Unicode. In practice, no client really supports it yet — and weird things have been seen on the wire. For example:

- Clients disagree on the length of the Unicode password string in `CaseSensitivePassword`. Some count the pair of nul bytes that terminate the string, others do not. (For comparison, the length of the ASCII `CaseInsensitivePassword` string does include the terminating nul, so it seems there is precedent.)
- In testing, more than one client stored the length of the Unicode password in the `CaseInsensitivePasswordLength` field... but that's where the ASCII password length is supposed to go. The Unicode password length should be in the `CaseSensitivePasswordLength` field. How should the server interpret the password in this situation — as ASCII or Unicode?
- One client added a nul byte at the beginning of the Unicode password string, probably intended as a padding byte to force word alignment. The extra nul byte was being read as the first byte of the `CaseSensitivePassword`, thus misaligning the Unicode string. Another client went further and counted the extra byte in the total length of the Unicode password string. As a result, the password length was given as an odd number of bytes (which should never happen).

Empirically, it would seem that Unicode plaintext passwords were never meant to be.

---

5. I don't know whether a Windows server can be configured to support Unicode plaintext passwords. To test against Samba, however, you need to use Samba version 3.0 or above. On the client side, Microsoft has a Knowledge Base article — and a patch — that addresses some of the message formatting problems in Windows 2000 (see *Microsoft Knowledge Base Article #257292*). Thanks to Nir Soffer for finding this article.

An interesting fact-ette that can be gleaned from this discussion is that there is a linkage between the password fields and the negotiation of Unicode. Simply put:

ASCII (OEM character set) <==> `CaseInsensitivePassword`
Unicode <==> `CaseSensitivePassword`

That is, ASCII plaintext passwords are stored in the `CaseInsensitivePassword` field, and Unicode plaintext passwords should be placed into the `CaseSensitivePassword` field. Indeed, Ethereal names these two fields, respectively, "ANSI Password" and "Unicode Password" instead of using the longer names shown above. This relationship carries over to the challenge/response passwords as well, as we shall soon see.

15.2.2 *Share Level Security with Plaintext Passwords*

We won't spend too much time on this. It is easy to see by looking at packet captures. Basically, in Share Level security mode the plaintext password is passed to the server in the `TREE CONNECT ANDX` request instead of the `SESSION SETUP ANDX`. In the NT LM 0.12 dialect, however, a valid username should also be placed into the `SESSION SETUP AccountName` field if at all possible. Doing so allows the server to map Share Level security to its own user-based authentication system.

Interesting Implementation Alert

Samba does not completely implement Share Level security. Though all of the required SMBs are supported, Samba does not provide any way to assign a password to a share.

Many SMB clients will provide a username (if one is available) in the `SESSION SETUP ANDX` *SMB even though it is not (technically) required at Share Level. If there is no username available, however, Samba will attempt (through various methods — some of which might be considered kludgey) to guess an appropriate username for the connection. Read through the* `smb.conf(5)` *manual page if you are interested in the details.*

15.3 LM Challenge/Response

In plaintext mode, the client proves that it knows the password by sending the password itself to the server. In challenge/response mode, the goal is to prove that the password is known without risking any exposure. It's a bit of a magic trick. Here's how it's done:

1. The server generates a random string of bytes — random enough that it is not likely to come up again in a very, very long time (thus preventing replay attacks). This string is called the *challenge*.
2. The challenge is sent to the client.
3. Both the client and server encrypt the challenge using a key that is derived from the user's password. The client sends its result back to the server. This is the *response*.
4. If the client's response matches the server's result, then the server knows (beyond a reasonable doubt) that the client knows the correct key. If they don't match, authentication fails.

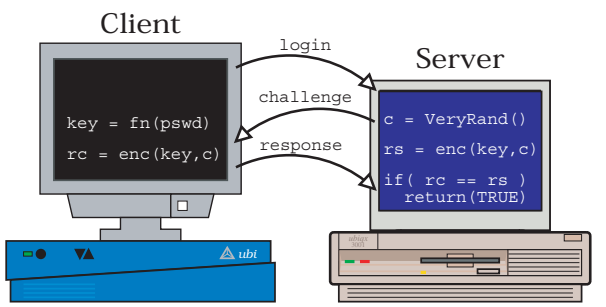


Figure 15.1: *Challenge/response*

The server generates a random challenge, which it sends to the client. Both systems encrypt the challenge using the secret encryption key. The client sends its result (`rc`) to the server. If the client's result matches the server's result (`rs`), then the two nodes have matching keys.

That's a rough, general overview of challenge/response. The details of its use in LAN Manager authentication are a bit more involved, but are fairly easy to explain. As we dig deeper, keep in mind that the goal is to protect the password while still allowing authentication to occur. Also remember that LM challenge/response was the first attempt to add encrypted password support to SMB.

15.3.1 *DES*

The formula used to generate the LM response makes use of the U.S. Department of Commerce **D**ata **E**ncryption **S**tandard (DES) function, in block mode. DES has been around a long time. There are a lot of references which describe it and a good number of implementations available, so we will not spend a whole lot of time studying DES itself.[6] For our purposes, the important thing to know is that the DES function — as used with SMB — takes two input parameters and returns a result, like so:

```
result = DES( key, source );
```

The `source` and `result` are both eight-byte blocks of data, the `result` being the DES encryption of the `source`. In the SNIA doc, as in the Leach/Naik draft, the `key` is described as being seven bytes (56 bits) long. Documentation on DES itself gives the length of the `key` as eight bytes (64 bits), but each byte contains a parity bit so there really are only 56 bits worth of "`key`" in the 64-bit `key`. As shown in Figure 15.2, there is a simple formula for mapping 56 bits into the required 64-bit format. The seven byte string is simply copied, seven *bits* at a time, into an eight byte array. A parity bit (odd parity) is inserted following each set of seven bits (but some existing DES implementations use zero and ignore the parity bit).

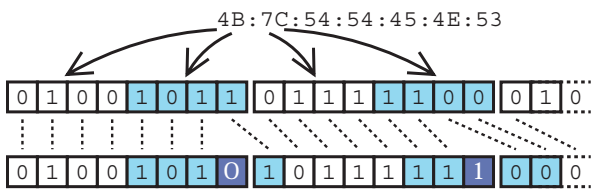


Figure 15.2: *DES key manglement*

Converting a seven byte key (56 bits) into an eight byte key with odd parity for use with DES. Some DES implementations perform this step internally.

The `key` is used by the DES algorithm to encrypt the `source`. Given the same `key` and `source`, DES will always return the same `result`.

---

6. If you are interested in the workings of DES, Bruce Schneier's *Applied Cryptography, Second Edition* provides a very complete discussion. See the References section.

15.3.2 *Creating the Challenge*

The challenge needs to be very random, otherwise the logon process could be made vulnerable to "replay" attacks.

A replay attack is fairly straightforward. The attacker captures the exchange between the server and the client and keeps track of the challenge, the response, and the username. The attacker then tries to log on, hoping that the challenge will be repeated (this step is easier if the challenge is at all predictable). If the server sends a challenge that is in the stored list, the attacker can use the recorded username and response to fake a logon. No password cracking required.

Given that the challenge is eight bytes (64 bits) long, and that random number generators are pretty good these days, it is probably best to create the challenge using a random number function. The better the random number generator, the lower the likelihood (approaching 1 in 2^{64}) that a particular challenge will be repeated.

The X/Open doc (which was written a long time ago) briefly describes a different approach to creating the challenge. According to that document, a seven-byte pseudo-random number is generated using an internal counter and the system time. That value is then used as the `key` in a call to `DES()`, like so:

```
Ckey = fn( time( NULL ), counter++ );
challenge = DES( Ckey, "????????" );
```

(The `source` string is honest-to-goodnessly given as eight question marks.)

That formula actually makes a bit of sense, though it's probably overkill. The pseudo-random `Ckey` is non-repeating (because it's based on the time), so the resulting `challenge` is likely to be non-repeating as well. Also note that the pseudo-random value is passed as the `key`, not the `source`, in the call to `DES()`. That makes it much more difficult to reverse and, since it changes all the time, reversing it is probably not useful anyway.

As Andrew Bartlett[7] points out, however, the time and counter inputs are easily guessed so the challenge is predictable, which is a potential weakness. Adding a byte or two of truly random "salt" to the `Ckey` in the recipe above would prevent such predictability.

---

7. ...without whom the Authentication section would never have been written.

Email

```
   From: Andrew Bartlett
     To: Chris Hertel
Subject: SMB Challenge...

Actually, given comments I've read on some SMB cracking sites, it
would not surprise me if MS still does (or at least did) use exactly
this for the challenges.

I still think you should address the X/Open function not as
'overkill' but as 'flawed'.
```

Using a plain random number generator is probably faster, easier, and safer.

15.3.3 *Creating the LM Hash*

LM challenge/response authentication prevents password theft by ensuring that the plaintext password is never transmitted across a network or stored on disk. Instead, a separate value known as the "LM Hash" is generated. It is the LM Hash that is stored on the server side for use in authentication, and used on the client side to create the response from the challenge.

The LM Hash is a sixteen byte string, created as follows:

1. The password, as entered by the user, is either padded with nuls (`0x00`) or trimmed to fourteen (14) bytes.[8]
   - Note that the 14-byte password string is not handled as a nul-terminated string. If the user enters 14 or more bytes, the last byte in the modified string will *not* be nul.
   - Also note that the password is given in the 8-bit OEM character set (extended ASCII), not Unicode.
2. The 14-byte password string is converted to all uppercase.
3. The uppercase 14-byte password string is chopped into two 7-byte keys.

8. Both the the X/Open doc and the expired Leach/Naik draft state that the padding character is a space, not a nul. They are incorrect. It really is a nul.

4. The seven-byte keys are each used to DES-encrypt the string constant "KGS!@#$%", which is known as the "magic" string.[9]
5. The two 8-byte results are then concatenated together to form the 16-byte LM Hash.

That outline would make a lot more sense as code, wouldn't it? Well, you're in luck. Listing 15.1 shows how the steps given above might be implemented.

Listing 15.1: LM Hash function

```
static const uchar SMB_LMHash_Magic[8] =
  { 'K', 'G', 'S', '!', '@', '#', '$', '%' };

uchar *smb_LMHash( const uchar *password, uchar *result )
  /* ------------------------------------------------- **
   * Generate an LM Hash.
   * password - pointer to the raw password string.
   * result   - pointer to at least 16 bytes of memory
   *            into which the LM Hash will be written.
   * Returns a pointer to the LM Hash (== result).
   * ------------------------------------------------- **
   */
  {
  uchar  tmp_pass[14] = { 0,0,0,0,0,0,0,0,0,0,0,0,0,0 };
  uchar *hash;
  uchar  K1[7];
  uchar  K2[7];
  int    i;

  /* Copy at most 14 bytes of password to tmp_pass.
   * If the string is shorter, the unused bytes will
   * be nul-filled.
   */
  (void)strncpy( tmp_pass, password, 14 );

  /* Convert to uppercase.
   */
  for( i = 0; i < 14; i++ )
    tmp_pass[i] = toupper( tmp_pass[i] );
```

9. The magic string was considered secret, and was not listed in the Leach/Naik draft. The story of Tridge and Jeremy's (pre-DMCA) successful effort to reverse-engineer this value is quite entertaining.

```
/* Split tmp_pass into two 7-byte keys.
 */
(void)memcpy( K1, tmp_pass, 7 );
(void)memcpy( K2, (tmp_pass + 7), 7 );

/* Use the keys to encrypt the 'magic' string.
 * Place each encrypted half into the result
 * buffer.
 */
hash = DES( K1, SMB_LMHash_Magic );
(void)memcpy( result, hash, 8 );
hash = DES( K2, SMB_LMHash_Magic );
(void)memcpy( (result + 8), hash, 8 );

/* Return a pointer to the result.
 */
return( result );
} /* smb_LMHash */
```

15.3.4 *Creating the LM Response*

Now we get to the actual logon. When a `NEGOTIATE PROTOCOL REQUEST` arrives from the client, the server generates a new challenge on the fly and hands it back in the `NEGOTIATE PROTOCOL RESPONSE`.

On the client side, the user is prompted for the password. The client generates the LM Hash from the password, and then uses the hash to DES-encrypt the challenge. Of course, it's not a straightforward DES operation. As you may have noticed, the LM Hash is 16 bytes but the `DES()` function requires 7-byte keys. Ah, well... Looks as though there's a bit more padding and chopping to do.

1. The password entered by the user is converted to a 16-byte LM Hash as described above.
2. The LM Hash is padded with five nul bytes, resulting in a string that is 21 bytes long.
3. The 21 byte string is split into three 7-byte keys.
4. The challenge is encrypted three times, once with each of the three keys derived from the LM Hash.
5. The results are concatenated together, forming a 24-byte string which is returned to the server. This, of course, is the response.

Once again, we provide demonstrative code. Listing 15.2 shows how the LM Response would be generated.

Listing 15.2: LM Response function

```
uchar *smb_LMResponse( const uchar *LMHash,
                            uchar *chal,
                            uchar *resp )
  /* ------------------------------------------------- **
   * Generate an LM Response
   * LMHash - pointer to the LM Hash of the password.
   * chal   - Pointer to the challenge.
   * resp   - pointer to at least 24 bytes of memory
   *          into which the LM response will be written.
   * Returns a pointer to the LM response (== resp).
   * ------------------------------------------------- **
   */
  {
  uchar  P21[21];
  uchar  K[7];
  uchar *result;
  int    i;

  /* Copy the LM Hash to P21 and pad with nuls to 21 bytes.
   */
  (void)memcpy( P21, LMHash, 16 );
  (void)memset( (P21 + 16), 0, 5 );

  /* A compact method of splitting P21 into three keys,
   * generating a DES encryption of the challenge for
   * each key, and combining the results.
   * (i * 7) will give 0, 7, 14 and
   * (i * 8) will give 0, 8, 16.
   */
  for( i = 0; i < 3; i++ )
    {
    (void)memcpy( K, (P21 + (i * 7)) , 7 );
    result = DES( K, chal );
    (void)memcpy( (resp + (i * 8)), result, 8 );
    }

  /* Return the response.
   */
  return( resp );
  } /* smb_LMResponse */
```

The server, which has the username and associated LM Hash tucked away safely in its authentication database, also generates the 24-byte response string. When the client's response arrives, the server compares its own value against the client's. If they match, then the client has authenticated.

Under User Level security, the client sends its LM Response in the `SESSION_SETUP_ANDX.CaseInsensitivePassword` field of the `SESSION SETUP` request (yes, the LM *response* is in the `SESSION SETUP REQUEST`). With Share Level security, the LM Response is placed in the `TREE_CONNECT_ANDX.Password` field.

15.3.5 *LM Challenge/Response: Once More with Feeling*

The details sometimes obfuscate the concepts, and vice versa. We have presented a general overview of the challenge/response mechanism, as well as the particular formulae of the LAN Manager scheme. Let's go through it once again, quickly, just to put the pieces together and cover anything that we may have missed.

The LM Hash

The LM Hash is derived from the password. It is used instead of the password so that the latter won't be exposed. A copy of the LM Hash is stored on the server (or Domain Controller) in the authentication database.

On the down side, the LM Hash is *password equivalent*. Because of the design in the LM challenge/response mechanism, a cracker[10] can use the LM Hash to break into a system. The password itself is not, in fact, needed. Thus, the LM Hash must be protected as if it were the password.

The Challenge

If challenge/response is required by the server, the `SecurityMode` field of the `NEGOTIATE PROTOCOL RESPONSE` will have bit `0x02` set, and the challenge will be found in the `EncryptionKey` field.

---

10. A "cracker," not a "hacker." The former is someone who cracks passwords or authentication schemes with the goal of cracking into a system (naughty). The latter is one who studies and fiddles with software and systems to see how they work and, possibly, to make them work better (nice). The popular media has mangled the distinction. Don't make the same mistake. If you are reading this book, you most likely are a hacker (and that's good).

Challenge/response may be used with either User Level or Share Level security.

The Logon

On the client side, the user will — at some point — be prompted for a password. The password is converted into the LM Hash. Meanwhile, the server (or NT Domain Controller) has its own copy of the LM Hash, stored in the authentication database. Both systems use the LM Hash to generate the LM Response from the challenge.

The LM Response

The client sends the LM Response to the server in either the `SESSION_SETUP_ANDX.CaseInsensitivePassword` field or the `TREE_CONNECT_ANDX.Password` field, depending upon the security level of the server. The server compares the client's response against its own to see if they match.

The `SESSION SETUP ANDX RESPONSE`

To finish up, the server will send back a `SESSION SETUP ANDX RESPONSE`. The `STATUS` field will indicate whether the logon was successful or not.

Well, that's a lot of work and it certainly goes a long way towards looking complicated. Unfortunately, looking complicated isn't enough to truly protect a password. LM challenge/response is an improvement over plaintext, but there are some problems with the formula and it turns out that it is not, in fact, a very *big* improvement.

Let's consider what an attacker might do to try and break into a system. We've already explained the replay attack. Other common garden varieties include the "dictionary" and the "brute force" attack, both of which simply try pushing possible passwords through the algorithm until one of them returns the same response as seen on the wire. The dictionary attack is typically faster because it uses a database of likely passwords, so tools tend to try this first. The brute force method tries all (remaining) possible combinations of bytes, which is usually a longer process. Unfortunately, all of the upper-casing, nul-padding, chopping, and concatenating used in the LM algorithm makes LM challenge/response very susceptible to these attacks. Here's why:

The LM Hash formula pads the original password with nul bytes. If the password is short enough (seven or fewer characters) then, when the 14-byte

padded password is split into two seven-byte DES keys, the second key will always be a string of seven nuls. Given the same input, DES produces the same output:

```
0xAAD3B435B51404EE = DES( "\0\0\0\0\0\0\0", "KGS!@#$%" )
```

which results in an LM Hash in which the second set of eight bytes are known:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
result 0								result 1							
??	??	??	??	??	??	??	??	AA	D3	B4	35	B5	14	04	EE

To create the LM Response, the LM Hash is padded with nuls to 21 bytes, and then split again into three DES keys:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
key 0							key 1							key 2						
??	??	??	??	??	??	??	??	AA	D3	B4	35	B5	14	04	EE	00	00	00	00	00

Now the problem is obvious. If the original password was seven bytes or less, then almost two-thirds of the encryption key used to generate the LM Response will be a known, constant value. The password cracking tools leverage this information to reduce the size of the *keyspace* (the set of possible passwords) that needs to be tested to find the password. Less obvious, but clear enough if you study the LM Response algorithm closely, is that short passwords are only part of the problem. Because the hash is created in pieces, it is possible to attack the password in 7-byte chunks even if it is longer than 7 bytes.

Converting to uppercase also diminishes the keyspace, because lowercase characters do not need to be tested at all. The smaller the keyspace, the faster a dictionary or brute-force attack can run through the possible options and discover the original password.[11]

---

11. Jeremy Allison proved it could be done with a little tool called `PWdump`. Mudge and other folks at the L0pht then expanded on the idea and built the now semi-infamous `L0phtCrack` tool. In July of 1997, Mudge posted a long and detailed description of the decomposition of LM challenge/response, a copy of which can be found at: `http://www.insecure.org/sploits/l0phtcrack.lanman.problems.html`. For a curious counterpoint, see *Microsoft Knowledge Base Article #147706*.

15.4 NTLM Challenge/Response

At some point in the evolution of Windows NT a new, improved challenge/response formula was introduced. It was similar to the LAN Manager version, with the following changes:

1. Instead of using the uppercase ASCII (OEM character set) password, NTLM challenge/response generates the hash from the mixed-case Unicode (UCS-2LE) representation of the password. This change alone makes the password much more difficult to crack.
2. Instead of the `DES()` function, NTLM uses the `MD4()` message digest function described in RFC 1320. This function produces a 16-byte hash (the NTLM Hash)[12] but requires no padding or trimming of the input (though the resulting 16-byte NTLM Hash is still padded with nuls to 21 bytes for use in generating the NTLM Response.)
3. The NTLM Response is sent to the server in the `SESSION_SETUP_ANDX.CaseSensitivePassword` field.

...and that's basically it. The rest of the formula is the same.

So what does it buy us?

The first advantage of NTLM is that the passwords are more complex. They're mixed case and in Unicode, which means that the keyspace is much larger. The second advantage over LM is that the `MD4()` function doesn't require fixed length input. That means no padding bytes and no chopping to over-simplify the keys. The NTLM Hash itself is more robust than the LM Hash, so the NTLM Response is much more difficult to reverse.

Unfortunately, the NTLM Response is still created using the same algorithm as is used with LM, which provides only 56-bit encryption. Worse, clients often include *both* the NTLM Response *and* the LM Response (derived from the weaker LM Hash) in the `SESSION SETUP ANDX REQUEST`. They do this to maintain backward compatibility with older servers. Even if the server refuses to accept the LM Response, the client has sent it. Ouch.

---

12. Andrew Bartlett prefers to call this the "NT Hash," stating that the NT Hash is passed through the LM response algorithm to produce the NTLM (NT+LM) response.

Brain Overflow Alert

The next section describes the NTLMv2 algorithm. It's not really that difficult, but it can get tedious — especially if your head is still swimming from the LM and NTLM algorithms. Jerry Carter of the Samba Team warns that your brain may explode if you try to understand it all the first time through. (Most veteran CIFS engineers have had this happen at least twice.)

*You may want to skim through Section 15.5 and possibly Section 15.9, which describes **M**essage **A**uthentication **C**odes (MACs). You can always come back and read them again after you've iced your cranium.*

15.5 NTLM Version 2

NTLMv2, as it's called, has some additional safeguards thrown into the recipe that make it more complex — and hopefully more secure — than its predecessors. There are, however, two small problems with NTLMv2:

- Good documentation on the inner workings of NTLMv2 is rare.
- Although it is widely available, NTLMv2 does not seem to be widely used.

Regarding the first point, Appendix B of Luke K. C. Leighton's book *DCE/RPC over SMB: Samba and Windows NT Domain Internals* provides a recipe for NTLMv2 authentication. We'll do our best to expand on Luke's description. The other option, of course, is to look at available Open Source code.

The second point is really a conjecture, based in part on the fact that it took a very long time to get NTLMv2 implemented in Samba and few seemed to care. Indeed, NTLMv2 support had already been added to Samba-TNG by Luke and crew, and needed only to be copied over. It seems that the delay in adding it to Samba was not a question of know-how, but of priorities.

Another factor is that NTLMv2 is not required by default on most Windows systems. When challenge/response is negotiated, even newer Windows versions will default to using the LM/NTLM combination unless they are specifically configured not to.

15.5.1 *The NTLMv2 Toolbox*

We have already fussed with the DES algorithm and toyed with the MD4 algorithm. Now we get to use the HMAC-MD5 Message Authentication Code hash. This one's a power tool with razor-sharp keys and swivel-action hashing. The kind of thing your Dad would never let you play with when you were a kid. Like all good tools, though, it's neither complex nor dangerous once you learn how it works.

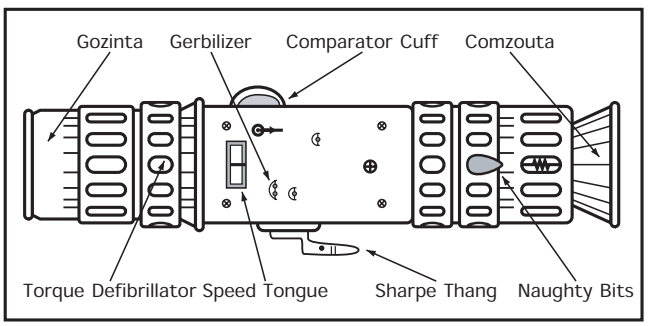


Figure 15.3: *HMAC-MD5*

The HMAC-MD5 is a popular tool for use in message authentication. It is lightweight, powerful, efficient, and ergonomic.

HMAC-MD5 is actually a combination of two different algorithms: HMAC and MD5. HMAC is a **M**essage **A**uthentication **C**ode (MAC) algorithm that takes a hashing function (such as MD5) and adds a secret key to the works so that the resulting hash can be used to verify the *authenticity* of the data. The MD5 algorithm is basically an industrial-strength version of MD4. Put them together and you get HMAC-MD5.

HMAC-MD5 is quite well documented,[13] and there are a lot of implementations available. It's also much less complicated than it appears in Figure 15.3, so we won't need to go into any of the details. For our purposes, what you need to know is that the `HMAC_MD5()` function takes a key and some source data as inputs, and returns a 16-byte (128-bit) output.

---

13. MD4 is explained in RFC 1320 and MD5 is in RFC 1321; HMAC in general, and HMAC-MD5 in particular, is written up in RFC 2104 — an embarrassment of riches! As usual with this sort of thing, a deeper understanding can be gained by reading about it in Bruce Schneier's *Applied Cryptography, Second Edition* (see the References section).

Hmmm... Well, it's not actually quite that simple. See, MD4, MD5, and HMAC-MD5 all work with variable-length input, so they also need to know how big their input parameters are. The function call winds up looking something like this:

```
hash16 = HMAC_MD5( Key, KeySize, Data, DataSize );
```

There is, as it turns out, more than one way to skin an HMAC-MD5. Some implementations use a whole set of functions to compute the result:

- the first function accepts the key and creates an initial *context*,
- the second function may be called repeatedly, each time passing the context and the next block of data,
- and the final function is used to close the context and return the resulting hash.

Conceptually, though, the multi-function approach is the same as the simpler example shown above. That is: `Key` and `Data` in, 16-byte hash out.

Not Quite Entirely Unlike Standard Alert

The HMAC-MD5 function can handle very large `Key` *inputs. Internally, though, there is a maximum keysize of 64 bytes. If the key is too long, the function uses the MD5 hash of the key instead. In other words, inside the* `HMAC_MD5()` *function there is some code that does this:*

```
if( KeySize > 64 )
  {
  Key = MD5( Key, KeySize );
  KeySize = 16;
  }
```

In his book, Luke explains that the function used by Windows systems is actually a variation on HMAC-MD5 known as HMACT64, which can be quickly defined as follows:

```
#define HMACT64( K, Ks, D, Ds ) \
        HMAC_MD5( K, ((Ks > 64)?64:Ks), D, Ds )
```

In other words, the `HMACT64()` *function is the same as* `HMAC_MD5()` *except that it truncates the input* `Key` *to 64 bytes rather than hashing it down to 16 bytes using the* `MD5()` *function as prescribed in the specification.*

As you read on, you will probably notice that the keys used by the NTLMv2 challenge/response algorithm are never more that 16 bytes, so the distinction is moot for our purposes. We bother to explain it only because `HMACT64()` *may be used*

elsewhere in CIFS (in some dark corner that we have not visited) and it might be a useful tidbit of information for you to have.

Another important tool is the older NTLM hash algorithm. It was described earlier but it is simple enough that we can present it again, this time in pseudo-code:

```
uchar *NTLMhash( uchar *password )
  {
  UniPasswd = UCS2LE( password );
  KeySize   = 2 * strlen( password );
  return( MD4( UniPasswd, KeySize ) );
  }
```

The ASCII password is converted to Unicode UCS-2LE format, which requires two bytes per character. The `KeySize` is simply the length of that (Unicode) password string, which we calculate here by doubling the ASCII string length (which is probably cheating). Finally, we generate the MD4 hash (that's MD*4*, not MD*5*) of the password, and that's all there is to it.

Note that the string terminator is not counted in the `KeySize`. That is common behavior for NTLM and NTLMv2 challenge/response when working with Unicode strings.

The NTLM Hash is of interest because the SMB/CIFS designers at Microsoft (if indeed such people truly exist any more, except in legend) used it to cleverly avoid upgrade problems. With LM and NTLM, the hash is created from the password. Under NTLMv2, however, the older NTLM (v1) Hash is used *instead of the password* to generate the new hash. A server or Domain Controller being upgraded to use NTLMv2 may already have the older NTLM hash values in its authentication database. The stored values can be used to generate the new hashes — no password required. That avoids the nasty chicken-and-egg problem of trying to upgrade to NTLMv2 Hashes on a system that only allows NTLMv2 authentication.

15.5.2 *The NTLMv2 Password Hash*

The NTLMv2 Hash is created from:

- the NTLM Hash (which, of course, is derived from the password),
- the user's username, and
- the name of the logon destination.

The process works as shown in the following pseudo-code example:

```
v1hash  = NTLMhash( password );
UniUser = UCS2LE( upcase( user ) );
UniDest = UCS2LE( upcase( destination ) );
data    = uni_strcat( UniUser, UniDest );
datalen = 2 * (strlen( user ) + strlen( destination ));
v2hash  = HMAC_MD5( v1hash, 16, data, datalen );
```

Let's clarify that, shall we?

v1hash

The NTLM Hash, calculated as described previously.

UniUser

The username, converted to uppercase UCS-2LE Unicode.

UniDest

The NetBIOS name of either the SMB server or NT Domain against which the user is trying to authenticate.

data

The two Unicode strings are concatenated and passed as the `Data` parameter to the `HMAC_MD5()` function.

datalen

The length of the concatenated Unicode strings, excluding the nul termination. Once again, doubling the ASCII string lengths is probably cheating.

v2hash

The NTLM Version 2 Hash.

A bit more explanation is required regarding the `destination` value (which gets converted to `UniDest`).

In theory, the client can use NTLMv2 challenge/response to log into a standalone server *or* to log into an NT Domain. In the former case, the server will have an authentication database of its very own, but an NT Domain logon requires authentication against the central database maintained by the Domain Controllers.

So, in theory, the destination name could be either the NetBIOS name of the standalone server *or* the NetBIOS name of the NT Domain (no NetBIOS

suffix byte in either case). In practice, however, the server logon doesn't seem to work reliably. The Windows systems used in testing were unable to use NTLMv2 authentication with one another when they were in standalone mode, but once they joined the NT Domain NTLMv2 logons worked just fine.[14]

15.5.3 *The NTLMv2 Response*

The NTLMv2 Response is calculated using the NTLMv2 Hash as the `Key`. The `Data` parameter is composed of the challenge plus a blob of data which we will refer to as "the blob." The blob will be explained shortly. For now, just think of it as a mostly-random bunch of garblement. The formula is shown in this pseudo-code example:

```
blob = RandomBytes( blobsize );
data = concat( ServerChallenge, 8, blob, blobsize );
hmac = HMAC_MD5( v2hash, 16, data, (8 + blobsize) );
v2resp = concat( hmac, 16, blob, blobsize );
```

Okay, let's take a closer look at that and see if we can force it to make some sense.

1. The first step is blob generation. The blob is normally around 64 bytes in size, give or take a few bytes. The pseudo-code above suggests that the bytes are entirely random, but in practice there is a formula (explained below) for creating the blob.
2. The next step is to append the blob to the end of the challenge. This, of course, is the same challenge sent by the server and used by all of the other challenge/response mechanisms.

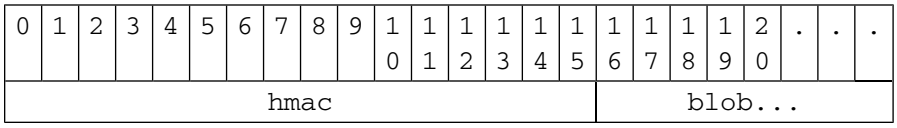
0	1	2	3	4	5	6	7	8	9	10	11	12	.	.	.
challenge								blob...							

3. The challenge and blob are HMAC'd using the NTLMv2 Hash as the key.

---

14. The lab in the basement is somewhat limited which, in turn, limits my ability to do rigorous testing of esoteric CIFS nuances. You should probably verify these results yourself. Andrew Bartlett (him again!) turned up an interesting quirk regarding the NTLMv2 Response calculation when authenticating against a standalone server. It seems that the NT Domain name is left blank in the v2hash calculation. That is: `destination = "";`

4. The NTLMv2 Response is created by appending the blob to the tail of the `HMAC_MD5()` result. That's 16 bytes of HMAC followed by `blobsize` bytes of blob.



If the client sends the NTLMv2 Response, it will take the place of the NTLM Response in the `SESSION_SETUP_ANDX.CaseSensitive-Password` field. Note that, unlike the older NTLM Response, the NTLMv2 Response algorithm uses 128-bit encryption all the way through.

15.5.4 *Creating The Blob*

If you have ever taken a college-level Invertebrate Zoology course, you may find the dissection of the blob to be nauseatingly familiar. The rest of you... try not to be squeamish. One more warning before we cut into this: The blob's structure may not matter at all. We'll explain why a little later on.

Okay, now that the disclaimers are out of the way, we can get back to work. The blob does have a structure, which is more or less as follows:

4 bytes

The value seen in testing is consistently `0x01010000`. (Note that those are nibbles, not bits.) The field is broken out as follows:

1 byte

Response type identification number. The only known value is `0x01`.

1 byte

The identification number of the maximum response type that the client understands. Again, the only known value is `0x01`.

2 bytes

Reserved. Must be zero (`0x0000`).

4 bytes

The value seen in testing is always `0x00000000`. This field may, however, be reserved for some purpose.

8 bytes

A timestamp, in the same 64-bit format as described back in Section 13.3.1 on page 227.

8 bytes

The "blip": An eight-byte random value, sometimes referred to as the "Client Challenge." More on this later, when we talk about LMv2 challenge/response.

4 bytes

Unknown.

Comments in the Samba-TNG code and other sources suggest that this is meant to be either a 4-byte field or a pair of 2-byte fields. These fields should contain offsets to other data. That interpretation is probably based on empirical observation, but in the testing done for this book there was no pattern to the data in these fields. It may be that some implementations provide offsets and others just fill this space with left-over buffer garbage. Variety is the spice of life.

variable length

A list of structures containing NetBIOS names in Unicode.

4 bytes

Unknown. (Appears to be more buffer garbage.)

The list of names near the end of the blob may contain the NT Domain and/or the server name. As with the names used to generate the NTLMv2 Hash, these are NetBIOS names in uppercase UCS-2LE Unicode with no string termination and no suffix byte. The name list also has a structure:

2 bytes

Name type.

`0x0000`

Indicates the end of the list.

0x0001
The name is a NetBIOS machine name (e.g. a server name).

0x0002
The name is an NT Domain NetBIOS name.

0x0003
The name is the server's DNS hostname.

0x0004
The name is a W2K Domain name (a DNS name).

2 bytes
The length, in bytes, of the name. If the name type is `0x0000`, then this field will also be `0x0000`.

variable length
The name, in uppercase UCS-2LE Unicode format.

The blob structure is probably related to (the same as?) data formats used in the more advanced security systems available under Extended Security.[15]

15.5.5 *Improved Security Through Confusion*

Now that we have the formula worked out, let's take a closer look at the NTLMv2 challenge/response algorithm and see how much better it is than NTLM.

With the exception of the password itself, all of the inputs to NTLMv2 are known or knowable from a packet capture. Even the blob can be read off the wire, since it is sent as part of the response. That means that the problem is still a not-so-simple case of solving for a single variable: the password.

---

15. Luke Kenneth Casson Leighton's book *DCE/RPC over SMB: Samba and Windows NT Domain Internals* gives an outline of the structure of the data blob used in NTLMv2 Response creation. Using Luke's book as a starting point, the details presented above were worked out during a late-night IRC session. My thanks to Andrew Bartlett, Richard Sharpe, and Vance Lankhaar for their patience, commitment, and sudden flashes of insight. Thanks also to Luke Howard for later clarifying some of the finer points.

The NTLMv2 Hash is derived directly from the NTLM (v1) Hash. Since there is no change to the initial input (the password), the keyspace is exactly the same. The only change is that the increased complexity of the algorithm means that there are more encryption hoops through which to jump compared to the simpler NTLM process. It takes more computer time to generate a v2 response, which doesn't impact a normal login but will slow down dictionary and brute force attacks against NTLMv2 (though Moore's Law may compensate). Weak passwords (those that are near the beginning of the password dictionary) are still vulnerable.

Another thing to consider is the blob. If the blob were zero length (empty), the NTLMv2 Response formula would reduce to:

```
v2resp = HMAC_MD5( v2hash, ServerChallenge );
```

which would still be pretty darn secure. So the question is this: Does the inclusion of the blob improve the NTLMv2 algorithm and, if so, how?

Well, see, it's like this... Instead of being produced by the key and challenge alone, the NTLMv2 Response involves the hash of a chunk of semi-random data. As a result, the same challenge will *not* always generate the same response. That's good, because it prevents replay attacks... in theory.

In practice, the randomness of the challenge should be enough to prevent replay attacks. Even if that were not the case, the only way that the blob could help would be if it, too, were non-repeating *and* if the server could somehow verify that the blob was not a repeat. That, quite possibly, is why the timestamp is included.

The timestamp could be used to let the server know that the blob is "fresh" — that is, that it was created a reasonably short amount of time before it was received. Fresh packets can't easily be forged because the response is HMAC-signed using the v2hash as the key (and that's based on the password which is the very thing the cracker doesn't know). Of course, the timestamp test won't work unless the client and server clocks are synchronized, which is not always the case.

In all likelihood the contents of the blob are never tested at all. There is code and commentary in the Samba-TNG source showing that they have done some testing, and that their results indicate that a completely random blob of bytes works just fine. If that's true, then the blob does little to improve the security of the algorithm except perhaps by adding a few more CPU cycles to the processing time.

Bottom line: NTLMv2 challenge/response provides only a minimal improvement over its predecessor.

This isn't the first time that we have put a lot of effort into figuring out some complex piece of the protocol only to discover that it's almost pointless, and it probably won't be the last time either.

Email

```
   From: Ronald Tschalär
     To: Chris Hertel
Subject: The point of client nonces

In section 15.5.5 you talk about the "client challenge" a bit, but
miss the point of it: the client nonce (as it should really more
correctly be called) is there to prevent precomputed dictionary
attacks by the server, and has nothing to do with replay attacks
against the server (which, as you correctly state, is what the
server-challenge is for).

If there's no client nonce, then a rogue server can pick a fixed
server-nonce (server-challenge), take dictionary, and precompute all
the responses. Then any time a client connects to it it sends the
fixed challenge, and upon receipt of the client's response it can do
a simple database lookup to find the password (assuming the password
was in the dictionary). However, if the client adds its own bit of
random stuff to the response computation, then this attack (by the
server) is not possible. Hence the client-nonce.

Even with client nonces a rogue server can still try to use a
dictionary to figure out your password, but the server has to run
the complete dictionary on each response, instead of being able to
precompute and use the results for all responses.
```

15.5.6 *Insult to Injury: LMv2*

There is yet one more small problem with the NTLMv2 Response, and that problem is known as *pass-through* authentication. Simply put, a server can *pass* the authentication process *through* to an NT Domain Controller. The trouble is that some servers that use pass-through assume that the response string is only 24 bytes long.

You may recall that both the LM and NTLM responses are, in fact, 24 bytes long. Because of the blob, however, the NTLMv2 response is much

longer. If a server truncates the response to 24 bytes before forwarding it to the NT Domain Controller almost all of the blob will be lost. Without the blob, the Domain Controller will have no way to verify the response so authentication will fail.

To compensate, a simpler response — known as the LMv2 response — is also calculated and returned alongside the NTLMv2 response. The formula is identical to that of NTLMv2, except that the blob is really small.

```
blip = RandomBytes( 8 );
data = concat( ServerChallenge, 8, blip, 8 );
hmac = HMAC_MD5( v2hash, 16, data, 16 );
LMv2resp = concat( hmac, 16, blip, 8 );
```

The "blip," as we've chosen to call it, is sometimes referred to as the "Client Challenge." If you go back and look, you'll find that the blip value is also included in the blob, just after the timestamp. It is fairly easy to spot in packet captures. The blip is 8 bytes long so that the resulting LMv2 Response will be 24 bytes, exactly the size needed for pass-through authentication.

If it is true that the contents of the blob are not checked, then the LMv2 Response isn't really any less secure than the NTLMv2 Response — even though the latter is bigger.

The LMv2 Response takes the place of the LM Response in the `SESSION_SETUP_ANDX.CaseInsensitivePassword` field.

15.5.7 *Choosing NTLMv2*

The use of NTLMv2 is *not* negotiated between the client and the server. There is nothing in the protocol to determine which challenge/response algorithms should be used.

So, um... how does the client know what to send, and how does the server know what to expect?

The default behavior for Windows clients is to send the LM and NTLM responses, and the default for Windows servers is to accept them. Changing these defaults requires fiddling in the Windows registry. Fortunately, the fiddles are well known and documented so we can go through them quickly and get them out of the way.[16]

16. A quick web search for "LMCompatibility" will turn up a lot of references, *Microsoft Knowledge Base Article #147706* among them.

The registry path to look at is:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\LSA
```

On Windows 9x the variable is called `LMCompatibility`, but on Windows NT and 2000 it is `LMCompatibilityLevel`. That variable may not be present in the registry, so you might have to add it. In general, it's best to follow Microsoft's instructions when editing the registry.[17]

The settings for `LMCompatibilityLevel` are as follows:

Level	Description	Client Implications	Server Implications
0	**The Default**	LM and NTLM responses are sent by the client.	The server or Domain Controller will compare the client's responses against the LM, NTLM, LMv2, and NTLMv2 responses. Any valid response is acceptable.
1	**NTLMv2 Session Security**	This level does nothing to change the algorithm used to generate the response. Instead, at this level and higher a feature called NTLMv2 Session Security is supported. Session Security is only used with Extended Security, and must be negotiated between the client and the server. Session Security is an advanced topic, and won't be covered here.	
2	**NTLM Authentication**	The LM Response is not sent by the client. Instead, the NTLM Response is sent in both password fields. Replacing the LM Response with the NTLM Response facilitates pass-through authentication. Servers need only hand the 24-byte contents of the `SESSION_SETUP_ANDX.CaseInsensitivePassword` field along to the Domain Controller.	The server or Domain Controller will accept a valid LM, NTLM, LMv2, or NTLMv2 response.

17. ...so that if something goes wrong you can blame them, and not me.

Level	Description	Client Implications	Server Implications
3	**NTLMv2 Authentication**	The client sends the LMv2 and NTLMv2 responses in place of the older LM and NTLM values.	The server or Domain Controller will accept a valid LM, NTLM, LMv2, or NTLMv2 response.
4	**NTLM Required**	The client sends the LMv2 and NTLMv2 responses.	At this level, the server or Domain Controller will not check LM Responses. It will compare responses using the NTLM, LMv2, and/or NTLMv2 algorithms.
5	**NTLMv2 Required**	The client sends the LMv2 and NTLMv2 responses.	The server or Domain Controller will compare the client's responses using the LMv2 and NTLMv2 algorithms only.

That's just a quick overview of the settings and their meanings. The important points are these:

- The password hash type is *not* negotiated on the wire, but determined by client and/or server configuration. If the client and server configurations are incompatible, authentication will fail.
- The SMB server or Domain Controller may try several comparisons in order to determine whether or not a given response is valid.

15.6 Extended Security: That Light at the End of the Tunnel

Our discussion of SMB authentication mechanisms is winding down now. There are a few more topics to be covered and a few others that will be carefully, but purposefully, avoided. Extended Security falls somewhere in between. We will dip our toes into its troubled waters, but we won't wade in too deep (or the monsters might get us).

One reason for trepidation is that — as of this writing — Extended Security is still an area of active research and development for the Samba Team and

others. Though much has been learned, and much has been implemented, the dark pools are still being explored and the fine points are still being examined. Another deterrent is that Extended Security represents a full set of sub-protocols — a whole, vast world of possibilities to be explored... some other day. As with MS-RPC (which we touched on just long enough to get our fingers burned), the topic is simply too large to cover here.

As suggested in Figure 15.4, Extended Security makes use of nested protocols. Go back to Section 13.3.2 on page 235 and take a look at the `NEGOTIATE_PROTOCOL_RESPONSE.SMB_DATA` structure. Note that the `ext_sec.SecurityBlob` field is nothing more than a block of bytes — and it's what's inside that block that matters. If the client and server agree to use Extended Security, then the whole `NEGOTIATE PROTOCOL RESPONSE / SESSION SETUP REQUEST` business becomes a transport for the authentication protocol.

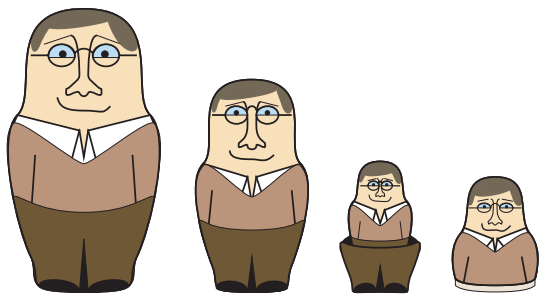


Figure 15.4: *Protocols nested like Russian dolls*

CIFS sub-protocols may be nested several layers deep. Extended Security, for example, is carried within SMB within NBT within TCP within IP.

In some cases the security exchange may require several packets and a few round trips to complete. When that happens, a single `NEGOTIATE PROTOCOL RESPONSE / SESSION SETUP REQUEST` pair will not be sufficient to handle it all. The solution to this dilemma is fairly simple: The server sends an error message to force the client to send another `SESSION SETUP REQUEST` containing the next chunk of data.

The process is briefly (and incompletely) described in Section 4.1.2 of the SNIA doc as part of the discussion of the `SESSION SETUP RESPONSE`. Simply put, as long as there are more Extended Security packets required, the server will reply to the `SESSION SETUP REQUEST` by sending a `NEGATIVE`

SESSION SETUP RESPONSE with an NT_STATUS value of 0xC0000016 (which is known as STATUS_MORE_PROCESSING_REQUIRED). The client then sends another SESSION SETUP REQUEST containing the additional data. This continues until the authentication protocol has completed.

There is no DOS error code equivalent for STATUS_MORE_PROCESSING_REQUIRED, something we have already whined about in the *Strange Behavior Alert* back in Section 12.1 on page 198. It seems that Extended Security expects that the client can handle NT_STATUS codes, which may be a significant issue for anyone trying to implement an SMB client.[18]

15.6.1 *The Extended Security Authentication Toolkit*

There are several different authentication protocols which may be carried within the SecurityBlob. Those protocols, in turn, are built on top of a whole pile of different languages and APIs and data transfer formats. The result is an alphabet soup of acronyms. Here's a taste:

ASN.1: Abstract Syntax Notation One

ASN.1 is a language used to define the structure and content of objects such as data records and protocol messages. If you are not familiar with ASN.1, you might think of it as a super-duper-hyper version of the typedef in C — only a lot more powerful. ASN.1 was developed as part of the Open Systems Interconnection (OSI) environment, and was originally used for writing specifications. More recently, though, tools have been developed that will generate software from ASN.1.

The development and promotion of the ASN.1 language is managed by the ASN.1 Consortium.

DER: Distinguished Encoding Rules of ASN.1

DER is a set of rules for encoding and decoding ASN.1 data. It provides a standard format for transport of data over a network so that the receiving end can convert the data back into its correct ASN.1 format. DER is a specialized form of a more general encoding known as BER (**B**asic

18. It might be worth doing some testing if you really want to use DOS codes in your implementation, but also want Extended Security. It may be possible to use the NT_STATUS codes for this exchange only, or you might try interpreting any unrecognized DOS error code as if it were STATUS_MORE_PROCESSING_REQUIRED.

Encoding **R**ules). DER is designed to work well with security protocols, and is used for encoding Kerberos and LDAP exchanges.

GSS-API: Generic Security Service Application Program Interface

As the name suggests, GSS-API is a generic interface to a set of security services. It makes it possible to write software that does not care what the underlying security mechanisms actually are. GSS-API is documented in RFC 2078.

Kerberos

("Kerberos" is a name, not an acronym.)

Kerberos is *the* preferred authentication system for SMB over naked TCP/IP transport. The use of Kerberos with CIFS is also tied in with GSS-API and SPNEGO.

LDAP: Lightweight Directory Access Protocol

Some folks at the University of Michigan realized that the DAP protocol (which was designed as part of the the Open Systems Interconnection (OSI) environment for use with the X.500 directory system) was just too big and hairy for general-purpose use, so they came up with a "lightweight" version, known as LDAP. LDAP was popularized in the mid-1990's, and support was included with directory service implementations such as Novell's NDS (**N**ovell **D**irectory **S**ervice). When Microsoft created their Active Directory system they followed Novell's lead and added LDAP support as well.

MIDL: Microsoft Interface Definition Language

MIDL is Microsoft's version of the **I**nterface **D**efinition **L**anguage (IDL). It is used to specify the parameters to function calls, particularly function calls made across a network — things like Remote Procedure Call (RPC). MIDL is also used to define the interfaces to Microsoft DLL library functions.

MS-RPC: Microsoft Remote Procedure Call

Paul Leach was one of the founders of Apollo Computer. At Apollo, he worked on a system for distributed computing that eventually became the DCE/RPC system. When Hewlett-Packard purchased Apollo, Leach went to Microsoft. That's probably why the MS-RPC system is so remarkably similar to the DCE/RPC system.

NDR: Network Data Representation

NDR is to DCE/RPC as DER (or BER) is to ASN.1. That is, NDR is an on-the-wire encoding for parameters passed via RPC. When an MS-RPC call is made on the client side, the parameters are converted into NDR format (*marshaled*) for transmission over the network. On the server side, the NDR formatted data is *unmarshaled* and passed into the called function. The process is then reversed to return the results.

NTLMSSP: NTLM Security Support Provider

It seems there are a few variations on the interpretation of the acronym. A quick web search for "NTLMSSP" turns up *NTLM Security Service Provider* and *NTLM Secure Service Provider* in addition to *NTLM Security Support Provider*. No matter. They all amount to the same thing.

NTLMSSP is a Windows authentication service that is accessed in much the same way as MS-RPC services. NTLMSSP authentication requests are formatted into a record structure and converted to NDR format for transport to the NTLMSSP authentication service provider. In addition to Extended Security, NTLMSSP authentication shows up in lots of odd places. It is even used by Microsoft Internet Explorer to authenticate HTTP (web) connections.

SPNEGO: Simple, Protected Negotiation

Also known as "The Simple and Protected GSS-API Negotiation Mechanism," SPNEGO is a protocol that underlies GSS-API. It is used to negotiate the security mechanism to be used between two systems. SPNEGO is documented in RFC 2478.

Quite a list, eh?

As you can see, there is a lot going on below the surface of Extended Security. We could try diving into a few of the above topics, but the waters are deep and the currents are strong and we would quickly be swept away. Out of necessity, we will spend a little time talking about Kerberos, but we won't swim out too far and we will be wearing a PFD (**P**ersonal **F**loatation **D**evice — don'cha just love acronyms?).

15.7 Kerberos

As already stated, we won't be going into depth about Kerberos. There is a lot of documentation available on the Internet and in print, so the wiser course is to suggest some starting points for research. There are, of course, several starting points presented in the References section of this very book. A good place to get your feet wet is Bruce Schneier's *Applied Cryptography, Second Edition*.

Kerberos version 5 is specified in RFC 1510, but this is CIFS we're talking about. Microsoft has made a few "enhancements" to the standard. The best known is probably the inclusion of a proprietary **P**rivilege **A**ccess **C**ertificate (PAC) which carries Windows-specific authorization information. Microsoft heard a lot of grumbling about the PAC, and in the end they did publish the information required by third-party implementors. They even did so under acceptable licensing terms (and the CIFS community sighed a collective sigh of relief). The PAC information is available in a **M**icro**s**oft **D**eveloper **N**etwork (MSDN) document entitled *Windows 2000 Authorization Data in Kerberos Tickets*.

There are a lot of Kerberos-related RFCs. The interesting ones for our purposes are:

- RFC 1964, which provides information about the use of Kerberos with GSS-API,
- RFC 3244, which covers Microsoft's Kerberos password-set and password-change protocols.

There is also (as of this writing) a set of Internet Drafts that cover Microsoft Kerberos features, including a draft for Kerberos authentication over HTTP.

Finally, a web search for "Microsoft" and "Kerberos" will toss up an abundant salad of opinions and references, both historical and contemporary. Where CIFS is concerned, it seems that there is always either too little or too much information. Microsoft-compatible Kerberos falls under the latter curse. There is a lot of stuff out there, and it is easy to get overwhelmed. If you plan to dive in, find a buddy. Don't swim alone.

15.8 Random Notes on W2K and NT Domain Authentication

We have been delicately dancing around the role of the Domain Controller in authentication. It's time to face the music.

The concept is fairly simple: Take the password database that is normally kept locally by a standalone server and move it to a central authority so that it can be shared by multiple servers, then call the whole thing a "Domain." The central authority that stores the shared database is, of course, the Domain Controller. As shown in Figure 15.5, the result is that the SMB fileserver must now consult the Domain Controller when a user tries to access SMB services.

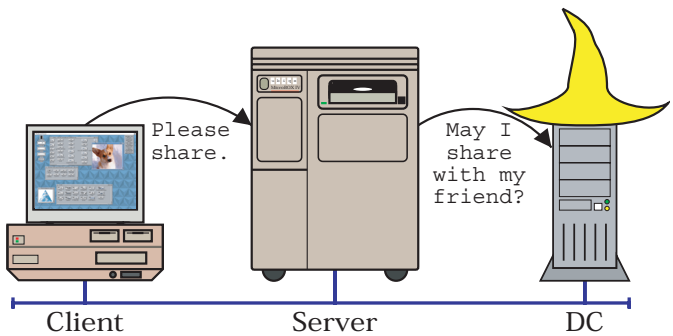


Figure 15.5: *Centralized authentication*

The **D**omain **C**ontroller (DC) wears a special hat. It keeps track of the common authentication database that is shared by the SMB servers in the Domain. The SMB servers query the DC when a client requests access to SMB services.

That general description applies to both NT and W2K Domains, even though the two are implemented in very different ways. Windows 2000 Domains are based on Active Directory and Kerberos, while Windows NT Domains make use of a **S**ecurity **A**ccounts **M**anager (SAM) Database and MS-RPC.

Let's see what bits of wisdom we can pull out of the hat regarding these two Domain systems...

15.8.1 *A Quick Look at W2K Domains*

As with Microsoft's Kerberos implementation, there is probably too much information available on this topic. A full description would also be very much beyond the stated scope of this book. So, as briefly as possible, here are some notes about W2K Domains and Domain Controllers:

- Windows 2000 Domains are based on the real thing: The Internet Domain Name Service (DNS). The DNS provides a hierarchical namespace, and W2K can take advantage of the DNS hierarchy to form collections of related W2K domains called "trees." Groups of separate W2K Domain trees are known as "forests."
- W2K Domain Controllers run the Active Directory service. **A**ctive **D**irectory (AD) is a database system that can be used to store all sorts of information, including user account data. The design of AD owes a lot to Novell's NDS architecture which, in turn, is based on OSI X.500.
- Data stored in the Active Directory may be accessed using the LDAP protocol.
- Microsoft's Kerberos implementation relies upon the data stored in the Active Directory. The two services are closely linked.

...and that barely begins to scratch the surface. CIFS client and server participation in a W2K Domain requires Kerberos support, but does not require a detailed understanding of Active Directory architecture. The above points are given here primarily for comparison with the NT Domain system notes, presented below.

15.8.2 *A Few Notes about NT Domains*

In contrast to W2K Domains, NT Domains have the following features:

- Windows NT Domains are built upon NetBIOS. The NetBIOS namespace is flat, not hierarchical, so there is no natural way to build relationships among NT Domains. Conceptually, NT Domains are standalone.
- NT authentication information is stored in the **S**ecurity **A**ccounts **M**anager (SAM) Database. The SAM is an extension of the Windows NT Registry database, and it is accessed using a Windows DLL.

- In an NT Domain, the shared SAM database is stored on the Domain Controller and may be accessed using RPC function calls. (Windows 2000, of course, stores the SAM data in the Active Directory, but it can also respond to the RPC calls for compatibility with the NT Domain system.)

There are two mechanisms that an SMB Server can use to ask a Domain Controller to validate a client logon attempt. These are known as *pass-through* and *NetLogon* authentication. The NetLogon mechanism uses MS-RPC, so we won't cover it here except to say that it provides a more intimate relationship between the SMB server and the Domain Controller than does the pass-through mechanism. There are several good sources for further reading listed in the References section. In particular:

- Start with the whitepaper *More Than You Ever Wanted to Know about NT Login Authentication*, by Philip Cox and Paul Hill. It provides a clear and succinct introduction to the Windows NT authentication system.
- Another good overview from a different perspective is provided in the whitepaper *CIFS Authentication and Security* by Bridget Allison (now Bridget Warwick).
- ...and once you're read that you'll be ready for the more in-depth NetLogon coverage in Luke's Leighton's book.

Pass-through, in contrast to NetLogon, is really quite simple. It is also documented in (yet) an(other) expired Leach/Naik IETF draft, titled *CIFS Domain Logon and Pass Through Authentication*, which can be found on Microsoft's CIFS FTP site (under the name `cifslog.txt`).

Basically, pass-through authentication is a man-in-the-middle mechanism. It goes like this:

- The client attempts to log on to the server, but the server has no SAM database so it, in turn, attempts to create an SMB session with the Domain Controller.
- The server sends a `NEGOTIATE PROTOCOL REQUEST` to the DC. The DC returns a challenge which the server passes back to the client.
- The client does the hard work and generates the various responses (LM, NTLM, etc.), which are sent to the server. The server simply passes them through to the DC in its own `SESSION SETUP REQUEST`.

- If the DC returns a `POSITIVE SESSION SETUP RESPONSE` to the server, then the server will return a `POSITIVE SESSION SETUP RESPONSE` to the client. Likewise with a negative response.

It should be easy to capture an example of pass-through authentication using your network sniffer. Windows 9x systems (and possibly other Windows varieties) do not support NetLogon so they always use the pass-through method if they are part of an NT Domain. Samba can be configured to use either method.

Radical Rodent Alert

There is an obscure Windows SMB file transfer mode implemented by Windows 98, Windows Me, and possibly other Windows flavors. This mode is known in the community as "rabbit-pellet" mode, and it is triggered by various subtle combinations of conditions. In testing, it appears as though delays in pass-through authentication may *be a factor.*

In rabbit-pellet mode the client will send a file to the server in very small chunks, somewhere between 512 and 1536 bytes each (give or take). The client will wait for a reply to each write, and will also send a flush request after every one or two writes. This slows down file transfers considerably.

The condition is rare, which is good because it's really annoying when it happens. It's also bad because it has been a very difficult problem to track down.

15.8.3 *It's Good to Have a Backup*

In the NT Domain system, there is a single Domain Controller that is *primarily* responsible for the maintenance of the domain's SAM database. This Domain Controller is known as the (surprise) **P**rimary **D**omain **C**ontroller (PDC).

The domain may also have zero or more **B**ackup **D**omain **C**ontrollers (BDCs). The BDCs keep read-only replicas of the PDC's SAM database. BDCs can be used for authentication just as the PDC can, and if the PDC is accidentally thrown out of a twelfth-story window into an active volcano, a BDC can be "promoted" to fill the role of the dearly departed PDC.

Windows 2000 Domains do things differently. They do not distinguish between Primary and Backup DCs. Instead, Active Directory makes use of something called "multimaster replication." Updates to any replica are propagated to all of the other replicas, so there is no need any longer to specify one copy of the database as the primary.

15.8.4 *Trust Me on This*

This is one of those concepts that we have to cover because — unless you're already familiar with it — you'll read about it somewhere else and think to yourself "What the heck is that all about?"

Somewhere back a few paragraphs it was stated that NT Domains are, conceptually, standalone entities... and so they are, but it is possible to introduce them to one another and get them to cooperate. The agreements forged between the domains are known as "Inter-Domain Trust Relationships."

Let's use an example to explain what this is all about.

Consider a large corporate organization with several divisions, departments, committees, consultants, and such-like. In this corporation, the Business Units Reassignment Planning Division runs the `BURP_DIV` domain, and the Displacement Entry Department calls theirs the `DISENTRY` domain.

Now, let's say that the `BURP_DIV` folks need access to files stored on `DISENTRY` servers (so they can move the files around a bit). One way to handle this would be to create accounts for the `BURP_DIV` users in the `DISENTRY` domain. That would cause a bit of a problem, however, because the `BURP_DIV` users would need two accounts, one per domain. That is likely to result in things like passwords, preferences, and web browser bookmarks getting a bit out of sync. Also, the Benefits Reduction Committee will want to know why all of the `BURP_DIV` employees are moonlighting in the `DISENTRY` department and how they could possibly be doing two jobs at once. It could become quite a mess, resulting in the hiring of dozens of consultants to ensure that the problem is properly ignored.

The better way to handle this situation is to create a *trust relationship* between the `DISENTRY` and `BURP_DIV` domains. With inter-domain trust established, the `BURP_DIV` folks can log on to `DISENTRY` servers using their `BURP_DIV` credentials. As shown in Figure 15.6, the `DISENTRY` Domain Controller will ask the `BURP_DIV` Domain Controllers to validate the logon.

Note that, in the non-extended-security version of the `SESSION SETUP REQUEST` message, there is a field called `PrimaryDomain`. This field identifies the NT domain against which the client wishes to authenticate. That is, the `PrimaryDomain` field should contain the name of the NT Domain to which the user belongs.

Windows 2000 domains also support trust relationships. This is useful for creating trust between two separate W2K Domain trees, or between W2K Domains and NT Domains.

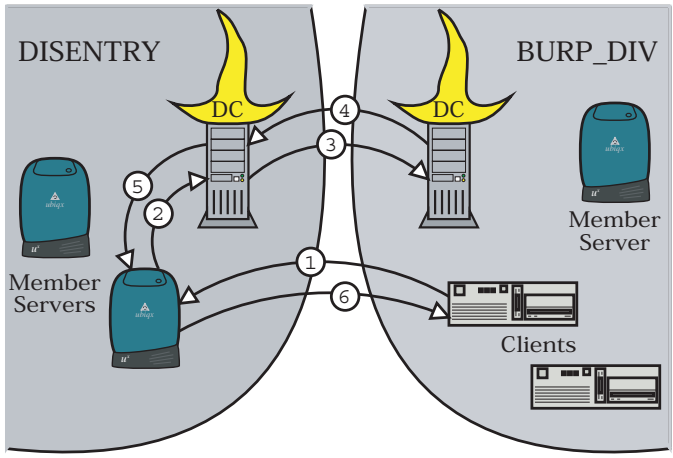


Figure 15.6: *Inter-domain trust*

1. A client in the `BURP_DIV` domain tries to access services on a `DISENTRY` server.
2. The server requests authentication services from a `DISENTRY` Domain Controller.
3. The `DISENTRY` Domain Controller trusts the `BURP_DIV` domain, so it requests authentication services from a `BURP_DIV` Domain Controller.
4. The `BURP_DIV` Domain Controller replies to the `DISENTRY` Domain Controller...
5. ...which replies to the server...
6. ...which replies to the client.

The mechanisms used to support inter-domain trust are very advanced topics, and won't be covered here.

15.9 Random Notes on Message Authentication Codes

Message **A**uthentication **C**odes (MACs) are used to prevent "pickle-in-the-middle" attacks (more commonly known as "man-in-the-middle" attacks).[19] This form of attack is simple to describe, but it can be difficult to pull off in practice (though wireless LAN technology has the potential to make it much easier). Figure 15.7 provides some visuals.

19. The latter name — though decidedly less Freudian — is somewhat gender-biased.

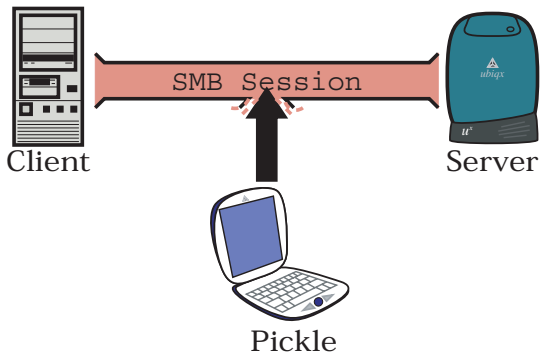


Figure 15.7a: *Attempting a man-in-the-middle attack*

The evil interloper attempts to interpose itself between the legitimate client and server.

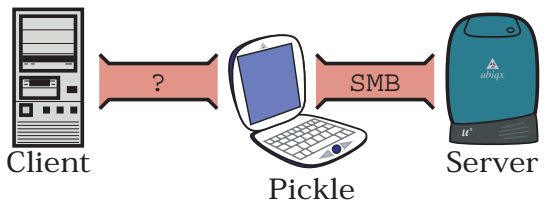


Figure 15.7b: *A successful man-in-the-middle attack*

If the evil interloper succeeds, the server will not notice that the "real" client is gone. The evil interloper may also try to impersonate the server to the client.

Generally speaking, in the pickle-in-the-middle attack an evil interloper allows the "real" client to authenticate with the server and then assumes ownership of the TCP/IP connection, thus bypassing the whole problem of needing to know the password.

There are a number of ways to hijack the TCP session, but with SMB that step isn't necessary. Instead, the evil interloper can simply impersonate the server to fool the client. For instance, if the evil interloper is on the same IP subnet as both the client and server (a B-mode network) then it can usurp the server's name by responding to broadcast name queries sent by the client faster than the server does. Server identity theft can also be accomplished by "poisoning" the NBNS database (or, possibly, the DNS) — that is, by somehow forcing it to swallow false information. A simple way to do that is to register the server's name — with the interloper's IP address — in the NBNS before

the server does (perhaps by registering the name while the server is down for maintenance or something).

In any case, when the client tries to open an SMB session with the server it may wind up talking to the evil interloper instead. The evil interloper will pass the authentication request through to the real server, then pass the challenge back to the client, and then pass the client's response to the server... and that... um... um... um... *that looks exactly like pass-through authentication*. In fact, the basic difference between pass-through authentication and this type of attack is the ownership of the box that is relaying the authentication. If the crackers control the box, consider it an attack.

This authentication stuff is fun, isn't it?

So, given a situation in which you are concerned about evil interlopers gaining access to your network, you need a mechanism that allows the client and server to prove to one another *on an ongoing basis* that they are the *real* client and server. That's what the MACs are supposed to do.

Caveat Emptor Alert

*As of this writing, SMB MAC signing is an active area of research for the Samba Team. The available documentation regarding **M**essage **A**uthentication **C**odes (MACs) disagrees to some extent with empirical results. The information presented in this section is the best currently available, derived from both the documentation* and *the testing being done by the Samba Team. As such, it is* probably *(but* not necessarily*) very close to reality. Doveryay, no proveryay.*

15.9.1 *Generating the Session Key*

The server and client each generate a special key, known as the *Session Key*. There are several potential uses for the Session Key, but we will only be looking at its use in MAC signing.

The Session Key is derived from the password hash — something that only the client and server should know. There are several hash types available: LM, NTLM, LMv2, and NTLMv2. The hash chosen is probably the most advanced hash that the two systems know they share. So, if the client sent an LM Response — but did *not* send an NTLM Response — then the Session Key will be based on the LM Hash. The LM Session Key is calculated as follows:

```
char eightnuls[8] = { 0, 0, 0, 0, 0, 0, 0, 0 };
LM_Session_Key = concat( LM_Hash, 8, eightnuls, 8 );
```

That is, take the first eight bytes of the LM Hash and add eight nul bytes to the end for a total of 16 bytes. Note that the resulting Session Key is not the same as the LM Hash itself. As stated earlier, the password hashes can be used to perform all of the authentication functions we have covered so far, so they must be protected as if they were the actual password. Overwriting the last eight bytes of the hash with zeros serves to obfuscate the hash (though this method is rather weak).

A different formula is used if the client *did* send an NTLM Response. The NTLM Session Key is calculated like so:

```
NTLM_Session_Key = MD4( NTLM_Hash );
```

which means that the NTLM Session Key is the MD4 of the MD4 of the Unicode password. The SNIA doc says there's only one MD4, but that would make the NTLM Session Key the same as the NTLM Hash. Andrew Bartlett of the Samba Team says there are two MD4s; the second does a fine job of protecting the password-equivalent NTLM Hash from exposure.

Moving along to LMv2 and NTLMv2, we find that the Session Key recipe is slightly more complex, but it's all stuff we have seen before. We need the following ingredients:

`v2hash`
: The NTLM Version 2 Hash, which we calculated back in Section 15.5.2 on page 279.

`hmac`
: The result of the `HMAC_MD5()` function using the `v2hash` as the key and the server challenge plus the blob (or blip) as the input data. The NTLMv2 `hmac` was calculated in Section 15.5.3 and sent as the first 16 bytes of the response. The LMv2 `hmac` was calculated in Section 15.5.6.

The LMv2 and NTLMv2 session keys are computed as follows:

```
LMv2_Session_Key   = HMAC_MD5( v2hash, 16,   lmv2_hmac, 16 );
NTLMv2_Session_Key = HMAC_MD5( v2hash, 16, ntlmv2_hmac, 16 );
```

The client is able to generate the Session Key because it knows the password and other required information (because the user entered the required information at the logon prompt). If the server is standalone, it will have the password hash and other required information in its local SAM database, and can generate the Session Key as well. On the other hand, if the server relies

upon a Domain Controller for authentication then it won't have the password hash and won't be able to generate the Session Key.

What's a server to do?

As we have already pointed out, the MAC protocol is designed to prevent a situation that looks exactly like pass-through authentication, so a pass-through server simply cannot do MAC signing. A NetLogon-capable server, however, has a special relationship with the Domain Controller. The NetLogon protocol is secured, so the Domain Controller can generate the Session Key and send it to the server. That's how an NT Domain member server gets hold of the Session Key without ever knowing the user's password or password hash.

15.9.2 *Sequence Numbers*

The client and server each maintain an integer counter which they initialize to zero. Both counters are incremented for every SMB message — that's once for a request and once for a reply. As a result, requests always have an even sequence number and replies always have an odd.

The zero-eth message is always a `SESSION SETUP ANDX` message, but it may not be the *first* `SESSION SETUP ANDX` of the session. Recall, from near the beginning of the Authentication section, that the client sometimes uses an anonymous or guest logon to access server information. Watch enough packet captures and you will see that MAC signing doesn't really start until after a real user logon occurs.

Also, it appears from testing that the MAC Signature in the zero-eth message is never checked (and that existing clients send a bogus MAC Signature in the zero-eth packet). That's okay, since the authenticity of the zero-eth message can be verified by the fact that it contains a valid response to the server challenge.

Once the MAC signing has been initialized within a session, all messages are numbered using the same counters and signed using the same Session Key. This is true even if additional `SESSION SETUP ANDX` exchanges occur.

15.9.3 *Calculating the MAC*

The MAC itself is calculated using the MD5 function. That's the plain MD5, not HMAC-MD5 and not MD4. The input to the MD5 function consists of three concatenated blocks of data:

- the Session Key,
- the response, and
- the SMB message.

We start by combining the Session Key and the response into a single value known as the MAC Key. For LM, NTLM, and LMv2 the MAC Key is created like so:

```
MAC_Key = concat( Session_Key, 16, Response, 24 );
```

The thing to note here is that all of the responses, with the exception of the NTLMv2 Response, are 24 bytes long. So, except for NTLMv2, all auth mechanisms produce a MAC Key that is 40 bytes long. (16 + 24 = 40). Unfortunately, the formula for creating the NTLMv2 MAC Key is not yet known. It is probably similar to the above, however. Possibly identical to the calculation of the LMv2 MAC Key, or possibly the concatenation of the Session Key with the first 28 bytes of the blob.

Okay, now you need to pay careful attention. The last few steps of MAC Signature calculation are a bit fiddly.

1. Start by re-acquainting yourself with the structure of the `SMB_HEADER.EXTRA` field, as described in Section 11.2.1 on page 181. We are particularly interested in the eight bytes labeled `Signature`.
2. The sequence number is written as a longword into the first four bytes of the `SMB_HEADER.EXTRA.Signature` field. The remaining four bytes are zeroed.
3. The MAC Signature is calculated as follows:

   ```
   data = concat( MAC_Key, MAC_Key_Len, SMB_Msg, SMB_Msg_Len );
   hash = MD5( data );
   MAC  = head( hash, 8 );
   ```

   In words: the MAC Signature is the first eight bytes of the MD5 of the MAC_Key plus the entire SMB message.
4. The eight bytes worth of MAC Signature are copied into the `SMB_HEADER.EXTRA.Signature` field, overwriting the sequence number.

...and that, to the best of our knowledge, is how it's done.

15.9.4 *Enabling and Requiring MAC Signing*

Windows NT systems offer four registry keys to control the use of SMB MAC signing. The first two manage server behavior, and the others represent client settings.

Server: `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\LanManServer\Parameters`

`EnableSecuritySignature`

The valid values are zero (`0`) and one (`1`). If zero, then MAC signing is disabled on the server side. If one, then the server will support MAC signing.

`RequireSecuritySignature`

The valid values are zero (`0`) and one (`1`). This parameter is ignored unless MAC signing is enabled via the `EnableSecurity-Signature` parameter. If zero, then MAC signing is optional and will only be used if the client also supports it. If one, then MAC signing is required. If the client does not support MAC signing then authentication will fail.

Client: `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Rdr\Parameters`

`EnableSecuritySignature`

The valid values are zero (`0`) and one (`1`). If zero, then MAC signing is disabled on the client side. If one, then the client will support MAC signing.

`RequireSecuritySignature`

The valid values are zero (`0`) and one (`1`). This parameter is ignored unless MAC signing is enabled via the `EnableSecurity Signature` parameter. If zero, then MAC signing is optional and will only be used if the server also supports it. If one, then MAC signing is required. If the server does not support MAC signing then authentication will fail.

Study those closely and you may detect some small amount of similarity between the client and server parameter settings. (Well, okay, they are mirror

images of one another.) Keep in mind that the client and server must have compatible settings or the `SESSION SETUP` will fail.

These options are also available under Windows 2000, but are managed using security policy settings.[20]

15.10 Non Sequitur Time

A mathematician, a physicist, and an engineer were sitting together in a teashop, sharing a pot of Lapsang Souchong and discussing the relationship between theory and practice. The mathematician said, "One of my students asked me today whether all odd numbers greater than one were prime numbers, so I provided this simple proof:

> Stated: All odd numbers greater than one are prime.
> 3 is prime,
> 5 is prime,
> 7 is prime,
> 9 is divisible by three, so it is odd but not prime.
> Contradiction; the statement is false."

"Interesting," replied the physicist. "Perhaps I have the same student. I was asked the same question today. I solved the problem using a thought-experiment, as Galileo might have done. Our experiment was as follows:

> By observation we can see that:
> 3 is prime,
> 5 is prime,
> 7 is prime,
> 9 is experimental error,
> 11 is prime,
> 13 is prime..."

---

20. Jean-Baptiste Marchand has done some digging and reports that starting with Windows 2000 the SMB redirector (rdr) has been redesigned, which may impact which registry keys are fiddled. The preferred way to configure SMB MAC signing in Windows 2000 is to use the Local Security Settings/Group Policy Management Console (whatever that is). Basically, this means that Windows 2000 and Windows XP have MAC signing settings comparable to those in Windows NT, but they are handled in a different way.

The engineer interrupted before the physicist could draw a conclusion, and said, "Out in the field we don't have time to mess with theory. We just define all odd numbers as prime and work from there. It's simpler that way."

Consider this as you contemplate what you have learned about SMB authentication.

15.11 Further Study

You should now have all you need to create an SMB session with an SMB server. As you become more comfortable with the system, you will likely become curious about the vast uncharted jungle of Extended Security. Don't be afraid to go exploring. With the background provided here, and the guidebooks listed in the References section, you are well prepared. If you get it all mapped out, do us all a favor: write it up so that everyone can share what you've learned.

A few more bits of advice before we move along...

1. **Know what you've got to work with.** This is one of Andrew Bartlett's rules of thumb. If you are trying to figure out how an encrypted token or key or somesuch is derived, consider the available functions and inputs. Existing tools and values are often reused. Just look through the calculation of the NTLMv2 Response and you'll see what we mean.
2. **Trust but verify.** Read the available documentation and make notes, but don't assume that the documentation is always right. The truth is on the wire. In some cases implementations stray from the specifications, and in other cases (e.g. this book) the documentation is a best-effort attempt at presenting what has been learned. There are few truly definitive sources. Another factor, as you are by now aware, is that there is a tremendous amount of variation in the CIFS world. Something may work correctly in one instance only to surprise you in another.
3. **Don't be surprised.** Don't go looking for weirdness in CIFS, but don't be surprised when you find it. If you expect bad behavior, you may miss the sane and obvious. A lot of CIFS does, in fact, make some sort of sense when you think about it. There are gotchas, though, so be prepared.

These guidelines are quite general, but they apply particularly well to the study of SMB security and authentication.