CHAPTER

26 Secure Application Design

This chapter covers the important security considerations that should be part of the development cycle of web applications, client applications, and remote administration, illustrating potential security issues and how to solve them.

After an application is written, it is deployed into an environment of some sort, where it remains for an extended period of time with only its original features to defend it from whatever threats, mistakes, or misuse it encounters. A malicious agent in the environment, on the other hand, has that same extended period of time to observe the application and tailor its attack techniques until something works. At this point, any number of undesirable things could happen. For example, there could be a breach, there could be a vulnerability disclosure, malware exploiting the vulnerability could be released, or the exploit technique could be sold to the highest bidder.

Most of these undesirable things eventually lead to customers who are unhappy with their software vendors, regardless of whether or not the customers were willing to pay for security before the incident occurred. For that reason, security is becoming more important to organizations that produce software, and building security into the software up front is easier (and cheaper) than waiting until the software is already out in the field and then providing security updates.

While the deployment environment can help protect the application to some extent, every application must be secure enough to protect itself from whatever meaningful attacks the deployment environment cannot prevent, for long enough for the operator to notice and respond to attacks in progress. This chapter describes techniques for developing applications that are secure enough for their intended use, during the development cycle, to save time and money down the road.

Secure Development Lifecycle

A secure development lifecycle (SDL, or sometimes SSDL, for secure software development lifecycle) is essentially a development process that includes security practices and decision-making inputs. In some cases, an SDL is a stand-alone process, such as in the case of

Microsoft's well-known Security Development Lifecycle (www.microsoft.com/security/sdl/default.aspx), but most organizations find that altering their existing practices and processes is easier and more efficient than creating and managing an additional, separate process.

Despite the name, which implies a single lifecycle, a typical SDL actually affects two to three lifecycles, the specifics of which vary by organization:

- The application lifecycle, in which an application begins as an idea and then is planned, designed, developed, tested, documented (hopefully), released, sometimes deployed and operated, maintained, and eventually "end-of-lifed."
- The employee lifecycle, in which an employee is selected, hired, brought on board, changes job responsibilities, and eventually leaves the organization.
- The project or contract lifecycle, if any development is outsourced, in which a contract is negotiated, results are accepted, and vendors are paid.

The SDL itself is created, operated, measured, and changed over time following a business process lifecycle. Sometimes people call the process of developing and maintaining an SDL and other application security activities an *application security assurance program.*

Typically, an SDL contains three primary elements:

- Security activities that don't exist at all in the original lifecycle; for instance, threat modeling
- Security modifications to existing activities; for instance, adding security checks to existing peer reviews of code
- Security criteria that should affect existing decisions; for instance, the number of open high-severity security issues when a decision to ship is made

Figure 26-1 shows the application lifecycle portion of an SDL for an organization that uses an Agile development lifecycle.

Like any other quality, adding security is cheapest if it is included from the beginning of the lifecycle. Like other bugs, security vulnerabilities are less expensive to fix the earlier they are resolved, and the cheapest thing to do is to avoid inserting bugs at all. Therefore, the pre-ship activities in an SDL usually focus on either preventing security bugs in each development deliverable or detecting security bugs in a deliverable that was just produced.

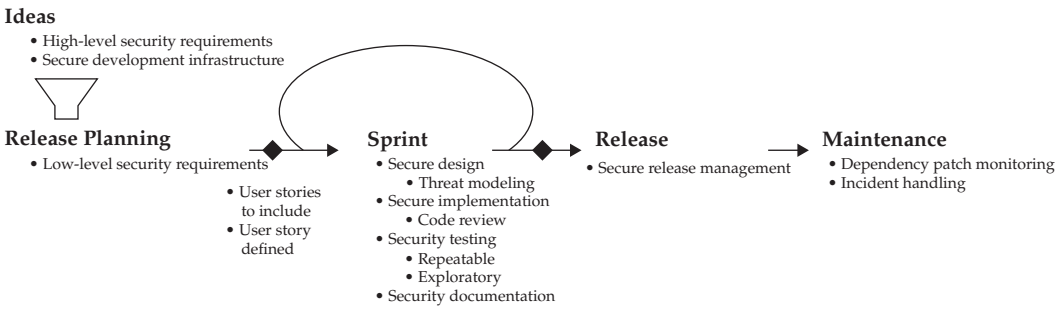

Figure 26-1 Secure development lifecycle in Agile

Waterfall SDLs frequently involve a security reviewer from outside the team, who must approve the application at different points in the process. Agile SDLs frequently provide access to security coaches from outside the team, so that the team has someone to consult when they need security help.

Finally, because different applications have different security requirements, it is common for an SDL to require all applications to determine their requirements, and then allow applications with lower security requirements to skip some security activities or perform checks less rigorously.

Application Security Practices

This section provides a brief overview of the practices and decisions that appear in some form in most secure development lifecycles.

Security Training

Typically, a security training program for development teams includes technical security-awareness training for everyone and role-specific training for most individuals. Role-specific training goes into more detail about the security activities a particular individual participates in, and the technologies in use (for developers).

Secure Development Infrastructure

At the beginning of a new project, source code repositories, file shares, and build servers must be configured for team members' exclusive access, bug tracking software must be configured to disclose security bugs only according to organization policies, project contacts must be registered in case any application security issues occur, and licenses for secure development tools must be acquired.

Security Requirements

Security requirements may include access control matrices, security objectives (which specify actions attackers with specific privileges should not be able to perform), abuse cases, references to policies and standards, logging requirements, security bug bars, assignment of a security risk or impact level, and low-level security requirements such as key sizes or how specific error conditions should be handled.

Secure Design

Secure design activities usually revolve around secure design principles and patterns. They also frequently include adding information about security properties and responsibilities to design documents. For more information on secure application design, see Chapter 27.

Threat Modeling

Threat modeling is a technique for reviewing the security properties of a design and identifying potential issues and fixes. Architects can perform it as a secure design activity, or independent design reviewers can perform it to verify architects' work. There is a variety of threat modeling methodologies to choose from. For more information, see Chapter 27.

Secure Coding

Secure coding includes using safe or approved versions of functions and libraries, eliminating unused code, following policies, handling data safely, managing resources correctly, handling events safely, and using security technology correctly. Chapter 27 covers these concepts in more detail.

Security Code Review

To find security issues by inspecting application code, development teams may use static analysis tools, manual code review, or a combination. Static analysis tools are very effective at finding some kinds of mechanical security issues but are usually ineffective at finding algorithmic issues like incorrect enforcement of business logic. Static analysis tools usually require tuning to avoid high numbers of false positives. Manual code review by someone other than the code author is more effective at finding issues that involve code semantics, but requires training and experience. Manual code review is also time-consuming and may miss mechanical issues that require tracing large numbers of lines of code or remembering many details.

Security Testing

To find security issues by running application code, developers and independent testers perform repeatable security testing, such as fuzzing and regression tests for past security issues, and exploratory security testing, such as penetration testing.

Security Documentation

When an application will be operated by someone other than the development team, the operator needs to understand what security the application needs the deployment environment to provide, what settings can affect security, and how to handle any error messages that have security impact. The operator also needs to know if a release fixes any vulnerabilities in previous releases.

Secure Release Management

When an application will be shipped, it should be built on a limited-access build server and packaged and distributed in such a way that the recipients can verify it is unchanged. Depending on the target platform, this may mean code signing or distributing signed checksums with the binaries.

Dependency Patch Monitoring

Any application that includes third-party code should monitor that external dependency for known security issues and updates, and issue a patch to update the application when any are discovered.

Product Security Incident Response

Like operational security incident response (as described in Chapter 33), product security incident response includes contacting people who should help respond, verifying and diagnosing the issue, figuring out and implementing a fix, and possibly managing public relations. It does not usually include forensics.

Decisions to Proceed

Any decision to ship an application or continue its development should take security into account. At ship time, the relevant question is whether the application can be reasonably expected to meet its security objectives. Frequently, this means that security validation activities have occurred and no critical or high-severity security issues remain open. Decisions to continue development should include some indicator of expected security risk, so that business stakeholders can draw conclusions regarding the expected business risk.

Web Application Security

This section covers web application security, including the vulnerabilities attackers can exploit in insecure web applications in order to compromise a web server or deface a web site, and how developers can avoid introducing these vulnerabilities.

There are several web application security concerns to be considered:

- SQL injection
- Forms and scripts
- Cookies and session management
- General attacks

NOTE In this chapter, the term *server-side scripts* refers to any available server-side programming technology, such as Java, ASP (Active Server Pages), PHP (PHP: Hypertext Preprocessor), or CGI (Common Gateway Interface).

SQL Injection

SQL (Structured Query Language) is standardized by the American National Standards Institute (ANSI) and serves as a common language for communicating with databases. Every database system adds some proprietary features to the basic ANSI SQL.

SQL injection is a technique to inject crafted SQL into user input fields that are part of web forms—it is mostly used to bypass custom logins to web sites. However, SQL injection can also be used to log in to or even to take over a web site, so it is important to secure against such attacks.

Simple Login Bypass

The most basic form of SQL injection is bypassing a login to a web site. Consider the following example, where the victim web site has a simple login form (see Figure 26-2):

```
...
<form action="login.asp" method="post">
<p>Username:<input type=text name="username" /></p>
<p>Password:<input type=password name="password" /></p>
<p><input type=submit name="submit" value="login" /></p>
</form>
...
```

This page requests two pieces of information from the user (username and password), and it submits the information in the fields to login.asp. The login.asp file looks like this:

```
dim adoConnection
set adoConnection=Server.CreateObject("ADODB.Connection")
...
dim strLoginSQL
strLoginSQL="select * from users where username=" &
Request.Form("username") & "' and password='" & Request.Form("password")
& "'"
dim adoResult
set adoResult=adoConnection.Execute(strLoginSQL)
If not adoResult.EOF Then
        'Everything went OK
Else
        'Login incorrect
End If
```

This script takes the entered username and password and places them into a SQL command that selects data from the users table based on the username and password. If the login is valid, the database will return the user's record. If not, it will return an empty record.

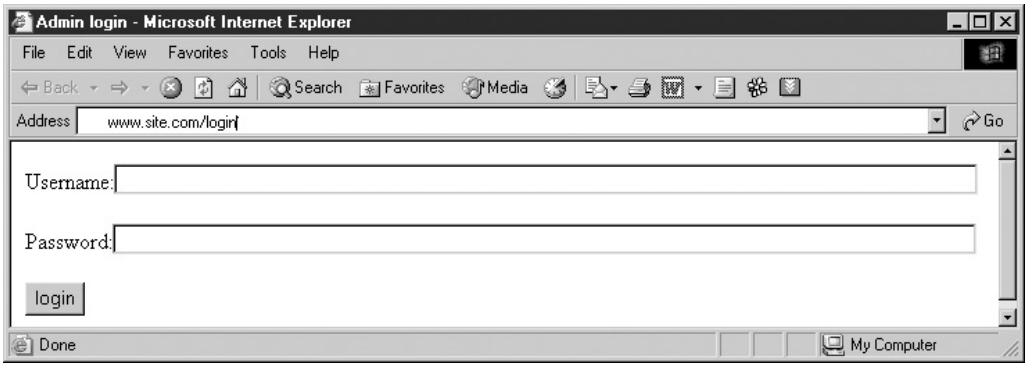

Figure 26-2 A typical login form for a web site

NOTE SQL injection is demonstrated here with ASP and ADO (ActiveX Data Objects), but it's a general problem that is not limited to these technologies.

The following SQL statement is built when a user enters **admin** as the username and **somepassword** as the password (as shown in Figure 26-3):

```
select * from users where username='admin' and password='somepassword'
```

Let's go over the query:

- `select` * means "give me all the data"
- `from users` means "take it from the table called *users*"
- `where username='admin' and password='somepassword'` means "find a row where both the username is *admin* and the password is *somepassword*"

The username and password are placed inside the SQL string without any sanity checks. (Sanity checks, known as "form field validation," are performed to make sure user input doesn't contain any characters an attacker could use to modify the SQL statement.) This means that an attacker can inject custom code into the user input fields without being detected.

In this case, the attacker will enter `'a' or "1"="1"` for the username, and any password at all, because it will be ignored (see Figure 26-4). The resulting SQL looks like this:

```
select * from users where username='a' or "1"="1"--' and
password='whatever'
```


The `--` stands for a code remark, which means that everything that follows will be disregarded (for example, the trailing apostrophe (`'`) will be ignored). This SQL phrase will always return data because `"1"="1"` is always true. The server will have to evaluate the statement "false and false or true," and because it will evaluate the "and" statement first, it'll become "false or true," which is true—the attacker will get access into the system.

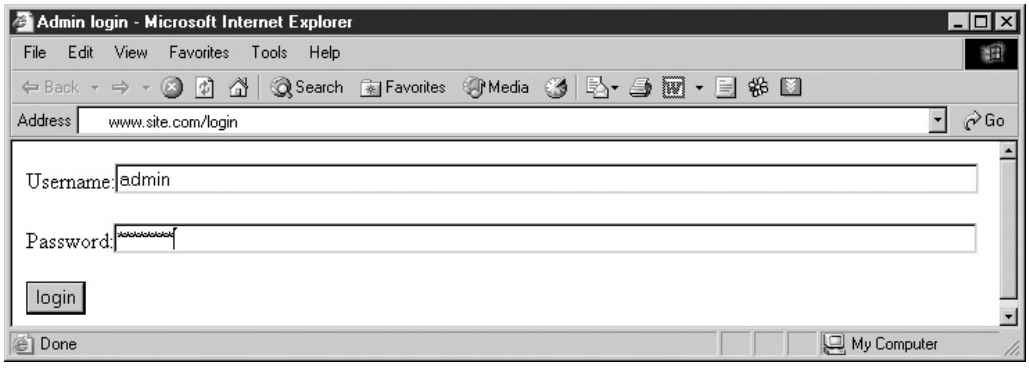

Figure 26-3 A user signing in using the login web form

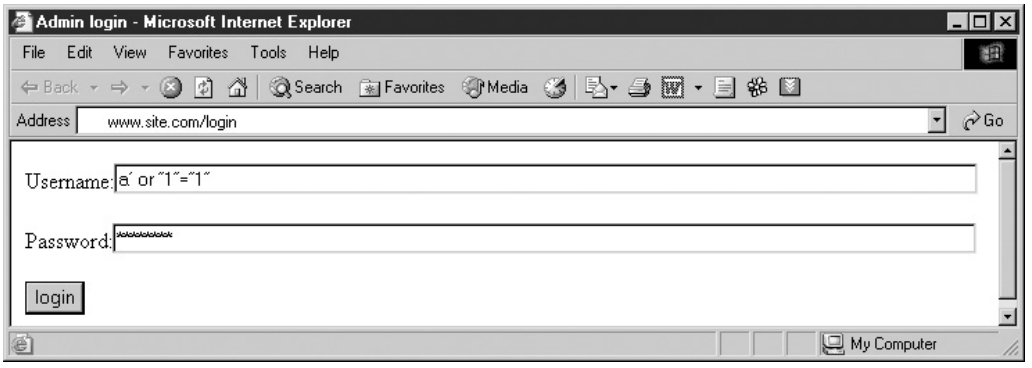

Figure 26-4 An attacker attacking the login web form with SQL injection

This attack was made possible because the programmer didn't filter the apostrophe (') inside the user input fields, which allowed the attacker to break the SQL syntax and enter custom code.

The following code solves this problem by filtering the apostrophes (every occurrence of ' in the user input is removed):

```
strLoginSQL="select * from users where username='" & Replace
 (Request.Form("username"),"'","") & "' and password='" & Replace
 (Request.Form("password"),"'","") & "'"
```


When SQL Injection Goes Bad

The previous example was very straightforward, but sometimes the SQL phrase is not so simple. Most login scripts check information in the user record: can the user log in, what is the level of subscription, and so on. A typical SQL login phrase can look like this:

```
Select * from users where username='someusername' and
password='somepassword' and active=1 and administrator=1
```

This SQL phrase looks for users that are also administrators and are active; the SQL in the previous example simply identified a user and didn't pay attention to whether the user was active or an administrator.

Attackers don't usually know the exact format of the SQL phrase (unless they managed to view the server-side script using a web server exploit), so they need to submit bad SQL in order to gain more information. For example, an attacker might submit **someusername** for the username and **a'aaa** (or any other value that isn't part of the SQL syntax) for the password. Because the resulting SQL is invalid, it will be rejected by the SQL server, which will send back an error that may look like this:

```
 [Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error (missing
operator) in query expression 'username=''' AND password='a'aaa and
active=1 and administrator=1'.
/login.asp, line 25
```

Now the attacker can see the SQL phrase and can craft better input, like **someusername** for the username and **'a' or '3'='3'** for the password, which will be interpreted like this:

```
Select * from users where username='someusername' and password='a' or
'3'='3' and active=1 and administrator=1
```

Procedure Invocations and SQL Administration

The attacker can use built-in stored procedures (functions supplied by the database to perform administrative and maintenance tasks) to write or read files, or to invoke programs in the database's computer. For example, the `xp_cmdshell` stored procedure invokes shell commands on the server's computer, like `dir`, `copy`, `move`, `rename`, and so on. Using the same scenario from the previous section, an attacker can enter **someusername** as the username and **a' exec master..xp_cmdshell 'del c:\winnt\system32*.dll'** as the password, which will cause the database to delete all DLLs in the specified directory. Table 26-1 lists some stored procedures and SQL commands that can be used to further elevate an attack.

Solutions for SQL Injection

Developers and administrators can take a number of different steps in order to solve the SQL injection problem.

These are some solutions for developers:

- Filter all input fields for apostrophes (') to prevent unauthorized logins.
- Filter all input fields for SQL commands like `insert`, `select`, `union`, `delete`, and `exec` to prevent server manipulation. (Make sure you do this after filtering for the apostrophes.)
- Limit input field length (which will limit attackers' options), and validate the input length with server-side scripts.

Stored Procedure	Usage
`xp_cmdshell`	Executes shell commands on the database computer's operating system
`xp_sendmail`	Sends an e-mail from the database's computer

`xp_regaddmultistring`
`xp_regdeletekey`
`xp_regdeletevalue`
`xp_regenumkeys`
`xp_regenumvalues`
`xp_regread`
`xp_regremovemultistring`
`xp_regwrite`
	Controls aspects of registry administration

`xp_servicecontrol`	Starts, stops, and pauses services. Can be used by an attacker to stop critical services or activate services that can be exploited, like the Telnet server service.

Table 26-1 Common SQL Server Stored Procedures That Are Abused by Attackers

- Use the option to filter "escape characters" (characters that can be used to inject SQL code, such as apostrophes) if the database offers that function.
- Place the database on a different computer than the web server. If the database is hacked, it'll be harder for the attacker to reach the web server.
- Limit the user privileges of the server-side script. A common practice is to use the administrative user when logging in from the server-side script to the database, but this can allow an attacker to run database tasks (such as modifying tables or running stored procedures) that require the administrative user. Assign a user with minimal privileges for this purpose.
- Delete all unneeded extended stored procedures to limit attackers' possibilities.
- Place the database in a separate container (behind a firewall), separated from the web container and application server.

Unlike developers, the administrator has no control over the code and can't make changes on behalf of the programmers. However, the administrator can mitigate the risks by running some tests and making sure that the code is secure:

- Make sure the web server returns a custom error page. This way, the server won't return the SQL error, which will make it harder for the attacker to gain data about the SQL query. (A custom error page should not contain any information that might aid the attacker, unlike the regular error page, which will return part of the SQL statement.)
- Deploy only web applications that separate the database from the web server.
- Hire an outside agency to perform penetration tests on the web server and to look for SQL injection exploits.
- Use a purpose-built automated scanning device to discover SQL injection exploits that result from programmers' mistakes.
- Deploy security solutions that validate user input and that filter SQL injection attempts.

Forms and Scripts

Forms are used to allow a user to enter input, but forms can also be used to manage sessions (discussed in the "Cookies and Session Management" section, later in this chapter) and to transfer crucial data within the session (such as a user or session identifier). Attackers can exploit the data embedded inside forms and can trick the web application into either exposing information about another user or to charge a lower price in e-commerce applications. Three methods of exploiting forms are these:

- Disabling client-side scripts
- Passing parameters in the URLs
- Passing parameters via hidden fields

Client-Side Scripts

Some developers use client-side scripts to validate input fields in various ways:

- Limit the size of the input fields
- Disallow certain characters (such as apostrophes)
- Perform other types of validation (these can be specific to each site)

By disabling client-side scripting (either JavaScript or VBScript), this validation can be easily bypassed. A developer should validate all fields at the server side. This may require additional resources on the server.

Passing Parameters via URLs

A form has two methods of passing data: `post` and `get`. The `post` command sends the data in the content stream and the `get` command sends the data in the URL. Attackers can exploit the `get` command to send invalid or incorrect data, or to send malicious code.

For example, suppose we have this kind of form:

```
...
<form action="login.asp" method="get">
<p>Username:<input type=text name="username" /></p>
<p>Password:<input type=password name=password /></p>
<p><input type="submit" name="submit" value="login" /></p>
</form>
...
```


Let's assume the user enters **someusername** as the username and **somepassword** as the password. The browser will be redirected to this URL:

http://thesite/login.asp?username=someusername?password=somepassword

An attacker can exploit this type of URL by simply modifying the URL's data (in the browser's address bar). This method can be used in e-commerce sites to change the prices of items. For example, look at the following URL:

http://somesite/checkout.asp?totalprice=100

The attacker could simply change the value of "totalprice" and perform a checkout that has a lower price than was intended. This can be done simply by changing the URL like this:

http://somesite/checkout.asp?totalprice=50

The web application will perform the checkout, but with $50 as the total price (instead of $100).

Another scenario is that, after the login, the user identification is sent using `get`, allowing an attacker to modify it and perform actions on another user's behalf. An example is shown in the following URL:

http://somesite/changeuserinfo.asp?user=134

The attacker could change the value of "user" and get the data of that user, if the user exists.

Passing Data via Hidden Fields

The post method sends the data using the `POST` HTTP command. Unlike get, this method doesn't reveal the data in the URL, but it can be exploited rather easily as well. Consider the following form:

```
...
<form action="checkout.asp" method="post">
<input type="hidden" name="UserID" value="102" />
<p><input type="submit" name="submit" value="checkout" /></p>
</form>
...
```

This form transmits the user identifier using `POST`. An attacker can save the HTML, modify the `UserID` field, modify the checkout.asp path (to link to the original site, like this: `<form action="http://example/checkout.asp"...`), run it (by double-clicking on the modified local version of the HTML page), and submit the modified data.

Solving Data-Transfer Problems

The developer can prevent attackers from modifying data that is supposed to be hidden by managing the session information, by using GUIDs, or by encrypting the information.

Managing Session Information Most server-side scripting technologies allow the developer to store session information about the user—this is the most secure method to save session-specific information because all the data is stored locally on the web server machine.

Using GUIDs A *globally unique identifier*, or *GUID*, is a 128-bit randomly generated number that has 2^{128} possible values. GUIDs can be used as user identifiers by the web application programmer. Assuming a web server has 4 billion users (about 2^{32}, which is more than the number of people who have Internet access), this means there are on average 2^{96} possible values per user ($2^{128}/2^{32} = 2^{96}$). Since 2^{96} is approximately 7 followed by 28 zeros, the attacker will have no chance of guessing, and thus accessing, a correct GUID.

Encrypting Data The developer can pass encrypted data rather than passing the data in cleartext. The data should be encrypted using a master key (a symmetric key that is stored only at the web server, and used to store data at the client side). If an attacker tries to modify the encrypted data, the client will detect that someone has tampered with the data.

NOTE Never use a derivative of the user's information as a hidden identifier, such as an MD5 hash of the username. Attackers will try to find such shortcuts and exploit them.

Cookies and Session Management

Web *sessions* are implemented differently by each server-side scripting technology, but in general they start when the user enters the web site, and they end when the user closes the browser or the session times out. Sessions are used to track user activities, such as a user adding items to their shopping cart—the site keeps track of the items by using the session identifier.

Sessions use cookies (data sent by the web site, per site or per page, stored by the user's browser). Each time the user visits a web site that sent a cookie, the browser will send the cookie back to the web site. (Although cookies can be used to track users' surfing behavior and are considered a major privacy threat, they are also the best medium for session management.) Sessions use cookies to identify users and pair them with an active session identifier.

Attackers can abuse both sessions and cookies, and this section will deal with the various risks:

- Session theft
- Managing sessions by sending data to the user
- Web server cookie attacks
- Securing sessions

Session Theft

Suppose that a user logs in to a web site that uses sessions. The web site tags the session as authenticated and allows the user to browse to secure areas for authenticated users. Using `post` or `get` in order to save a weak session identifier or other relevant identifying data (such as e-mail addresses) is not the best choice. Instead, the web site can use cookies in order to save sensitive data, but an attacker can exploit this as well. Server-side cookies are another alternative.

Let's assume the web site uses e-mail addresses as the identifying data. After the user has logged in, the system will send the browser a cookie containing the user's e-mail address. For every page this user will visit, the browser will transmit the cookie containing the user's e-mail address. The site checks the data in the cookie and allows the user to go where their profile permits.

An attacker could modify the data in the cookie, however. Assume the cookie contains someemail@site.com, and each time we access the site we can automatically access restricted areas. If the attacker changes the e-mail address in his cookie (located on his computer) to be **someotheremail@site.com**, the next time the attacker accesses the site, it will think he is the user someotheremail and allow him to access that user's data.

NOTE Amazon saves user information in a cookie, and allows users to see their recent activities (without logging in). However, Amazon encrypts the content of the cookie, making it harder for an attacker to hijack a session.

Managing Sessions Without Sending Data to the User

Some users disable cookies (to protect their privacy), which means they also don't allow session management (which requires cookies). Unless the site is using the less secure get or

post methods to manage sessions, the only way to keep track of users is by using their IP address as an identifier. However, this method has many problems:

- Some users surf through Network Address Translation (NAT), such as corporate users, and they will share one or a limited number of IP addresses.
- Some users surf through anonymous proxies, and they will share this proxy IP address (though some proxies do send the address of the client, thus allowing the web site to use it for session management).
- Some users use dial-up connections and share an IP address pool, which means that when a user disconnects, the next connected user will get that IP address. (This problem can be solved with a short IP timeout, so that after the time expires, the IP address will not be linked to a session.)

NOTE Don't be afraid to require cookies on your site in order to perform actions that require session tracking—remember, your web site's security comes first.

Securing Session Tracking

The best way to secure session tracking is to use a hard-to-guess identifier that is not derived from the user's data, such as an encrypted string or GUID, and to tie this identifier to the IP address of the user. In cases where multiple users share a single IP address, the session identifier can be used to distinguish them.

In addition, a short timeout can be used to delete an active session after the time limit has elapsed. This means that if the user doesn't close the browser gracefully (as in the case of a computer or browser crash), the session is closed by the server.

Web Server Cookie Attacks

An attacker can exhaust the resources of a web server using cookie management by opening many connections from dedicated software. Since this software will not send "close" events as a browser does when it is closed, the session will not be deleted until a timeout elapses. During this time, the session's information is saved either in the memory or in the hard drive, consuming resources.

The solution to this problem is to configure a firewall so that it does not allow more than a particular number of connections per second, which will prevent an attacker from initiating an unlimited number of connections.

General Attacks

Some attacks aren't part of any specific category, but they still pose a significant risk to web applications. Among these are vulnerable scripts, attempts to brute-force logins, and buffer overflows.

Vulnerable Scripts

Some publicly used scripts (which are essentially the same as web applications) contain bugs that allow attackers to view or modify files or even take over the web server's computer. The best way to find out if the web server contains such scripts is to run a vulnerability

scanner, either freeware or commercial. If such a script is found, it should be either updated (with a non-vulnerable version) or replaced with an alternative script.

Brute-Forcing Logins

An attacker can try to brute-force the login (either a standard web login or a custom ASP) using a dictionary. There are a number of ways to combat brute-force attacks:

- Limit the number of connections per second per IP address (define this either at the firewall level or at the server-side script level)
- Force users to choose strong passwords that contain upper- and lowercase letters and digits

Buffer Overflows

Buffer overflows can be used to gain control over the web server. The attacker sends a large input that contains assembly code, and if the script is vulnerable, this string is executed and usually runs a Trojan that will allow the attacker to take over the computer.

Web Application Security Conclusions

Web applications are harder to secure than client applications because, unlike web servers that have four or five major vendors, there are a huge number of web applications and custom scripts, and each may contain a potential exploit. The best way for developers to secure their applications is to use the proposed security measures and use software that scans code and alerts you to potential security problems. Administrators need to periodically scan their web sites for vulnerabilities.

Client Application Security

Application security is mainly controlled by the developer of the application. The administrator can tighten the security for some applications, but if the application is not secure by nature, it's not always possible to secure it.

Writing a secure application is difficult, because every aspect of the application, like the GUI, network connectivity, OS interaction, and sensitive data management, requires extensive security knowledge in order to secure it. Most programmers don't possess this knowledge or don't consider the security of the application important enough to justify extra work.

From the administrator's point of view, there are a number of security issues to keep in mind:

- Running privileges
- Administration
- Application updates
- Integration with OS security
- Adware and spyware
- Network access

Running Privileges

An administrator should strive to run an application with the fewest privileges possible. Doing so protects the computer against several threats:

- If the application is exploited by attackers, they will have the privileges of the application. If the privileges are low enough, the attackers won't be able to take the attack further.
- Low privileges protect the computer from an embedded Trojan (in the application) because the Trojan will have fewer options at its disposal.
- When an application has low privileges, the user won't be able to save data in sensitive areas (such as areas belonging to the OS) or even access key network resources.

NOTE While developing an application, programmers tend to make assumptions in order to cut development time. Some of these assumptions result in applications that require administrative privileges to work. This may cut programming time, but it reduces the ability of the administrator to keep systems secure. When ordinary users are given administrative privileges, they can remove or go around security configurations, thus subverting any security that might be in place.

Installing Applications

When installing an application, it's usually necessary to have higher privileges or even administrative privileges, because the installer may need to access sensitive OS directories and make registry and hardware changes.

NOTE It's best to install the application on a testing computer that has a similar configuration to the actual computer that requires the installation. This way, you can see if any problems arise before installing the application on a live computer.

Circumventing Administrative Privilege Requirements

If an application requires administrative privileges but there is no obvious reason why it needs them, or if you just don't trust the application, you can run it within a sandbox. A sandbox is a security application that intercepts the system calls of the application that it is running and makes sure the application will have access only to the resources the administrator has allowed. Thus, sandboxes can limit access to the registry, OS data directory, and network usage. This isolates the application from sensitive OS areas and other user-defined locations, such as those containing sensitive data.

Application Administration

Most applications offer some type of interface for administration (mostly for application configuration), and each administration method poses security risks that must be addressed, such as these:

- INI/Conf file
- GUI
- Web-based control

INI/Conf Files

The most basic method of administrating an application is to control it via text-based files. To secure such an application, the administrator needs to limit access to the configuration files either by using built-in OS access management, if the files are stored locally, or by using authentication to log in to the remote storage place (making sure the authentication method is secured).

GUIs

Most applications have a GUI for administrating them. In addition to providing security at the GUI level, the administrator should provide security for the communications between the GUI and the application.

When the GUI is physically located on the same computer as the application, the administrator should give the GUI the least possible privileges (the application can run with higher privileges if necessary).

When the GUI controls a remote system, the most important issue is how the GUI controls the application; this topic will be discussed in the "Remote Administration Security" section of this chapter.

Web-Based Control

A popular way to allow application administration is via a web interface, which doesn't require a dedicated client and can be used from multiple platforms. Web interface remote administration is covered in the "Remote Administration Using a Web Interface" section of this chapter.

Integration with OS Security

When an application is integrated with OS security, it can use the security information of the OS, and even modify it when needed. This is sometimes required by an application, or it may be supplied as an optional feature. There are both advantages and disadvantages to OS security integration.

Importance of OS Security Integration

OS security integration allows an application to either import or access in real time the OS's list of users and their privileges. Imagine an organization with a thousand employees that need access to a central enterprise resource planning (ERP) application. The administrator could manually enter all the thousand users into the ERP's administrative console, along with their privileges, but this method is time consuming and will require double management afterward. If the organization has more than one central system that requires manual user entry, this scenario would be even worse.

Manual Import of Security Information

An application may allow the administrator to import all the user information and use it to manage authentication for the application. Although this method may speed up application deployment, there is still double administration afterward. For example, when an employee leaves the organization, the administrator has to delete the user both from the organization's user list and from the application list.

Another question to consider is how the application stores its user information. Is it protected? Encrypted? Stored in cleartext? If you don't trust your application's data storage security, you can encrypt the entire hard drive.

Automatic Integration of Security Information

Automatic integration of security information allows the application to query the OS in real time for user credentials. This way, both the initial deployment time and the double administrative issues are solved. There are two problems with this option, though:

- If the OS's user database is deleted or lost, the application can't be accessed.
- The network connection between the application and the OS user database must be secured to prevent attackers from either eavesdropping on the line or using a fake server to gain information about users' credentials.

Using OS Security for Authorization

An application can use OS security to authorize sessions. In this scenario, the application sets up a special directory or resource (like shared memory, a mail slot, or name pipes) that can be accessed only by users who possess certain privileges, and the OS protects access to that directory or resource.

Keeping OS Security Integration Optional

Sometimes it's necessary to deploy a small application that will be used by only one or two users—consider what will happen if the application forces us to integrate with the OS security (using one of the methods previously discussed) in a corporation with a thousand users. It will only decrease the security (if it uses an insecure method) and decrease deployment speed (because we have only one or two users). In addition, the administrator may be reluctant to give an application the ability to modify (and potentially damage) the user directory.

Application Updates

Keeping applications up to date with the latest security patches is one of the most important security measures that you can take.

This section covers some mechanisms for easily updating applications:

- Manual updates
- Automatic updates
- Semi-automated updates
- Physical updates

Manual Updates

Manual updates require the administrator to physically download a file (or use a supplied media, like a CD) and install the update on the relevant system. This option is the least preferable because it forces administrators to spend extra time to patch a working system. Manual updates are very common for open source programs (such as Apache).

Automatic Updates

When an application uses automatic updates, it checks with its web site every so often for an update, and if one exists, it downloads it and installs it on the system. There are two problems with this method:

- **Bandwidth usage** Consider an organization with a thousand computers that run the same antivirus software, which updates itself daily. Every day, a copy of the same update is downloaded to each of the thousand computers running this program.
- **Installing problematic patches** Sometimes patches (software updates released by the vendor to fix security problems and bugs) can cause more harm than good, because patches are made in a hurry to solve a critical issue. The developers can't foresee all possible environment scenarios, and the patch may stop the application or cause it to behave improperly. That's why testing is imperative.

Semi-Automated Updates

Some applications allow the administrator to decide when to download an update. After the update is downloaded, the application distributes the update to all the connected clients.

Physical Updates

It's possible to update the system using an update received physically. A motivated attacker can create a "fake" patch by forging an update that looks just like the original but contains a Trojan or other malicious software. To secure against this kind of attack, the administrator can check for the size and CRC32 signature of the update at the vendor's site and compare it to the physical copy.

Remote Administration Security

Most of today's applications offer remote administration as part of their features, and it's crucial that it be secure. If an attacker manages to penetrate the administration facilities, other security measures can be compromised or bypassed.

Reasons for Remote Administration

Remote administration is needed for various reasons:

- **Relocated servers** An administrator needs an interface to administer any relocated web servers (computers that belong to an organization but that are physically located at the ISP).
- **Outsourced services** Managing security products requires knowledge that some organizations don't possess, so they often outsource their entire security management to a firm specializing in that area. In order to save costs, that firm needs to manage all the security products through the Internet.
- **Physical distance** An administrator may need to manage a large number of computers in the organization. Some organizations span several buildings (or cities), and physically attending the computers can be a tedious and time-consuming task. Additionally, physical access may be limited to the actual data centers.

Remote Administration Using a Web Interface

Using a web interface to remotely administer an application or a computer has many advantages, but it also has its costs, and some advantages are also disadvantages.

These are some advantages of remote web administration:

- **Quick development time** Developing a web interface is faster than developing a GUI client, in terms of development, debugging, and deployment.
- **OS support** A web interface can be accessed from all the major OSs by using a browser (unless the developers used an OS-specific solution, like ActiveX, which only runs on Windows).
- **Accessibility** A web interface can be accessed from any location on the Internet. An administrator can administrate even if he's not in the office.
- **User learning curve** An administrator knows how to use a browser, so the learning curve for the administrator will be shorter.

Although remote web administration has some disadvantages, they are usually not critical for most administrators. However, they should be noted:

- **Accessibility** Because web administration is accessible from anywhere on the Internet, it's also accessible to an attacker who may try to hack it.
- **Browser control** Because a browser controls the interface, an attacker doesn't need to deploy a specific product control GUI (which might be hard to come by).
- **Support** Web-based applications are typically easier to support and maintain.

Authenticating Web-Based Remote Administration

When connecting to the remote web administration interface, the first hurdle to clear is the authentication process. If the authentication is weak, an attacker can bypass it and take control of the application or computer.

HTTP Authentication Methods

Before delving into the problem of remote administration, it's important to go over the current methods available to authenticate HTTP connections:

- **Basic authentication** When a page requires basic authentication, it replies to the browser with error code 401 (unauthorized) and specifies that basic authentication is required. The browser encodes the username and password using BASE64 encoding and sends it back to the server. If the login is correct, the server returns message number 200, which means everything is OK. If the login fails, it replies with the same 401 error as before.
- **Digest authentication** Digest authentication uses MD5 to hash the username and password, using a challenge supplied by the web server.
- **Secure Sockets Layer (SSL)** SSL can be configured to require a client certificate (optional) and authenticate a user only if they have a known certificate.

- **Encrypted basic authentication** Basic authentication can be used in conjunction with regular SSL, thus encrypting the entire session, including the BASE64 encoded username and password (which is very weak encoding, easy to decode—this is not encryption).
- **CAPTCHA** This is a popular method of verifying that the person on the other end is a human being, by showing a distorted image of letters and numbers and requiring the user to type them in correctly.

Securing Web-Based Remote Administration

The best solution for securely logging in to a web-administered server is to use either SSL, which checks for client certificates, or encrypted basic authentication. (SSL can also authenticate the server against a third-party certificate authority to ensure it is the server you meant to connect to.) Another option is to use secured custom logins (implemented with server-side scripts), but they may contain web exploits.

Custom Remote Administration

Some applications are controlled remotely via a GUI or through console applications, such as SQL Server, Exchange Server, firewalls, and intrusion detection systems (IDSs). An application may also control clients with probes, as an IDS does. Proprietary network connections have a few security issues that need to be addressed (as web connections do). Just like remote web administration, custom remote administration has both advantages and disadvantages.

Advantages and Disadvantages

These are the advantages of custom remote administration:

- **Complex graphics** Sometimes the console needs to display complex graphics that can't be shown using a regular web administration interface.
- **Authentication and encryption** The application may use either a stronger authentication method or a stronger encryption method to secure the session (perhaps using a greater key length that isn't supported by SSL).
- **Availability** Since the application can only be controlled from a dedicated GUI, the attacker will need to install it at his computer (and accessing or installing it may not be possible).

Although custom remote administration has some disadvantages, they usually are not critical for most administrators. However, they should be noted:

- **Specific OS** Some vendors will require a specific OS to run the controlling GUI, and the administrator will have to install it if it isn't already installed (this may involve additional costs if the OS is not free).
- **Unavailability** The application can be administered only from computers on which the GUI is installed, and if the administrator is not in the office, it may not be possible to administer it from other computers.

Session Security

It's important that the session between the client (GUI or console) and the application be secure. Otherwise, attackers may be able to gain information, steal credentials, or even conduct a replay attack. If the session is known to be insecure, the administrator can easily relay it through a VPN or a secure tunnel (SSH).

Authentication

It's important that authentication take place and that it isn't based upon easily forged assumptions, like the IP or MAC address of the computer.

The sequence of the authentication process is also critical: Is the session secured before the credentials are sent? Are the credentials sent in cleartext format? The best way to exchange login information is either after the session is secured, or using a known method like EAP for insecure sessions.

Using OS Networking Services

Some applications use OS networking services, such as remote procedure calls (RPC) or Distributed Component Object Model (DCOM), which allows the administrator to add data integrity, encryption, and authentication. If you don't trust the OS security measures, you can tunnel the network connection through a VPN connection.

To conclude, just like web application connectivity, we can't force an application to communicate securely if it doesn't support the option. The solution is either to use a VPN or to tunnel the data session through a secure session (SSH).

Summary

Unlike network security, which uses devices like vulnerability scanners, firewalls, and IDSs and relies on processes like patching to compensate for security vulnerabilities that pre-exist in the technologies on the network, application security needs to be done right from the start because it's much harder to actively fix security problems in the field than it is to do so in the programmer's chair. Training, corporate standards, reviews at the design phase, and formal code reviews can all help ensure that security is integrated from the beginning in any new application.

Every programmer who isn't focused on security when writing an application, whether web-based or client, can leave the application vulnerable to outside attackers. Because application security problems primarily result from human errors and omissions (on the part of the programmers), the best solution is education.

To produce an application that is secure enough, define "secure enough" near the beginning of the development process. Keep this definition in mind when you construct each deliverable. As each deliverable is completed, check it for security issues. At the end of the development process, ship it only if the application meets your definition of secure enough.

References

Clarke, Justin, et al. *SQL Injection Attacks and Defense*. 2nd ed. Syngress, 2012.

Davis, Noopur. *Secure Software Development Life Cycle Processes: A Technology Scouting Report*. Software Engineering Institute, Carnegie Mellon University, 2005.

Heiderich, Mario, et al. *Web Application Obfuscation*. Syngress, 2010.

Hope, Paco, and Ben Walther. *Web Security Testing Cookbook: Systematic Techniques to Find Problems Fast*. O'Reilly, 2008.

Howard, Michael, David LeBlanc, and John Viega. *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. McGraw-Hill, 2009.

Scambray, Joel, Vincent Liu, and Caleb Sima. *Hacking Exposed Web Applications*. 3rd ed. McGraw-Hill, 2010.

Shema, Mike. *Seven Deadliest Web Application Attacks*. Syngress, 2010.

Stuttard, Dafydd, and Marcus Pinto. *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*. Wiley, 2007.

Sullivan, Bryan, and Vincent Liu. *Web Application Security, A Beginner's Guide*. McGraw-Hill, 2011.

Zalewski, Michal. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2011.