# Communicating with SVCCTL (Service Control)

## Some initial words (You can skip this if you only want to know how it's done )

In this tutorial I want to create a Java Program that is able to access a remote Windows machines Service Registry and for example start and stop services on that.

Access will be done during DCE/RPC calls transported using SMB named pipes. In contrast to allmost all other techniques this has the smallest configuration impact on the targeted machine. All you have to have configured are the "File and Printer Sharing" and have the administrative Shares enabled. In most cases Windows is allready configured correctly (http://everydaynerd.com/how-to/fix/windows-7-enable-admin-share).

There seem to be several Java libraries that offer services in the DCE/RPC sector. The one I liked most was JCIFS, which seems to be the base of most more complex solutions and seems to the only not-dead DCE/RPC project. I could not find a single other library that had any update in the last 3-4 years. Unfortunately support I got on mailinglists, forums, etc. for all projects was allmost non-existent. At least with JCIFS I got a response on the list ... even if it was no real help. One thing that convinced me was that JInterop is based on JCIFS and this seems to be widely used.

While doing the basic stuff with JCIFS was pretty easy, finding out how to do DCE/RPC calls was allmost impossible. Unfortunately JCIFS allready had stuff for a lot of DCE/RPC services, but how to use them wasn't explained anywhere and nither Google nor the Mailinglist was able or willing to explain how to do it. So I had to look how JCIFS did stuff internally. So the source was with me, but it turned out it was more the dark side of the source ... and there were no Cookies (This was regarding the methodology of how to use it, but this seems to be more a DCE/RPC thing than a JCIFS thing).

The svcctl service I wanted to acces didn't have any client implementation in JCIFS so I had to create my own. Luckily this code can be generated automatically from so-called IDL files (http://en.wikipedia.org/wiki/Microsoft_Interface_Definition_Language). This interface definition was a well protected secret of Microsoft. The guys at the Samba project did quite a good job of reverse engineering this. Luckily Microsoft had to publish the specification in 2002 in order to avoid major trouble with the European Union (http://www.theregister.co.uk/2002/03/19/why_microsofts_eu_concession/).

Based upon an IDL it is possible to generate the code for communicating with MS services automatically. One implementation of such a compiler is MIDLC (http://jcifs.samba.org/src/) which is available from the JCIFS site.

Fortunately midlc allready comes with a set of idl files. Even with an idl for the svcctl service. From having a look at them the svcctl.idl seems to be implemented as far as the Samba guys managed to do this in the early stages (Not all service functions are available). Unfortunately the ones I was looking for were missing (CreateServiceW, DeleteServiceW, StopServiceW, ...).

So If you now think: "Hey ... so I'll get the official IDL file from Microsoft and use that" (http://msdn.microsoft.com/en-us/library/cc245860(v=prot.10).aspx) ... well what should I say ... the formats are slightly different and therefore you can't use them out of the box. Michal from the JCIFS Mailinglist explained that midlc understands a subset of the official Microsoft IDL format, but with the extension of the OpNums which provide the operation-code that has to be transmitted with every request (http://www.hsc.fr/ressources/articles/win_net_srv/msrpc_svcctl.html). So even if the official IDLs are not directly usable, they will certainly help extending the svcctl.idl shipped with midlc.

So far, so good ... Now I used midlc on a linux machine to generate the stubs for communicating with SVCCTL (I couln't get it to compile on Windows). I used the default files first because I had to learn how do do these calls first and I didn't want to get stuck, just because my IDL was wrong.

Generating the file was pretty easy but unfortunateley I wasted about 5 full days on trying to find out how to use the generated stub classes. All I got was "DCERPC_FAULT_PROTO_ERROR" without any explanation from Windows, what was actually going wrong. That was when I temporarily abandoned my quest. Thanks to Amazon I was able to purchase one used print of the Book "DCE/RPC over SMB - Samba and Windows NT Domain Internals" (With a neat stamp that this book is the property of the Syracuse University). This tought me a lot about how these DCE/RPC calls have to be performed in general and I found the book quite entertaining. Still I could not work out why I was getting these stupid errors. I even started to compare my WireShark dumps with those of PSExec (http://technet.microsoft.com/de-de/sysinternals/bb897553) as this is a tool that does something very similar. But it seems to be doing it completely different. So I was even more confused afterwards.

Beeing completely frustrated I even offered JCIFS professionals a bounty (not the chocolate bar) for providing me with a solution, but not a single "professional" even answered ... "Hail to the glory of open-source!!!!".

Luckily my employee Björn stumbled over Alfresco JLan (http://svn.alfresco.com/repos/alfresco-open-mirror/alfresco/HEAD/root/projects/alfresco-jlan/) and discovered that it could do a lot more than simply copy files from one server to another. It actually came with a full implementation of the SVCCTL Service. This allowed him to build the Software I was trying to build for so long. It took quite some time to figgure out how to build such an application with JLan, but he managed to get it going.

The general functionality was implemented pretty fast after that and he sarted implementing some logic that sent quite large amounts of data over Named Pipes. This is where he began having trouble. Errors complaning about the size of data packets keept on occuring, ruining the data-transfer. He avoided this by breaking everything down to really small pieces of data, but I was never really satisfied with this and it didn't look as if there was any support available from anywhere . So this was only a Proof-Of-Concept solution.

Then one day I had the Idea to use a stripped-down minimal implementation of a JLan program and to have a look at what it does in WireShark and to try to replicate what JLan did in JCIFS. With this approach it took another two days and I was actually able to start a Windows service on a

remote machine using JCIFS.

The rest of this tutorial will explain in detail the steps needed to do this. Hopefully this document might save the sanity of the one or the other developer also going nuts on something similar

## Generating the Stub

> **INFO**
> You can actually skip this part if you only want to fetch the generated and patched file as I posted the entire content in the following chapter.

Ok ... so before we can start coding we have to generate the stub code, that we will use do encode/decode the DCE/RPC requests. This is done using the midlc tool (http://jcifs.samba.org/src/midlc-0.6.1.tar.gz). Download it to a linux machine and build it:

**Building midlc**

```
$ cd libmba-0.9.1
$ make ar
$ cd ..
$ make
```

You can leave away the "make install" documented in the README.txt as this failled to execute and isn't really needed.

Now you have the "midlc" binary and can start generating code. So if you simply execute the "midlc" binary, it will present you the options:

**Building midlc**

```
cdutz@ubuntu:~/Test/midlc-0.6.1$ ./midlc
usage: ./midlc [-v|-d] [-s <symtab>] [-t jcifs|java|samba|c] [-o outfile]
[-Dmacro=defn] <filename>
```

Unfortunately I was complaining about missing documentation of the parameters here before and I am sorry for that because I was corrected by Michael on the mailinglist. It turned out that the explanation of how to use the tool simply came AFTER all of the change-history. I was assuming no more usage information was to be expected after the first "how to compile block". So just scroll down to the end of the midlcs README.txt and you will find this:

**Building midlc**

```
RUNNING
usage: ./midlc [-v|-d] [-s <symtab>] [-t c|java] [-o outfile]
[-Dmacro=defn] <f$
-v verbose   - List partial parse tree
-d debug     - List full parse tree and insert symbol comments into stub
-s <symtab>  - Specify primative data type symbol table (default is
               currently symtabjava.txt)
-t c|java    - Specify the code type generated (currently only java
               is supported)
-o <outfile> - Specify name of output file. Otherwise outfile is same
               name as input pathname but with different extension
-Dmacro=defn - Specify a preprocessor macro. All idl files are first
               processed by the preprocessor. Additionally setting the
               special macro 'package' will insert a 'package xyz;'
               statement at the top of the generated Java stub.
<filename>   - The pathname of the input IDL file.
```

The good thing ist that it claims to only support "java" format, but there is the previously undocumented "**jcifs**" format, which works nicely. So stick to that.

The other important option was "-o". This controls the name of the output file. So if you set that option to "myCoolFile", then midlc will generate myCoolFile.java".

Inside the midlc package there is a directory "idl" which contains several IDL files for different Windows services. The one I was interested in was "idl/svcctl.idl", so I used that one to generate the initial stub:

**Building midlc**

```
cdutz@ubuntu:~/Test/midlc-0.6.1$ ./midlc -t jcifs -o svcctl idl/svcctl.idl
```

The resulting "svcctl.java" was now the starting point of my JCIFS-based SVCCTL client.

## Making DCE/RPC calls from Java using JCIFS

So if you generated the code as described in the previous chapter and you use this unmodified, all you will get are "DCERPC_FAULT_PROTO_ERROR" errors. As I had absolutely no clue on how to use the Stub classes and perform DCE/RPC requests, these Errors were driving me nuts. Expecially when comparing my WireShark dumps with those of PSExec. But when I started comparing them to my JLan POC I was able to reverse enginer what to do in JCIFS. I did this by looing at what packets were sent by JLan and then to search through the JCIFS source to find something that looked similar and somehow figgured out how to use them.

One thing I found relatively anoying was the wide usage of packet-level accessability. So I had to implement the code for accessing SVCCTL in a class in the "jcifs.smb" package to be able to access some methods of the SmbTransport class.

After setting up the initial handshake stuff identical to the way JLan did it, I wanted to actually perfom my first SVCCTL call. The result was my ever-so-beloved "DCERPC_FAULT_PROTO_ERROR". But this time comparing the bytes generated by JLan and JCIFS showed me that the packet-type "ptype" being sent was set to "0xff" (-1) instead of "0" (Request). So I modified the generated classes so manually set "ptype" to "0" and was actually able to perform my calls without any problems 🙂

So now comes the code for starting the "Remoteregistry" service on a remote Windows system using JCIFS:

```
package jcifs.smb;
import de.cware.utils.libPsExec.msrpc.svcctl;
import jcifs.UniAddress;
import jcifs.dcerpc.DcerpcBinding;
```

```
import jcifs.dcerpc.DcerpcHandle;
import jcifs.dcerpc.rpc;
import jcifs.netbios.NbtAddress;
import jcifs.netbios.NbtException;
import jcifs.netbios.NbtSocket;
/**
 * Created with IntelliJ IDEA.
 * User: cdutz
 * Date: 08.03.12
 * Time: 10:32
 */
public class Test {
    // Without registering our class at DcerpcBinding we are not able to
use the pipe "\pipe\svcctl".
    static {
        DcerpcBinding.addInterface("svcctl", svcctl.getSyntax());
    }
    public static void main(String[] args) throws Exception {
        // Create a NbtSocket to the remote machine.
        // It seems the socket itself isn't needed at all, it's just
        // for testing if the nbtAddress was generated correctly.
        //
        // The structure of this code was strongly inspired by JLan
        final String host = "192.168.178.35";
        NbtAddress nbtAddress = NbtAddress.getByName(host);
        NbtSocket nbtSocket;
        try {
            nbtSocket = new NbtSocket(nbtAddress, 139);
        } catch (NbtException nbte) {
            if ((nbte.errorClass == 2) && (nbte.errorCode == 130)) {
                final NbtAddress[] nbtAddresses =
NbtAddress.getAllByAddress(host);
                nbtAddress = nbtAddresses[0];
                nbtSocket = new NbtSocket(nbtAddress, 139);
            } else {
                throw nbte;
            }
        }
        nbtSocket.close();
        // Don't know what this is needed for, but JLan did it.
        SmbTransport smbTransport = SmbTransport.getSmbTransport(new
UniAddress(nbtAddress), 445);
        smbTransport.connect();
        SmbSession smbSession =
smbTransport.getSmbSession(getNtlmAuthentication());
        SmbTree ipcShare = smbSession.getSmbTree("IPC$", null);
        ipcShare.treeConnect(null, null);

        try {
            // Open the SCManager on the remote machine and get a handle
            // for that open instance (scManagerHandle).
            rpc.policy_handle scManagerHandle = new rpc.policy_handle();
            svcctl.OpenSCManager openSCManagerRpc = new
```

```
svcctl.OpenSCManager("\\\\" + host, null,
                    (0x000F0000 | 0x0001 | 0x0002 | 0x0004 | 0x0008 |
0x0010 | 0x0020), scManagerHandle);
            DcerpcHandle handle = DcerpcHandle.getHandle(
                    "ncacn_np:" + host + "[\\pipe\\svcctl]",
getNtlmAuthentication());
            handle.sendrecv(openSCManagerRpc);
            if (openSCManagerRpc.retval != 0) {
                throw new SmbException(openSCManagerRpc.retval, true);
            }
            // Get a handle of the "Remoteregistry" service (svcHandle).
            rpc.policy_handle svcHandle = new rpc.policy_handle();
            svcctl.OpenService openServiceRpc = new svcctl.OpenService(
                    scManagerHandle, "Remoteregistry", 0x0F01FF,
svcHandle);
            handle.sendrecv(openServiceRpc);
            if(openServiceRpc.retval != 0) {
                throw new SmbException(openServiceRpc.retval, true);
            }
            // Use the service handle and tell the remote machine to start
the service.
            svcctl.StartService startServiceRpc = new
svcctl.StartService(svcHandle, 0, new String[0]);
            handle.sendrecv(startServiceRpc);
            if(startServiceRpc.retval != 0) {
                throw new SmbException(startServiceRpc.retval, true);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    private static NtlmPasswordAuthentication getNtlmAuthentication() {
        return new NtlmPasswordAuthentication("localhost", "testadmin",
```

```
"testpass");
    }
}
```

For the numeric codes used in the example, please have a look at Microsofts specification document at: http://msdn.microsoft.com/en-us/library/cc245832(v=prot.13).aspx

As I mentioned, I simply replicated the calls done by JLan ... after having a running POC I had another look at the code and noticed that there seemed to be no relation at all between the smbTransport and the DCE/RPC requests. I assumed this "simply has to be done". But now I simply commented parts out, that I assumed to be obsolete. The following code conatins all code needed to perform the actions I wanted:

```
package my.other.cool.package;
import de.cware.utils.libPsExec.msrpc.svcctl;
import jcifs.dcerpc.DcerpcBinding;
import jcifs.dcerpc.DcerpcHandle;
import jcifs.dcerpc.rpc;
/**
 * Created with IntelliJ IDEA.
 * User: cdutz
 * Date: 08.03.12
 * Time: 10:32
 */
public class Test {
    // Without registering our class at DcerpcBinding we are not able to
use the pipe "\pipe\svcctl".
    static {
        DcerpcBinding.addInterface("svcctl", svcctl.getSyntax());
    }
    public static void main(String[] args) throws Exception {
        final String host = "192.168.178.35";
        try {
            // Open the SCManager on the remote machine and get a handle
            // for that open instance (scManagerHandle).
            rpc.policy_handle scManagerHandle = new rpc.policy_handle();
            svcctl.OpenSCManager openSCManagerRpc = new
svcctl.OpenSCManager("\\\\" + host, null,
                    (0x000F0000 | 0x0001 | 0x0002 | 0x0004 | 0x0008 |
0x0010 | 0x0020), scManagerHandle);
            DcerpcHandle handle = DcerpcHandle.getHandle(
                    "ncacn_np:" + host + "[\\pipe\\svcctl]",
getNtlmAuthentication());
            handle.sendrecv(openSCManagerRpc);
            if (openSCManagerRpc.retval != 0) {
                throw new SmbException(openSCManagerRpc.retval, true);
            }

            // Get a handle of the "Remoteregistry" service (svcHandle).
            rpc.policy_handle svcHandle = new rpc.policy_handle();
            svcctl.OpenService openServiceRpc = new svcctl.OpenService(
                    scManagerHandle, "Remoteregistry", 0x0F01FF,
svcHandle);
            handle.sendrecv(openServiceRpc);
            if(openServiceRpc.retval != 0) {
```

```
                throw new SmbException(openServiceRpc.retval, true);
            }

            // Use the service handle and tell the remote machine to start
the service.
            // This will fail if the service is already running.
            svcctl.StartService startServiceRpc = new
svcctl.StartService(svcHandle, 0, new String[0]);
            handle.sendrecv(startServiceRpc);
            if(startServiceRpc.retval != 0) {
                throw new SmbException(startServiceRpc.retval, true);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    private static NtlmPasswordAuthentication getNtlmAuthentication() {
        return new NtlmPasswordAuthentication("localhost", "testadmin",
```

```
"testpass");
    }
}
```

As now all references to NbtAddress were eliminated, my code is now also free to run in any package I like 🙂

Here comes the code that was generated by midlc from the default svcctl.idl (So you don't have to generate it, you can simply use it)

```
package jcifs.derpc;
import com.sun.xml.internal.messaging.saaj.util.ByteInputStream;
import jcifs.dcerpc.*;
import jcifs.dcerpc.ndr.*;
import java.io.DataInputStream;
import java.util.List;
public class svcctl {
    public static String getSyntax() {
        return "367abb81-9844-35f1-ad32-98f038001003:2.0";
    }
    public static final int SC_MANAGER_ALL_ACCESS = 0x3f;
    public static class CloseServiceHandle extends DcerpcMessage {
        public int getOpnum() { return 0x00; }
        public int retval;
        public rpc.policy_handle handle;
        public CloseServiceHandle(rpc.policy_handle handle) {
            this.handle = handle;
            this.ptype = 0;
        }
        public void encode_in(NdrBuffer _dst) throws NdrException {
            handle.encode(_dst);
        }
        public void decode_out(NdrBuffer _src) throws NdrException {
            handle.decode(_src);
            retval = (int)_src.dec_ndr_long();
        }
    }
    public static class OpenSCManager extends DcerpcMessage {
        public int getOpnum() { return 0x0f; }
        public int retval;
        public String machine_name;
        public String database_name;
        public int access_mask;
        public rpc.policy_handle handle;
        public OpenSCManager(String machine_name,
                    String database_name,
                    int access_mask,
                    rpc.policy_handle handle) {
            this.machine_name = machine_name;
            this.database_name = database_name;
            this.access_mask = access_mask;
            this.handle = handle;
            this.ptype = 0;
        }
```

```
        public void encode_in(NdrBuffer _dst) throws NdrException {
            _dst.enc_ndr_referent(machine_name, 1);
            if (machine_name != null) {
                _dst.enc_ndr_string(machine_name);
            }
            _dst.enc_ndr_referent(database_name, 1);
            if (database_name != null) {
                _dst.enc_ndr_string(database_name);
            }
            _dst.enc_ndr_long(access_mask);
        }
        public void decode_out(NdrBuffer _src) throws NdrException {
            handle.decode(_src);
            retval = (int)_src.dec_ndr_long();
        }
    }
    public static class OpenService extends DcerpcMessage {
        public int getOpnum() { return 0x10; }
        public int retval;
        public rpc.policy_handle scmanager_handle;
        public String service_name;
        public int access_mask;
        public rpc.policy_handle handle;
        public OpenService(rpc.policy_handle scmanager_handle,
                    String service_name,
                    int access_mask,
                    rpc.policy_handle handle) {
            this.scmanager_handle = scmanager_handle;
            this.service_name = service_name;
            this.access_mask = access_mask;
            this.handle = handle;
            this.ptype = 0;
        }
        public void encode_in(NdrBuffer _dst) throws NdrException {
            scmanager_handle.encode(_dst);
            _dst.enc_ndr_string(service_name);
            _dst.enc_ndr_long(access_mask);
        }
        public void decode_out(NdrBuffer _src) throws NdrException {
            handle.decode(_src);
            retval = (int)_src.dec_ndr_long();
        }
    }
    public static class EnumServicesStatus extends DcerpcMessage {
        public int getOpnum() { return 0x0e; }
        public int retval;
        public rpc.policy_handle handle;
        public int type;
        public int state;
        public int buf_size;
        public byte[] service;
        public int bytes_needed;
        public int services_returned;
```

```
        public int resume_handle;
        public EnumServicesStatus(rpc.policy_handle handle,
                    int type,
                    int state,
                    int buf_size,
                    byte[] service,
                    int bytes_needed,
                    int services_returned,
                    int resume_handle) {
            this.handle = handle;
            this.type = type;
            this.state = state;
            this.buf_size = buf_size;
            this.service = service;
            this.bytes_needed = bytes_needed;
            this.services_returned = services_returned;
            this.resume_handle = resume_handle;
            this.ptype = 0;
        }
        public void encode_in(NdrBuffer _dst) throws NdrException {
            handle.encode(_dst);
            _dst.enc_ndr_long(type);
            _dst.enc_ndr_long(state);
            _dst.enc_ndr_long(buf_size);
            _dst.enc_ndr_long(resume_handle);
        }
        public void decode_out(NdrBuffer _src) throws NdrException {
            int _services = _src.dec_ndr_long();
            int _servicei = _src.index;
            _src.advance(1 * _services);
            if (service == null) {
                if (_services < 0 || _services > 0xFFFF) throw new
NdrException( NdrException.INVALID_CONFORMANCE );
                service = new byte[_services];
            }
            _src = _src.derive(_servicei);
            for (int _i = 0; _i < _services; _i++) {
                service[_i] = (byte)_src.dec_ndr_small();
            }
            bytes_needed = (int)_src.dec_ndr_long();
            services_returned = (int)_src.dec_ndr_long();
            resume_handle = (int)_src.dec_ndr_long();
            retval = (int)_src.dec_ndr_long();
        }
    }
    public static class StartService extends DcerpcMessage {
        public int getOpnum() { return 0x13; }
        public int retval;
        public rpc.policy_handle handle;
        public int num_service_args;
        public String[] service_arg_vectors;
        public StartService(rpc.policy_handle handle, int num_service_args,
String[] service_arg_vectors) {
```

```
            this.handle = handle;
            this.num_service_args = num_service_args;
            this.service_arg_vectors = service_arg_vectors;
            this.ptype = 0;
        }
        public void encode_in(NdrBuffer _dst) throws NdrException {
            handle.encode(_dst);
            _dst.enc_ndr_long(num_service_args);
            _dst.enc_ndr_referent(service_arg_vectors, 1);
            if (service_arg_vectors != null) {
                int _service_arg_vectorss = num_service_args;
                _dst.enc_ndr_long(_service_arg_vectorss);
                for (int _i = 0; _i < _service_arg_vectorss; _i++) {
                    _dst.enc_ndr_referent(service_arg_vectors[_i], 1);
                }
                for (int _i = 0; _i < _service_arg_vectorss; _i++) {
                    if (service_arg_vectors[_i] != null) {
                        _dst.enc_ndr_string(service_arg_vectors[_i]);
                    }
                }
            }
        }
        public void decode_out(NdrBuffer _src) throws NdrException {
            retval = (int)_src.dec_ndr_long();
        }
    }
    public static class service_status extends NdrObject {
        public int service_type;
        public int current_state;
        public int controls_accepted;
        public int win32_exit_code;
        public int service_specific_exit_code;
        public int check_point;
        public int wait_hint;
        public void encode(NdrBuffer _dst) throws NdrException {
            _dst.align(4);
            _dst.enc_ndr_long(service_type);
            _dst.enc_ndr_long(current_state);
            _dst.enc_ndr_long(controls_accepted);
            _dst.enc_ndr_long(win32_exit_code);
            _dst.enc_ndr_long(service_specific_exit_code);
            _dst.enc_ndr_long(check_point);
            _dst.enc_ndr_long(wait_hint);
        }
        public void decode(NdrBuffer _src) throws NdrException {
            _src.align(4);
            service_type = (int)_src.dec_ndr_long();
            current_state = (int)_src.dec_ndr_long();
            controls_accepted = (int)_src.dec_ndr_long();
            win32_exit_code = (int)_src.dec_ndr_long();
            service_specific_exit_code = (int)_src.dec_ndr_long();
            check_point = (int)_src.dec_ndr_long();
            wait_hint = (int)_src.dec_ndr_long();
```

```
        }
    }
    public static final int SERVICE_CONTROL_STOP = 1;
    public static final int SERVICE_CONTROL_PAUSE = 2;
    public static final int SERVICE_CONTROL_CONTINUE = 3;
    public static final int SERVICE_CONTROL_INTERROGATE = 4;
    public static class ControlService extends DcerpcMessage {
        public int getOpnum() { return 0x01; }
        public int retval;
        public rpc.policy_handle handle;
        public int control;
        public service_status status;
        public ControlService(rpc.policy_handle handle, int control,
service_status status) {
            this.handle = handle;
            this.control = control;
            this.status = status;
            this.ptype = 0;
        }
        public void encode_in(NdrBuffer _dst) throws NdrException {
            handle.encode(_dst);
            _dst.enc_ndr_long(control);
        }
        public void decode_out(NdrBuffer _src) throws NdrException {
            status.decode(_src);
            retval = (int)_src.dec_ndr_long();
```

```
            }
        }
    }
```

The only difference to the original was setting the "ptype" property to "0" in the constructor of every call.

# Extending the default svcctl.idl

The Samba guys could only reverse engineer service calls that were actually used by official tools. I found the Book: "DCE/RPC over SMB - Samba and Windows NT Domain Internals" from Luke Kenneth Casson Leighton (Yes ... that's only one guy 😊) to describe this process of reverse-engineering the SMB protocol very nicely. It seems that there were no tools available that made use of the ability to remotely create or delete services therefore they could never get network captures of this and therefore never integrate these calls into their IDL.

But now we have the official IDL from Microsoft and we can use this to complete the idl-definition and use this to complete the Stub for the SVCCTL DCE/RPC stub.

I did the extension step by step by extending it with one call at a time. As soon as the call I was woring on worked fine, I took the next one. It turned out to be pretty simple.

I will now describe how I did the extension for one particular Command and will post my finished IDL as well as the generated svcctl.java after that.

## Adding "CreateService"

In order to add a new command, I needed to find out the operation number for that call. I used this link to find that out. In case of the CreateService call this was 0x0c (Please note that there are Commands ending with "W" and ones ending with "A". As far as I understood the ones with "W" are the synchrounus ones and the ones with "A" are the asynchronous ones. I focussed on the synchronous "W" ones).

Next step was to get the corresponding IDL fragment from the original Microsoft IDL found here.

**Original IDL fragment**

```
DWORD
RCreateServiceW(
    [in] SC_RPC_HANDLE  hSCManager,
    [in,string,range(0, SC_MAX_NAME_LENGTH)]
                wchar_t *  lpServiceName,
    [in,string,unique,range(0, SC_MAX_NAME_LENGTH)]
                wchar_t *  lpDisplayName,
    [in] DWORD   dwDesiredAccess,
    [in] DWORD   dwServiceType,
    [in] DWORD   dwStartType,
    [in] DWORD   dwErrorControl,
    [in,string, range(0, SC_MAX_PATH_LENGTH)]
                wchar_t *  lpBinaryPathName,
    [in,string,unique,range(0, SC_MAX_NAME_LENGTH)]
                wchar_t *  lpLoadOrderGroup,
    [in,out,unique] LPDWORD  lpdwTagId,
    [in,unique,size_is(dwDependSize)] LPBYTE  lpDependencies,
    [in, range (0, SC_MAX_DEPEND_SIZE)] DWORD  dwDependSize,
    [in,string,unique,range(0, SC_MAX_ACCOUNT_NAME_LENGTH)]
                wchar_t *  lpServiceStartName,
    [in,unique,size_is(dwPwSize)] LPBYTE  lpPassword,
    [in, range(0, SC_MAX_PWD_SIZE)] DWORD   dwPwSize,
    [out] LPSC_RPC_HANDLE lpServiceHandle
    );
```

Comparing this to commands that were allready implemented I saw that I have to remove the "range" stuff first. Which type in this IDL I had to replace with which type of the midlc IDL was pretty simply by consequently looking up the types. The resulting IDL fragment for midlc was this:

**Transformaed MIDLC IDL fragment**

```
[op(0x0c)]
    int CreateServiceW([in] policy_handle *scmanager_handle,
            [in,string] wchar_t *service_name,
            [in,string,unique] wchar_t *display_name,
            [in] uint32_t access_mask,
            [in] uint32_t service_type,
            [in] uint32_t start_type,
            [in] uint32_t error_control,
            [in,string] wchar_t *binary_path_name,
            [in,string,unique] wchar_t *load_order_group,
            [in,out,unique] uint32_t *lpdwTagId,
            [in,unique,size_is(dependency_size)] uint8_t *lpDependencies,
            [in] uint32_t dependency_size,
            [in,string,unique] wchar_t *lpServiceStartName,
            [in,unique,size_is(password_size)] uint8_t *password,
            [in] uint32_t password_size,
            [out] policy_handle *service_handle);
```

Midlc generated the following code from this:

**Generated command stub**

```
public static class CreateServiceW extends DcerpcMessage {
        public int getOpnum() { return 0x0c; }
        public int retval;
        public rpc.policy_handle scmanager_handle;
        public String service_name;
        public String display_name;
        public int access_mask;
        public int service_type;
        public int start_type;
        public int error_control;
        public String binary_path_name;
        public String load_order_group;
        public NdrLong lpdwTagId;
        public byte[] lpDependencies;
        public int dependency_size;
        public String lpServiceStartName;
        public byte[] password;
        public int password_size;
        public rpc.policy_handle service_handle;
        public CreateServiceW(rpc.policy_handle scmanager_handle,
                    String service_name,
                    String display_name,
                    int access_mask,
                    int service_type,
                    int start_type,
                    int error_control,
                    String binary_path_name,
                    String load_order_group,
                    NdrLong lpdwTagId,
                    byte[] lpDependencies,
                    int dependency_size,
                    String lpServiceStartName,
                    byte[] password,
                    int password_size,
                    rpc.policy_handle service_handle) {
            this.scmanager_handle = scmanager_handle;
            this.service_name = service_name;
            this.display_name = display_name;
            this.access_mask = access_mask;
            this.service_type = service_type;
            this.start_type = start_type;
            this.error_control = error_control;
            this.binary_path_name = binary_path_name;
            this.load_order_group = load_order_group;
            this.lpdwTagId = lpdwTagId;
            this.lpDependencies = lpDependencies;
            this.dependency_size = dependency_size;
            this.lpServiceStartName = lpServiceStartName;
```

```
        this.password = password;
        this.password_size = password_size;
        this.service_handle = service_handle;
    }
    public void encode_in(NdrBuffer _dst) throws NdrException {
        scmanager_handle.encode(_dst);
        _dst.enc_ndr_string(service_name);
        _dst.enc_ndr_referent(display_name, 1);
        if (display_name != null) {
            _dst.enc_ndr_string(display_name);
        }
        _dst.enc_ndr_long(access_mask);
        _dst.enc_ndr_long(service_type);
        _dst.enc_ndr_long(start_type);
        _dst.enc_ndr_long(error_control);
        _dst.enc_ndr_string(binary_path_name);
        _dst.enc_ndr_referent(load_order_group, 1);
        if (load_order_group != null) {
            _dst.enc_ndr_string(load_order_group);
        }
        _dst.enc_ndr_referent(lpdwTagId, 1);
        if (lpdwTagId != null) {
            lpdwTagId.encode(_dst);
        }
        _dst.enc_ndr_referent(lpDependencies, 1);
        if (lpDependencies != null) {
            int _lpDependenciess = dependency_size;
            _dst.enc_ndr_long(_lpDependenciess);
            int _lpDependenciesi = _dst.index;
            _dst.advance(1 * _lpDependenciess);
            _dst = _dst.derive(_lpDependenciesi);
            for (int _i = 0; _i < _lpDependenciess; _i++) {
                _dst.enc_ndr_small(lpDependencies[_i]);
            }
        }
        _dst.enc_ndr_long(dependency_size);
        _dst.enc_ndr_referent(lpServiceStartName, 1);
        if (lpServiceStartName != null) {
            _dst.enc_ndr_string(lpServiceStartName);
        }
        _dst.enc_ndr_referent(password, 1);
        if (password != null) {
            int _passwords = password_size;
            _dst.enc_ndr_long(_passwords);
            int _passwordi = _dst.index;
            _dst.advance(1 * _passwords);
            _dst = _dst.derive(_passwordi);
            for (int _i = 0; _i < _passwords; _i++) {
                _dst.enc_ndr_small(password[_i]);
            }
        }
        _dst.enc_ndr_long(password_size);
    }
```

```
public void decode_out(NdrBuffer _src) throws NdrException {
    int _lpdwTagIdp = _src.dec_ndr_long();
    if (_lpdwTagIdp != 0) {
        lpdwTagId.decode(_src);
    }
    service_handle.decode(_src);
    retval = (int)_src.dec_ndr_long();
```

```
        }
    }
```

The problem was, that all generated classes were missing the initialization of the "ptype" property, so I had to extend the Constructor manually to this (Only the last assignment had to be changed):

**Generated command stub**

```
public CreateServiceW(rpc.policy_handle scmanager_handle,
                      String service_name,
                      String display_name,
                      int access_mask,
                      int service_type,
                      int start_type,
                      int error_control,
                      String binary_path_name,
                      String load_order_group,
                      NdrLong lpdwTagId,
                      byte[] lpDependencies,
                      int dependency_size,
                      String lpServiceStartName,
                      byte[] password,
                      int password_size,
                      rpc.policy_handle service_handle) {
            this.scmanager_handle = scmanager_handle;
            this.service_name = service_name;
            this.display_name = display_name;
            this.access_mask = access_mask;
            this.service_type = service_type;
            this.start_type = start_type;
            this.error_control = error_control;
            this.binary_path_name = binary_path_name;
            this.load_order_group = load_order_group;
            this.lpdwTagId = lpdwTagId;
            this.lpDependencies = lpDependencies;
            this.dependency_size = dependency_size;
            this.lpServiceStartName = lpServiceStartName;
            this.password = password;
            this.password_size = password_size;
            this.service_handle = service_handle;
            this.ptype = 0;
        }
```

## How to use the commands

In order to use the SVCCTL interface, I have to tell JCIFS about it. Without this, all I will get will be "Bad endpoint: \pipe\svcctl" errors. This is done by adding a static initializer bock in your code:

**Register the SVCCTL interface**

```
// Without registering our class at DcerpcBinding we are not able to use
the pipe "\pipe\svcctl".
    static {
        DcerpcBinding.addInterface("svcctl", svcctl.getSyntax());
    }
```

After this you will be able to get a DCE/RPC handle, which will be used for all interactions with the remote system:

**Get a handle to the SVCCTL interface**

```
DcerpcHandle handle = DcerpcHandle.getHandle(
                    "ncacn_np:" + host + "[\\pipe\\svcctl]",
getNtlmAuthentication());



...
Somewhere else in the code
...

    private static NtlmPasswordAuthentication getNtlmAuthentication() {
        return new NtlmPasswordAuthentication("domain", "username",
"password");
    }
```

In order to be able to perform tasks on the remote registry, you have to open the service manager and get a handle to that:

**Get a handle to the service manager**

```
... initialize the svcctl handle as described in the previous code block
...

            // Open the SCManager on the remote machine and get a handle
            // for that open instance (scManagerHandle).
            rpc.policy_handle scManagerHandle = new rpc.policy_handle();
            svcctl.OpenSCManager openSCManagerRpc = new
svcctl.OpenSCManager("\\\\" + host, null,
                    (0x000F0000 | 0x0001 | 0x0002 | 0x0004 | 0x0008 |
0x0010 | 0x0020), scManagerHandle);
            handle.sendrecv(openSCManagerRpc);
            if (openSCManagerRpc.retval != 0) {
                throw new SmbException(openSCManagerRpc.retval, true);
            }
```

If the return value was 0 then scManagerHandle will contain a valid handle for the service manager.

You can now use this handle to perform tasks on the remote service registry ... for example get a handle for a service.

**Get a handle for the "Remoteregistry" service**

```
rpc.policy_handle svcHandle = new rpc.policy_handle();
            svcctl.OpenService openServiceRpc = new
svcctl.OpenService(scManagerHandle,
                    "Remoteregistry", svcctl.SC_MANAGER_ALL_ACCESS,
svcHandle);
            handle.sendrecv(openServiceRpc);


            // If the service didn't exist...
            if(openServiceRpc.retval == 1060) {
            }
            // The service existed ... svcHandle will now contain a handle
to the requested service.
            else if(openServiceRpc.retval == 0) {

            }
     // Something else went wrong.
     else {
                throw new SmbException(openServiceRpc.retval, true);
            }
```

So if we were able to get a handle to the "Remoteregistry" service, we can query it's state:

**Query the state of the "Remoteregistry" service**

```
// Query the state of the service.
            svcctl.service_status status = new svcctl.service_status();
            svcctl.QueryServiceStatus queryServiceStatusRpc = new
svcctl.QueryServiceStatus(
                    svcHandle, status);
            handle.sendrecv(queryServiceStatusRpc);
     // Something went wrong.
            if(queryServiceStatusRpc.retval != 0) {
                throw new SmbException(queryServiceStatusRpc.retval, true);
            }
            // The service is currently running ...
            if(status.current_state != svcctl.SC_STATE_SERVICE_RUNNING) {

            }
     // The service is currently not running (could be a wide variaty of
states it is in)
            else {

            }
```

So I hope you get an idea of how these calls are done.

Get the complete IDL from here.

Get the complete svcctl.java class here.

Get the complete example code here.