SAMBA Developers Guide

Table of Contents

Table of Contents

Table of Contents

SAMBA Developers Guide

SAMBA Team

Abstract

Last Update : Mon Sep 30 15:23:53 CDT 2002

This book is a collection of documents that might be useful for people developing samba or those interested in doing so. It's nothing more than a collection of documents written by samba developers about the internals of various parts of samba and the SMB protocol. It's still incomplete. The most recent version of this document can be found at http://devel.samba.org/. Please send updates to jelmer@samba.org.

Table of Contents

Definition of NetBIOS Protocol and Name Resolution Modes

NETBIOS

NetBIOS runs over the following tranports: TCP/IP; NetBEUI and IPX/SPX. Samba only uses NetBIOS over TCP/IP. For details on the TCP/IP NetBIOS Session Service NetBIOS Datagram Service, and NetBIOS Names, see rfc1001.txt and rfc1002.txt.

NetBEUI is a raw NetBIOS frame protocol implementation that allows NetBIOS datagrams to be sent out over the 'wire' embedded within LLC frames. NetBEUI is not required when using NetBIOS over TCP/IP protocols and it is preferable NOT to install NetBEUI if it can be avoided.

IPX/SPX is also not required when using NetBIOS over TCP/IP, and it is preferable NOT to install the IPX/SPX transport unless you are using Novell servers. At the very least, it is recommended that you do not install 'NetBIOS over IPX/SPX'.

[When installing Windows 95, you will find that NetBEUI and IPX/SPX are installed as the default protocols. This is because they are the simplest to manage: no Windows 95 user−configuration is required].

NetBIOS applications (such as samba) offer their services (for example, SMB file and print sharing) on a NetBIOS name. They must claim this name on the network before doing so. The NetBIOS session service will then accept connections on the application's behalf (on the NetBIOS name claimed by the application). A NetBIOS session between the application and the client can then commence.

NetBIOS names consist of 15 characters plus a 'type' character. This is similar, in concept, to an IP address and a TCP port number, respectively. A NetBIOS−aware application on a host will offer different services under different NetBIOS name types, just as a host will offer different TCP/IP services on different port numbers.

NetBIOS names must be claimed on a network, and must be defended. The use of NetBIOS names is most suitable on a single subnet; a Local Area Network or a Wide Area Network.

NetBIOS names are either UNIQUE or GROUP. Only one application can claim a UNIQUE NetBIOS name on a network.

There are two kinds of NetBIOS Name resolution: Broadcast and Point−to−Point.

BROADCAST NetBIOS

Clients can claim names, and therefore offer services on successfully claimed names, on their broadcast−isolated subnet. One way to get NetBIOS services (such as browsing: see ftp.microsoft.com/drg/developr/CIFS/browdiff.txt; and SMB file/print sharing: see cifs4.txt) working on a LAN or WAN is to make your routers forward all broadcast packets from TCP/IP ports 137, 138 and 139.

This, however, is not recommended. If you have a large LAN or WAN, you will find that some of your hosts spend 95 percent of their time dealing with broadcast traffic. [If you have IPX/SPX on your LAN or WAN, you will find that this is already happening: a packet analyzer will show, roughly every twelve minutes, great swathes of broadcast traffic!].

NBNS NetBIOS

rfc1001.txt describes, amongst other things, the implementation and use of, a 'NetBIOS Name Service'. NT/AS offers 'Windows Internet Name Service' which is fully rfc1001/2 compliant, but has had to take specific action with certain NetBIOS names in order to make it useful. (for example, it deals with the registration of <1c> <1d> <1e> names all in different ways. I recommend the reading of the Microsoft WINS Server Help files for full details).

The use of a WINS server cuts down on broadcast network traffic for NetBIOS name resolution. It has the effect of pulling all the broadcast isolated subnets together into a single NetBIOS scope, across your LAN or WAN, while avoiding the use of TCP/IP broadcast packets.

When you have a WINS server on your LAN, WINS clients will be able to contact the WINS server to resolve NetBIOS names. Note that only those WINS clients that have registered with the same WINS server will be visible. The WINS server _can_ have static NetBIOS entries added to its database (usually for security reasons you might want to consider putting your domain controllers or other important servers as static entries, but you should not rely on this as your sole means of security), but for the most part, NetBIOS names are registered dynamically.

This provides some confusion for lots of people, and is worth mentioning here: a Browse Server is NOT a WINS Server, even if these services are implemented in the same application. A Browse Server _needs_ a WINS server because a Browse Server is a WINS client, which is _not_ the same thing].

Clients can claim names, and therefore offer services on successfully claimed names, on their broadcast−isolated subnet. One way to get NetBIOS services (such as browsing: see ftp.microsoft.com/drg/developr/CIFS/browdiff.txt; and SMB file/print sharing: see cifs6.txt) working on a LAN or WAN is to make your routers forward all broadcast packets from TCP/IP ports 137, 138 and 139. You will find, however, if you do this on a large LAN or a WAN, that your network is completely swamped by NetBIOS and browsing packets, which is why WINS was developed to minimise the necessity of broadcast traffic.

WINS Clients therefore claim names from the WINS server. If the WINS server allows them to register a name, the client's NetBIOS session service can then offer services on this name. Other WINS clients will then contact the WINS server to resolve a NetBIOS name.

Samba Architecture

Introduction

This document gives a general overview of how Samba works internally. The Samba Team has tried to come up with a model which is the best possible compromise between elegance, portability, security and the constraints imposed by the very messy SMB and CIFS protocol.

It also tries to answer some of the frequently asked questions such as:

1. Is Samba secure when running on Unix? The xyz platform? What about the root priveliges issue?
2. Pros and cons of multithreading in various parts of Samba
3. Why not have a separate process for name resolution, WINS, and browsing?

Multithreading and Samba

People sometimes tout threads as a uniformly good thing. They are very nice in their place but are quite inappropriate for smbd. nmbd is another matter, and multi−threading it would be very nice.

The short version is that smbd is not multithreaded, and alternative servers that take this approach under Unix (such as Syntax, at the time of writing) suffer tremendous performance penalties and are less robust. nmbd is not threaded either, but this is because it is not possible to do it while keeping code consistent and portable across 35 or more platforms. (This drawback also applies to threading smbd.)

The longer versions is that there are very good reasons for not making smbd multi−threaded. Multi−threading would actually make Samba much slower, less scalable, less portable and much less robust. The fact that we use a separate process for each connection is one of Samba's biggest advantages.

Threading smbd

A few problems that would arise from a threaded smbd are:

1. It's not only to create threads instead of processes, but you must care about all variables if they have to be thread specific (currently they would be global).
2. if one thread dies (eg. a seg fault) then all threads die. We can immediately throw robustness out the window.
3. many of the system calls we make are blocking. Non−blocking equivalents of many calls are either not available or are awkward (and slow) to use. So while we block in one thread all clients are waiting. Imagine if one share is a slow NFS filesystem and the others are fast, we will end up slowing all clients to the speed of NFS.
4. you can't run as a different uid in different threads. This means we would have to switch uid/gid on _every_ SMB packet. It would be horrendously slow.
5. the per process file descriptor limit would mean that we could only support a limited number of clients.
6. we couldn't use the system locking calls as the locking context of fcntl() is a process, not a thread.

Threading nmbd

This would be ideal, but gets sunk by portability requirements.

Andrew tried to write a test threads library for nmbd that used only ansi−C constructs (using setjmp and longjmp). Unfortunately some OSes defeat this by restricting longjmp to calling addresses that are shallower than the current address on the stack (apparently AIX does this). This makes a truly portable threads library impossible. So to support all our current platforms we would have to code nmbd both with and without threads, and as the real aim of threads is to make the code clearer we would not have gained anything. (it is a myth that threads make things faster. threading is like recursion, it can make things clear but the same thing can always be done faster by some other method)

Chris tried to spec out a general design that would abstract threading vs separate processes (vs other methods?) and make them accessible through some general API. This doesn't work because of the data sharing requirements of the protocol (packets in the future depending on packets now, etc.) At least, the code would work but would be very clumsy, and besides the fork() type model would never work on Unix. (Is there an OS that it would work on, for nmbd?)

A fork() is cheap, but not nearly cheap enough to do on every UDP packet that arrives. Having a pool of processes is possible but is nasty to program cleanly due to the enormous amount of shared data (in complex structures) between the processes. We can't rely on each platform having a shared memory system.

nbmd Design

Originally Andrew used recursion to simulate a multi−threaded environment, which use the stack enormously and made for really confusing debugging sessions. Luke Leighton rewrote it to use a queuing system that keeps state information on each packet. The first version used a single structure which was used by all the pending states. As the initialisation of this structure was done by adding arguments, as the functionality developed, it got pretty messy. So, it was replaced with a higher−order function and a pointer to a user−defined memory block. This suddenly made things much simpler: large numbers of functions could be made static, and modularised. This is the same principle as used in NT's kernel, and achieves the same effect as threads, but in a single process.

Then Jeremy rewrote nmbd. The packet data in nmbd isn't what's on the wire. It's a nice format that is very amenable to processing but still keeps the idea of a distinct packet. See "struct packet_struct" in nameserv.h. It has all the detail but none of the on−the−wire mess. This makes it ideal for using in disk or memory−based databases for browsing and WINS support.

The samba DEBUG system

New Output Syntax

The syntax of a debugging log file is represented as:

```
>debugfile< :== { >debugmsg< }

>debugmsg<  :== >debughdr< '\n' >debugtext<

>debughdr<  :== '[' TIME ',' LEVEL ']' FILE ':' [FUNCTION] '(' LINE ')'

>debugtext< :== { >debugline< }

>debugline< :== TEXT '\n'
```

TEXT is a string of characters excluding the newline character.

LEVEL is the DEBUG level of the message (an integer in the range 0..10).

TIME is a timestamp.

FILE is the name of the file from which the debug message was generated.

FUNCTION is the function from which the debug message was generated.

LINE is the line number of the debug statement that generated the message.

Basically, what that all means is:

1. A debugging log file is made up of debug messages.
2. Each debug message is made up of a header and text. The header is separated from the text by a newline.
3. The header begins with the timestamp and debug level of the message enclosed in brackets. The filename, function, and line number at which the message was generated follow. The filename is terminated by a colon, and the function name is terminated by the parenthesis which contain the line number. Depending upon the compiler, the function name may be missing (it is generated by the __FUNCTION__ macro, which is not universally implemented, dangit).
4. The message text is made up of zero or more lines, each terminated by a newline.

Here's some example output:

```
    [1998/08/03 12:55:25, 1] nmbd.c:(659)
      Netbios nameserver version 1.9.19-prealpha started.
      Copyright Andrew Tridgell 1994-1997
    [1998/08/03 12:55:25, 3] loadparm.c:(763)
      Initializing global parameters
```

Note that in the above example the function names are not listed on the header line. That's because the example above was generated on an SGI Indy, and the SGI compiler doesn't support the __FUNCTION__ macro.

The DEBUG() Macro

Use of the DEBUG() macro is unchanged. DEBUG() takes two parameters. The first is the message level, the second is the body of a function call to the Debug1() function.

That's confusing.

Here's an example which may help a bit. If you would write

```
printf( "This is a %s message.\n", "debug" );
```

to send the output to stdout, then you would write

```
DEBUG( 0, ( "This is a %s message.\n", "debug" ) );
```

to send the output to the debug file. All of the normal printf() formatting escapes work.

Note that in the above example the DEBUG message level is set to 0. Messages at level 0 always print. Basically, if the message level is less than or equal to the global value DEBUGLEVEL, then the DEBUG statement is processed.

The output of the above example would be something like:

```
    [1998/07/30 16:00:51, 0] file.c:function(128)
      This is a debug message.
```

Each call to DEBUG() creates a new header *unless* the output produced by the previous call to DEBUG() did not end with a '\n'. Output to the debug file is passed through a formatting buffer which is flushed every time a newline is encountered. If the buffer is not empty when DEBUG() is called, the new input is simply appended.

...but that's really just a Kludge. It was put in place because DEBUG() has been used to write partial lines. Here's a simple (dumb) example of the kind of thing I'm talking about:

```
    DEBUG( 0, ("The test returned " ) );
    if( test() )
      DEBUG(0, ("True") );
    else
      DEBUG(0, ("False") );
    DEBUG(0, (".\n") );
```

Without the format buffer, the output (assuming test() returned true) would look like this:

```
    [1998/07/30 16:00:51, 0] file.c:function(256)
      The test returned
    [1998/07/30 16:00:51, 0] file.c:function(258)
      True
    [1998/07/30 16:00:51, 0] file.c:function(261)
      .
```

Which isn't much use. The format buffer kludge fixes this problem.

The DEBUGADD() Macro

In addition to the kludgey solution to the broken line problem described above, there is a clean solution. The DEBUGADD() macro never generates a header. It will append new text to the current debug message even if the format buffer is empty. The syntax of the DEBUGADD() macro is the same as that of the DEBUG() macro.

```
    DEBUG( 0, ("This is the first line.\n" ) );
    DEBUGADD( 0, ("This is the second line.\nThis is the third line.\n" ) );
```

Produces

```
    [1998/07/30 16:00:51, 0] file.c:function(512)
      This is the first line.
      This is the second line.
      This is the third line.
```

The DEBUGLVL() Macro

One of the problems with the DEBUG() macro was that DEBUG() lines tended to get a bit long. Consider this example from nmbd_sendannounce.c:

```
  DEBUG(3,("send_local_master_announcement: type %x for name %s on subnet %s for workgroup %s\n",
            type, global_myname, subrec->subnet_name, work->work_group));
```

One solution to this is to break it down using DEBUG() and DEBUGADD(), as follows:

```
  DEBUG( 3, ( "send_local_master_announcement: " ) );
  DEBUGADD( 3, ( "type %x for name %s ", type, global_myname ) );
  DEBUGADD( 3, ( "on subnet %s ", subrec->subnet_name ) );
  DEBUGADD( 3, ( "for workgroup %s\n", work->work_group ) );
```

A similar, but arguably nicer approach is to use the DEBUGLVL() macro. This macro returns True if the message level is less than or equal to the global DEBUGLEVEL value, so:

```
  if( DEBUGLVL( 3 ) )
    {
    dbgtext( "send_local_master_announcement: " );
    dbgtext( "type %x for name %s ", type, global_myname );
    dbgtext( "on subnet %s ", subrec->subnet_name );
    dbgtext( "for workgroup %s\n", work->work_group );
    }
```

(The dbgtext() function is explained below.)

There are a few advantages to this scheme:

1. The test is performed only once.
2. You can allocate variables off of the stack that will only be used within the DEBUGLVL() block.
3. Processing that is only relevant to debug output can be contained within the DEBUGLVL() block.

New Functions

dbgtext()

This function prints debug message text to the debug file (and possibly to syslog) via the format buffer. The function uses a variable argument list just like printf() or Debug1(). The input is printed into a buffer using the vslprintf() function, and then passed to format_debug_text(). If you use DEBUGLVL() you will probably print the body of the message using dbgtext().

---

dbghdr()

This is the function that writes a debug message header. Headers are not processed via the format buffer. Also note that if the format buffer is not empty, a call to dbghdr() will not produce any output. See the comments in dbghdr() for more info.

It is not likely that this function will be called directly. It is used by DEBUG() and DEBUGADD().

---

format_debug_text()

This is a static function in debug.c. It stores the output text for the body of the message in a buffer until it encounters a newline. When the newline character is found, the buffer is written to the debug file via the Debug1() function, and the buffer is reset. This allows us to add the indentation at the beginning of each line of the message body, and also ensures that the output is written a line at a time (which cleans up syslog output).

Coding Suggestions

So you want to add code to Samba ...

One of the daunting tasks facing a programmer attempting to write code for Samba is understanding the various coding conventions used by those most active in the project. These conventions were mostly unwritten and helped improve either the portability, stability or consistency of the code. This document will attempt to document a few of the more important coding practices used at this time on the Samba project. The coding practices are expected to change slightly over time, and even to grow as more is learned about obscure portability considerations. Two existing documents `samba/source/internals.doc` and `samba/source/architecture.doc` provide additional information.

The loosely related question of coding style is very personal and this document does not attempt to address that subject, except to say that I have observed that eight character tabs seem to be preferred in Samba source. If you are interested in the topic of coding style, two oft−quoted documents are:

http://lxr.linux.no/source/Documentation/CodingStyle

http://www.fsf.org/prep/standards_toc.html

But note that coding style in Samba varies due to the many different programmers who have contributed.

Following are some considerations you should use when adding new code to Samba. First and foremost remember that:

Portability is a primary consideration in adding function, as is network compatability with de facto, existing, real world CIFS/SMB implementations. There are lots of platforms that Samba builds on so use caution when adding a call to a library function that is not invoked in existing Samba code. Also note that there are many quite different SMB/CIFS clients that Samba tries to support, not all of which follow the SNIA CIFS Technical Reference (or the earlier Microsoft reference documents or the X/Open book on the SMB Standard) perfectly.

Here are some other suggestions:

1. use d_printf instead of printf for display text reason: enable auto−substitution of translated language text
2. use SAFE_FREE instead of free reason: reduce traps due to null pointers
3. don't use bzero use memset, or ZERO_STRUCT and ZERO_STRUCTP macros reason: not POSIX
4. don't use strcpy and strlen (use safe_* equivalents) reason: to avoid traps due to buffer overruns
5. don't use getopt_long, use popt functions instead reason: portability
6. explicitly add const qualifiers on parm passing in functions where parm is input only (somewhat controversial but const can be #defined away)
7. when passing a va_list as an arg, or assigning one to another please use the VA_COPY() macro reason: on some platforms, va_list is a struct that must be initialized in each function...can SEGV if you don't.
8. discourage use of threads reason: portability (also see architecture.doc)
9. don't explicitly include new header files in C files − new h files should be included by adding them once to includes.h reason: consistency
10. don't explicitly extern functions (they are autogenerated by "make proto" into proto.h) reason: consistency

11. use endian safe macros when unpacking SMBs (see byteorder.h and internals.doc) reason: not everyone uses Intel
12. Note Unicode implications of charset handling (see internals.doc). See pull_* and push_* and convert_string functions. reason: Internationalization
13. Don't assume English only reason: See above
14. Try to avoid using in/out parameters (functions that return data which overwrites input parameters) reason: Can cause stability problems
15. Ensure copyright notices are correct, don't append Tridge's name to code that he didn't write. If you did not write the code, make sure that it can coexist with the rest of the Samba GPLed code.
16. Consider usage of DATA_BLOBs for length specified byte−data. reason: stability
17. Take advantage of tdbs for database like function reason: consistency
18. Don't access the SAM_ACCOUNT structure directly, they should be accessed via pdb_get...() and pdb_set...() functions. reason: stability, consistency
19. Don't check a password directly against the passdb, always use the check_password() interface. reason: long term pluggability
20. Try to use asprintf rather than pstrings and fstrings where possible
21. Use normal C comments / * instead of C++ comments // like this. Although the C++ comment format is part of the C99 standard, some older vendor C compilers do not accept it.
22. Try to write documentation for API functions and structures explaining the point of the code, the way it should be used, and any special conditions or results. Mark these with a double−star comment start / ** so that they can be picked up by Doxygen, as in this file.
23. Keep the scope narrow. This means making functions/variables static whenever possible. We don't want our namespace polluted. Each module should have a minimal number of externally visible functions or variables.
24. Use function pointers to keep knowledge about particular pieces of code isolated in one place. We don't want a particular piece of functionality to be spread out across lots of places − that makes for fragile, hand to maintain code. Instead, design an interface and use tables containing function pointers to implement specific functionality. This is particularly important for command interpreters.
25. Think carefully about what it will be like for someone else to add to and maintain your code. If it would be hard for someone else to maintain then do it another way.

The suggestions above are simply that, suggestions, but the information may help in reducing the routine rework done on new code. The preceeding list is expected to change routinely as new support routines and macros are added.

Samba Internals

Character Handling

This section describes character set handling in Samba, as implemented in Samba 3.0 and above

In the past Samba had very ad−hoc character set handling. Scattered throughout the code were numerous calls which converted particular strings to/from DOS codepages. The problem is that there was no way of telling if a particular char* is in dos codepage or unix codepage. This led to a nightmare of code that tried to cope with particular cases without handlingt the general case.

The new functions

The new system works like this:

1. all char* strings inside Samba are "unix" strings. These are multi−byte strings that are in the charset defined by the "unix charset" option in smb.conf.
2. there is no single fixed character set for unix strings, but any character set that is used does need the following properties:

   a. must not contain NULLs except for termination
   b. must be 7−bit compatible with C strings, so that a constant string or character in C will be byte−for−byte identical to the equivalent string in the chosen character set.
   c. when you uppercase or lowercase a string it does not become longer than the original string
   d. must be able to correctly hold all characters that your client will throw at it

   For example, UTF−8 is fine, and most multi−byte asian character sets are fine, but UCS2 could not be used for unix strings as they contain nulls.
3. when you need to put a string into a buffer that will be sent on the wire, or you need a string in a character set format that is compatible with the clients character set then you need to use a pull_ or push_ function. The pull_ functions pull a string from a wire buffer into a (multi−byte) unix string. The push_ functions push a string out to a wire buffer.
4. the two main pull_ and push_ functions you need to understand are pull_string and push_string. These functions take a base pointer that should point at the start of the SMB packet that the string is in. The functions will check the flags field in this packet to automatically determine if the packet is marked as a unicode packet, and they will choose whether to use unicode for this string based on that flag. You may also force this decision using the STR_UNICODE or STR_ASCII flags. For use in smbd/ and libsmb/ there are wrapper functions clistr_ and srvstr_ that call the pull_/push_ functions with the appropriate first argument.

   You may also call the pull_ascii/pull_ucs2 or push_ascii/push_ucs2 functions if you know that a particular string is ascii or unicode. There are also a number of other convenience functions in charcnv.c that call the pull_/push_ functions with particularly common arguments, such as pull_ascii_pstring()
5. The biggest thing to remember is that internal (unix) strings in Samba may now contain multi−byte characters. This means you cannot assume that characters are always 1 byte long. Often this means that you will have to convert strings to ucs2 and back again in order to do some (seemingly) simple task. For examples of how to do this see functions like strchr_m(). I know this is very slow, and we will eventually speed it up but right now we want this stuff correct not fast.
6. all lp_ functions now return unix strings. The magic "DOS" flag on parameters is gone.

7. all vfs functions take unix strings. Don't convert when passing to them

---

Macros in byteorder.h

This section describes the macros defined in byteorder.h. These macros are used extensively in the Samba code.

CVAL(buf,pos)

returns the byte at offset pos within buffer buf as an unsigned character.

PVAL(buf,pos)

returns the value of CVAL(buf,pos) cast to type unsigned integer.

SCVAL(buf,pos,val)

sets the byte at offset pos within buffer buf to value val.

SVAL(buf,pos)

returns the value of the unsigned short (16 bit) little−endian integer at offset pos within buffer buf. An integer of this type is sometimes refered to as "USHORT".

IVAL(buf,pos)

returns the value of the unsigned 32 bit little−endian integer at offset pos within buffer buf.

SVALS(buf,pos)

returns the value of the signed short (16 bit) little−endian integer at offset pos within buffer buf.

IVALS(buf,pos)

returns the value of the signed 32 bit little−endian integer at offset pos within buffer buf.

SSVAL(buf,pos,val)

sets the unsigned short (16 bit) little−endian integer at offset pos within buffer buf to value val.

SIVAL(buf,pos,val)

sets the unsigned 32 bit little−endian integer at offset pos within buffer buf to the value val.

SSVALS(buf,pos,val)

sets the short (16 bit) signed little−endian integer at offset pos within buffer buf to the value val.

SIVALS(buf,pos,val)

sets the signed 32 bit little−endian integer at offset pos withing buffer buf to the value val.

RSVAL(buf,pos)

returns the value of the unsigned short (16 bit) big−endian integer at offset pos within buffer buf.

RIVAL(buf,pos)

returns the value of the unsigned 32 bit big−endian integer at offset pos within buffer buf.

RSSVAL(buf,pos,val)

sets the value of the unsigned short (16 bit) big−endian integer at offset pos within buffer buf to value val. refered to as "USHORT".

RSIVAL(buf,pos,val)

sets the value of the unsigned 32 bit big−endian integer at offset pos within buffer buf to value val.

LAN Manager Samba API

This section describes the functions need to make a LAN Manager RPC call. This information had been obtained by examining the Samba code and the LAN Manager 2.0 API documentation. It should not be considered entirely reliable.

```
call_api(int prcnt, int drcnt, int mprcnt, int mdrcnt,
        char *param, char *data, char **rparam, char **rdata);
```

This function is defined in client.c. It uses an SMB transaction to call a remote api.

Parameters

The parameters are as follows:

1. prcnt: the number of bytes of parameters begin sent.
2. drcnt: the number of bytes of data begin sent.
3. mprcnt: the maximum number of bytes of parameters which should be returned
4. mdrcnt: the maximum number of bytes of data which should be returned
5. param: a pointer to the parameters to be sent.
6. data: a pointer to the data to be sent.
7. rparam: a pointer to a pointer which will be set to point to the returned paramters. The caller of

call_api() must deallocate this memory.
8. rdata: a pointer to a pointer which will be set to point to the returned data. The caller of call_api() must deallocate this memory.

These are the parameters which you ought to send, in the order of their appearance in the parameter block:

1. An unsigned 16 bit integer API number. You should set this value with SSVAL(). I do not know where these numbers are described.
2. An ASCIIZ string describing the parameters to the API function as defined in the LAN Manager documentation. The first parameter, which is the server name, is ommited. This string is based uppon the API function as described in the manual, not the data which is actually passed.
3. An ASCIIZ string describing the data structure which ought to be returned.
4. Any parameters which appear in the function call, as defined in the LAN Manager API documentation, after the "Server" and up to and including the "uLevel" parameters.
5. An unsigned 16 bit integer which gives the size in bytes of the buffer we will use to receive the returned array of data structures. Presumably this should be the same as mdrcnt. This value should be set with SSVAL().
6. An ASCIIZ string describing substructures which should be returned. If no substructures apply, this string is of zero length.

The code in client.c always calls call_api() with no data. It is unclear when a non−zero length data buffer would be sent.

Return value

The returned parameters (pointed to by rparam), in their order of appearance are:

1. An unsigned 16 bit integer which contains the API function's return code. This value should be read with SVAL().
2. An adjustment which tells the amount by which pointers in the returned data should be adjusted. This value should be read with SVAL(). Basically, the address of the start of the returned data buffer should have the returned pointer value added to it and then have this value subtracted from it in order to obtain the currect offset into the returned data buffer.
3. A count of the number of elements in the array of structures returned. It is also possible that this may sometimes be the number of bytes returned.

When call_api() returns, rparam points to the returned parameters. The first if these is the result code. It will be zero if the API call suceeded. This value by be read with "SVAL(rparam,0)".

The second parameter may be read as "SVAL(rparam,2)". It is a 16 bit offset which indicates what the base address of the returned data buffer was when it was built on the server. It should be used to correct pointer before use.

The returned data buffer contains the array of returned data structures. Note that all pointers must be adjusted before use. The function fix_char_ptr() in client.c can be used for this purpose.

The third parameter (which may be read as "SVAL(rparam,4)") has something to do with indicating the amount of data returned or possibly the amount of data which can be returned if enough buffer space is allowed.

Code character table

Certain data structures are described by means of ASCIIz strings containing code characters. These are the code characters:

1. W a type byte little−endian unsigned integer
2. N a count of substructures which follow
3. D a four byte little−endian unsigned integer
4. B a byte (with optional count expressed as trailing ASCII digits)
5. z a four byte offset to a NULL terminated string
6. l a four byte offset to non−string user data
7. b an offset to data (with count expressed as trailing ASCII digits)
8. r pointer to returned data buffer???
9. L length in bytes of returned data buffer???
10. h number of bytes of information available???

The smb.conf file

Lexical Analysis

Basically, the file is processed on a line by line basis. There are four types of lines that are recognized by the lexical analyzer (params.c):

1. Blank lines − Lines containing only whitespace.
2. Comment lines − Lines beginning with either a semi−colon or a pound sign (';' or '#').
3. Section header lines − Lines beginning with an open square bracket ('[').
4. Parameter lines − Lines beginning with any other character. (The default line type.)

The first two are handled exclusively by the lexical analyzer, which ignores them. The latter two line types are scanned for

1. − Section names
2. − Parameter names
3. − Parameter values

These are the only tokens passed to the parameter loader (loadparm.c). Parameter names and values are divided from one another by an equal sign: '='.

Handling of Whitespace

Whitespace is defined as all characters recognized by the isspace() function (see ctype(3C)) except for the newline character ('\n') The newline is excluded because it identifies the end of the line.

1. The lexical analyzer scans past white space at the beginning of a line.
2. Section and parameter names may contain internal white space. All whitespace within a name is compressed to a single space character.
3. Internal whitespace within a parameter value is kept verbatim with the exception of carriage return characters ('\r'), all of which are removed.
4. Leading and trailing whitespace is removed from names and values.

Handling of Line Continuation

Long section header and parameter lines may be extended across multiple lines by use of the backslash character ('\\'). Line continuation is ignored for blank and comment lines.

If the last (non−whitespace) character within a section header or on a parameter line is a backslash, then the next line will be (logically) concatonated with the current line by the lexical analyzer. For example:

```
        param name = parameter value string \
        with line continuation.
```

Would be read as

```
    param name = parameter value string     with line continuation.
```

Note that there are five spaces following the word 'string', representing the one space between 'string' and '\\' in the top line, plus the four preceeding the word 'with' in the second line. (Yes, I'm counting the indentation.)

Line continuation characters are ignored on blank lines and at the end of comments. They are *only* recognized within section and parameter lines.

Line Continuation Quirks

Note the following example:

```
        param name = parameter value string \
    \
    with line continuation.
```

The middle line is *not* parsed as a blank line because it is first concatonated with the top line. The result is

```
param name = parameter value string         with line continuation.
```

The same is true for comment lines.

```
        param name = parameter value string \
        ; comment \
    with a comment.
```

This becomes:

```
param name = parameter value string     ; comment     with a comment.
```

On a section header line, the closing bracket (']') is considered a terminating character, and the rest of the line is ignored. The lines

```
        [ section   name ] garbage \
    param  name  = value
```

are read as

```
        [section name]
    param name = value
```

Syntax

The syntax of the smb.conf file is as follows:

```
  <file>            :==  { <section> } EOF
  <section>         :==  <section header> { <parameter line> }
  <section header>  :==  '[' NAME ']'
  <parameter line>  :==  NAME '=' VALUE NL
```

Basically, this means that

1. a file is made up of zero or more sections, and is terminated by an EOF (we knew that).

2. A section is made up of a section header followed by zero or more parameter lines.
3. A section header is identified by an opening bracket and terminated by the closing bracket. The enclosed NAME identifies the section.
4. A parameter line is divided into a NAME and a VALUE. The *first* equal sign on the line separates the NAME from the VALUE. The VALUE is terminated by a newline character (NL = '\n').

About params.c

The parsing of the config file is a bit unusual if you are used to lex, yacc, bison, etc. Both lexical analysis (scanning) and parsing are performed by params.c. Values are loaded via callbacks to loadparm.c.

NetBIOS in a Unix World

Introduction

This is a short document that describes some of the issues that confront a SMB implementation on unix, and how Samba copes with them. They may help people who are looking at unix<−>PC interoperability.

It was written to help out a person who was writing a paper on unix to PC connectivity.

Usernames

The SMB protocol has only a loose username concept. Early SMB protocols (such as CORE and COREPLUS) have no username concept at all. Even in later protocols clients often attempt operations (particularly printer operations) without first validating a username on the server.

Unix security is based around username/password pairs. A unix box should not allow clients to do any substantive operation without some sort of validation.

The problem mostly manifests itself when the unix server is in "share level" security mode. This is the default mode as the alternative "user level" security mode usually forces a client to connect to the server as the same user for each connected share, which is inconvenient in many sites.

In "share level" security the client normally gives a username in the "session setup" protocol, but does not supply an accompanying password. The client then connects to resources using the "tree connect" protocol, and supplies a password. The problem is that the user on the PC types the username and the password in different contexts, unaware that they need to go together to give access to the server. The username is normally the one the user typed in when they "logged onto" the PC (this assumes Windows for Workgroups). The password is the one they chose when connecting to the disk or printer.

The user often chooses a totally different username for their login as for the drive connection. Often they also want to access different drives as different usernames. The unix server needs some way of divining the correct username to combine with each password.

Samba tries to avoid this problem using several methods. These succeed in the vast majority of cases. The methods include username maps, the service%user syntax, the saving of session setup usernames for later validation and the derivation of the username from the service name (either directly or via the user= option).

File Ownership

The commonly used SMB protocols have no way of saying "you can't do that because you don't own the file". They have, in fact, no concept of file ownership at all.

This brings up all sorts of interesting problems. For example, when you copy a file to a unix drive, and the file is world writeable but owned by another user the file will transfer correctly but will receive the wrong date. This is because the utime() call under unix only succeeds for the owner of the file, or root, even if the file is world writeable. For security reasons Samba does all file operations as the validated user, not root, so the utime() fails. This can stuff up shared development diectories as programs like "make" will not get file time comparisons right.

There are several possible solutions to this problem, including username mapping, and forcing a specific username for particular shares.

Passwords

Many SMB clients uppercase passwords before sending them. I have no idea why they do this. Interestingly WfWg uppercases the password only if the server is running a protocol greater than COREPLUS, so obviously it isn't just the data entry routines that are to blame.

Unix passwords are case sensitive. So if users use mixed case passwords they are in trouble.

Samba can try to cope with this by either using the "password level" option which causes Samba to try the offered password with up to the specified number of case changes, or by using the "password server" option which allows Samba to do its validation via another machine (typically a WinNT server).

Samba supports the password encryption method used by SMB clients. Note that the use of password encryption in Microsoft networking leads to password hashes that are "plain text equivalent". This means that it is *VERY* important to ensure that the Samba smbpasswd file containing these password hashes is only readable by the root user. See the documentation ENCRYPTION.txt for more details.

Locking

The locking calls available under a DOS/Windows environment are much richer than those available in unix. This means a unix server (like Samba) choosing to use the standard fcntl() based unix locking calls to implement SMB locking has to improvise a bit.

One major problem is that dos locks can be in a 32 bit (unsigned) range. Unix locking calls are 32 bits, but are signed, giving only a 31 bit range. Unfortunately OLE2 clients use the top bit to select a locking range used for OLE semaphores.

To work around this problem Samba compresses the 32 bit range into 31 bits by appropriate bit shifting. This seems to work but is not ideal. In a future version a separate SMB lockd may be added to cope with the problem.

It also doesn't help that many unix lockd daemons are very buggy and crash at the slightest provocation. They normally go mostly unused in a unix environment because few unix programs use byte range locking. The stress of huge numbers of lock requests from dos/windows clients can kill the daemon on some systems.

The second major problem is the "opportunistic locking" requested by some clients. If a client requests opportunistic locking then it is asking the server to notify it if anyone else tries to do something on the same file, at which time the client will say if it is willing to give up its lock. Unix has no simple way of implementing opportunistic locking, and currently Samba has no support for it.

Deny Modes

When a SMB client opens a file it asks for a particular "deny mode" to be placed on the file. These modes (DENY_NONE, DENY_READ, DENY_WRITE, DENY_ALL, DENY_FCB and DENY_DOS) specify what actions should be allowed by anyone else who tries to use the file at the same time. If DENY_READ is placed on the file, for example, then any attempt to open the file for reading should fail.

Unix has no equivalent notion. To implement this Samba uses either lock files based on the files inode and placed in a separate lock directory or a shared memory implementation. The lock file method is clumsy and consumes processing and file resources, the shared memory implementation is vastly prefered and is turned on by default for those systems that support it.

Trapdoor UIDs

A SMB session can run with several uids on the one socket. This happens when a user connects to two shares with different usernames. To cope with this the unix server needs to switch uids within the one process. On some unixes (such as SCO) this is not possible. This means that on those unixes the client is restricted to a single uid.

Note that you can also get the "trapdoor uid" message for other reasons. Please see the FAQ for details.

Port numbers

There is a convention that clients on sockets use high "unprivilaged" port numbers (>1000) and connect to servers on low "privilaged" port numbers. This is enforced in Unix as non−root users can't open a socket for listening on port numbers less than 1000.

Most PC based SMB clients (such as WfWg and WinNT) don't follow this convention completely. The main culprit is the netbios nameserving on udp port 137. Name query requests come from a source port of 137. This is a problem when you combine it with the common firewalling technique of not allowing incoming packets on low port numbers. This means that these clients can't query a netbios nameserver on the other side of a low port based firewall.

The problem is more severe with netbios node status queries. I've found that WfWg, Win95 and WinNT3.5 all respond to netbios node status queries on port 137 no matter what the source port was in the request. This works between machines that are both using port 137, but it means it's not possible for a unix user to do a node status request to any of these OSes unless they are running as root. The answer comes back, but it goes to port 137 which the unix user can't listen on. Interestingly WinNT3.1 got this right − it sends node status responses back to the source port in the request.

Protocol Complexity

There are many "protocol levels" in the SMB protocol. It seems that each time new functionality was added to a Microsoft operating system, they added the equivalent functions in a new protocol level of the SMB protocol to "externalise" the new capabilities.

This means the protocol is very "rich", offering many ways of doing each file operation. This means SMB servers need to be complex and large. It also means it is very difficult to make them bug free. It is not just Samba that suffers from this problem, other servers such as WinNT don't support every variation of every call and it has almost certainly been a headache for MS developers to support the myriad of SMB calls that are available.

There are about 65 "top level" operations in the SMB protocol (things like SMBread and SMBwrite). Some of these include hundreds of sub−functions (SMBtrans has at least 120 sub−functions, like DosPrintQAdd and NetSessionEnum). All of them take several options that can change the way they work. Many take dozens of possible "information levels" that change the structures that need to be returned. Samba supports all but 2 of the "top level" functions. It supports only 8 (so far) of the SMBtrans sub−functions. Even NT doesn't support them all.

Samba currently supports up to the "NT LM 0.12" protocol, which is the one preferred by Win95 and WinNT3.5. Luckily this protocol level has a "capabilities" field which specifies which super−duper new−fangled options the server suports. This helps to make the implementation of this protocol level much easier.

There is also a problem with the SMB specications. SMB is a X/Open spec, but the X/Open book is far from ideal, and fails to cover many important issues, leaving much to the imagination. Microsoft recently renamed the SMB protocol CIFS (Common Internet File System) and have published new specifications. These are far superior to the old X/Open documents but there are still undocumented calls and features. This specification is actively being worked on by a CIFS developers mailing list hosted by Microsft.

Tracing samba system calls

This file describes how to do a system call trace on Samba to work out what its doing wrong. This is not for the faint of heart, but if you are reading this then you are probably desperate.

Actually its not as bad as the the above makes it sound, just don't expect the output to be very pretty :−)

Ok, down to business. One of the big advantages of unix systems is that they nearly all come with a system trace utility that allows you to monitor all system calls that a program is making. This is extremely using for debugging and also helps when trying to work out why something is slower than you expect. You can use system tracing without any special compilation options.

The system trace utility is called different things on different systems. On Linux systems its called strace. Under SunOS 4 its called trace. Under SVR4 style systems (including solaris) its called truss. Under many BSD systems its called ktrace.

The first thing you should do is read the man page for your native system call tracer. In the discussion below I'll assume its called strace as strace is the only portable system tracer (its available for free for many unix types) and its also got some of the nicest features.

Next, try using strace on some simple commands. For example, **strace ls** or **strace echo hello**.

You'll notice that it produces a LOT of output. It is showing you the arguments to every system call that the program makes and the result. Very little happens in a program without a system call so you get lots of output. You'll also find that it produces a lot of "preamble" stuff showing the loading of shared libraries etc. Ignore this (unless its going wrong!)

For example, the only line that really matters in the **strace echo hello** output is:

```
write(1, "hello\n", 6)                  = 6
```

all the rest is just setting up to run the program.

Ok, now you're familiar with strace. To use it on Samba you need to strace the running smbd daemon. The way I tend ot use it is to first login from my Windows PC to the Samba server, then use smbstatus to find which process ID that client is attached to, then as root I do **strace −p PID** to attach to that process. I normally redirect the stderr output from this command to a file for later perusal. For example, if I'm using a csh style shell:

strace −f −p 3872 >& strace.out

or with a sh style shell:

strace −f −p 3872 > strace.out 2>&1

Note the "−f" option. This is only available on some systems, and allows you to trace not just the current process, but any children it forks. This is great for finding printing problems caused by the "print command" being wrong.

Once you are attached you then can do whatever it is on the client that is causing problems and you will capture all the system calls that smbd makes.

So how do you interpret the results? Generally I search through the output for strings that I know will appear when the problem happens. For example, if I am having touble with permissions on a file I would search for that files name in the strace output and look at the surrounding lines. Another trick is to match up file descriptor numbers and "follow" what happens to an open file until it is closed.

Beyond this you will have to use your initiative. To give you an idea of what you are looking for here is a piece of strace output that shows that `/dev/null` is not world writeable, which causes printing to fail with Samba:

```
[pid 28268] open("/dev/null", O_RDWR)   = −1 EACCES (Permission denied)
[pid 28268] open("/dev/null", O_WRONLY) = −1 EACCES (Permission denied)
```

The process is trying to first open `/dev/null` read−write then read−only. Both fail. This means `/dev/null` has incorrect permissions.

NT Domain RPC's

Introduction

This document contains information to provide an NT workstation with login services, without the need for an NT server. It is the sgml version of http://mailhost.cb1.com/~lkcl/cifsntdomain.txt, controlled by Luke.

It should be possible to select a domain instead of a workgroup (in the NT workstation's TCP/IP settings) and after the obligatory reboot, type in a username, password, select a domain and successfully log in. I would appreciate any feedback on your experiences with this process, and any comments, corrections and additions to this document.

The packets described here can be easily derived from (and are probably better understood using) Netmon.exe. You will need to use the version of Netmon that matches your system, in order to correctly decode the NETLOGON, lsarpc and srvsvc Transact pipes. This document is derived from NT Service Pack 1 and its corresponding version of Netmon. It is intended that an annotated packet trace be produced, which will likely be more instructive than this document.

Also needed, to fully implement NT Domain Login Services, is the document describing the cryptographic part of the NT authentication. This document is available from comp.protocols.smb; from the ntsecurity.net digest and from the samba digest, amongst other sources.

A copy is available from:

http://ntbugtraq.rc.on.ca/SCRIPTS/WA.EXE?A2=ind9708;L=ntbugtraq;O=A;P=2935

http://mailhost.cb1.com/~lkcl/crypt.html

A c−code implementation, provided by Linus Nordberg of this protocol is available from:

http://samba.org/cgi−bin/mfs/01/digest/1997/97aug/0391.html

http://mailhost.cb1.com/~lkcl/crypt.txt

Also used to provide debugging information is the Check Build version of NT workstation, and enabling full debugging in NETLOGON. This is achieved by setting the following REG_SZ registry key to 0x1ffffff:

```
HKLM\SYSTEM\CurrentControlSet\Services\Netlogon\Parameters
```

Incorrect direct editing of the registry can cause your machine to fail. Then again, so can incorrect implementation of this protocol. See "Liability:" above.

Bear in mind that each packet over−the−wire will have its origin in an API call. Therefore, there are likely to be structures, enumerations and defines that are usefully documented elsewhere.

This document is by no means complete or authoritative. Missing sections include, but are not limited to:

1. Mappings of RIDs to usernames (and vice−versa).
2. What a User ID is and what a Group ID is.

3. The exact meaning/definition of various magic constants or enumerations.
4. The reply error code and use of that error code when a workstation becomes a member of a domain (to be described later). Failure to return this error code will make the workstation report that it is already a member of the domain.
5. the cryptographic side of the NetrServerPasswordSet command, which would allow the workstation to change its password. This password is used to generate the long−term session key. [It is possible to reject this command, and keep the default workstation password].

Sources

cket Traces from Netmonitor (Service Pack 1 and above)

ul Ashton and Luke Leighton's other "NT Domain" doc.

FS documentation − cifs6.txt

FS documentation − cifsrap2.txt

Credits

Paul Ashton: loads of work with Net Monitor; understanding the NT authentication system; reference implementation of the NT domain support on which this document is originally based.

Duncan Stansfield: low−level analysis of MSRPC Pipes.

Linus Nordberg: producing c−code from Paul's crypto spec.

Windows Sourcer development team

Notes and Structures

Notes

1. In the SMB Transact pipes, some "Structures", described here, appear to be 4−byte aligned with the SMB header, at their start. Exactly which "Structures" need aligning is not precisely known or documented.
2. In the UDP NTLOGON Mailslots, some "Structures", described here, appear to be 2−byte aligned with the start of the mailslot, at their start.
3. Domain SID is of the format S−revision−version−auth1−auth2...authN. e.g S−1−5−123−456−789−123−456. the 5 could be a sub−revision.
4. any undocumented buffer pointers must be non−zero if the string buffer it refers to contains characters. exactly what value they should be is unknown. 0x0000 0002 seems to do the trick to indicate that the buffer exists. a NULL buffer pointer indicates that the string buffer is of zero length. If the buffer pointer is NULL, then it is suspected that the structure it refers to is NOT put into (or taken out of) the SMB data stream. This is empirically derived from, for example, the LSA SAM Logon response packet, where if the buffer pointer is NULL, the user information is not inserted into the data stream. Exactly what happens with an array of buffer pointers is not known, although an educated guess can be made.
5. an array of structures (a container) appears to have a count and a pointer. if the count is zero, the pointer is also zero. no further data is put into or taken out of the SMB data stream. if the count is non−zero, then the pointer is also non−zero. immediately following the pointer is the count again, followed by an array of container sub−structures. the count appears a third time after the last sub−structure.

Enumerations

MSRPC Header type

command number in the msrpc packet header

MSRPC_Request:	0x00
MSRPC_Response:	0x02
MSRPC_Bind:	0x0B
MSRPC_BindAck:	0x0C

MSRPC Packet info

The meaning of these flags is undocumented

FirstFrag:	0x01
LastFrag:	0x02
NotaFrag:	0x04
RecRespond:	0x08
NoMultiplex:	0x10
NotForIdemp:	0x20
NotforBcast:	0x40
NoUuid:	0x80

Structures

VOID *

sizeof VOID* is 32 bits.

char

sizeof char is 8 bits.

UTIME

UTIME is 32 bits, indicating time in seconds since 01jan1970. documented in cifs6.txt (section 3.5 page, page 30).

NTTIME

NTTIME is 64 bits. documented in cifs6.txt (section 3.5 page, page 30).

DOM_SID (domain SID structure)

UINT32
: num of sub−authorities in domain SID

UINT8
: SID revision number

UINT8
: num of sub−authorities in domain SID

UINT8[6]
: 6 bytes for domain SID − Identifier Authority.

UINT16[n_subauths]
: domain SID sub−authorities

Note: the domain SID is documented elsewhere.

STR (string)

STR (string) is a char[] : a null−terminated string of ascii characters.

UNIHDR (unicode string header)

UINT16
: length of unicode string

UINT16
: max length of unicode string

UINT32
: 4 − undocumented.

UNIHDR2 (unicode string header plus buffer pointer)

UNIHDR
: unicode string header

*VOID**
: undocumented buffer pointer

UNISTR (unicode string)

UINT16[]
: null−terminated string of unicode characters.

NAME (length−indicated unicode string)

UINT32
: length of unicode string

UINT16[]
: null−terminated string of unicode characters.

UNISTR2 (aligned unicode string)

UINT8[]
: padding to get unicode string 4−byte aligned with the start of the SMB header.

UINT32
: max length of unicode string

UINT32
: 0 − undocumented

UINT32
: length of unicode string

UINT16[]
: string of uncode characters

OBJ_ATTR (object attributes)

UINT32
: 0x18 − length (in bytes) including the length field.

*VOID**
: 0 − root directory (pointer)

*VOID**
: 0 − object name (pointer)

UINT32
: 0 − attributes (undocumented)

*VOID**
: 0 − security descriptior (pointer)

UINT32
: 0 − security quality of service

POL_HND (LSA policy handle)

char[20]
: policy handle

DOM_SID2 (domain SID structure, SIDS stored in unicode)

UINT32
: 5 − SID type

UINT32
: 0 − undocumented

UNIHDR2
: domain SID unicode string header

UNISTR
: domain SID unicode string

Note: there is a conflict between the unicode string header and the unicode string itself as to which to use to indicate string length. this will need to be resolved.

Note: the SID type indicates, for example, an alias; a well−known group etc. this is documented somewhere.

DOM_RID (domain RID structure)

UINT32
: 5 − well−known SID. 1 − user SID (see ShowACLs)

UINT32
: 5 − undocumented

UINT32
: domain RID

UINT32
: 0 − domain index out of above reference domains

LOG_INFO (server, account, client structure)

Note: logon server name starts with two '\' characters and is upper case.

Note: account name is the logon client name from the LSA Request Challenge, with a $ on the end of it, in upper case.

*VOID**
: undocumented buffer pointer

UNISTR2
: logon server unicode string

UNISTR2
: account name unicode string

UINT16
: sec_chan − security channel type

UNISTR2
: logon client machine unicode string

CLNT_SRV (server, client names structure)

Note: logon server name starts with two '\' characters and is upper case.

*VOID**
: undocumented buffer pointer

UNISTR2
: logon server unicode string

*VOID**
: undocumented buffer pointer

UNISTR2
: logon client machine unicode string

CREDS (credentials + time stamp)

char[8]
: credentials

UTIME
: time stamp

CLNT_INFO2 (server, client structure, client credentials)

Note: whenever this structure appears in a request, you must take a copy of the client−calculated credentials received, because they will beused in subsequent credential checks. the presumed intention is to maintain an authenticated request/response trail.

CLNT_SRV
: client and server names

UINT8[]
: ???? padding, for 4−byte alignment with SMB header.

*VOID**
: pointer to client credentials.

CREDS
: client−calculated credentials + client time

CLNT_INFO (server, account, client structure, client credentials)

Note: whenever this structure appears in a request, you must take a copy of the client−calculated credentials received, because they will be used in subsequent credential checks. the presumed intention is to maintain an authenticated request/response trail.

LOG_INFO
: logon account info

CREDS
: client−calculated credentials + client time

ID_INFO_1 (id info structure, auth level 1)

*VOID**
: ptr_id_info_1

UNIHDR
: domain name unicode header

UINT32
: param control

UINT64
: logon ID

UNIHDR
: user name unicode header

UNIHDR
: workgroup name unicode header

char[16]
: arc4 LM OWF Password

char[16]

arc4 NT OWF Password

UNISTR2
: domain name unicode string

UNISTR2
: user name unicode string

UNISTR2
: workstation name unicode string

SAM_INFO (sam logon/logoff id info structure)

Note: presumably, the return credentials is supposedly for the server to verify that the credential chain hasn't been compromised.

CLNT_INFO2
: client identification/authentication info

*VOID**
: pointer to return credentials.

CRED
: return credentials − ignored.

UINT16
: logon level

UINT16
: switch value

```
        switch (switch_value)
        case 1:
        {
            ID_INFO_1     id_info_1;
        }
```

GID (group id info)

UINT32
: group id

UINT32
: user attributes (only used by NT 3.1 and 3.51)

DOM_REF (domain reference info)

*VOID**
: undocumented buffer pointer.

UINT32
: num referenced domains?

*VOID**
: undocumented domain name buffer pointer.

UINT32
: 32 − max number of entries

UINT32
: 4 − num referenced domains?

UNIHDR2
: domain name unicode string header

UNIHDR2[num_ref_doms−1]
: referenced domain unicode string headers

UNISTR
: domain name unicode string

DOM_SID[num_ref_doms]
: referenced domain SIDs

DOM_INFO (domain info, levels 3 and 5 are the same))

UINT8[]
: ??? padding to get 4−byte alignment with start of SMB header

UINT16
: domain name string length * 2

UINT16
: domain name string length * 2

*VOID**
: undocumented domain name string buffer pointer

*VOID**
: undocumented domain SID string buffer pointer

UNISTR2
: domain name (unicode string)

DOM_SID
: domain SID

USER_INFO (user logon info)

Note: it would be nice to know what the 16 byte user session key is for.

NTTIME
: logon time

NTTIME
: logoff time

NTTIME
: kickoff time

NTTIME
: password last set time

NTTIME
: password can change time

NTTIME
: password must change time

UNIHDR
: username unicode string header

UNIHDR
: user's full name unicode string header

UNIHDR
: logon script unicode string header

UNIHDR
: profile path unicode string header

UNIHDR
: home directory unicode string header

UNIHDR
: home directory drive unicode string header

UINT16
: logon count

UINT16
: bad password count

UINT32
: User ID

UINT32
: Group ID

UINT32
: num groups

*VOID**
: undocumented buffer pointer to groups.

UINT32
: user flags

char[16]
: user session key

UNIHDR
: logon server unicode string header

UNIHDR
: logon domain unicode string header

*VOID**
: undocumented logon domain id pointer

char[40]
: 40 undocumented padding bytes. future expansion?

UINT32
: 0 − num_other_sids?

*VOID**
: NULL − undocumented pointer to other domain SIDs.

UNISTR2
: username unicode string

UNISTR2
: user's full name unicode string

UNISTR2
: logon script unicode string

UNISTR2
: profile path unicode string

UNISTR2
: home directory unicode string

UNISTR2
: home directory drive unicode string

UINT32
: num groups

GID[num_groups]
: group info

UNISTR2
: logon server unicode string

UNISTR2
: logon domain unicode string

DOM_SID

domain SID
DOM_SID[num_sids]
other domain SIDs?

SH_INFO_1_PTR (pointers to level 1 share info strings)

Note: see cifsrap2.txt section5, page 10.

0 for shi1_type indicates a Disk.
1 for shi1_type indicates a Print Queue.
2 for shi1_type indicates a Device.
3 for shi1_type indicates an IPC pipe.
0x8000 0000 (top bit set in shi1_type) indicates a hidden share.

*VOID**
shi1_netname − pointer to net name
UINT32
shi1_type − type of share. 0 − undocumented.
*VOID**
shi1_remark − pointer to comment.

SH_INFO_1_STR (level 1 share info strings)

UNISTR2
shi1_netname − unicode string of net name
UNISTR2
shi1_remark − unicode string of comment.

SHARE_INFO_1_CTR

share container with 0 entries:

UINT32
0 − EntriesRead
UINT32
0 − Buffer

share container with > 0 entries:

UINT32
EntriesRead
UINT32
non−zero − Buffer
UINT32
EntriesRead
SH_INFO_1_PTR[EntriesRead]
share entry pointers
SH_INFO_1_STR[EntriesRead]
share entry strings

UINT8[]
: padding to get unicode string 4−byte aligned with start of the SMB header.

UINT32
: EntriesRead

UINT32
: 0 − padding

SERVER_INFO_101

Note: see cifs6.txt section 6.4 − the fields described therein will be of assistance here. for example, the type listed below is the same as fServerType, which is described in 6.4.1.

SV_TYPE_WORKSTATION	0x00000001 All workstations
SV_TYPE_SERVER	0x00000002 All servers
SV_TYPE_SQLSERVER	0x00000004 Any server running with SQL server
SV_TYPE_DOMAIN_CTRL	0x00000008 Primary domain controller
SV_TYPE_DOMAIN_BAKCTRL	0x00000010 Backup domain controller
SV_TYPE_TIME_SOURCE	0x00000020 Server running the timesource service
SV_TYPE_AFP	0x00000040 Apple File Protocol servers
SV_TYPE_NOVELL	0x00000080 Novell servers
SV_TYPE_DOMAIN_MEMBER	0x00000100 Domain Member
SV_TYPE_PRINTQ_SERVER	0x00000200 Server sharing print queue
SV_TYPE_DIALIN_SERVER	0x00000400 Server running dialin service.
SV_TYPE_XENIX_SERVER	0x00000800 Xenix server
SV_TYPE_NT	0x00001000 NT server
SV_TYPE_WFW	0x00002000 Server running Windows for
SV_TYPE_SERVER_NT	0x00008000 Windows NT non DC server
SV_TYPE_POTENTIAL_BROWSER	0x00010000 Server that can run the browser service
SV_TYPE_BACKUP_BROWSER	0x00020000 Backup browser server
SV_TYPE_MASTER_BROWSER	0x00040000 Master browser server
SV_TYPE_DOMAIN_MASTER	0x00080000 Domain Master Browser server
SV_TYPE_LOCAL_LIST_ONLY	

0x40000000 Enumerate only entries marked "local"

SV_TYPE_DOMAIN_ENUM
: 0x80000000 Enumerate Domains. The pszServer and pszDomain parameters must be NULL.

UINT32
: 500 − platform_id

*VOID**
: pointer to name

UINT32
: 5 − major version

UINT32
: 4 − minor version

UINT32
: type (SV_TYPE_... bit field)

*VOID**
: pointer to comment

UNISTR2
: sv101_name − unicode string of server name

UNISTR2
: sv_101_comment − unicode string of server comment.

UINT8[]
: padding to get unicode string 4−byte aligned with start of the SMB header.

MSRPC over Transact Named Pipe

For details on the SMB Transact Named Pipe, see cifs6.txt

MSRPC Pipes

The MSRPC is conducted over an SMB Transact Pipe with a name of `\PIPE\`. You must first obtain a 16 bit file handle, by sending a SMBopenX with the pipe name `\PIPE\srvsvc` for example. You can then perform an SMB Trans, and must carry out an SMBclose on the file handle once you are finished.

Trans Requests must be sent with two setup UINT16s, no UINT16 params (none known about), and UINT8 data parameters sufficient to contain the MSRPC header, and MSRPC data. The first UINT16 setup parameter must be either 0x0026 to indicate an RPC, or 0x0001 to indicate Set Named Pipe Handle state. The second UINT16 parameter must be the file handle for the pipe, obtained above.

The Data section for an API Command of 0x0026 (RPC pipe) in the Trans Request is the RPC Header, followed by the RPC Data. The Data section for an API Command of 0x0001 (Set Named Pipe Handle state) is two bytes. The only value seen for these two bytes is 0x00 0x43.

MSRPC Responses are sent as response data inside standard SMB Trans responses, with the MSRPC Header, MSRPC Data and MSRPC tail.

It is suspected that the Trans Requests will need to be at least 2−byte aligned (probably 4−byte). This is standard practice for SMBs. It is also independent of the observed 4−byte alignments with the start of the MSRPC header, including the 4−byte alignment between the MSRPC header and the MSRPC data.

First, an SMBtconX connection is made to the IPC$ share. The connection must be made using encrypted passwords, not clear−text. Then, an SMBopenX is made on the pipe. Then, a Set Named Pipe Handle State must be sent, after which the pipe is ready to accept API commands. Lastly, and SMBclose is sent.

To be resolved:

lkcl/01nov97 there appear to be two additional bytes after the null−terminated \PIPE\ name for the RPC pipe. Values seen so far are listed below:

```
        initial SMBopenX request:         RPC API command 0x26 params:
        "\\PIPE\\lsarpc"                  0x65 0x63; 0x72 0x70; 0x44 0x65;
        "\\PIPE\\srvsvc"                  0x73 0x76; 0x4E 0x00; 0x5C 0x43;
```

Header

[section to be rewritten, following receipt of work by Duncan Stansfield]

Interesting note: if you set packed data representation to 0x0100 0000 then all 4−byte and 2−byte word ordering is turned around!

The start of each of the NTLSA and NETLOGON named pipes begins with:

offset: 00

Variable type: UINT8

Variable data: 5 − RPC major version

offset: 01

Variable type: UINT8

Variable data: 0 − RPC minor version

offset: 02

Variable type: UINT8

Variable data: 2 − RPC response packet

offset: 03

Variable type: UINT8

Variable data: 3 − (FirstFrag bit−wise or with LastFrag)

offset: 04

Variable type: UINT32

Variable data: 0x1000 0000 − packed data representation

offset: 08

Variable type: UINT16

Variable data: fragment length − data size (bytes) inc header and tail.

offset: 0A

Variable type: UINT16

Variable data: 0 − authentication length

offset: 0C

Variable type: UINT32

Variable data: call identifier. matches 12th UINT32 of incoming RPC data.

offset: 10

Variable type: UINT32

Variable data: allocation hint − data size (bytes) minus header and tail.

offset: 14

Variable type: UINT16

Variable data: 0 − presentation context identifier

offset: 16

Variable type: UINT8

Variable data: 0 − cancel count

offset: 17

Variable type: UINT8

Variable data: in replies: 0 − reserved; in requests: opnum − see #defines.

offset: 18

Variable type: ......

Variable data: start of data (goes on for allocation_hint bytes)

RPC_Packet for request, response, bind and bind acknowledgement

UINT8 versionmaj
    reply same as request (0x05)
UINT8 versionmin
    reply same as request (0x00)
UINT8 type
    one of the MSRPC_Type enums
UINT8 flags
    reply same as request (0x00 for Bind, 0x03 for Request)
UINT32 representation
    reply same as request (0x00000010)
UINT16 fraglength
    the length of the data section of the SMB trans packet
UINT16 authlength
UINT32 callid
    call identifier. (e.g. 0x00149594)
** stub USE TvPacket*
    the remainder of the packet depending on the "type"

Interface identification

the interfaces are numbered. as yet I haven't seen more than one interface used on the same pipe name srvsvc

```
abstract (0x4B324FC8, 0x01D31670, 0x475A7812, 0x88E16EBF, 0x00000003)
transfer (0x8A885D04, 0x11C91CEB, 0x0008E89F, 0x6048102B, 0x00000002)
```

RPC_Iface RW

UINT8 byte[16]
    16 bytes of number
UINT32 version
    the interface number

RPC_ReqBind RW

the remainder of the packet after the header if "type" was Bind in the response header, "type" should be BindAck

UINT16 maxtsize
    maximum transmission fragment size (0x1630)
UINT16 maxrsize
    max receive fragment size (0x1630)
UINT32 assocgid
    associated group id (0x0)
UINT32 numelements
    the number of elements (0x1)
UINT16 contextid
    presentation context identifier (0x0)
UINT8 numsyntaxes

the number of syntaxes (has always been 1?)(0x1)

UINT8[]
: 4−byte alignment padding, against SMB header

** abstractint USE RPC_Iface*
: num and vers. of interface client is using

** transferint USE RPC_Iface*
: num and vers. of interface to use for replies

RPC_Address RW

UINT16 length
: length of the string including null terminator

** port USE string*
: the string above in single byte, null terminated form

RPC_ResBind RW

the response to place after the header in the reply packet

UINT16 maxtsize
: same as request

UINT16 maxrsize
: same as request

UINT32 assocgid
: zero

** secondaddr USE RPC_Address*
: the address string, as described earlier

UINT8[]
: 4−byte alignment padding, against SMB header

UINT8 numresults
: the number of results (0x01)

UINT8[]
: 4−byte alignment padding, against SMB header

UINT16 result
: result (0x00 = accept)

UINT16 reason
: reason (0x00 = no reason specified)

** transfersyntax USE RPC_Iface*
: the transfer syntax from the request

RPC_ReqNorm RW

the remainder of the packet after the header for every other other request

UINT32 allochint
: the size of the stub data in bytes

UINT16 prescontext
: presentation context identifier (0x0)

UINT16 opnum
: operation number (0x15)

** stub USE TvPacket*
    a packet dependent on the pipe name (probably the interface) and the op number)

RPC_ResNorm RW

UINT32 allochint
    # size of the stub data in bytes
UINT16 prescontext
    # presentation context identifier (same as request)
UINT8 cancelcount
    # cancel count? (0x0)
UINT8 reserved
    # 0 − one byte padding
** stub USE TvPacket*
    # the remainder of the reply

Tail

The end of each of the NTLSA and NETLOGON named pipes ends with:

......
    end of data
UINT32
    return code

RPC Bind / Bind Ack

RPC Binds are the process of associating an RPC pipe (e.g \PIPE\lsarpc) with a "transfer syntax" (see RPC_Iface structure). The purpose for doing this is unknown.

Note: The RPC_ResBind SMB Transact request is sent with two uint16 setup parameters. The first is 0x0026; the second is the file handle returned by the SMBopenX Transact response.

Note: The RPC_ResBind members maxtsize, maxrsize and assocgid are the same in the response as the same members in the RPC_ReqBind. The RPC_ResBind member transfersyntax is the same in the response as the

Note: The RPC_ResBind response member secondaddr contains the name of what is presumed to be the service behind the RPC pipe. The mapping identified so far is:

initial SMBopenX request:
    RPC_ResBind response:
"\\PIPE\\srvsvc"
    "\\PIPE\\ntsvcs"
"\\PIPE\\samr"
    "\\PIPE\\lsass"
"\\PIPE\\lsarpc"
    "\\PIPE\\lsass"
"\\PIPE\\wkssvc"
    "\\PIPE\\wksvcs"
"\\PIPE\\NETLOGON"

"\\PIPE\\NETLOGON"

Note: The RPC_Packet fraglength member in both the Bind Request and Bind Acknowledgment must contain the length of the entire RPC data, including the RPC_Packet header.

Request:

RPC_Packet
RPC_ReqBind

Response:

RPC_Packet
RPC_ResBind

NTLSA Transact Named Pipe

The sequence of actions taken on this pipe are:

Establish a connection to the IPC$ share (SMBtconX). use encrypted passwords.
Open an RPC Pipe with the name "\\PIPE\\lsarpc". Store the file handle.
Using the file handle, send a Set Named Pipe Handle state to 0x4300.
Send an LSA Open Policy request. Store the Policy Handle.
Using the Policy Handle, send LSA Query Info Policy requests, etc.
Using the Policy Handle, send an LSA Close.
Close the IPC$ share.

Defines for this pipe, identifying the query are:

LSA Open Policy:
0x2c
LSA Query Info Policy:
0x07
LSA Enumerate Trusted Domains:
0x0d
LSA Open Secret:
0xff
LSA Lookup SIDs:
0xfe
LSA Lookup Names:
0xfd
LSA Close:
0x00

LSA Open Policy

Note: The policy handle can be anything you like.

Request

*VOID**
: buffer pointer

UNISTR2
: server name − unicode string starting with two '\'s

OBJ_ATTR
: object attributes

UINT32
: 1 − desired access

Response

POL_HND
: LSA policy handle

return
: 0 − indicates success

LSA Query Info Policy

Note: The info class in response must be the same as that in the request.

Request

POL_HND
: LSA policy handle

UINT16
: info class (also a policy handle?)

Response

*VOID**
: undocumented buffer pointer

UINT16
: info class (same as info class in request).

```
switch (info class)
case 3:
case 5:
{
DOM_INFO domain info, levels 3 and 5 (are the same).
}

return    0 − indicates success
```

LSA Enumerate Trusted Domains

Request

no extra data

Response

UINT32
: 0 − enumeration context

UINT32
: 0 − entries read

UINT32
: 0 − trust information

return
: 0x8000 001a − "no trusted domains" success code

LSA Open Secret

Request

no extra data

Response

UINT32
: 0 − undocumented

UINT32
: 0 − undocumented

UINT32
: 0 − undocumented

UINT32
: 0 − undocumented

UINT32
: 0 − undocumented

return 0x0C00 0034 − "no such secret" success code

LSA Close

Request

POL_HND
: policy handle to be closed

Response

POL_HND
    0s − closed policy handle (all zeros)

return 0 − indicates success

LSA Lookup SIDS

Note: num_entries in response must be same as num_entries in request.

Request

POL_HND
    LSA policy handle
UINT32
    num_entries
*VOID**
    undocumented domain SID buffer pointer
*VOID**
    undocumented domain name buffer pointer
VOID[num_entries] undocumented domain SID pointers to be looked up.*
    DOM_SID[num_entries] domain SIDs to be looked up.
char[16]
    completely undocumented 16 bytes.

Response

DOM_REF
    domain reference response
UINT32
    num_entries (listed above)
*VOID**
    undocumented buffer pointer
UINT32
    num_entries (listed above)
DOM_SID2[num_entries]
    domain SIDs (from Request, listed above).
UINT32
    num_entries (listed above)

return 0 − indicates success

LSA Lookup Names

Note: num_entries in response must be same as num_entries in request.

Request

POL_HND
    LSA policy handle
UINT32
    num_entries
UINT32
    num_entries
*VOID**
    undocumented domain SID buffer pointer
*VOID**
    undocumented domain name buffer pointer
NAME[num_entries]
    names to be looked up.
char[]
    undocumented bytes − falsely translated SID structure?

Response

DOM_REF
    domain reference response
UINT32
    num_entries (listed above)
*VOID**
    undocumented buffer pointer
UINT32
    num_entries (listed above)
DOM_RID[num_entries]
    domain SIDs (from Request, listed above).
UINT32
    num_entries (listed above)

return 0 − indicates success

NETLOGON rpc Transact Named Pipe

The sequence of actions taken on this pipe are:

tablish a connection to the IPC$ share (SMBtconX). use encrypted passwords.
en an RPC Pipe with the name "\\PIPE\\NETLOGON". Store the file handle.
ing the file handle, send a Set Named Pipe Handle state to 0x4300.
eate Client Challenge. Send LSA Request Challenge. Store Server Challenge.
lculate Session Key. Send an LSA Auth 2 Challenge. Store Auth2 Challenge.
lc/Verify Client Creds. Send LSA Srv PW Set. Calc/Verify Server Creds.
lc/Verify Client Creds. Send LSA SAM Logon . Calc/Verify Server Creds.
lc/Verify Client Creds. Send LSA SAM Logoff. Calc/Verify Server Creds.
ose the IPC$ share.

Defines for this pipe, identifying the query are

LSA Request Challenge:	0x04
LSA Server Password Set:	0x06
LSA SAM Logon:	0x02
LSA SAM Logoff:	0x03
LSA Auth 2:	0x0f
LSA Logon Control:	0x0e

LSA Request Challenge

Note: logon server name starts with two '\' characters and is upper case.

Note: logon client is the machine, not the user.

Note: the initial LanManager password hash, against which the challenge is issued, is the machine name itself (lower case). there will becalls issued (LSA Server Password Set) which will change this, later. refusing these calls allows you to always deal with the same password (i.e the LM# of the machine name in lower case).

Request

*VOID**
: undocumented buffer pointer

UNISTR2
: logon server unicode string

UNISTR2
: logon client unicode string

char[8]
: client challenge

Response

char[8]
: server challenge

return 0 − indicates success

LSA Authenticate 2

Note: in between request and response, calculate the client credentials, and check them against the client−calculated credentials (this process uses the previously received client credentials).

Note: neg_flags in the response is the same as that in the request.

Note: you must take a copy of the client−calculated credentials received here, because they will be used in subsequent authentication packets.

Request

LOG_INFO
: client identification info

char[8]
: client−calculated credentials

UINT8[]
: padding to 4−byte align with start of SMB header.

UINT32
: neg_flags − negotiated flags (usual value is 0x0000 01ff)

Response

char[8]
: server credentials.

UINT32
: neg_flags − same as neg_flags in request.

return 0 − indicates success. failure value unknown.

LSA Server Password Set

Note: the new password is suspected to be a DES encryption using the old password to generate the key.

Note: in between request and response, calculate the client credentials, and check them against the client−calculated credentials (this process uses the previously received client credentials).

Note: the server credentials are constructed from the client−calculated credentials and the client time + 1 second.

Note: you must take a copy of the client−calculated credentials received here, because they will be used in subsequent authentication packets.

Request

CLNT_INFO
: client identification/authentication info

char[]
: new password − undocumented.

Response

CREDS
: server credentials. server time stamp appears to be ignored.

return 0 − indicates success; 0xC000 006a indicates failure

LSA SAM Logon

Note: valid_user is True iff the username and password hash are valid for the requested domain.

Request

SAM_INFO
: sam_id structure

Response

*VOID**
: undocumented buffer pointer

CREDS
: server credentials. server time stamp appears to be ignored.

```
if (valid_user)
{
        UINT16      3 - switch value indicating USER_INFO structure.
    VOID*     non-zero - pointer to USER_INFO structure
    USER_INFO user logon information

    UINT32    1 - Authoritative response; 0 - Non-Auth?

    return    0 - indicates success
}
else
{
        UINT16    0 - switch value.  value to indicate no user presumed.
    VOID*     0x0000 0000 - indicates no USER_INFO structure.

    UINT32    1 - Authoritative response; 0 - Non-Auth?

    return    0xC000 0064 - NT_STATUS_NO_SUCH_USER.
}
```

LSA SAM Logoff

Note: presumably, the SAM_INFO structure is validated, and a (currently undocumented) error code returned if the Logoff is invalid.

Request

SAM_INFO
: sam_id structure

Response

*VOID**
: undocumented buffer pointer

CREDS
: server credentials. server time stamp appears to be ignored.

return 0 − indicates success. undocumented failure indication.

\\MAILSLOT\NET\NTLOGON

Note: mailslots will contain a response mailslot, to which the response should be sent. the target NetBIOS name is REQUEST_NAME<20>, where REQUEST_NAME is the name of the machine that sent the request.

Query for PDC

Note: NTversion, LMNTtoken, LM20token in response are the same as those given in the request.

Request

UINT16
: 0x0007 − Query for PDC

STR
: machine name

STR
: response mailslot

UINT8[]
: padding to 2−byte align with start of mailslot.

UNISTR
: machine name

UINT32
: NTversion

UINT16
: LMNTtoken

UINT16
: LM20token

Response

UINT16
: 0x000A − Respose to Query for PDC

STR
: machine name (in uppercase)

UINT8[]
: padding to 2−byte align with start of mailslot.

UNISTR
: machine name

UNISTR
: domain name

UINT32
: NTversion (same as received in request)

UINT16
: LMNTtoken (same as received in request)

UINT16
: LM20token (same as received in request)

SAM Logon

Note: machine name in response is preceded by two '\' characters.

Note: NTversion, LMNTtoken, LM20token in response are the same as those given in the request.

Note: user name in the response is presumably the same as that in the request.

Request

UINT16
: 0x0012 − SAM Logon

UINT16
: request count

UNISTR
: machine name

UNISTR
: user name

STR
: response mailslot

UINT32
: alloweable account

UINT32
: domain SID size

char[sid_size]
: domain SID, of sid_size bytes.

UINT8[]
: ???? padding to 4? 2? −byte align with start of mailslot.

UINT32
: NTversion

UINT16
: LMNTtoken

UINT16
: LM20token

Response

UINT16
: 0x0013 − Response to SAM Logon

UNISTR
: machine name

UNISTR
: user name − workstation trust account

UNISTR
: domain name

UINT32
: NTversion

UINT16
: LMNTtoken

UINT16

LM20token

SRVSVC Transact Named Pipe

Defines for this pipe, identifying the query are:

Net Share Enum
: 0x0f

Net Server Get Info
: 0x15

Net Share Enum

Note: share level and switch value in the response are presumably the same as those in the request.

Note: cifsrap2.txt (section 5) may be of limited assistance here.

Request

*VOID**
: pointer (to server name?)

UNISTR2
: server name

UINT8[]
: padding to get unicode string 4−byte aligned with the start of the SMB header.

UINT32
: share level

UINT32
: switch value

*VOID**
: pointer to SHARE_INFO_1_CTR

SHARE_INFO_1_CTR
: share info with 0 entries

UINT32
: preferred maximum length (0xffff ffff)

Response

UINT32
: share level

UINT32
: switch value

*VOID**
: pointer to SHARE_INFO_1_CTR

SHARE_INFO_1_CTR
: share info (only added if share info ptr is non−zero)

return 0 − indicates success

Net Server Get Info

Note: level is the same value as in the request.

Request

UNISTR2
: server name

UINT32
: switch level

Response

UINT32
: switch level

*VOID**
: pointer to SERVER_INFO_101

SERVER_INFO_101
: server info (only added if server info ptr is non−zero)

return 0 − indicates success

Cryptographic side of NT Domain Authentication

Definitions

Add(A1,A2)	Intel byte ordered addition of corresponding 4 byte words in arrays A1 and A2
E(K,D)	DES ECB encryption of 8 byte data D using 7 byte key K
lmowf()	Lan man hash
ntowf()	NT hash
PW	md4(machine_password) == md4(lsadump $machine.acc) == pwdump(machine$) (initially) == md4(lmowf(unicode(machine)))
ARC4(K,Lk,D,Ld)	ARC4 encryption of data D of length Ld with key K of length Lk
v[m..n(,l)]	subset of v from bytes m to n, optionally padded with zeroes to length l
Cred(K,D)	E(K[7..7,7],E(K[0..6],D)) computes a credential
Time()	4 byte current time
Cc,Cs	8 byte client and server challenges Rc,Rs: 8 byte client and server credentials

---

Protocol

C−>S ReqChal,Cc S−>C Cs

C & S compute session key Ks = E(PW[9..15],E(PW[0..6],Add(Cc,Cs)))

C: Rc = Cred(Ks,Cc) C−>S Authenticate,Rc S: Rs = Cred(Ks,Cs), assert(Rc == Cred(Ks,Cc)) S−>C Rs C: assert(Rs == Cred(Ks,Cs))

On joining the domain the client will optionally attempt to change its password and the domain controller may refuse to update it depending on registry settings. This will also occur weekly afterwards.

C: Tc = Time(), Rc' = Cred(Ks,Rc+Tc) C−>S ServerPasswordSet,Rc',Tc, arc4(Ks[0..7,16],lmowf(randompassword()) C: Rc = Cred(Ks,Rc+Tc+1) S: assert(Rc' == Cred(Ks,Rc+Tc)), Ts = Time() S: Rs' = Cred(Ks,Rs+Tc+1) S−>C Rs',Ts C: assert(Rs' == Cred(Ks,Rs+Tc+1)) S: Rs = Rs'

User: U with password P wishes to login to the domain (incidental data such as workstation and domain omitted)

C: Tc = Time(), Rc' = Cred(Ks,Rc+Tc) C−>S NetLogonSamLogon,Rc',Tc,U, arc4(Ks[0..7,16],16,ntowf(P),16), arc4(Ks[0..7,16],16,lmowf(P),16) S: assert(Rc' == Cred(Ks,Rc+Tc)) assert(passwords match those in SAM) S: Ts = Time()

S−>C Cred(Ks,Cred(Ks,Rc+Tc+1)),userinfo(logon script,UID,SIDs,etc) C: assert(Rs == Cred(Ks,Cred(Rc+Tc+1)) C: Rc = Cred(Ks,Rc+Tc+1)

Comments

On first joining the domain the session key could be computed by anyone listening in on the network as the machine password has a well known value. Until the machine is rebooted it will use this session key to encrypt NT and LM one way functions of passwords which are password equivalents. Any user who logs in before the machine has been rebooted a second time will have their password equivalent exposed. Of course the new machine password is exposed at this time anyway.

None of the returned user info such as logon script, profile path and SIDs *appear* to be protected by anything other than the TCP checksum.

The server time stamps appear to be ignored.

The client sends a ReturnAuthenticator in the SamLogon request which I can't find a use for. However its time is used as the timestamp returned by the server.

The password OWFs should NOT be sent over the network reversibly encrypted. They should be sent using ARC4(Ks,md4(owf)) with the server computing the same function using the owf values in the SAM.

SIDs and RIDs

SIDs and RIDs are well documented elsewhere.

A SID is an NT Security ID (see DOM_SID structure). They are of the form:

```
revision−NN−SubAuth1−SubAuth2−SubAuth3...
revision−0xNNNNNNNNNNNN−SubAuth1−SubAuth2−SubAuth3...
```

currently, the SID revision is 1. The Sub−Authorities are known as Relative IDs (RIDs).

Well−known SIDs

Universal well−known SIDs

Null SID	S−1−0−0
World	S−1−1−0
Local	S−1−2−0
Creator Owner ID	S−1−3−0
Creator Group ID	S−1−3−1
Creator Owner Server ID	S−1−3−2
Creator Group Server ID	S−1−3−3
(Non−unique IDs)	S−1−4

NT well−known SIDs

NT Authority	S−1−5
Dialup	S−1−5−1
Network	S−1−5−2
Batch	S−1−5−3
Interactive	S−1−5−4
Service	S−1−5−6
AnonymousLogon(aka null logon session)	S−1−5−7
Proxy	S−1−5−8
ServerLogon(aka domain controller account)	S−1−5−8
(Logon IDs)	S−1−5−5−X−Y

(NT non−unique IDs)
S−1−5−0x15−...
(Built−in domain)
s−1−5−0x20

Well−known RIDS

A RID is a sub−authority value, as part of either a SID, or in the case of Group RIDs, part of the DOM_GID structure, in the USER_INFO_1 structure, in the LSA SAM Logon response.

Well−known RID users

Groupname: DOMAIN_USER_RID_ADMIN

????: 0x0000

RID: 01F4

Groupname: DOMAIN_USER_RID_GUEST

????: 0x0000

RID: 01F5

Well−known RID groups

Groupname: DOMAIN_GROUP_RID_ADMINS

????: 0x0000

RID: 0200

Groupname: DOMAIN_GROUP_RID_USERS

????: 0x0000

RID: 0201

Groupname: DOMAIN_GROUP_RID_GUESTS

????: 0x0000

RID: 0202

Well−known RID aliases

Groupname: DOMAIN_ALIAS_RID_ADMINS

????: 0x0000

RID: 0220

Groupname: DOMAIN_ALIAS_RID_USERS

????: 0x0000

RID: 0221

Groupname: DOMAIN_ALIAS_RID_GUESTS

????: 0x0000

RID: 0222

Groupname: DOMAIN_ALIAS_RID_POWER_USERS

????: 0x0000

RID: 0223

Groupname: DOMAIN_ALIAS_RID_ACCOUNT_OPS

????: 0x0000

RID: 0224

Groupname: DOMAIN_ALIAS_RID_SYSTEM_OPS

????: 0x0000

RID: 0225

Groupname: DOMAIN_ALIAS_RID_PRINT_OPS

????: 0x0000

RID: 0226

Groupname: DOMAIN_ALIAS_RID_BACKUP_OPS

????: 0x0000

RID: 0227

Groupname: DOMAIN_ALIAS_RID_REPLICATOR

????: 0x0000

RID: 0228

Samba Printing Internals

Abstract

The purpose of this document is to provide some insight into Samba's printing functionality and also to describe the semantics of certain features of Windows client printing.

Printing Interface to Various Back ends

Samba uses a table of function pointers to seven functions. The function prototypes are defined in the `printif` structure declared in `printing.h`.

- retrieve the contents of a print queue
- pause the print queue
- resume a paused print queue
- delete a job from the queue
- pause a job in the print queue
- result a paused print job in the queue
- submit a job to the print queue

Currently there are only two printing back end implementations defined.

- a generic set of functions for working with standard UNIX printing subsystems
- a set of CUPS specific functions (this is only enabled if the CUPS libraries were located at compile time).

Print Queue TDB's

Samba provides periodic caching of the output from the "lpq command" for performance reasons. This cache time is configurable in seconds. Obviously the longer the cache time the less often smbd will be required to exec a copy of lpq. However, the accuracy of the print queue contents displayed to clients will be diminished as well.

The list of currently opened print queue TDB's can be found be examining the list of tdb_print_db structures (see print_db_head in printing.c). A queue TDB is opened using the wrapper function printing.c:get_print_db_byname(). The function ensures that smbd does not open more than MAX_PRINT_DBS_OPEN in an effort to prevent a large print server from exhausting all available file descriptors. If the number of open queue TDB's exceeds the MAX_PRINT_DBS_OPEN limit, smbd falls back to a most recently used algorithm for maintaining a list of open TDB's.

There are two ways in which a a print job can be entered into a print queue's TDB. The first is to submit the job from a Windows client which will insert the job information directly into the TDB. The second method is to have the print job picked up by executing the "lpq command".

```
/* included from printing.h */
struct printjob {
        pid_t pid; /* which process launched the job */
        int sysjob; /* the system (lp) job number */
        int fd; /* file descriptor of open file if open */
        time_t starttime; /* when the job started spooling */
        int status; /* the status of this job */
        size_t size; /* the size of the job so far */
        int page_count; /* then number of pages so far */
        BOOL spooled; /* has it been sent to the spooler yet? */
        BOOL smbjob; /* set if the job is a SMB job */
        fstring filename; /* the filename used to spool the file */
        fstring jobname; /* the job name given to us by the client */
        fstring user; /* the user who started the job */
        fstring queuename; /* service number of printer for this job */
        NT_DEVICEMODE *nt_devmode;
};
```

The current manifestation of the printjob structure contains a field for the UNIX job id returned from the "lpq command" and a Windows job ID (32−bit bounded by PRINT_MAX_JOBID). When a print job is returned by the "lpq command" that does not match an existing job in the queue's TDB, a 32−bit job ID above the <*vance doesn't know what word is missing here*> is generating by adding UNIX_JOB_START to the id reported by lpq.

In order to match a 32−bit Windows jobid onto a 16−bit lanman print job id, smbd uses an in memory TDB to match the former to a number appropriate for old lanman clients.

When updating a print queue, smbd will perform the following steps (refer to `print.c:print_queue_update()`):

1. Check to see if another smbd is currently in the process of updating the queue contents by checking the pid stored in `LOCK/`*`printer_name`*. If so, then do not update the TDB.
2. Lock the mutex entry in the TDB and store our own pid. Check that this succeeded, else fail.
3. Store the updated time stamp for the new cache listing
4. Retrieve the queue listing via "lpq command"
5.
```
        foreach job in the queue
        {
                if the job is a UNIX job, create a new entry;
                if the job has a Windows based jobid, then
                {
                        Lookup the record by the jobid;
                        if the lookup failed, then
                                treat it as a UNIX job;
                        else
                                update the job status only
                }
        }
```
6. Delete any jobs in the TDB that are not in the in the lpq listing
7. Store the print queue status in the TDB
8. update the cache time stamp again

Note that it is the contents of this TDB that is returned to Windows clients and not the actual listing from the "lpq command".

The NT_DEVICEMODE stored as part of the printjob structure is used to store a pointer to a non−default DeviceMode associated with the print job. The pointer will be non−null when the client included a Device Mode in the OpenPrinterEx() call and subsequently submitted a job for printing on that same handle. If the client did not include a Device Mode in the OpenPrinterEx() request, the nt_devmode field is NULL and the job has the printer's device mode associated with it by default.

Only non−default Device Mode are stored with print jobs in the print queue TDB. Otherwise, the Device Mode is obtained from the printer object when the client issues a GetJob(level == 2) request.

ChangeID & Client Caching of Printer Information

[To be filled in later]

Windows NT/2K Printer Change Notify

When working with Windows NT+ clients, it is possible for a print server to use RPC to send asynchronous change notification events to clients for certain printer and print job attributes. This can be useful when the client needs to know that a new job has been added to the queue for a given printer or that the driver for a printer has been changed. Note that this is done entirely orthogonal to cache updates based on a new ChangeID for a printer object.

The basic set of RPC's used to implement change notification are

- RemoteFindFirstPrinterChangeNotifyEx (RFFPCN)
- RemoteFindNextPrinterChangeNotifyEx (RFNPCN)
- FindClosePrinterChangeNotify(FCPCN)
- ReplyOpenPrinter
- ReplyClosePrinter
- RouteRefreshPrinterChangeNotify (RRPCN)

One additional RPC is available to a server, but is never used by the Windows spooler service:

- RouteReplyPrinter()

The opnum for all of these RPC's are defined in include/rpc_spoolss.h

Windows NT print servers use a bizarre method of sending print notification event to clients. The process of registering a new change notification handle is as follows. The 'C' is for client and the 'S' is for server. All error conditions have been eliminated.

```
C:      Obtain handle to printer or to the printer
        server via the standard OpenPrinterEx() call.
S:      Respond with a valid handle to object

C:      Send a RFFPCN request with the previously obtained
        handle with either (a) set of flags for change events
        to monitor, or (b) a PRINTER_NOTIFY_OPTIONS structure
        containing the event information to monitor.  The windows
        spooler has only been observed to use (b).
S:      The <* another missing word*> opens a new TCP session to the client (thus requiring
        all print clients to be CIFS servers as well) and sends
        a ReplyOpenPrinter() request to the client.
C:      The client responds with a printer handle that can be used to
        send event notification messages.
S:      The server replies success to the RFFPCN request.

C:      The windows spooler follows the RFFPCN with a RFNPCN
        request to fetch the current values of all monitored
        attributes.
```

```
S:      The server replies with an array SPOOL_NOTIFY_INFO_DATA
        structures (contained in a SPOOL_NOTIFY_INFO structure).

C:      If the change notification handle is ever released by the
        client via a FCPCN request, the server sends a ReplyClosePrinter()
        request back to the client first.  However a request of this
        nature from the client is often an indication that the previous
        notification event was not marshalled correctly by the server
        or a piece of data was wrong.
S:      The server closes the internal change notification handle
        (POLICY_HND) and does not send any further change notification
        events to the client for that printer or job.
```

The current list of notification events supported by Samba can be found by examining the internal tables in srv_spoolss_nt.c

- printer_notify_table[]
- job_notify_table[]

When an event occurs that could be monitored, smbd sends a message to itself about the change. The list of events to be transmitted are queued by the smbd process sending the message to prevent an overload of TDB usage and the internal message is sent during smbd's idle loop (refer to printing/notify.c and the functions send_spoolss_notify2_msg() and print_notify_send_messages()).

The decision of whether or not the change is to be sent to connected clients is made by the routine which actually sends the notification. (refer to srv_spoolss_nt.c:recieve_notify2_message()).

Because it possible to receive a listing of multiple changes for multiple printers, the notification events must be split into categories by the printer name. This makes it possible to group multiple change events to be sent in a single RPC according to the printer handle obtained via a ReplyOpenPrinter().

The actual change notification is performed using the RRPCN request RPC. This packet contains

- the printer handle registered with the client's spooler on which the change occurred
- The change_low value which was sent as part of the last RFNPCN request from the client
- The SPOOL_NOTIFY_INFO container with the event information

A `SPOOL_NOTIFY_INFO` contains:

- the version and flags field are predefined and should not be changed
- The count field is the number of entries in the SPOOL_NOTIFY_INFO_DATA array

The `SPOOL_NOTIFY_INFO_DATA` entries contain:

- The type defines whether or not this event is for a printer or a print job
- The field is the flag identifying the event
- the notify_data union contains the new valuie of the attribute
- The enc_type defines the size of the structure for marshalling and unmarshalling
- (a) the id must be 0 for a printer event on a printer handle. (b) the id must be the job id for an event on a printer job (c) the id must be the matching number of the printer index used in the response packet to the RFNPCN when using a print server handle for notification. Samba currently uses the snum of the printer for this which can break if the list of services has been modified since the notification handle was registered.

- The size is either (a) the string length in UNICODE for strings, (b) the size in bytes of the security descriptor, or (c) 0 for data values.

Samba WINS Internals

WINS Failover

The current Samba codebase possesses the capability to use groups of WINS servers that share a common namespace for NetBIOS name registration and resolution. The formal parameter syntax is

```
        WINS_SERVER_PARAM       = SERVER [ SEPARATOR SERVER_LIST ]
        WINS_SERVER_PARAM       = "wins server"
        SERVER                  = ADDR[:TAG]
        ADDR                    = ip_addr | fqdn
        TAG                     = string
        SEPARATOR               = comma | \s+
        SERVER_LIST             = SERVER [ SEPARATOR SERVER_LIST ]
```

A simple example of a valid wins server setting is

```
[global]
        wins server = 192.168.1.2 192.168.1.3
```

In the event that no TAG is defined in for a SERVER in the list, smbd assigns a default TAG of "*". A TAG is used to group servers of a shared NetBIOS namespace together. Upon startup, nmbd will attempt to register the netbios name value with one server in each tagged group.

An example using tags to group WINS servers together is show here. Note that the use of interface names in the tags is only by convention and is not a technical requirement.

```
[global]
        wins server = 192.168.1.2:eth0 192.168.1.3:eth0 192.168.2.2:eth1
```

Using this configuration, nmbd would attempt to register the server's NetBIOS name with one WINS server in each group. Because the "eth0" group has two servers, the second server would only be used when a registration (or resolution) request to the first server in that group timed out.

NetBIOS name resolution follows a similar pattern as name registration. When resolving a NetBIOS name via WINS, smbd and other Samba programs will attempt to query a single WINS server in a tagged group until either a positive response is obtained at least once or until a server from every tagged group has responded negatively to the name query request. If a timeout occurs when querying a specific WINS server, that server is marked as down to prevent further timeouts and the next server in the WINS group is contacted. Once marked as dead, Samba will not attempt to contact that server for name registration/resolution queries for a period of 10 minutes.