CHAPTER 19

# Finding Vulnerabilities in Source Code

So far, the attack techniques we have described have all involved interacting with a live running application and have largely consisted of submitting crafted input to the application and monitoring its responses. This chapter examines an entirely different approach to finding vulnerabilities — reviewing the application's source code.

In various situations it may be possible to perform a source code audit to help attack a target web application:

- Some applications are open source, or use open source components, enabling you to download their code from the relevant repository and scour it for vulnerabilities.
- If you are performing a penetration test in a consultancy context, the application owner may grant you access to his or her source code to maximize the effectiveness of your audit.
- You may discover a file disclosure vulnerability within an application that enables you to download its source code (either partially or in its entirety).
- Most applications use some client-side code such as JavaScript, which is accessible without requiring any privileged access.

It is often believed that to carry out a code review, you must be an experienced programmer and have detailed knowledge of the language being used. However, this need not be the case. Many higher-level languages can be read

and understood by someone with limited programming experience. Also, many types of vulnerabilities manifest themselves in the same way across all the languages commonly used for web applications. The majority of code reviews can be carried out using a standard methodology. You can use a cheat sheet to help understand the relevant syntax and APIs that are specific to the language and environment you are dealing with. This chapter describes the core methodology you need to follow and provides cheat sheets for some of the languages you are likely to encounter.

# Approaches to Code Review

You can take a variety of approaches to carrying out a code review to help maximize your effectiveness in discovering security flaws within the time available. Furthermore, you can often integrate your code review with other test approaches to leverage the inherent strengths of each.

## Black-Box Versus White-Box Testing

The attack methodology described in previous chapters is often described as a *black-box* approach to testing. This involves attacking the application from the outside and monitoring its inputs and outputs, with no prior knowledge of its inner workings. In contrast, a *white-box* approach involves looking inside the application's internals, with full access to design documentation, source code, and other materials.

Performing a white-box code review can be a highly effective way to discover vulnerabilities within an application. With access to source code, it is often possible to quickly locate problems that would be extremely difficult or time-consuming to detect using only black-box techniques. For example, a backdoor password that grants access to any user account may be easy to identify by reading the code but nearly impossible to detect using a password-guessing attack.

However, code review usually is not an effective substitute for black-box testing. Of course, in one sense, all the vulnerabilities in an application are "in the source code," so it must in principle be possible to locate all those vulnerabilities via code review. However, many vulnerabilities can be discovered more quickly and efficiently using black-box methods. Using the automated fuzzing techniques described in Chapter 14, it is possible to send an application hundreds of test cases per minute, which propagate through all relevant code paths and return a response immediately. By sending triggers for common vulnerabilities to every field in every form, it is often possible to find within minutes a mass of problems that would take days to uncover via code review. Furthermore, many enterprise-class applications have a complex structure with numerous

layers of processing of user-supplied input. Different controls and checks are implemented at each layer, and what appears to be a clear vulnerability in one piece of source code may be fully mitigated by code elsewhere.

In most situations, black-box and white-box techniques can complement and enhance each other. Often, having found a prima facie vulnerability through code review, the easiest and most effective way to establish whether it is real is to test for it on the running application. Conversely, having identified some anomalous behavior on a running application, often the easiest way to investigate its root cause is to review the relevant source code. If feasible, therefore, you should aim to combine a suitable mix of black- and white-box techniques. Allow the time and effort you devote to each to be guided by the application's behavior during hands-on testing, and the size and complexity of the codebase.

## Code Review Methodology

Any reasonably functional application is likely to contain many thousands of lines of source code, and in most cases the time available for you to review it is likely to be restricted, perhaps to only a few days. A key objective of effective code review, therefore, is to identify as many security vulnerabilities as possible, given a certain amount of time and effort. To achieve this, you must take a structured approach, using various techniques to ensure that the "low-hanging fruit" within the codebase is quickly identified, leaving time to look for issues that are more subtle and harder to detect.

In the authors' experience, a threefold approach to auditing a web application codebase is effective in identifying vulnerabilities quickly and easily. This methodology comprises the following elements:

1. Tracing user-controllable data from its entry points into the application, and reviewing the code responsible for processing it.
2. Searching the codebase for signatures that may indicate the presence of common vulnerabilities, and reviewing these instances to determine whether an actual vulnerability exists.
3. Performing a line-by-line review of inherently risky code to understand the application's logic and find any problems that may exist within it. Functional components that may be selected for this close review include the key security mechanisms within the application (authentication, session management, access control, and any application-wide input validation), interfaces to external components, and any instances where native code is used (typically C/C++).

We will begin by looking at the ways in which various common web application vulnerabilities appear at the level of source code and how these can be

most easily identified when performing a review. This will provide a way to search the codebase for signatures of vulnerabilities (step 2) and closely review risky areas of code (step 3).

We will then look at some of the most popular web development languages to identify the ways in which an application acquires user-submitted data (through request parameters, cookies, and so on). We will also see how an application interacts with the user session, the potentially dangerous APIs that exist within each language, and the ways in which each language's configuration and environment can affect the application's security. This will provide a way to trace user-controllable data from its entry point to the application (step 1) as well as provide some per-language context to assist with the other methodology steps. Finally, we will discuss some tools that are useful when performing code review.

**NOTE When carrying out a code audit, you should always bear in mind that applications may extend library classes and interfaces, may implement wrappers to standard API calls, and may implement custom mechanisms for security-critical tasks such as storing per-session information. Before launching into the detail of a code review, you should establish the extent of such customization and tailor your approach to the review accordingly.**

# Signatures of Common Vulnerabilities

Many types of web application vulnerabilities have a fairly consistent signature within the codebase. This means that you can normally identify a good portion of an application's vulnerabilities by quickly scanning and searching the codebase. The examples presented here appear in various languages, but in most cases the signature is language-neutral. What matters is the programming technique being employed, more than the actual APIs and syntax.

## Cross-Site Scripting

In the most obvious examples of XSS, parts of the HTML returned to the user are explicitly constructed from user-controllable data. Here, the target of an `HREF` link is constructed using strings taken directly from the query string in the request:

```
String link = "<a href=" + HttpUtility.UrlDecode(Request.QueryString
["refURL"]) + "&SiteID=" + SiteId + "&Path=" + HttpUtility.UrlEncode
(Request.QueryString["Path"]) + "</a>";
objCell.InnerHtml = link;
```

The usual remedy for cross-site scripting, which is to HTML-encode potentially malicious content, cannot be subsequently applied to the resulting concatenated

string, because it already contains valid HTML markup. Any attempt to sanitize the data would break the application by encoding the HTML that the application itself has specified. Hence, the example is certainly vulnerable unless filters are in place elsewhere that block requests containing XSS exploits within the query string. This filter-based approach to stopping XSS attacks is often flawed. If it is present, you should closely review it to identify any ways to work around it (see Chapter 12).

In more subtle cases, user-controllable data is used to set the value of a variable that is later used to build the response to the user. Here, the class member variable `m_pageTitle` is set to a value taken from the request query string. It will presumably be used later to create the `<title>` element within the returned HTML page:

```
private void setPageTitle(HttpServletRequest request) throws
    ServletException
{
    String requestType = request.getParameter("type");

    if ("3".equals(requestType) && null!=request.getParameter("title"))
        m_pageTitle = request.getParameter("title");

    else m_pageTitle = "Online banking application";
}
```

When you encounter code like this, you should closely review the processing subsequently performed on the `m_pageTitle` variable. You should see how it is incorporated into the returned page to determine whether the data is suitably encoded to prevent XSS attacks.

The preceding example clearly demonstrates the value of a code review in finding some vulnerabilities. The XSS flaw can be triggered only if a different parameter (`type`) has a specific value (`3`). Standard fuzz testing and vulnerability scanning of the relevant request may well fail to detect the vulnerability.

## SQL Injection

SQL injection vulnerabilities most commonly arise when various hard-coded strings are concatenated with user-controllable data to form a SQL query, which is then executed within the database. Here, a query is constructed using data taken directly from the request query string:

```
StringBuilder SqlQuery = newStringBuilder("SELECT name, accno FROM
TblCustomers WHERE " + SqlWhere);

if(Request.QueryString["CID"] != null &&
    Request.QueryString["PageId"] == "2")
{
    SqlQuery.Append(" AND CustomerID = ");
```

```
    SqlQuery.Append(Request.QueryString["CID"].ToString());
}
...
```

A simple way to quickly identify this kind of low-hanging fruit within the codebase is to search the source for the hard-coded substrings, which are often used to construct queries out of user-supplied input. These substrings usually consist of snippets of SQL and are quoted in the source, so it can be profitable to search for appropriate patterns composed of quotation marks, SQL keywords, and spaces. For example:

```
"SELECT
"INSERT
"DELETE
" AND
" OR
" WHERE
" ORDER BY
```

In each case, you should verify whether these strings are being concatenated with user-controllable data in a way that introduces SQL injection vulnerabilities. Because SQL keywords are processed in a case-insensitive manner, the searches for these terms should also be case-insensitive. Note that a space may be appended to each of these search terms to reduce the incidence of false positives in the results.

## Path Traversal

The usual signature for path traversal vulnerabilities involves user-controllable input being passed to a filesystem API without any validation of the input or verification that an appropriate file has been selected. In the most common case, user data is appended to a hard-coded or system-specified directory path, enabling an attacker to use dot-dot-slash sequences to step up the directory tree to access files in other directories. For example:

```
public byte[] GetAttachment(HttpRequest Request)
{
    FileStream fsAttachment = new FileStream(SpreadsheetPath +
        HttpUtility.UrlDecode(Request.QueryString["AttachName"]),
        FileMode.Open, FileAccess.Read, FileShare.Read);

    byte[] bAttachment = new byte[fsAttachment.Length];
    fsAttachment.Read(FileContent, 0,
        Convert.ToInt32(fsAttachment.Length,
        CultureInfo.CurrentCulture));

    fsAttachment.Close();
```

```
    return bAttachment;
}
```

You should closely review any application functionality that enables users to upload or download files. You need to understand how filesystem APIs are being invoked in response to user-supplied data and determine whether crafted input can be used to access files in an unintended location. Often, you can quickly identify relevant functionality by searching the codebase for the names of any query string parameters that relate to filenames (`AttachName` in the current example). You also can search for all file APIs in the relevant language and review the parameters passed to them. (See later sections for listings of the relevant APIs in common languages.)

## Arbitrary Redirection

Various phishing vectors such as arbitrary redirects are often easy to spot through signatures in the source code. In this example, user-supplied data from the query string is used to construct a URL to which the user is redirected:

```
private void handleCancel()
{
    httpResponse.Redirect(HttpUtility.UrlDecode(Request.QueryString[
        "refURL"]) + "&SiteCode=" +
        Request.QueryString["SiteCode"].ToString() +
        "&UserId=" + Request.QueryString["UserId"].ToString());
}
```

Often, you can find arbitrary redirects by inspecting client-side code, which of course does not require any special access to the application's internals. Here, JavaScript is used to extract a parameter from the URL query string and ultimately redirect to it:

```
url = document.URL;

index = url.indexOf('?redir=');
target = unescape(url.substring(index + 7, url.length));
target = unescape(target);

if ((index = target.indexOf('//')) > 0) {
    target = target.substring (index + 2, target.length);
    index = target.indexOf('/');
    target = target.substring(index, target.length);
}
target = unescape(target);
document.location = target;
```

As you can see, the author of this script knew the script was a potential target for redirection attacks to an absolute URL on an external domain. The script

checks whether the redirection URL contains a double slash (as in `http://`). If it does, the script skips past the double slash to the first single slash, thereby converting it into a relative URL. However, the script then makes a final call to the `unescape()` function, which unpacks any URL-encoded characters. Performing canonicalization after validation often leads to a vulnerability (see Chapter 2). In this instance an attacker can cause a redirect to an arbitrary absolute URL with the following query string:

```
?redir=http:%25252f%25252fwahh-attacker.com
```

## OS Command Injection

Code that interfaces with external systems often contains signatures indicating code injection flaws. In the following example, the `message` and `address` parameters have been extracted from user-controllable form data and are passed directly into a call to the UNIX `system` API:

```
void send_mail(const char *message, const char *addr)
{
    char sendMailCmd[4096];
    snprintf(sendMailCmd, 4096, "echo '%s' | sendmail %s", message, addr);
    system(sendMailCmd);
    return;
}
```

## Backdoor Passwords

Unless they have been deliberately concealed by a malicious programmer, backdoor passwords that have been used for testing or administrative purposes usually stand out when you review credential validation logic. For example:

```
private UserProfile validateUser(String username, String password)
{
    UserProfile up = getUserProfile(username);

    if (checkCredentials(up, password) ||
            "oculiomnium".equals(password))
        return up;

    return null;
}
```

Other items that may be easily identified in this way include unreferenced functions and hidden debug parameters.

## Native Software Bugs

You should closely review any native code used by the application for classic vulnerabilities that may be exploitable to execute arbitrary code.

### *Buffer Overflow Vulnerabilities*

These typically employ one of the unchecked APIs for buffer manipulation, of which there are many, including `strcpy`, `strcat`, `memcpy`, and `sprintf`, together with their wide-char and other variants. An easy way to identify low-hanging fruit within the codebase is to search for all uses of these APIs and verify whether the source buffer is user-controllable. You also should verify whether the code has explicitly ensured that the destination buffer is large enough to accommodate the data being copied into it (because the API itself does not do so).

Vulnerable calls to unsafe APIs are often easy to identify. In the following example, the user-controllable string `pszName` is copied into a fixed-size stack-based buffer without checking that the buffer is large enough to accommodate it:

```
BOOL CALLBACK CFiles::EnumNameProc(LPTSTR pszName)
{
    char strFileName[MAX_PATH];
    strcpy(strFileName, pszName);
    ...
}
```

Note that just because a safe alternative to an unchecked API is employed, this is no guarantee that a buffer overflow will not occur. Sometimes, due to a mistake or misunderstanding, a checked API is used in an unsafe manner, as in the following "fix" of the preceding vulnerability:

```
BOOL CALLBACK CFiles::EnumNameProc(LPTSTR pszName)
{
    char strFileName[MAX_PATH];
    strncpy(strFileName, pszName, strlen(pszName));
    ...
}
```

Therefore, a thorough code audit for buffer overflow vulnerabilities typically entails a close line-by-line review of the entire codebase, tracing every operation performed on user-controllable data.

### *Integer Vulnerabilities*

These come in many forms and can be extremely subtle, but some instances are easy to identify from signatures within the source code.

Comparisons between signed and unsigned integers often lead to problems. In the following "fix" to the previous vulnerability, a signed integer (`len`) is compared with an unsigned integer (`sizeof(strFileName)`). If the user can engineer a situation where `len` has a negative value, this comparison will succeed, and the unchecked `strcpy` will still occur:

```
BOOL CALLBACK CFiles::EnumNameProc(LPTSTR pszName, int len)
{
    char strFileName[MAX_PATH];

    if (len < sizeof(strFileName))
        strcpy(strFileName, pszName);
    ...
}
```

### *Format String Vulnerabilities*

Typically you can identify these quickly by looking for uses of the `printf` and `FormatMessage` families of functions where the format string parameter is not hard-coded but is user-controllable. The following call to `fprintf` is an example:

```
void logAuthenticationAttempt(char* username);
{
    char tmp[64];
    snprintf(tmp, 64, "login attempt for: %s\n", username);
    tmp[63] = 0;
    fprintf(g_logFile, tmp);
}
```

## Source Code Comments

Many software vulnerabilities are actually documented within source code comments. This often occurs because developers are aware that a particular operation is unsafe, and they record a reminder to fix the problem later, but they never get around to doing so. In other cases, testing has identified some anomalous behavior within the application that was commented within the code but never fully investigated. For example, the authors encountered the following within an application's production code:

```
char buf[200]; // I hope this is big enough
...
strcpy(buf, userinput);
```

Searching a large codebase for comments indicating common problems is frequently an effective source of low-hanging fruit. Here are some search terms that have proven useful:

- `bug`
- `problem`
- `bad`
- `hope`
- `todo`
- `fix`
- `overflow`
- `crash`
- `inject`
- `xss`
- `trust`

# The Java Platform

This section describes ways to acquire user-supplied input, ways to interact with the user's session, potentially dangerous APIs, and security-relevant configuration options on the Java platform.

## Identifying User-Supplied Data

Java applications acquire user-submitted input via the `javax.servlet.http.HttpServletRequest` interface, which extends the `javax.servlet.ServletRequest` interface. These two interfaces contain numerous APIs that web applications can use to access user-supplied data. The APIs listed in Table 19-1 can be used to obtain data from the user request.

**Table 19-1:** APIs Used to Acquire User-Supplied Data on the Java Platform

| API | DESCRIPTION |
|---|---|
| `getParameter`<br>`getParameterNames`<br>`getParameterValues`<br>`getParameterMap` | Parameters within the URL query string and the body of a `POST` request are stored as a map of `String` names to `String` values, which can be accessed using these APIs. |
| `getQueryString` | Returns the entire query string contained within the request and can be used as an alternative to the `getParameter` APIs. |
| `getHeader`<br>`getHeaders`<br>`getHeaderNames` | HTTP headers in the request are stored as a map of `String` names to `String` values and can be accessed using these APIs. |
| `getRequestURI`<br>`getRequestURL` | These APIs return the URL contained within the request, including the query string. |
| `getCookies` | Returns an array of `Cookie` objects, which contain details of the cookies received in the request, including their names and values. |
| `getRequestedSessionId` | Used as an alternative to `getCookies` in some cases; returns the session ID value submitted within the request. |
| `getInputStream`<br>`getReader` | These APIs return different representations of the raw request received from the client and therefore can be used to access any of the information obtained by all the other APIs. |
| `getMethod` | Returns the method used in the HTTP request. |
| `getProtocol` | Returns the protocol used in the HTTP request. |
| `getServerName` | Returns the value of the HTTP `Host` header. |
| `getRemoteUser`<br>`getUserPrincipal` | If the current user is authenticated, these APIs return details of the user, including his login name. If users can choose their own username during self-registration, this may be a means of introducing malicious input into the application's processing. |

## Session Interaction

Java Platform applications use the `javax.servlet.http.HttpSession` interface to store and retrieve information within the current session. Per-session storage is a map of string names to object values. The APIs listed in Table 19-2 are used to store and retrieve data within the session.

**Table 19-2:** APIs Used to Interact with the User's Session on the Java Platform

| API | DESCRIPTION |
|---|---|
| `setAttribute` | Used to store data within the current session. |
| `putValue` | |
| `getAttribute` | Used to query data stored within the current session. |
| `getValue` | |
| `getAttributeNames` | |
| `getValueNames` | |

## Potentially Dangerous APIs

This section describes some common Java APIs that can introduce security vulnerabilities if used in an unsafe manner.

### *File Access*

The main class used to access files and directories in Java is `java.io.File`. From a security perspective, the most interesting uses of this class are calls to its constructor, which may take a parent directory and filename, or simply a pathname.

Whichever form of the constructor is used, path traversal vulnerabilities may exist if user-controllable data is passed as the filename parameter without checking for dot-dot-slash sequences. For example, the following code opens a file in the root of the `C:\` drive on Windows:

```
String userinput = "..\\boot.ini";
File f = new File("C:\\temp", userinput);
```

The classes most commonly used for reading and writing file contents in Java are:

- `java.io.FileInputStream`
- `java.io.FileOutputStream`
- `java.io.FileReader`
- `java.io.FileWriter`

These classes take a `File` object in their constructors or may open a file themselves via a filename string, which may again introduce path traversal vulnerabilities if user-controllable data is passed as this parameter. For example:

```
String userinput = "..\\boot.ini";
FileInputStream fis = new FileInputStream("C:\\temp\\" + userinput);
```

### *Database Access*

The following are the APIs most commonly used for executing an arbitrary string as a SQL query:

- `java.sql.Connection.createStatement`
- `java.sql.Statement.execute`
- `java.sql.Statement.executeQuery`

If user-controllable input is part of the string being executed as a query, it is probably vulnerable to SQL injection. For example:

```
String username = "admin' or 1=1--";
String password = "foo";
Statement s = connection.createStatement();
s.executeQuery("SELECT * FROM users WHERE username = "' + username +
    "' AND password = "' + password + "'");
```

executes this unintended query:

```
SELECT * FROM users WHERE username = 'admin' or 1=1--' AND password = 'foo'
```

The following APIs are a more robust and secure alternative to the ones previously described. They allow an application to create a precompiled SQL statement and set the value of its parameter placeholders in a secure and type-safe way:

- `java.sql.Connection.prepareStatement`
- `java.sql.PreparedStatement.setString`
- `java.sql.PreparedStatement.setInt`
- `java.sql.PreparedStatement.setBoolean`
- `java.sql.PreparedStatement.setObject`
- `java.sql.PreparedStatement.execute`
- `java.sql.PreparedStatement.executeQuery`

and so on.

If used as intended, these are not vulnerable to SQL injection. For example:

```
String username = "admin' or 1=1--";
String password = "foo";
Statement s = connection.prepareStatement(
    "SELECT * FROM users WHERE username = ? AND password = ?");
s.setString(1, username);
s.setString(2, password);
s.executeQuery();
```

results in a query that is equivalent to the following:

```
SELECT * FROM users WHERE username = 'admin'' or 1=1--' AND
password = 'foo'
```

### Dynamic Code Execution

The Java language itself does not contain any mechanism for dynamic evaluation of Java source code, although some implementations (notably within database products) provide a facility to do this. If the application you are reviewing constructs any Java code on the fly, you should understand how this is done and determine whether any user-controllable data is being used in an unsafe way.

### OS Command Execution

The following APIs are the means of executing external operating system commands from within a Java application:

- `java.lang.runtime.Runtime.getRuntime`
- `java.lang.runtime.Runtime.exec`

If the user can fully control the string parameter passed to `exec`, the application is almost certainly vulnerable to arbitrary command execution. For example, the following causes the Windows `calc` program to run:

```
String userinput = "calc";
Runtime.getRuntime.exec(userinput);
```

However, if the user controls only part of the string passed to `exec`, the application may not be vulnerable. In the following example, the user-controllable data is passed as command-line arguments to the notepad process, causing it to attempt to load a document called `| calc`:

```
String userinput = "| calc";
Runtime.getRuntime.exec("notepad " + userinput);
```

The `exec` API itself does not interpret shell metacharacters such as `&` and `|`, so this attack fails.

Sometimes, controlling only part of the string passed to `exec` may still be sufficient for arbitrary command execution, as in the following subtly different example (note the missing space after `notepad`):

```
String userinput = "\\..\\system32\\calc";
Runtime.getRuntime().exec("notepad" + userinput);
```

Often, in this type of situation, the application is vulnerable to something other than code execution. For example, if an application executes the program `wget` with a user-controllable parameter as the target URL, an attacker may be able to pass dangerous command-line arguments to the `wget` process. For example, the attacker might cause `wget` to download a document and save it to an arbitrary location in the filesystem.

### *URL Redirection*

The following APIs can be used to issue an HTTP redirect in Java:

- `javax.servlet.http.HttpServletResponse.sendRedirect`
- `javax.servlet.http.HttpServletResponse.setStatus`
- `javax.servlet.http.HttpServletResponse.addHeader`

The usual means of causing a redirect response is via the `sendRedirect` method, which takes a string containing a relative or absolute URL. If the value of this string is user-controllable, the application is probably vulnerable to a phishing vector.

You should also be sure to review any uses of the `setStatus` and `addHeader` APIs. Given that a redirect simply involves a 3xx response containing an HTTP `Location` header, an application may implement redirects using these APIs.

### *Sockets*

The `java.net.Socket` class takes various forms of target host and port details in its constructors. If the parameters passed are user-controllable in any way, the application may be exploitable to cause network connections to arbitrary hosts, either on the Internet or on the private DMZ or internal network on which the application is hosted.

## Configuring the Java Environment

The `web.xml` file contains configuration settings for the Java Platform environment and controls how applications behave. If an application is using container-managed security, authentication and authorization are declared in `web.xml` against each resource or collection of resources to be secured, outside the application code. Table 19-3 shows configuration options that may be set in the `web.xml` file.

Servlets can enforce programmatic checks with `HttpServletRequest.isUserInRole` to access the same role information from within the servlet code. A

mapping entry `security-role-ref` links the built-in role check with the corresponding container role.

In addition to `web.xml`, different application servers may use secondary deployment files (for example, `weblogic.xml`) containing other security-relevant settings. You should include these when examining the environment's configuration.

**Table 19-3:** Security-Relevant Configuration Settings for the Java Environment

| SETTING | DESCRIPTION |
|---|---|
| `login-config` | Authentication details can be configured within the `login-config` element.<br><br>The two categories of authentication are `forms-based` (the page is specified by the `form-login-page` element) and `Basic Auth` or `Client-Cert`, specified within the `auth-method` element.<br><br>If forms-based authentication is used, the specified form must have the action defined as `j_security_check` and must submit the parameters `j_username` and `j_password`. Java applications recognize this as a login request. |
| `security-constraint` | If the `login-config` element is defined, resources can be restricted using the `security-constraint` element. This can be used to define the resources to be protected.<br><br>Within the `security-constraint` element, resource collections can be defined using the `url-pattern` element. For example:<br><br>`<url-pattern>/admin/*</url-pattern>`<br><br>These are accessible to roles and principals defined in the `role-name` and `principal-name` elements, respectively. |
| `session-config` | The session timeout (in minutes) can be configured within the `session-timeout` element. |
| `error-page` | The application's error handling is defined within the `error-page` element. HTTP error codes and Java exceptions can be handled on an individual basis through the `error-code` and `exception-type` elements. |
| `init-param` | Various initialization parameters are configured within the `init-param` element. These may include security-specific settings such as `listings`, which should be set to `false`, and `debug`, which should be set to `0`. |

# ASP.NET

This section describes methods of acquiring user-supplied input, ways of interacting with the user's session, potentially dangerous APIs, and security-relevant configuration options on the ASP.NET platform.

## Identifying User-Supplied Data

ASP.NET applications acquire user-submitted input via the `System.Web.HttpRequest` class. This class contains numerous properties and methods that web applications can use to access user-supplied data. The APIs listed in Table 19-4 can be used to obtain data from the user request.

**Table 19-4:** APIs Used to Acquire User-Supplied Data on the ASP.NET Platform

| API | DESCRIPTION |
|---|---|
| `Params` | Parameters within the URL query string, the body of a `POST` request, HTTP cookies, and miscellaneous server variables are stored as maps of string names to string values. This property returns a combined collection of all these parameter types. |
| `Item` | Returns the named item from within the `Params` collection. |
| `Form` | Returns a collection of the names and values of form variables submitted by the user. |
| `QueryString` | Returns a collection of the names and values of variables within the query string in the request. |
| `ServerVariables` | Returns a collection of the names and values of a large number of ASP server variables (akin to CGI variables). This includes the raw data of the request, query string, request method, HTTP `Host` header, and so on. |
| `Headers` | HTTP headers in the request are stored as a map of string names to string values and can be accessed using this property. |
| `Url`<br>`RawUrl` | Return details of the URL contained within the request, including the query string. |
| `UrlReferrer` | Returns information about the URL specified in the HTTP `Referer` header in the request. |

| API | DESCRIPTION |
|---|---|
| `Cookies` | Returns a collection of `Cookie` objects, which contain details of the cookies received in the request, including their names and values. |
| `Files` | Returns a collection of files uploaded by the user. |
| `InputStream`<br>`BinaryRead` | Return different representations of the raw request received from the client and therefore can be used to access any of the information obtained by all the other APIs. |
| `HttpMethod` | Returns the method used in the HTTP request. |
| `Browser`<br>`UserAgent` | Return details of the user's browser, as submitted in the HTTP `User-Agent` header. |
| `AcceptTypes` | Returns a string array of client-supported MIME types, as submitted in the HTTP `Accept` header. |
| `UserLanguages` | Returns a string array containing the languages accepted by the client, as submitted in the HTTP `Accept-Language` header. |

## Session Interaction

ASP.NET applications can interact with the user's session to store and retrieve information in various ways.

The `Session` property provides a simple way to store and retrieve information within the current session. It is accessed in the same way as any other indexed collection:

```
Session["MyName"] = txtMyName.Text;              // store user's name
lblWelcome.Text = "Welcome "+Session["MyName"]; // retrieve user's name
```

ASP.NET profiles work much like the `Session` property does, except that they are tied to the user's profile and therefore actually persist across different sessions belonging to the same user. Users are reidentified across sessions either through authentication or via a unique persistent cookie. Data is stored and retrieved in the user profile as follows:

```
Profile.MyName = txtMyName.Text;               // store user's name
lblWelcome.Text = "Welcome " + Profile.MyName; // retrieve user's name
```

The `System.Web.SessionState.HttpSessionState` class provides another way to store and retrieve information within the session. It stores information

as a mapping from string names to object values, which can be accessed using the APIs listed in Table 19-5.

**Table 19-5:** APIs Used to Interact with the User's Session on the ASP.NET Platform

| API | DESCRIPTION |
|---|---|
| `Add` | Adds a new item to the session collection. |
| `Item` | Gets or sets the value of a named item in the collection. |
| `Keys` | Return the names of all items in the collection. |
| `GetEnumerator` | |
| `CopyTo` | Copies the collection of values to an array. |

## Potentially Dangerous APIs

This section describes some common ASP.NET APIs that can introduce security vulnerabilities if used in an unsafe manner.

### *File Access*

`System.IO.File` is the main class used to access files in ASP.NET. All of its relevant methods are static, and it has no public constructor.

The 37 methods of this class all take a filename as a parameter. Path traversal vulnerabilities may exist in every instance where user-controllable data is passed in without checking for dot-dot-slash sequences. For example, the following code opens a file in the root of the `C:\` drive on Windows:

```
string userinput = "..\\boot.ini";
FileStream fs = File.Open("C:\\temp\\" + userinput,
    FileMode.OpenOrCreate);
```

The following classes are most commonly used to read and write file contents:

- `System.IO.FileStream`
- `System.IO.StreamReader`
- `System.IO.StreamWriter`

They have various constructors that take a file path as a parameter. These may introduce path traversal vulnerabilities if user-controllable data is passed. For example:

```
string userinput = "..\\foo.txt";
FileStream fs = new FileStream("F:\\tmp\\" + userinput,
    FileMode.OpenOrCreate);
```

### Database Access

Numerous APIs can be used for database access within ASP.NET. The following are the main classes that can be used to create and execute a SQL statement:

- `System.Data.SqlClient.SqlCommand`
- `System.Data.SqlClient.SqlDataAdapter`
- `System.Data.Oledb.OleDbCommand`
- `System.Data.Odbc.OdbcCommand`
- `System.Data.SqlServerCe.SqlCeCommand`

Each of these classes has a constructor that takes a string containing a SQL statement. Also, each has a `CommandText` property that can be used to get and set the current value of the SQL statement. When a command object has been suitably configured, it is executed via a call to one of the various `Execute` methods.

If user-controllable input is part of the string being executed as a query, the application is probably vulnerable to SQL injection. For example:

```
string username = "admin' or 1=1--";
string password = "foo";
OdbcCommand c = new OdbcCommand("SELECT * FROM users WHERE username = '"
    + username + "' AND password = '" + password + "'", connection);
c.ExecuteNonQuery();
```

executes this unintended query:

```
SELECT * FROM users WHERE username = 'admin' or 1=1--'
    AND password = 'foo'
```

Each of the classes listed supports prepared statements via their `Parameters` property, which allows an application to create a SQL statement containing parameter placeholders and set their values in a secure and type-safe way. If used as intended, this mechanism is not vulnerable to SQL injection. For example:

```
string username = "admin' or 1=1--";
string password = "foo";
OdbcCommand c = new OdbcCommand("SELECT * FROM users WHERE username =
    @username AND password = @password", connection);
c.Parameters.Add(new OdbcParameter("@username", OdbcType.Text).Value =
username);
c.Parameters.Add(new OdbcParameter("@password", OdbcType.Text).Value =
password);
c.ExecuteNonQuery();
```

results in a query that is equivalent to the following:

```
SELECT * FROM users WHERE username = 'admin'' or 1=1--'
    AND password = 'foo'
```

### Dynamic Code Execution

The VBScript function `Eval` takes a string argument containing a VBScript expression. The function evaluates this expression and returns the result. If user-controllable data is incorporated into the expression to be evaluated, it might be possible to execute arbitrary commands or modify the application's logic.

The functions `Execute` and `ExecuteGlobal` take a string containing ASP code, which they execute just as if the code appeared directly within the script itself. The colon delimiter can be used to batch multiple statements. If user-controllable data is passed into the `Execute` function, the application is probably vulnerable to arbitrary command execution.

### OS Command Execution

The following APIs can be used in various ways to launch an external process from within an ASP.NET application:

- `System.Diagnostics.Start.Process`
- `System.Diagnostics.Start.ProcessStartInfo`

A filename string can be passed to the static `Process.Start` method, or the `StartInfo` property of a `Process` object can be configured with a filename before calling `Start` on the object. If the user can fully control the filename string, the application is almost certainly vulnerable to arbitrary command execution. For example, the following causes the Windows `calc` program to run:

```
string userinput = "calc";
Process.Start(userinput);
```

If the user controls only part of the string passed to `Start`, the application may still be vulnerable. For example:

```
string userinput = "..\\..\\..\\Windows\\System32\\calc";
Process.Start("C:\\Program Files\\MyApp\\bin\\"  + userinput);
```

The API does not interpret shell metacharacters such as `&` and `|`, nor does it accept command-line arguments within the filename parameter. Therefore, this kind of attack is the only one likely to succeed when the user controls only a part of the filename parameter.

Command-line arguments to the launched process can be set using the `Arguments` property of the `ProcessStartInfo` class. If only the `Arguments` parameter is user-controllable, the application may still be vulnerable to something other than code execution. For example, if an application executes the program `wget` with a user-controllable parameter as the target URL, an attacker may be able to pass dangerous command-line parameters to the `wget` process. For

example, the process might download a document and save it to an arbitrary location on the filesystem.

### URL Redirection

The following APIs can be used to issue an HTTP redirect in ASP.NET:

- `System.Web.HttpResponse.Redirect`
- `System.Web.HttpResponse.Status`
- `System.Web.HttpResponse.StatusCode`
- `System.Web.HttpResponse.AddHeader`
- `System.Web.HttpResponse.AppendHeader`
- `Server.Transfer`

The usual means of causing a redirect response is via the `HttpResponse.Redirect` method, which takes a string containing a relative or absolute URL. If the value of this string is user-controllable, the application is probably vulnerable to a phishing vector.

You should also be sure to review any uses of the `Status/StatusCode` properties and the `AddHeader/AppendHeader` methods. Given that a redirect simply involves a 3xx response containing an HTTP `Location` header, an application may implement redirects using these APIs.

The `Server.Transfer` method is also sometimes used to perform redirection. However, this does not in fact cause an HTTP redirect. Instead, it simply changes the page being processed on the server in response to the current request. Accordingly, it cannot be subverted to cause redirection to an off-site URL, so it is usually less useful to an attacker.

### Sockets

The `System.Net.Sockets.Socket` class is used to create network sockets. After a `Socket` object has been created, it is connected via a call to the `Connect` method, which takes the IP and port details of the target host as its parameters. If this host information can be controlled by the user in any way, the application may be exploitable to cause network connections to arbitrary hosts, either on the Internet or on the private DMZ or internal network on which the application is hosted.

## Configuring the ASP.NET Environment

The `Web.config` XML file in the web root directory contains configuration settings for the ASP.NET environment, listed in Table 19-6, and controls how applications behave.

**Table 19-6:** Security-Relevant Configuration Settings for the ASP.NET Environment

| SETTING | DESCRIPTION |
|---|---|
| `httpCookies` | Determines the security settings associated with cookies. If the `httpOnlyCookies` attribute is set to `true`, cookies are flagged as `HttpOnly` and therefore are not directly accessible from client-side scripts. If the `requireSSL` attribute is set to `true`, cookies are flagged as `secure` and therefore are transmitted by browsers only within HTTPS requests. |
| `sessionState` | Determines how sessions behave. The value of the `timeout` attribute determines the time in minutes after which an idle session will be expired. If the `regenerateExpiredSessionId` element is set to `true` (which is the default), a new session ID is issued when an expired session ID is received. |
| `compilation` | Determines whether debugging symbols are compiled into pages, resulting in more verbose debug error information. If the `debug` attribute is set to `true`, debug symbols are included. |
| `customErrors` | Determines whether the application returns detailed error messages in the event of an unhandled error. If the `mode` attribute is set to `On` or `RemoteOnly`, the page identified by the `defaultRedirect` attribute is displayed to application users in place of detailed system-generated messages. |
| `httpRuntime` | Determines various runtime settings. If the `enableHeaderChecking` attribute is set to `true` (which is the default), ASP.NET checks request headers for potential injection attacks, including cross-site scripting. If the `enableVersionHeader` attribute is set to `true` (which is the default), ASP.NET outputs a detailed version string, which may be of use to an attacker in researching vulnerabilities in specific versions of the platform. |

If sensitive data such as database connection strings is stored in the configuration file, it should be encrypted using the ASP.NET "protected configuration" feature.

# PHP

This section describes ways to acquire user-supplied input, ways to interact with the user's session, potentially dangerous APIs, and security-relevant configuration options on the PHP platform.

## Identifying User-Supplied Data

PHP uses a range of array variables to store user-submitted data, as listed in Table 19-7.

**Table 19-7:** Variables Used to Acquire User-Supplied Data on the PHP Platform

| VARIABLE | DESCRIPTION |
|---|---|
| `$_GET`<br>`$HTTP_GET_VARS` | Contains the parameters submitted in the query string. These are accessed by name. For example, in the following URL:<br>`https://wahh-app.com/search.php?query=foo`<br>the value of the `query` parameter is accessed using:<br>`$_GET['query']` |
| `$_POST`<br>`$HTTP_POST_VARS` | Contains the parameters submitted in the request body. |
| `$_COOKIE`<br>`$HTTP_COOKIE_VARS` | Contains the cookies submitted in the request. |
| `$_REQUEST` | Contains all the items in the `$_GET`, `$_POST`, and `$_COOKIE` arrays. |
| `$_FILES`<br>`$HTTP_POST_FILES` | Contains the files uploaded in the request. |
| `$_SERVER['REQUEST_METHOD']` | Contains the method used in the HTTP request. |
| `$_SERVER['QUERY_STRING']` | Contains the full query string submitted in the request. |
| `$_SERVER['REQUEST_URI']` | Contains the full URL contained in the request. |
| `$_SERVER['HTTP_ACCEPT']` | Contains the contents of the HTTP `Accept` header. |
| `$_SERVER['HTTP_ACCEPT_CHARSET']` | Contains the contents of the HTTP `Accept-charset` header. |
| `$_SERVER['HTTP_ACCEPT_ENCODING']` | Contains the contents of the HTTP `Accept-encoding` header. |
| `$_SERVER['HTTP_ACCEPT_LANGUAGE']` | Contains the contents of the HTTP `Accept-language` header. |
| `$_SERVER['HTTP_CONNECTION']` | Contains the contents of the HTTP `Connection` header. |
| `$_SERVER['HTTP_HOST']` | Contains the contents of the HTTP `Host` header. |

*Continued*

**Table 19-7** *(continued)*

| VARIABLE | DESCRIPTION |
|---|---|
| `$_SERVER['HTTP_REFERER']` | Contains the contents of the HTTP `Referer` header. |
| `$_SERVER['HTTP_USER_AGENT']` | Contains the contents of the HTTP `User-agent` header. |
| `$_SERVER['PHP_SELF']` | Contains the name of the currently executing script. Although the script name itself is outside an attacker's control, path information can be appended to this name. For example, if a script contains the following code:<br>`<form action="<?= $_ SERVER['PHP_SELF'] ?>">`<br>an attacker can craft a cross-site scripting attack as follows:<br>`/search.php/"><script>`<br>and so on. |

You should keep in mind various anomalies when attempting to identify ways in which a PHP application is accessing user-supplied input:

- `$GLOBALS` is an array containing references to all variables that are defined in the script's global scope. It may be used to access other variables by name.
- If the configuration directive `register_globals` is enabled, PHP creates global variables for all request parameters — that is, everything contained in the `$_REQUEST` array. This means that an application may access user input simply by referencing a variable that has the same name as the relevant parameter. If an application uses this method of accessing user-supplied data, there may be no way to identify all instances of this other than via a careful line-by-line review of the codebase to find variables used in this way.
- In addition to the standard HTTP headers identified previously, PHP adds an entry to the `$_SERVER` array for any custom HTTP headers received in the request. For example, supplying the header:

  ```
  Foo: Bar
  ```

  causes:

  ```
  $_SERVER['HTTP_FOO'] = "Bar"
  ```

- Input parameters whose names contain subscripts in square brackets are automatically converted into arrays. For example, requesting this URL:

  ```
  https://wahh-app.com/search.php?query[a]=foo&query[b]=bar
  ```

  causes the value of the `$_GET['query']` variable to be an array containing two members. This may result in unexpected behavior within the application if an array is passed to a function that expects a scalar value.

## Session Interaction

PHP uses the `$_SESSION` array as a way to store and retrieve information within the user's session. For example:

```
$_SESSION['MyName'] = $_GET['username'];       // store user's name
echo "Welcome " . $_SESSION['MyName'];         // retrieve user's name
```

The `$HTTP_SESSION_VARS` array may be used in the same way.

If `register_globals` is enabled (as discussed in the later section "Configuring the PHP Environment"), global variables may be stored within the current session as follows:

```
$MyName = $_GET['username'];
session_register("MyName");
```

## Potentially Dangerous APIs

This section describes some common PHP APIs that can introduce security vulnerabilities if used in an unsafe manner.

### *File Access*

PHP implements a large number of functions for accessing files, many of which accept URLs and other constructs that may be used to access remote files.

The following functions are used to read or write the contents of a specified file. If user-controllable data is passed to these APIs, an attacker may be able to exploit these to access arbitrary files on the server filesystem.

- `fopen`
- `readfile`
- `file`
- `fpassthru`
- `gzopen`

- `gzfile`
- `gzpassthru`
- `readgzfile`
- `copy`
- `rename`
- `rmdir`
- `mkdir`
- `unlink`
- `file_get_contents`
- `file_put_contents`
- `parse_ini_file`

The following functions are used to include and evaluate a specified PHP script. If an attacker can cause the application to evaluate a file he controls, he can achieve arbitrary command execution on the server.

- `include`
- `include_once`
- `require`
- `require_once`
- `virtual`

Note that even if it is not possible to include remote files, command execution may still be possible if there is a way to upload arbitrary files to a location on the server.

The PHP configuration option `allow_url_fopen` can be used to prevent some file functions from accessing remote files. However, by default this option is set to `1` (meaning that remote files are allowed), so the protocols listed in Table 19-8 can be used to retrieve a remote file.

**Table 19-8:** Network Protocols That Can Be Used to Retrieve a Remote File

| PROTOCOL | EXAMPLE |
|---|---|
| HTTP, HTTPS | `http://wahh-attacker.com/bad.php` |
| FTP | `ftp://user:password@wahh-attacker.com/bad.php` |
| SSH | `ssh2.shell://user:pass@wahh-attacker.com:22/xterm`<br>`ssh2.exec://user:pass@wahh-attacker.com:22/cmd` |

Even if `allow_url_fopen` is set to `0`, the methods listed in Table 19-9 may still enable an attacker to access remote files (depending on the extensions installed).

**Table 19-9:** Methods That May Allow Access to Remote Files Even If `allow_url_fopen` Is Set to 0

| METHOD | EXAMPLE |
|---|---|
| SMB | `\\wahh-attacker.com\bad.php` |
| PHP input/output streams | `php://filter/resource=http://wahh-attacker.com/bad.php` |
| Compression streams | `compress.zlib://http://wahh-attacker.com/bad.php` |
| Audio streams | `ogg://http://wahh-attacker.com/bad.php` |

**NOTE** **PHP 5.2 and later releases have a new option, `allow_url_include`, which is disabled by default. This default configuration prevents any of the preceding methods from being used to specify a remote file when calling one of the file include functions.**

### *Database Access*

The following functions are used to send a query to a database and retrieve the results:

- `mysql_query`
- `mssql_query`
- `pg_query`

The SQL statement is passed as a simple string. If user-controllable input is part of the string parameter, the application is probably vulnerable to SQL injection. For example:

```
$username = "admin' or 1=1--";
$password = "foo";
$sql="SELECT * FROM users WHERE username = '$username'
    AND password = '$password'";
$result = mysql_query($sql, $link)
```

executes this unintended query:

```
SELECT * FROM users WHERE username = 'admin' or 1=1--'
    AND password = 'foo'
```

The following functions can be used to create prepared statements. This allows an application to create a SQL query containing parameter placeholders and set their values in a secure and type-safe way:

- mysqli->prepare
- stmt->prepare
- stmt->bind_param
- stmt->execute
- odbc_prepare

If used as intended, this mechanism is not vulnerable to SQL injection. For example:

```
$username = "admin' or 1=1--";
$password = "foo";
$sql = $db_connection->prepare(
    "SELECT * FROM users WHERE username = ? AND password = ?");
$sql->bind_param("ss", $username, $password);
$sql->execute();
```

results in a query that is equivalent to the following:

```
SELECT * FROM users WHERE username = 'admin'' or 1=1--'
    AND password = 'foo'
```

### *Dynamic Code Execution*

The following functions can be used to dynamically evaluate PHP code:

- eval
- call_user_func
- call_user_func_array
- call_user_method
- call_user_method_array
- create_function

The semicolon delimiter can be used to batch multiple statements. If user-controllable data is passed into any of these functions, the application is probably vulnerable to script injection.

The function `preg_replace`, which performs a regular expression search and replace, can be used to run a specific piece of PHP code against every match if called with the `/e` option. If user-controllable data appears in the PHP that is dynamically executed, the application is probably vulnerable.

Another interesting feature of PHP is the ability to invoke functions dynamically via a variable containing the function's name. For example, the following code invokes the function specified in the `func` parameter of the query string:

```
<?php
    $var=$_GET['func'];
    $var();
?>
```

In this situation, a user can cause the application to invoke an arbitrary function (without parameters) by modifying the value of the `func` parameter. For example, invoking the `phpinfo` function causes the application to output a large amount of information about the PHP environment, including configuration options, OS information, and extensions.

### OS Command Execution

These functions can be used to execute operating system commands:

- `exec`
- `passthru`
- `popen`
- `proc_open`
- `shell_exec`
- `system`
- The backtick operator (`` ` ``)

In all these cases, commands can be chained together using the `|` character. If user-controllable data is passed unfiltered into any of these functions, the application is probably vulnerable to arbitrary command execution.

### URL Redirection

The following APIs can be used to issue an HTTP redirect in PHP:

- `http_redirect`
- `header`
- `HttpMessage::setResponseCode`
- `HttpMessage::setHeaders`

The usual way to cause a redirect is through the `http_redirect` function, which takes a string containing a relative or absolute URL. If the value of

this string is user-controllable, the application is probably vulnerable to a phishing vector.

Redirects can also be performed by calling the `header` function with an appropriate `Location` header, which causes PHP to deduce that an HTTP redirect is required. For example:

```
header("Location: /target.php");
```

You should also review any uses of the `setResponseCode` and `setHeaders` APIs. Given that a redirect simply involves a 3xx response containing an HTTP `Location` header, an application may implement redirects using these APIs.

### *Sockets*

The following APIs can be used to create and use network sockets in PHP:

- `socket_create`
- `socket_connect`
- `socket_write`
- `socket_send`
- `socket_recv`
- `fsockopen`
- `pfsockopen`

After a socket is created using `socket_create`, it is connected to a remote host via a call to `socket_connect`, which takes the target's host and port details as its parameters. If this host information is user-controllable in any way, the application may be exploitable to cause network connections to arbitrary hosts, either on the public Internet or on the private DMZ or internal network on which the application is hosted.

The `fsockopen` and `pfsockopen` functions can be used to open sockets to a specified host and port and return a file pointer that can be used with regular file functions such as `fwrite` and `fgets`. If user data is passed to these functions, the application may be vulnerable, as described previously.

## Configuring the PHP Environment

PHP configuration options are specified in the `php.ini` file, which uses the same structure as Windows INI files. Various options can affect an application's security. Many options that have historically caused problems have been removed from the latest version of PHP.

### Register Globals

If the `register_globals` directive is enabled, PHP creates global variables for all request parameters. Given that PHP does not require variables to be initialized before use, this option can easily lead to security vulnerabilities in which an attacker can cause a variable to be initialized to an arbitrary value.

For example, the following code checks a user's credentials and sets the `$authenticated` variable to `1` if they are valid:

```
if (check_credentials($username, $password))
{
    $authenticated = 1;
}
...
if ($authenticated)
{
    ...
```

Because the `$authenticated` variable is not first explicitly initialized to `0`, an attacker can bypass the login by submitting the request parameter `authenticated=1`. This causes PHP to create the global variable `$authenticated` with a value of `1` before the credentials check is performed.

**NOTE From PHP 4.2.0 onward, the `register_globals` directive is disabled by default. However, because many legacy applications depend on `register_globals` for their normal operation, you may often find that this directive has been explicitly enabled in `php.ini`. The `register_globals` option was removed in PHP 6.**

### Safe Mode

If the `safe_mode` directive is enabled, PHP places restrictions on the use of some dangerous functions. Some functions are disabled, and others are subject to limitations on their use. For example:

- The `shell_exec` function is disabled because it can be used to execute operating system commands.
- The `mail` function has the parameter `additional_parameters` disabled because unsafe use of this parameter may lead to SMTP injection flaws (see Chapter 10).
- The `exec` function can be used only to launch executables within the configured `safe_mode_exec_dir`. Metacharacters within the command string are automatically escaped.

**NOTE** **Not all dangerous functions are restricted by safe mode, and some restrictions are affected by other configuration options. Furthermore, there are various ways to bypass some safe mode restrictions. Safe mode should not be considered a panacea to security issues within PHP applications. Safe mode has been removed from PHP version 6.**

### *Magic Quotes*

If the `magic_quotes_gpc` directive is enabled, any single quote, double quote, backslash, and `NULL` characters contained within request parameters are automatically escaped using a backslash. If the `magic_quotes_sybase` directive is enabled, single quotes are instead escaped using a single quote. This option is designed to protect vulnerable code containing unsafe database calls from being exploitable via malicious user input. When reviewing the application codebase to identify any SQL injection flaws, you should be aware of whether magic quotes are enabled, because this affects the application's handling of input.

Using magic quotes does not prevent all SQL injection attacks. As described in Chapter 9, an attack that injects into a numeric field does not need to use single quotation marks. Furthermore, data whose quotes have been escaped may still be used in a second-order attack when it is subsequently read back from the database.

The magic quotes option may result in undesirable modification of user input, when data is being processed in a context that does not require any escaping. This can result in the addition of slashes that need to be removed using the `stripslashes` function.

Some applications perform their own escaping of relevant input by passing individual parameters through the `addslashes` function only when required. If magic quotes are enabled in the PHP configuration, this approach results in double-escaped characters. Doubled-up slashes are interpreted as literal backslashes, leaving the potentially malicious character unescaped.

Because of the limitations and anomalies of the magic quotes option, it is recommended that prepared statements be used for safe database access and that the magic quotes option be disabled.

**NOTE** **The magic quotes option has been removed from PHP version 6.**

### *Miscellaneous*

Table 19-10 lists some miscellaneous configuration options that can affect the security of PHP applications.

**Table 19-10:** Miscellaneous PHP Configuration Options

| OPTION | DESCRIPTION |
|---|---|
| `allow_url_fopen` | If disabled, this directive prevents some file functions from accessing remote files (as described previously). |
| `allow_url_include` | If disabled, this directive prevents the PHP file include functions from being used to include a remote file. |
| `display_errors` | If disabled, this directive prevents PHP errors from being reported to the user's browser. The `log_errors` and `error_log` options can be used to record error information on the server for diagnostic purposes. |
| `file_uploads` | If enabled, this directive causes PHP to allow file uploads over HTTP. |
| `upload_tmp_dir` | This directive can be used to specify the temporary directory used to store uploaded files. This can be used to ensure that sensitive files are not stored in a world-readable location. |

# Perl

This section describes ways to acquire user-supplied input, ways to interact with the user's session, potentially dangerous APIs, and security-relevant configuration options on the Perl platform.

The Perl language is notorious for allowing developers to perform the same task in a multitude of ways. Furthermore, numerous Perl modules can be used to meet different requirements. Any unusual or proprietary modules in use should be closely reviewed to identify whether they use any powerful or dangerous functions and thus may introduce the same vulnerabilities as if the application made direct use of those functions.

CGI.pm is a widely used Perl module for creating web applications. It provides the APIs you are most likely to encounter when performing a code review of a web application written in Perl.

## Identifying User-Supplied Data

The functions listed in Table 19-11 are all members of the CGI query object.

**Table 19-11:** CGI Query Members Used to Acquire User-Supplied Data

| FUNCTION | DESCRIPTION |
|---|---|
| `param`<br>`param_fetch` | Called without parameters, `param` returns a list of all the parameter names in the request.<br>Called with the name of a parameter, `param` returns the value of that request parameter.<br>The `param_fetch` method returns an array of the named parameters. |
| `Vars` | Returns a hash mapping of parameter names to values. |
| `cookie`<br>`raw_cookie` | The value of a named cookie can be set and retrieved using the `cookie` function.<br>The `raw_cookie` function returns the entire contents of the HTTP `Cookie` header, without any parsing having been performed. |
| `self_url`<br>`url` | Return the current URL, in the first case including any query string. |
| `query_string` | Returns the query string of the current request. |
| `referer` | Returns the value of the HTTP `Referer` header. |
| `request_method` | Returns the value of the HTTP method used in the request. |
| `user_agent` | Returns the value of the HTTP `User-agent` header. |
| `http`<br>`https` | Return a list of all the HTTP environment variables derived from the current request. |
| `ReadParse` | Creates an array named `%in` that contains the names and values of all the request parameters. |

## Session Interaction

The Perl module `CGISession.pm` extends the `CGI.pm` module and provides support for session tracking and data storage. For example:

```
$q->session_data("MyName"=>param("username"));  // store user's name
print "Welcome " . $q->session_data("MyName");  // retrieve user's name
```

## Potentially Dangerous APIs

This section describes some common Perl APIs that can introduce security vulnerabilities if used in an unsafe manner.

### File Access

The following APIs can be used to access files in Perl:

- `open`
- `sysopen`

The `open` function reads and writes the contents of a specified file. If user-controllable data is passed as the filename parameter, an attacker may be able to access arbitrary files on the server filesystem.

Furthermore, if the filename parameter begins or ends with the pipe character, the contents of this parameter are passed to a command shell. If an attacker can inject data containing shell metacharacters such as the pipe or semicolon, he may be able to perform arbitrary command execution. For example, in the following code, an attacker can inject into the `$useraddr` parameter to execute system commands:

```
$useraddr = $query->param("useraddr");
open (MAIL, "| /usr/bin/sendmail $useraddr");
print MAIL "To: $useraddr\n";
...
```

### Database Access

The `selectall_arrayref` function sends a query to a database and retrieves the results as an array of arrays. The `do` function executes a query and simply returns the number of rows affected. In both cases, the SQL statement is passed as a simple string.

If user-controllable input comprises part of the string parameter, the application is probably vulnerable to SQL injection. For example:

```
my $username = "admin' or 1=1--";
my $password = "foo";
my $sql="SELECT * FROM users WHERE username = '$username' AND password =
 '$password'";
my $result = $db_connection->selectall_arrayref($sql)
```

executes this unintended query:

```
SELECT * FROM users WHERE username = 'admin' or 1=1--'
    AND password = 'foo'
```

The functions `prepare` and `execute` can be used to create prepared statements, allowing an application to create a SQL query containing parameter

placeholders and set their values in a secure and type-safe way. If used as intended, this mechanism is not vulnerable to SQL injection. For example:

```
my $username = "admin' or 1=1--";
my $password = "foo";
my $sql = $db_connection->prepare("SELECT * FROM users
    WHERE username = ? AND password = ?");
$sql->execute($username, $password);
```

results in a query that is equivalent to the following:

```
SELECT * FROM users WHERE username = 'admin'' or 1=1--'
    AND password = 'foo'
```

### Dynamic Code Execution

`eval` can be used to dynamically execute a string containing Perl code. The semicolon delimiter can be used to batch multiple statements. If user-controllable data is passed into this function, the application is probably vulnerable to script injection.

### OS Command Execution

The following functions can be used to execute operating system commands:

- `system`
- `exec`
- `qx`
- The backtick operator (`` ` ``)

In all these cases, commands can be chained together using the `|` character. If user-controllable data is passed unfiltered into any of these functions, the application is probably vulnerable to arbitrary command execution.

### URL Redirection

The `redirect` function, which is a member of the CGI query object, takes a string containing a relative or absolute URL, to which the user is redirected. If the value of this string is user-controllable, the application is probably vulnerable to a phishing vector.

### *Sockets*

After a socket is created using `socket`, it is connected to a remote host via a call to `connect`, which takes a `sockaddr_in` structure composed of the target's host and port details. If this host information is user-controllable in any way, the application may be exploitable to cause network connections to arbitrary hosts, either on the Internet or on the private DMZ or internal network on which the application is hosted.

## Configuring the Perl Environment

Perl provides a taint mode that helps prevent user-supplied input from being passed to potentially dangerous functions. You can execute Perl programs in taint mode by passing the `-T` flag to the Perl interpreter as follows:

```
#!/usr/bin/perl -T
```

When a program is running in taint mode, the interpreter tracks each item of input received from outside the program and treats it as tainted. If another variable has its value assigned on the basis of a tainted item, it too is treated as tainted. For example:

```
$path = "/home/pubs"           # $path is not tainted
$filename = param("file");     # $filename is from request parameter and
                               # is tainted
$full_path = $path.$filename;  # $full_path now tainted
```

Tainted variables cannot be passed to a range of powerful commands, including `eval`, `system`, `exec`, and `open`. To use tainted data in sensitive operations, the data must be "cleaned" by performing a pattern-matching operation and extracting the matched substrings. For example:

```
$full_path =~ m/^([a-zA-Z1-9]+)$/;  # match alphanumeric submatch
                                    # in $full_path
$clean_full_path = $1;              # set $clean_full_path to the
                                    # first submatch
                                    # $clean_full_path is untainted
```

Although the taint mode mechanism is designed to help protect against many kinds of vulnerabilities, it is effective only if developers use appropriate regular expressions when extracting clean data from tainted input. If an expression is too liberal and extracts data that may cause problems in the context in which it

will be used, the taint mode protection fails, and the application is still vulnerable. In effect, the taint mode mechanism reminds programmers to perform suitable validation on all input before using it in dangerous operations. It cannot guarantee that the input validation implemented will be adequate.

## JavaScript

Client-side JavaScript can, of course, be accessed without requiring any privileged access to the application, enabling you to perform a security-focused code review in any situation. A key focus of this review is to identify any vulnerabilities such as DOM-based XSS, which are introduced on the client component and leave users vulnerable to attack (see Chapter 12). A further reason for reviewing JavaScript is to understand what kinds of input validation are implemented on the client, and also how dynamically generated user interfaces are constructed.

When reviewing JavaScript, you should be sure to include both `.js` files and scripts embedded in HTML content.

The key APIs to focus on are those that read from DOM-based data and that write to or otherwise modify the current document, as shown in Table 19-12.

**Table 19-12:** JavaScript APIs That Read from DOM-Based Data

| API | DESCRIPTION |
|---|---|
| `document.location`<br>`document.URL`<br>`document.URLUnencoded`<br>`document.referrer`<br>`window.location` | Can be used to access DOM data that may be controllable via a crafted URL, and may therefore represent an entry point for crafted data to attack other application users. |
| `document.write()`<br>`document.writeln()`<br>`document.body.innerHtml`<br>`eval()`<br>`window.execScript()`<br>`window.setInterval()`<br>`window.setTimeout()` | Can be used to update the document's contents and to dynamically execute JavaScript code. If attacker-controllable data is passed to any of these APIs, this may provide a way to execute arbitrary JavaScript within a victim's browser. |

# Database Code Components

Web applications increasingly use databases for much more than passive data storage. Today's databases contain rich programming interfaces, enabling substantial business logic to be implemented within the database tier itself. Developers frequently use database code components such as stored procedures, triggers, and user-defined functions to carry out key tasks. Therefore, when you review the source code to a web application, you should ensure that all logic implemented in the database is included in the scope of the review.

Programming errors in database code components can potentially result in any of the various security defects described in this chapter. In practice, however, you should watch for two main areas of vulnerabilities. First, database components may themselves contain SQL injection flaws. Second, user input may be passed to potentially dangerous functions in unsafe ways.

## SQL Injection

Chapter 9 described how prepared statements can be used as a safe alternative to dynamic SQL statements to prevent SQL injection attacks. However, even if prepared statements are properly used throughout the web application's own code, SQL injection flaws may still exist if database code components construct queries from user input in an unsafe manner.

The following is an example of a stored procedure that is vulnerable to SQL injection in the `@name` parameter:

```
CREATE PROCEDURE show_current_orders
    (@name varchar(400) = NULL)
AS
DECLARE @sql nvarchar(4000)
SELECT @sql = 'SELECT id_num, searchstring FROM searchorders WHERE ' +
              'searchstring = ''' + @name + '''';
EXEC (@sql)
GO
```

Even if the application passes the user-supplied `name` value to the stored procedure in a safe manner, the procedure itself concatenates this directly into a dynamic query and therefore is vulnerable.

Different database platforms use different methods to perform dynamic execution of strings containing SQL statements. For example:

- **MS-SQL** — `EXEC`
- **Oracle** — `EXECUTE IMMEDIATE`

- **Sybase** — `EXEC`
- **DB2** — `EXEC SQL`

Any appearance of these expressions within database code components should be closely reviewed. If user input is being used to construct the SQL string, the application may be vulnerable to SQL injection.

**NOTE** **On Oracle, stored procedures by default run with the permissions of the definer, rather than the invoker (as with SUID programs on UNIX). Hence, if the application uses a low-privileged account to access the database, and stored procedures were created using a DBA account, a SQL injection flaw within a procedure may enable you to escalate privileges and perform arbitrary database queries.**

## Calls to Dangerous Functions

Customized code components such as stored procedures are often used to perform unusual or powerful actions. If user-supplied data is passed to a potentially dangerous function in an unsafe way, this may lead to various kinds of vulnerabilities, depending on the nature of the function. For example, the following stored procedure is vulnerable to command injection in the `@loadfile` and `@loaddir` parameters:

```
Create import_data (@loadfile varchar(25), @loaddir varchar(25) )
as
begin
select @cmdstring = "$PATH/firstload " + @loadfile + " " + @loaddir
exec @ret = xp_cmdshell @cmdstring
...
...
End
```

The following functions may be potentially dangerous if invoked in an unsafe way:

- Powerful default stored procedures in MS-SQL and Sybase that allow execution of commands, registry access, and so on
- Functions that provide access to the filesystem
- User-defined functions that link to libraries outside the database
- Functions that result in network access, such as through `OpenRowSet` in MS-SQL or a database link in Oracle

# Tools for Code Browsing

The methodology we have described for performing a code review essentially involves reading the source code and searching for patterns indicating the capture of user input and the use of potentially dangerous APIs. To carry out a code review effectively, it is preferable to use an intelligent tool to browse the codebase. You need a tool that understands the code constructs in a particular language, provides contextual information about specific APIs and expressions, and facilitates your navigation.

In many languages, you can use one of the available development studios, such as Visual Studio, NetBeans, or Eclipse. In addition, various generic code-browsing tools support numerous languages and are optimized for viewing of code rather than development. The authors' preferred tool is Source Insight, shown in Figure 19-1. It supports easy browsing of the source tree, a versatile search function, a preview pane to display contextual information about any selected expression, and speedy navigation through the codebase.

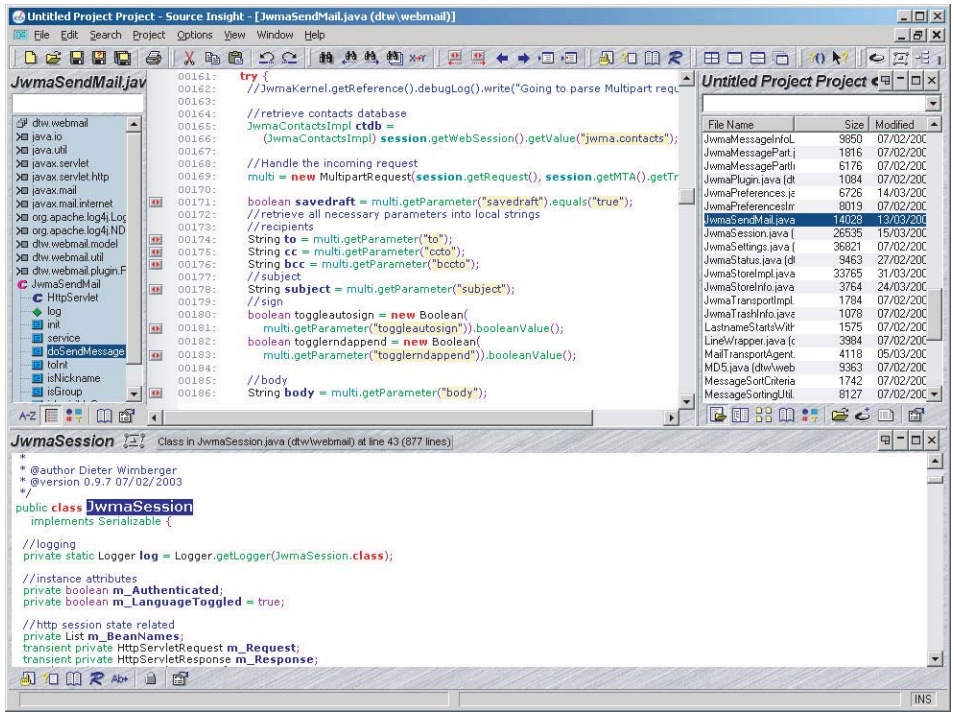


**Figure 19-1:** Source Insight being used to search and browse the source code for a web application

## Summary

Many people who have substantial experience with testing web applications interactively, exhibit an irrational fear of looking inside an application's codebase to discover vulnerabilities directly. This fear is understandable for people who are not programmers, but it is rarely justified. Anyone who is familiar with dealing with computers can, with a little investment, gain sufficient knowledge and confidence to perform an effective code audit. Your objective in reviewing an application's codebase need not be to discover "all" the vulnerabilities it contains, any more than you would set yourself this unrealistic goal when performing hands-on testing. More reasonably, you can aspire to understand some of the key processing that the application performs on user-supplied input and recognize some of the signatures that point toward potential problems. Approached in this way, code review can be an extremely useful complement to the more familiar black-box testing. It can improve the effectiveness of that testing and reveal defects that may be extremely difficult to discover when you are dealing with an application entirely from the outside.

## Questions

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

1. List three categories of common vulnerabilities that often have easily recognizable signatures within source code.
2. Why can identifying all sources of user input sometimes be challenging when reviewing a PHP application?
3. Consider the following two methods of performing a SQL query that incorporates user-supplied input:

```
// method 1
String artist = request.getParameter("artist").replaceAll("'", "''");
String genre = request.getParameter("genre").replaceAll("'", "''");
String album = request.getParameter("album").replaceAll("'", "''");
Statement s = connection.createStatement();
s.executeQuery("SELECT * FROM music WHERE artist = '" + artist +
    '" AND genre = '" + genre + '" AND album = '" + album + "'");

// method 2
String artist = request.getParameter("artist");
String genre = request.getParameter("genre");
String album = request.getParameter("album");
Statement s = connection.prepareStatement(
    "SELECT * FROM music WHERE artist = '" + artist +
    "' AND genre = ? AND album = ?");
```

```
s.setString(1, genre);
s.setString(2, album);
s.executeQuery();
```

Which of these methods is more secure, and why?

4. You are reviewing the codebase of a Java application. During initial reconnaissance, you search for all uses of the `HttpServletRequest.getParameter` API. The following code catches your eye:

```
private void setWelcomeMessage(HttpServletRequest request) throws
    ServletException
{
    String name = request.getParameter("name");

    if (name == null)
        name = "";

    m_welcomeMessage = "Welcome " + name +"!";
}
```

What possible vulnerability might this code indicate? What further code analysis would you need to perform to confirm whether the application is indeed vulnerable?

5. You are reviewing the mechanism that an application uses to generate session tokens. The relevant code is as follows:

```
public class TokenGenerator
{
    private java.util.Random r = new java.util.Random();

    public synchronized long nextToken()
    {
        long l = r.nextInt();
        long m = r.nextInt();

        return l + (m << 32);
    }
}
```

Are the application's session tokens being generated in a predictable way? Explain your answer fully.