SECURING PHP WEB APPLICATIONS

Easy, Powerful Code Security Techniques for Every PHP Developer

TRICIA BALLAD
WILLIAM BALLAD
This chapter covers the need for encryption, its importance in data security, and what can happen if it fails or if encryption of vital data isn’t implemented. We will revisit the code from Chapter 7, “Authentication,” and show you how to better secure the application.

What Is Encryption?

Encryption is the process of transforming information into something that is unreadable to anyone not possessing special knowledge. This transformation requires two crucial pieces of data: the cipher and the key. In the world of programming, the cipher is an algorithm. The special knowledge you must have to read the encrypted data is called the key. There are several ciphers, or encryption algorithms, that are available for you to use in your own application.

There are two major types of encryption: symmetric key and asymmetric or public key. Each type has multiple variations, each with its own strengths and weaknesses. We will try to help you understand when to use either type. As of PHP 6.0, PHP supports symmetric and asymmetric key encryption natively.

In a public key encryption scheme, there are two keys. One is kept private by the receiver; this is used to decrypt the message. The other key is supplied by the receiver to the sender; this is the public key and is used to encrypt the message. Only someone with the matching private key can then decrypt what is sent. The sender and the receiver have different keys. That is what makes this form of encryption asymmetric. This method is very good when you have lots of senders, such as with e-mail or for
digital signatures and SSL. These methods of encryption are not natively implemented in PHP until PHP 6.0, but you can add extensions to add SSL or call some public key ciphers as external functions. Figure 8.1 shows how public key encryption works.

In symmetric key encryption both the sender and the receiver share a key. This key is then used by the algorithm to encrypt or decrypt the information. The major drawback of this method is key management. Everyone who needs to decrypt the message must have the key, and all must remember which key is for which message. This method is very useful for encrypting data that another application will read or in situations where the sender and receiver are static. If you are in a situation where there will be multiple users of the key, this method is not ideal. Figure 8.2 shows how symmetric encryption works.

**Figure 8.1** Diagram of asymmetric encryption.

**Figure 8.2** Diagram of symmetric encryption.
There is also a useful variant of symmetric encryption called one-way encryption, where you encrypt the message with no intention of ever decrypting it. Figure 8.3 shows this type of one-way encryption.

One-way encryption can be used in password situations where two pieces of information match when encrypted. We will look at one form of symmetric encryption that involves using large hash tables. This is very useful for data integrity checking because any minor change in an object will cause a large change in the resulting hash.

**Choosing an Encryption Type**

When you are trying to decide how to secure your data, there are a few main points to consider:

- Algorithm strength
- Application speed versus data security
- Use of the encrypted data

In the following sections, we’ll look at each in a bit of detail.

**Algorithm Strength**

There are many algorithms to choose from. The PHP built-in `mcrypt()` function has over 20 different encryption options, and there are third-party libraries that add even
more. This can be rather bewildering, so it’s important to remember that key length and predictability of the algorithm determine its strength. That simply means the longer the key (the more bits it uses), the longer it will take someone to break it. But there is a stipulation. If the algorithm is predictable, the number of guesses needed to break the encryption can be greatly reduced. No one expects you to keep up with all of the cryptology news as to which method is easier to crack. Unless you’re one of those people who does calculus for fun, you probably have more interesting things to do. As long as you stick with the newest algorithms, you should be OK. Currently, 3DES, AES, and Blowfish are our recommendations.

For hashing, the PHP implementations of MD5 and SHA1 will work, but be aware that MD5 can be compromised. If you need a strong hash you may need to look at a third-party implementation.

On occasion, especially in older easy security guides, XOR or ROTX will be mentioned. These are bit manipulations that can make the data look encrypted, but they are very basic and easily guessed. If you are trying to secure your data, do not use these. They are both examples of data obfuscation as opposed to true encryption.

**Speed versus Security**

The question to ask yourself concerning this issue is “How secure does my data need to be?” The bigger the key, the longer it will take to encrypt and decrypt the data. This can cause a noticeable slowdown in the time it takes your application to load and process data. If you are looking at data that needs to be very secure, you may want to use multiple methods of encryption.

A big part of addressing this issue comes down to what data is being encrypted and why. Do you just want to keep the casual user from viewing the text, or are you trying to secure the information from determined attackers? If it’s a question of casual observation, you may be able to get away with obfuscation instead of encryption. Another aspect of this is simply the likelihood of viewers. If it’s a closed system, data security may be handled by physical security. For example, if the data is being stored on a server with no connection whatsoever with the outside world, it may be enough to simply lock the server room and monitor who has physical access to the server. You may not even need encryption in this scenario.

**Use of the Data**

Ask yourself this question: “How is the data going to be used?” Something like a password that needs to be secret and verified works well with hashing. Are you looking to send or receive the information from a third party? If so, asymmetric encryption may
be the way to go. If your application will be encrypting and decrypting the information, then symmetric encryption would be best.

**Password Security**

In Chapter 7, “Authentication,” we discussed the importance of choosing a strong password. Although this is important, it is not the only thing that needs to be done to secure your users' logins. If either your database or flat file is compromised, plain-text passwords will be exposed to the attacker. To truly secure passwords we need to encrypt them.

Let’s look again at our three criteria for choosing an encryption type, but this time in the context of our example application. This is a publicly accessible system so we need a strong algorithm, but it is just a guestbook so we don’t need to go nuts. Nothing like a credit card or Social Security number is getting stored. The consequences of a data breach are fairly minor—a user could get locked out of his or her account, or someone could post a comment to the guestbook under another user’s name. All told, the worst-case scenario really isn’t a crisis situation, just a hassle.

We need the algorithm to work very quickly, as this is a Web application. No one is willing to wait to get to our page. The data is going to be a password, not something we will ever need to decrypt. If the user forgets his or her password, we will just initiate the process of creating a new one.

Knowing these things, we will choose the MD5 hash to encrypt our passwords. MD5 can be compromised, but that still takes a significant amount of time. MD5 is quick, easy to implement, and secure enough for our purposes. If your situation calls for more security, SHA1 will work as well, or implement SHA2 with a third-party library. No matter what you implement, if you need a strongly secured password, you need to have a password retention policy. A six-month or shorter mandatory password life will greatly reduce the chances that someone can brute-force the password.

**Patching the Application to Encrypt Passwords**

Adding encryption to user authentication in the guestbook application will happen in three steps:

1. Modify the user table in the database.
2. Create the encryption and salting functions.
3. Modify the password validation.
Breaking the task into discrete steps helps ensure that we can consider each part of the problem carefully and avoid introducing security holes into our application. The salting function is used to introduce an element of randomness into the encryption. Without it, anyone who knows the username and password could generate the same encrypted string as our encryption function. Adding salt to the algorithm is an easy way to make the system more secure.

MODIFYING THE USER TABLE

We need to add a column to the user table. The new column will hold a random number used to encrypt the password. Table 8.1 outlines the characteristics of this new field.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>NULL?</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>salt</td>
<td>Varchar(30)</td>
<td>No</td>
<td> </td>
</tr>
</tbody>
</table>

Once we’re finished with the database, we’ll tackle the application code.

CREATE THE ENCRYPTION AND SALTING FUNCTIONS

Next, we’ll create a very simple function that encrypts the password. We’re making the assumption that the password has already been through data validation by the time it gets to the encryption function, so we’re not going to worry about that. This function is very simple, yet powerful enough for our purposes. First, we concatenate the username, salt, and password into a simple plain-text string. Then we pass that string through the built-in `md5()` function and return the results. It’s really that simple.

```php
function encryptPassword($plaintext_password, $username, $salt) {
    // At this point we can assume that the plaintext_password has already
    // been through validation, so there's no need to worry about tainting
    $str = $username.$salt.$password;
    return md5($str);
}
```

To generate the salt for our encryption algorithm, we simply return a random number between 0 and 1,028.
function createSalt() {
    return rand(1028);
}

**Modify the Password Validation System**

The final step in encrypting the passwords in our guestbook application is to make a few minor modifications to the existing password and login system. First, we rewrote the password function to pass the plain-text password through our new encryptPassword() function.

function password($plaintext_password = NULL) {
    if($plaintext_password) {
        $this->_password = encryptPassword($this->_username, $this->_salt, $plaintext_password);
    }
    return $this->_password;
}

Then we used the createSalt() and encryptPassword() functions in our login function as well.

function login($username, $plaintext_password) {
    $dbh = getDatabaseHandle();
    $selected_db = mysql_select_db("guestbook", $dbh);
    $sql = "select username, password from Users where username = "$username";
    $result = mysql_query($sql, $dbh);
    $userinfo = mysql_fetch_array($dbh);
    $salt = createSalt();
    $password = encryptPassword($userinfo['password'], $salt, $plaintext_password);
    if($userinfo['password'] == $password) {  // User is authenticated
        $user = new User($username);
        $user->_sessionID = _generateSessionID(); // Also stores // sessionID in DB
        return $user;
    } else {
        return FALSE;
    }
}
WRAPPING IT UP

In this chapter, we covered the need for encryption. We discussed how to decide on the right type of encryption for your application by understanding your data, and we covered a very common encryption scenario. This is a good start and should be enough to get you up and running with your own applications, but it is just a quick overview. Encryption and cryptography are huge topics that would require their own book to cover in depth. If you plan to store sensitive data, such as credit card numbers or Social Security numbers, we highly recommend that you familiarize yourself with encryption more thoroughly by reading one (or more) of the books listed in the Appendix, “Additional Resources.”
In this chapter, we cover session security. We look at what a session variable is and why it is used, then show you how to defend against the three major types of session attacks: hijacking, fixation, and injection.

What Is a Session Variable?

HTTP is stateless by design. This has some advantages but leaves us with a major problem when dealing with dynamic Web pages. How do we maintain a user’s identity across multiple pages? How do we pass data from page to page? This is where session variables come in; they enable you to track session information about the user through various pages on your site. PHP sessions are like server-side cookie files. Each one stores variables that are unique to the user request that created it and ideally can be accessed only on subsequent requests from that user. Of course, hackers try to turn this functionality into a vulnerability to gain access to resources. Therefore, there are session attacks that you must attempt to counter.

Major Types of Session Attacks

There are three types of attacks that you need to be wary about when using session variables:
• Session fixation
• Session hijacking
• Session poisoning (injection)

Luckily, there are some clear ways to defend against these attacks. It all comes down to session management.

It is also important to note that in a shared server environment anyone with access to the server can access the PHP session files. These people will not be able to identify what Web site each session belongs to, but they can get sensitive information out of the variables. It is very important not to store critical information in session variables because they simply aren’t secure enough to safeguard it. If you have sensitive data that must be passed around your site, store it in the database. This method is slower than storing data in the session, but it is significantly more secure.

**Session Fixation**

Session fixation is simply a method of obtaining a valid session identifier without the need to predict or capture one. It enables a malicious user to easily impersonate a legitimate user by forcing the session ID. It is the simplest and most effective method for a malicious user to obtain a valid session ID.

The attack itself is very basic. The hacker forms a link or redirect that sends the user to your site with the session ID preset:

```
<a href=http://YOUR_HOST/index.php?PHPSESSID=1234> Click here </a>
```

When users click on that link or are redirected there, they connect to your site with a session ID that has been set by the attacker. The attacker can now wait for the users to log in and access your site using their credentials, as shown in Figure 9.1.

PHP has a very good defense for this type of attack in the built-in `session_regenerate_id()` function. This function generates a new session file for the user, gets rid of the old one, and issues a new session cookie if your site utilizes them. Anytime your users get their credentials challenged, say at login or when they are changing their password, it’s a good idea to run `session_regenerate_id`. This will greatly mitigate fixation attacks.

Another good tool for dealing with session fixation is to make sure you set a session time-out in the php.ini file. For more information on this, see Chapter 13, “Securing PHP on the Server.”
These methods are not a 100 percent guarantee that an attacker can’t get your users’ session IDs. Hackers could get very lucky and guess a valid ID, or they could snoop it off the network. Guessing isn’t very likely because of the way PHP assigns session IDs. To defend against network snooping, you could use SSL/TSL. This does add a lot of overhead to your site, so you need to determine how secure your site needs to be. You may also want to make sure that you challenge users when they access very sensitive material, or that you do not fully display sensitive data such as credit card numbers.

**Session Hijacking**

After a successful session fixation attack, a malicious user has your user’s session. What does the attacker do with it? This is where session hijacking comes in. In a hijacking attack, the malicious user tries to access your site utilizing a valid session ID, as shown in Figure 9.2.

Obviously the steps we took to defend against fixation will give us some protection, especially regenerating the session ID on a regular basis, but you will still be vulnerable to a sophisticated attack. There are a number of steps we can take to defend against a session hijacking. Some are easily circumvented, and others don’t always allow legitimate users to access your site. You need to weigh security and usability
heavily when defending your site. The key is making it very difficult to hijack a user session. There are three common methods for session defense:

- User agent verification
- IP address verification
- Secondary token

User agent verification is a very basic way of verifying the user’s identity. When you create the session ID, you could grab the HTTP_USER_AGENT variable. Then you could verify it on each new page view. Unfortunately, if the session has been hijacked, the malicious agent could have grabbed the user agent info and spoofed it. A better method would be to store the hash of the user agent string. Better yet would be to store the hash plus a seed and verify that. See Chapter 8, “Encryption,” for more information on hashing data. There is another problem with user agent verification; in some specific circumstances the user agent data may not be consistent. Depending on how the user is connected, some proxy servers manipulate the user agent information. For this reason, you may just want to force users to reenter their password if the verification fails as opposed to kicking them out of their session.
IP address verification is very similar to user agent verification. In fact, in some cases it is more secure, as the attacker may know the user agent and be able to spoof the header. You store the users’ IP when you first generate their session, and then on every page load you verify that IP address. There are two major drawbacks to this method. A lot of locations are behind a NAT proxy, so it is possible that the attacker and the user both have the same IP address. The other issue comes from large ISPs like AOL. A number of them, and AOL specifically, have massive proxy setups that send the user out via a different IP address with every page request. If you know where your users are coming from or are willing to set up a different site for AOL users, this method can be very effective. In fact, if your users will be coming from only a small number of IP addresses, this method is great. But generally the drawbacks to IP verification make it unusable.

In token verification, you set up two points of verification. You create a token for the users utilizing a different method from the session ID. When they first log in, create a hash of that token and store it in their session. You can then verify it on every page load. You can also regenerate this token frequently, allowing only a very short window for the attacker to guess it.

None of these methods are foolproof, but all add to your overall security. Having more than one method of verifying your users’ session is always a good idea.

Session Poisoning

This should actually be called session injection, as it is just one more variable injection type of attack. If you allow user input into session variables, make sure you validate the data. Turn register globals off, and see Part III of this book for an in-depth look at dealing with injection attacks.

Patching the Application to Secure the Session

Securing the session capabilities in the application requires two steps:

1. To defeat session hijacking, we implement the secondary token method.
2. To defeat session fixation, we regenerate both the session and the token at crucial points.

Most of the work occurs within the user object, so we’ll start there. First, we rename the $sessionID private variable to $tokenId. We will not be storing the
actual session ID in the user object but rather the token ID. We also update the _generateSessionID() function to use the token, rather than the session variable. We also rename it to _generateTokenID():

```php
function _generateTokenID() {
    $tokenID = rand(10000, 9999999);
    $dbh = getDatabaseHandle();
    $selected_db = mysql_select_db("guestbook", $dbh);
    $sql = "update Users set tokenID = $tokenID where Username = $username";
    $result = mysql_query($sql, $dbh);
    $success = mysql_affected_rows($dbh);
    if($success == 1) {
        $cookieName = "guestbook_cookie";
        $value = $tokenID;
        $expire = 0;
        $secure = TRUE;
        $httponly = TRUE;
        if(setcookie($cookieName, $value, $expire, "", ",", $secure, $httponly)) {
            return $tokenID;
        } else {
            return NULL;
        }
    }
}
```

The code we added is shown in bold. Basically what we’re doing here is creating a token ID and storing it as a cookie in the user’s browser.

Next, we create two token functions, checkToken() and _deleteToken(), as shown here:

```php
function _deleteToken() {
    if(setcookie("guestbook_cookie", ",", time - 3600)) {
        $this->_tokenID = NULL;
        return TRUE;
    }
    return FALSE;
}

function checkToken() {
    if($_COOKIE['guestbook_cookie'] && $_COOKIE['guestbook_cookie'] == $this->_tokenID) {
        return TRUE;
    }
    return FALSE;
}
```
Finally, we retrofit the `login()` and `logout()` functions to create or destroy both the session and the token.

```php
function login($username, $plaintext_password) {
    $dbh = getDatabaseHandle();
    $selected_db = mysql_select_db("guestbook", $dbh);
    $sql = "select username, password from Users where username = $username";
    $result = mysql_query($sql, $dbh);
    $userinfo = mysql_fetch_array($dbh);
    $salt = createSeed();
    $password = encryptPassword($userinfo["password"], $salt, $plaintext_password);
    if($userinfo["password"] == $password) { //User is authenticated
        $user = new User($username);
        $user->_tokenID = _generateTokenID(); // Also stores tokenID in DB
        session_regenerate_id();
        return $user;
    } else {
        return FALSE;
    }
}

function logout() {
    // Invalidate both the session and the token
    session_destroy();
    $dbh = getDatabaseHandle();
    $selected_db = mysql_select_db("guestbook", $dbh);
    if(!_deleteToken()) {
        logError($dbh, "could not delete token cookie", 5);
    }
    $username = $this->_username;
    $sql = "update Users set TokenID = NULL where Username = $username";
```
```php
$result = mysqli_query($sql, $dbh);
$success = mysqli_affected_rows($dbh);
return $success;
```

In the application code, we’ve added code to create the token cookie and start the session before any HTML is sent to the browser. At the end, we invalidate the token cookie and destroy the session. As a final housekeeping task, we’ve changed the `sessionId` column name to `tokenId` in the database.

**Wrapping It Up**

In this chapter, we talked about the three types of session attacks: fixation, hijacking, and poisoning or injection. Session poisoning is just another form of injection attack, which we have covered in quite a bit of depth elsewhere.
In this chapter, we cover a special type of injection attack called cross-site scripting, or XSS. This is a special type of code injection attack (remember those from Chapter 5, “Input Validation”?) that doesn’t affect your system as much as it affects your users. Our example guestbook is exactly the type of site that is vulnerable to these attacks.

**What Is XSS?**

XSS is just a special case of code injection. In this type of attack, the malicious user embeds HTML or other client-side script into your Web site. The attack looks like it is coming from your Web site, which the user trusts. This enables the attacker to bypass a lot of the client’s security, gain sensitive information from the user, or deliver a malicious application. There are two types of XSS attacks:

- Reflected or nonpersistent
- Stored or persistent

**Reflected XSS**

This is the most common type of XSS and the easiest for a malicious attacker to pull off. The attacker uses social engineering techniques to get a user to click on a link to your site. The link has malicious code embedded in it. Your site then redisplay the
attack, and the user’s browser parses it as if it were from a trusted site. This method can be used to deliver a virus or malformed cookie (used to hijack sessions later) or grab data from the user’s system. One famous example of this was found in Google’s search results. The malicious code would be tacked onto the end of a search link. When the user clicked on the link, the code would get displayed as part of the search string. The user’s browser would parse this and compromise his or her system.

Defend against this as you would any variable injection attack. Before you display any user-generated data, validate the input. Do not trust anything that the user’s browser sends you.

**STORED XSS**

This is a less common but far more devastating type of attack. One instance of a stored XSS attack can affect any number of users. This type of attack happens when users are allowed to input data that will get redisplayed, such as a message board, guestbook, etc. Malicious users put HTML or client-side code inside their post. This code is then stored in your application like any other post. Every time that data is accessed, a user has the potential to be compromised. Most of the time this is a link that still requires social engineering to compromise your users, but more sophisticated attackers will launch attacks without the user doing any more than loading your page.

This is all scary stuff, but the defense is the same: If you allow user input, validate it before you store it in your application.

**PATCHING THE APPLICATION TO PREVENT XSS ATTACKS**

There are two ways we can handle patching our application. One is far easier and more secure but gives the user less flexibility. The other method allows a much wider range of user input but is much harder to implement securely. Once again, we have to weigh the usability of our application against security concerns.

We have decided that we don’t really need fancy posts in our guestbook so we will go the easier, more secure route. We will simply disallow HTML and all scripting in any user input (name, message, etc.) field. Any input that contains scripting code will be discarded with an error message. Just to be on the safe side, we will also escape all special characters such as ( and < to their HTML entities. Luckily for us, our sanitation API already does this, and we are already passing our variables through the sanitizer. In patching the application to sanitize all user input variables, we actually closed two potential security holes—general variable injection and XSS.
The fix gets a lot trickier if you want to allow scripts and HTML to be embedded in user inputs. There are two ways to do this, both of which are a little beyond the scope of this book and our application. You could discard any user-inputted code and allow HTML only via buttons on your page, giving the user a very limited set of code elements to use. You still have to validate the user input, because even limiting the user to a predefined subset of HTML isn’t foolproof. A sophisticated attacker can get around this precaution by nesting malicious code within the allowed HTML. If you allow users to include links in their posts, there is no way to defend against XSS—unless you personally have the time to manually check each and every link a user posts.

There is one more option: You can create filters that try to validate user input and filter out the malicious code while keeping the good input. This involves a rather tricky set of regular expressions that are well beyond the scope of this book. Luckily, there are some open-source projects already taking on this task. None of them are completely foolproof, because by the time a filter is created to identify one type of malicious code, several others have been created. Filters do have their place, as long as you realize that they aren’t a guarantee of security. If you decide to try to filter out malicious code from user input, we suggest looking into the following projects:

- OWASP’s PHP filters: www.owasp.org/index.php/OWASP_PHP_Filters. This project includes filters for all types of attacks.
- PHP IDS: http://php-ids.org. This is an intrusion detection system with the capability to report the types of attacks to you, but you need to configure how the system will respond to various circumstances.
- htmLawed: www.bioinformatics.org/phplabware/internal_utilities/htmLawed/index.php. This is an open-source PHP HTML filter.
- HTML Purifier: http://htmlpurifier.org/. This filter implements a whitelist approach to PHP filtering.

Wrapping It Up

Cross-site scripting is a hot buzzword in PHP security circles, but don’t let it intimidate you. It’s really just a new and interesting way of exploiting a variable injection attack. As long as you’re vigilant about sanitizing your variables, you should have no problems with XSS.
Index

Symbols
$ (dollar sign), 59
* (star), 63
{ } (curly brackets), 59, 63
+ (plus sgn), 63, 64

A
a-zA-Z, regular expressions, 59
Access Control List (ACL), securing Web root, 179
Actors diagram
designing security with, 260, 262
identifying points of failure in, 272
Acunetix Web Vulnerability Scanner testing interface, 247–254
Administrative Tools folder, 102–103, 108–109
Administrative Tools Services MMC, 177–178
Administrative users
changing username/password on MySQL, 163–164
granting privileges to, 100–101, 115
viewing and deleting user accounts/comments, 14
workflow diagrams for, 260–262, 272
Advanced button, Windows properties, 80–82
AES encryption, 124
Alerts
automated testing, 235
intrusion detection system, 73
keeping up with security, 144
for latest stable version of Web server, 147
ModSecurity, 215
paying attention to latest security, 44–46
reviewing during scanning, 252–253
system test, 223
Algorithm strength, 123–124
Allow permission, 77–79
allow_url_fopen directive, php.ini file, 72–73, 90–91
Anonymous users
allowing access to Web site, 180
Anonymous users (continued)
allowing comments from, 13–15
authentication systems vs., 269
no need to authenticate, 100–101
removing from SQL Server, 202–204
workflow diagram for, 260–262, 272
Apache server, 147–159
disabling unneeded options, 153–154
enabling ModSecurity, 154–159
giving own user and group to, 149–151
hiding version number/other information, 151
restricting to own directory structure, 152–153
upgrading or installing latest version, 147–149
using SuExec for shared hosting, 214–215
API (Application Programming Interface)
for authentication, 119–120
customizing for system calls, 31–32
customizing for user input validation, 32
defined, 289
sanitizing data to prevent buffer overflows, 49
for user-uploaded image files, 88–90
Application pools, 181–184
Application Programming Interface. See API (Application Programming Interface)
Applications
data sources for, 48
gaining access to server through insecure, 5–6, 10
hackers targeting minor, 9
hardening your, 6–7
making life difficult for spammers, 22–23
Applications, designing securely from the beginning, 257–271
concept summary, 257–260
data design, 260–267
file upload, 270
filesystem access, 271
identifying points of failure, 269
infrastructure functions, 267–268
login and logout, 269–270
user input, 270–271
workflow and actors diagram, 260
Applications, securing existing, 273–278
hardening checklist, 276–277
having code peer-reviewed, 278
using three-stage deployment, 273–275
using version control, 275–276
variable sanitation, 277
Arbitrary code attacks, from buffer overflows, 42
Asymmetric (public) key encryption, 121–122
Authentication
adding encryption to. See encryption
directory-based, 101–114
goals of creating, 95
identifying login/logout points of failure, 269–270
image recognition, 99–100
patching application for, 117–120
privileges, 100–101
SQL Server, 192
storing information in user database table, 114–115
storing usernames and passwords, 115–117
types of, 95–97
usernames and passwords, 97–99
using Web Vulnerability Scanner, 250–251
writing with Zend, 208
AutoAttack tool, CAL9000 toolkit, 245
Automated testing. See Testing, automated

B
Backup
length constraints on database, 56
storing information in user database, 118–119
Basic Multilingual Plane, 43, 289
Biometric analysis, 96
Black-box testing, 277, 289
Blank input
brainstorming boundary conditions, 18–19
overview of, 15–18
Blowfish encryption, 124
Books, as resources, 286–288
Boundary conditions
automated testing of, 219–220, 223–224
as buffer overflow, 45
building error-handling mechanism for, 23–26
determining, 18–19
Breach Security Labs, 155–159
Buffer, 40–41, 289
Buffer overflows, 37–52
computer science of, 39–41
consequences of, 42
with excessively long input, 55
fuzz testing for, 227
identifying points of failure, 270–271
memory allocation and PHP, 42–44
overview of, 37–39
patching application, 49–52
paying attention to latest security alerts, 44–46
sanitizing variables to prevent, 46–49

C
C libraries, underlying PHP, 39
CAL9000 toolkit
AutoAttack tool, 245
Cheat Sheets tool, 242–243
Checklist tool, 244–245
Encode/Decode tool, 237–239
HTTP Requests tool, 239
HTTP Responses tool, 240–241
Misc Tools, 243–244
obtaining, 234–235
Scratch Pad tool, 242
using, 235
XSS Attacks tool, 236–237
CAPTCHA (Completely Automated Public
Turing Test to tell Computers and
Humans Apart), 99–100, 289
CERT (Computer Emergency Response
Team), 9, 46–47
CGIs, and SuExec, 215
changeFilePrivs( ) function, 88–89
Character class (within regular expres-
sion), 59–61, 289
Checklist tool, CAL9000 toolkit, 244–245
checkToken() function, 134
chmod( ) function, 87
Classes, security alert, 45
Commas, and spammers, 22–23
Comments, 56–57
Completely Automated Public Turing
Test to tell Computers and Humans
Apart (CAPTCHA), 99–100, 289
Computer Browser Properties dialog, IIS, 178
Computer Emergency Response Team (CERT), 9, 46–47
Computer Management, Administrative Tools folder, 102–103
Consistency
  in building error-handling mechanism, 19–23
  in naming, 281
  when writing self-documenting code, 280–281
Constraints, database and logical, 56–57
Cookie button, PowerFuzzer, 231
Cracker, 4–5, 289
createSalt( ) function, 127
Creative Commons license, 207, 289
Cross-site scripting. See XSS (cross-site scripting)
Cryptography. See Encryption
Curly brackets ({}), 59, 63
CVS, 275–276

D
Data
  basing encryption type on, 124–125
  checking length of, 48–49
  choosing for testing, 223–224
  designing security for, 260–267
  making assumptions about user, 55
  sanitizing to prevent buffer overflows, 48–49
  sources of, 48
  tainted, 57–58
Data dictionary
  database constraints and, 56
  identifying points of failure, 269
  setting up, 264–266
Databases
  deleting sample MySQL, 165
  deleting sample SQL Server, 204–205
  placing constraints on length of stored data, 56
  running latest stable version of server, 49–50
  securing SQL Server. See SQL Server storing authentication information in, 114–115
Databases Security Uses folder, SSMSE, 202–203
Decoding plain text, with CAL9000 toolkit, 238
deleteToken() function, 134
Deny permission
  changing in Windows, 77
  directory-based authentication, 107
  overriding Allow permission, 78–79
Deployment, of existing applications, 273–275
Design phase. See Applications, designing security at beginning
Development box, 273–274
Development releases, PHP, 212
Directory-based authentication, 101–114
Directory structure
  hackers navigating, 7–8
  opening local files, 70–71
  restricting Apache to its own, 152–153
  securing Web root, 179
  storing needed files in separate directory within, 70–71
Directory traversal attack, 153
display_errors, hardening php.ini, 217–218
DMZ, 200, 290
Documentation
of length constraints on database, 56
writing self-documenting code, 280–281

Dollar sign ($), 59

DoS (denial-of-service) attacks
from buffer overflows, 42
defined, 289–290
fuzz testing for, 227
using system resources for, 29

Download mirror
MySQL, 161–162
PowerFuzzer, 229

E

Editing, object in Windows file permissions, 86–87

Encapsulation
allowing file uploads using, 89
data design using, 263
error handling with, 32
in filesystem access, 70
of system calls, 27, 278

Encode/Decode tab, CAL9000 toolkit, 237–239

Encryption, 121–128
choosing type of, 123–125
defining, 121–123
password security, 125
patching application to encrypt passwords, 125–127
username and password, 115

encryptPassword() function, 127

Error handling, 13–26
brainstorming boundary conditions, 18–19
building mechanism for, 19–23
encountering erroneous data, 23–24
guestbook application, 13–15
making system easy to use, 24–26
SQL injection attack, 16–18

Error-logging, SQL Server, 194

Error messages, writing, 23–24

Escape, defined, 21, 290
escapeshellarg() command, 30–31
escapeshellcmd() command, 30

Execute permissions, 76

Exploit testing. See Testing, exploit

expose_php, hardening php.ini, 217

Extensibility, with custom API, 31

F

Features
disabling unnecessary SQL Server, 197
keeping tight rein on new, 279–280

file_get_contents() function, 71

Filenames
checking variable sanitation, 51–52
escapeshellcmd() and escapeshellarg() securing, 30–31
malicious users of system calls and, 28
opening local files, 71
security myth of changing, 7–9
validating user input, 32–34

$_FILES Superglobal array, 74

Filesystem access, 69–91
allowing user-uploaded image files, 88–90
creating and storing files, 73–75
designing security from beginning, 271
opening local files, 69–71
opening remote files, 71
permissions in PHP, 87
permissions in UNIX, Linux and MAC OS X, 76
permissions in Windows. See Windows file permissions, changing
Filesysterm access (continued)
  preventing remote attacks, 72–73
  summary review, 90–91
Filters
  for malicious code in user input, 139
  testing effectiveness of. See testing, exploit
Firefox, for CAL9000 toolkit, 234
Firewalls, 5–6
Fixation sessions. See Session fixation
Footprint
  defined, 290
  reducing IIS server, 177–178
  reducing SQL Server, 195, 200
Forms
  for user-uploaded image files, 90
  for users to upload files, 74–75
Fuzz testing
  installing and configuring PowerFuzzer, 227–230
  overview of, 226–227
  using PowerFuzzer, 231–233
G
Generally Available Release, 160, 290
_generateSessionID() function, 134
_generateTokenID() function, 134–135
Gibson Research Corporation (GRC),
  password generator, 164
Glossary, 289–292
Granularity, of Windows file permissions,
  77–79, 85–87
GRC (Gibson Research Corporation),
  password generator, 164
Greedy modifiers, regular expressions, 63
Groups
  authentication, 102–106
  for each application in Apache, 149–151
  Web file authentication, 111–114
  Windows file authentication, 104–110
  Windows permission, 78, 84
Guestbook application
  adding buffer overflow prevention, 49–52
  adding encryption, 125–127
  adding session security, 133–136
  adding system calls API, 32–33
  adding user authentication, 117–119
  allowing user-uploaded files, 88–90
  concept summary for, 258–259
  defined, 13
  designing data dictionary, 264–266
  designing infrastructure functions, 267
  designing long-term data storage, 263–267
  designing workflow, 260–262
  preventing XSS attacks, 138
  primary code listing, 14–15
  program summary, 13–14
GUI, setting permissions using, 83–85
H
Hackers
  defined, 290
  targeting minor applications, 9
  targeting sessions, 9
  use of term in this book, 4–5
  using insecure applications, 5–7
  using obfuscation against, 7–9
Hard drive, Web root on nonsystem, 179
Harden an application
  checklist, 276–277
  defined, 290
  tools for programmers, 6
Hardened-PHP Group, 4
Hardened-PHP Project, 42–43, 46
Hardware, Optional updates, 187–188
Heap, 40, 290
High priority Windows updates, 187
Hijacking, session
defending against, 131–133
identifying login/logout points of failure, 270
patching application for, 133–136
Home Directory tab, 186–187
.htaccess files, 101
HTML
accepting from users safely, 21
preventing XSS attacks, 138–139
stripping from user input, 20–21
HTML Purifier filter, 139
htmlentities( ) function, 21, 42–44
htmlspecialchars( ) function, 21, 42–44
HTTP Requests tool, CAL9000 toolkit, 239, 244
HTTP Responses tool, CAL9000 toolkit, 240–241
HTTP, stateless, 129
httpd.conf file, Apache
copying old version of, 149
creating users and groups, 149–151
disabling unneeded options, 153–154
hiding version number/other information, 151
restricting to own directory structure, 152–153

I
IDE (integrated development environment)
defined, 290
resources for, 288
writing code using, 281–282
Identity dialog box, 181–182
IDS (intrusion detection system)
defined, 290
for malicious code, 139
for self-created files, 73
using ModSecurity as, 215–216
if( ) statement, 51–52
IIS (Internet Information Server)
reducing footprint on Web, 177–178
securing Web root, 179–187
securing Windows server environment, 167
updating operating system, 168–177
IIS Manager
creating Web sites in, 179–180
enabling only needed Web services, 185–187
setting permissions on existing sites, 109
setting up sandboxes for each Web site, 181–184
Image files
creating upload form for, 90
patching application to allow user-uploaded, 88–89
testing that file is correct type, 74–75
Image recognition, for authentication, 99–100
Infrastructure functions, designing, 267–268
Inheritance, Windows, 79–82
Initialization, variable, 33
Injection attack
from buffer overflows, 42
checking length of inputs to detect, 55
cross-site scripting as, 137–139
defined, 290
identifying points of failure, 270–271
session poisoning as, 133
Input validation, 53–67
  assumptions about expected user data, 55
  common patterns of, 65–67
  database constraints, 56
  logical constraints, 56–57
  patching guestbook application, 32
  regular expressions and, 58–65
  tainted data, 57–58
  testing effectiveness of. See testing, exploit
users signing guestbook comments, 53–54
users who give you more than you asked for, 54–55
Install Updates button, Windows, 174–175
Integrated development environment. See
  IDE (integrated development environment)
Internet Information Server. See IIS
  (Internet Information Server)
Intrusion detection system. See IDS
  (intrusion detection system)
IP address verification, 132–133
IP Encoder tool, CAL9000 toolkit, 244
isAdmin column, user database, 114–115,
  118
ISPs, and IP address verification, 133
is_uploaded_file( ) function, 74–75

K
Kernel, 145–146

L
Lazy modifiers, regular expressions, 64
Library functions, writing code using, 281
Licenses
  SQL Server, 188
  Windows Updates, 176
Linux
  changing file permissions in, 76–87
  securing server environment, 144–146
  username and password system in, 101
Local filesystem, accessing, 69–71
Local vulnerability, and security alerts, 45
Logical constraints, 56–57
login( ) function, 119, 134
Login, identifying points of failure,
  269–270
logout( ) function, 134
Logout, identifying points of failure,
  269–270
Lost passwords, 98–99

M
MAC OS X
  file permissions, 76–87
  securing server, 144–146
  username and password system, 101
Maintenance, of self-created files, 73
MAX_FILE_SIZE directive, upload
  forms, 90
mcrypt( ) function, 123–124
MD5 algorithm, 124, 125
Memory allocation, 40–44
Metacharacters, and regular expressions, 60
Misc Tools tab, CAL9000 toolkit, 243–244
ModSecurity
  as IDS for self-created files, 73
  installing/enabling for Apache, 154–159
  securing PHP with, 215–216
move_uploaded_file( ) function, 75
movieFile( ) function, 32–33, 88–90
Multilayered security approach, 4
mv command, movieFile( ) function, 32–33
My Computer, securing Web root, 179
MySQL
  changing admin username and password, 163–164
  creating new accounts for each application, 164–165
  deleting default database users, 164
  deleting sample databases, 165
  disabling remote access, 163
  upgrading or installing latest version, 159–163

N
Name field
  assumptions about expected data, 55
  placing logical constraints on, 56–57
  signing guestbook comments, 53–54
  testing for excessively long input, 54–55
Naming conventions
  separating tainted from validated data, 57–58
  writing self-documenting code using consistency, 281
NetBIOS, disabling for IIS server, 177
Network security, 5–7, 10
New Scan button, Web Vulnerability Scanner, 248
NTFS permissions, Web file authentication, 112

O
Obfuscation
  security myth of, 7–9
  using encryption vs., 124
  writing self-documenting code vs., 280–281
OCR (optical character reader), 100, 290
One-way encryption, 123
open_basedir, hardening php.ini, 217
Opening
  local filesystem, 69–71
  remote filesystem, 71
Operating systems
  inherent insecurity of, 143–144
  installing latest version of MySQL, 160–162
  updating, 168–177
  updating UNIX, Linux or MAC OS X, 145–146
  verifying running of latest stable version, 49–50
Optical character reader (OCR), 100, 290
OptionCart, 9
OWASP PHP filters, 139

P
Packets, 154, 290
Passphrases, 116, 290
Passwords. See also Usernames and passwords
  identifying login/logout points of failure, 269
  password retention policy, 125, 290
  securing SQL Server SA account, 200–202
Patches, 144, 167
Patterns, input validation, 65
PCRE (Perl Compatible Regular Expressions) library, 66–67, 290
PEAR (PHP Extension and Application Repository)
  CAPTCHA libraries, 100
  defined, 290
  overview of, 285–286
Peer reviewers, 278, 283–284
Penetration testing, 225–226
Performance, ModSecurity and, 216
Perl Compatible Regular Expressions (PCRE) library, 66–67, 290
Permissions
changing safely, 76
denying to users, 107–108
IIS server, 184, 186
PHP, 87
restrictive, 75
selecting for groups, 109–110
UNIX, Linux and MAC OS X, 76
user-uploaded image files, 88–89
Windows. See Windows file permissions, changing
PHP
buffer overflow vulnerabilities in, 37–39
changing file permissions in, 87
as inherently insecure language, 3–4
memory allocation and, 42–44
verifying running of latest stable version, 49–51
PHP Extension and Application Repository. See PEAR (PHP Extension and Application Repository)
PHP IDS Web site, 139
PHP, securing on server, 207–218
hardening php.ini, 216–218
with ModSecurity, 215–216
using latest version, 207–208, 212–213
using safe_mode, 213–214
using SuExec, 214–215
using Suhosin patch and extension, 213
using Zend Framework and Optimizer, 208–211
php.ini file
disabling PHP access to remote files, 71
hardening, 216–218
preventing remote filesystem attacks, 72–73, 90–91
session fixation defense in, 130–131
storing uploaded files in, 74
using ModSecurity to secure, 216
using safe_mode in, 213–214
ping, 29, 291
ping flood attacks, 291
Plus sign (+), 63, 64
Points of failure, designing security, 269
Poisoning, session, 133
POSIX, 66, 291
PowerFuzzer, 227–233
preg_match() function, 65–66
Primary code listing, guestbook application, 14–15
Privileges, 100–101
Programmer, becoming better, 279–284
avoid feature creep, 279–280
finding good peer reviewer, 283–284
using right tools, 282–283
write self-documenting code, 280–281
Programming languages, inherent insecurity of, 143–144
Properties. See also Permissions
configuring Web file authentication, 111–114
configuring Windows file authentication, 102–110
securing SQL Server, 200–201
Proprietary test suites
benefits and features of, 246
overview of, 246
scanning application with, 247–254
Public (asymmetric) key encryption, 121–122
Published alerts, 46
INDEX

R
Read permissions, 76
Really Bad Idea (term), 71
reflected XSS attacks, 137–138
Registered (authenticated) users, granting privileges to, 100–101
register_globals, hardening php.ini, 216, 217
Regular expressions (regex)
  character classes, 60–61
  defined, 291
  greedy modifiers, 63
  input validation patterns, 65–67
  lazy modifiers, 64
  metacharacters, 60–62
  overview of, 58–59
  preventing spammers with, 22–23
  testing with CAL9000 toolkit, 236
Releases
  MySQL, 159
  PHP development, 212
  UNIX, Linux or MAC OS X, 145
Remote access, disabling MySQL, 163
Remote exploits, from buffer overflows, 42
Remote filesystem
  accessing, 71
  preventing attacks on, 72–73
Remote vulnerability, security alerts, 45
Report button, Web Vulnerability Scanner, 252–254
Reporting style, Web Vulnerability Scanner Reporter, 252–253
Resetting passwords, 99
Resources
  Apache, current release of, 147–148
  Apache, disabling unneeded options, 154
  CAL9000 toolkit, 234
  CAPTCHA libraries, 100
  CVS, 276
  filters for malicious code, 139
  Gibson Research Corporation password generator, 164
  ModSecurity, 155, 159, 215–216
  MySQL, current release of, 159–160
  PEAR, 285–286
  PowerFuzzer, 227, 229
  SQL Server Management Studio Express, 198
  Suhosin patch and extension, 213
  Visual SourceSafe, 275
  Zend Core Website, 209–211
Review Other Updates button, Windows, 170
Rootkit
  defined, 291
  remote filesystem access, 71
  as uploading vulnerability, 270
ROTX bit manipulation, avoiding, 124
S
safe_mode, securing PHP, 213–214, 217
Salt, 126–127, 291
Sandboxes
  defined, 291
  securing existing applications, 273–274
  setting up for each Web site, 181–184
Sanitation, data
  creating custom API for system call, 31–32
  preventing remote filesystem attacks, 72–73
Sanitation, variable. See Variable sanitation
Scan button, PowerFuzzer test, 232
Scan wizard, Web Vulnerability Scanner, 248–252
Scratch Pad tab, CAL9000 toolkit, 242
Script kiddie, 69, 291
Scripts
  - defeating spammers with CAPTCHA, 100
  - methodically traversing directory structures with, 7–9
  - preventing XSS attacks, 138–139
Scroogle Search tool, CAL9000 toolkit, 244
Security advisory sources, 45–47
Security alerts, 44–46, 144
Security badges, 96
Security, common misconceptions, 3–10
  - about minor applications, 9
  - about native session management, 9
  - about obscurity, 7–9
  - about single points of failure, 10
  - reality check, 3–5
  - as server issue, 5–7
Security Logins folder, SSMSE, 200–201
Security tab, Windows GUI, 83–84
Security tab, Windows properties, 80–82
Security updates, 187–188
SecurityFocus, 45–46
Self-created files, preventing attacks on, 73
Self-documenting code, writing, 280–281
Semicolons, and spammers, 22–23
Servers, 143–166
  - Apache. See Apache server
  - application hardening checklist, 276
  - MySQL, 159–165
  - programming languages, OS and, 143–144
  - securing UNIX, Linux or MAC OS X, 144–146
  - security myth, 5–6
  - verifying latest stable version, 49–50
ServerSignature to Off, Apache, 151
ServerTokens to Prod, Apache, 151
Service packs, updating operating system, 168–177
Services
  - disabling unneeded IIS server, 177–178
  - disabling unneeded SQL Server, 196
  - installing updates for necessary Windows, 172–173
Session fixation, 130–131, 133–136
Session hijacking
  - defending against, 131–133
  - identifying login/logout points of failure, 270
  - patching application for, 133–136
Session IDs, in session fixation, 130–131
Session poisoning, 133
Session security, 129–136
  - defining session variables, 129
  - patching application for, 133–136
  - session fixation, 130
  - session hijacking, 131–133
  - session poisoning, 133
  - types of session attacks, 129–130
Session variables, 129
  - session.cookie_lifetime, hardening
    php.ini, 217
SessionID column, user database,
  114–115, 118
  - session_regenerate_id function, 130–131
Set User ID (SUID) bit, 28, 29
SHA algorithm, 125
SimpleTest framework, 221
SMP, disabling for IIS server, 177
Software, Optional updates, 187–188
Spaghetti code, 284, 291
Spammers
checking length of inputs to detect, 55
making life difficult for, 22–23
using image recognition to defeat automated scripts of, 99–100
Speed, encryption based on, 124–125
SQL injection
defined, 291
fuzz testing for, 227
how it works, 16–18
identifying points of failure, 270
on stored usernames and passwords, 117
SQL Server
defined, 187
installing SQL Server Management Studio Express, 198–200
installing/upgrading to latest version, 187–200
securing Windows server environment, 167
setting up DMZ, 200
steps in hardening, 200–205
updating operating system, 168–177
SQL Server Enterprise Edition, 188–198
SQL Server Express Edition, 188–198
SQL Server Management Studio Express (SSMSE), 198–200
Square brackets ([ ]), 59
SSL/TSL, 131
SSMSE (SQL Server Management Studio Express), 198–200
Stack, 40–41, 291
Star (*), 63
Stateless, defined, 291
Stateless HTTP, 129
Storage
designing long-term, 263–267
safe file, 75
of self-created files in separate filesystem, 73
storing data securely, 278
Stored XSS attacks, 138
striptags() function, 20–21
strlen() function, 48–49
Subdirectories, setting permissions on, 110
Sudo command, 28, 29
SuExec, securing PHP with, 214–215
Suhosin patch and extension, to PHP, 213
SUID (Set User ID) bit, 28, 29
Superglobals, 74, 291–292
Surface Area Configuration tool, SQL Server, 195–198
Swipe cards, 96
Symmetric key encryption, 122–123
System calls, 27–34
defined, 27
encapsulating, 278
overview of, 27–28
patching guestbook application, 32–34
securing with escapeshellarg(), 30–31
securing with escapeshellcmd(), 30
using system binaries with SUID bit or sudo, 28–29
using system resources, 29–30
System calls API, 31–32, 51–52
System functions, validating data from, 48
System resources, system calls using, 29–30
System tests, 222–223
T
Tainted data, 57–58, 65
Tainted_prefix, 58
Test suites. See Proprietary test suites
Testing
penetration, 225–226
securing existing applications with,
274–275
for unexpected input, 20–21
Testing, automated, 219–224
choosing solid data, 223–224
framework for, 220–221
performing system tests, 223
performing unit tests, 222–223
resources for, 288
security implications of, 219–220
Testing, exploit, 225–254
defining, 225–226
fuzzing, overview of, 226–227
installing and configuring PowerFuzzer,
227–230
resources for, 288
testing toolkits, 233–234
using CAL9000 toolkit. See CAL9000
toolkit
using PowerFuzzer, 231–233
using proprietary test suites, 246–254
warnings about tools of, 226
Testing toolkits, 233–234. See also
CAL9000 toolkit
Third-party libraries, encryption, 123–124
3DES Encryption, 124
/tmp Directory, 74–75
tmp_name variable, 74
Token verification, 132–136
Trust, Internet security and, 4
U
Unicode, 43, 292
Unit tests, 222–223, 268
UNIX
changing file permissions in, 76–87
securing server environment in, 144–146
username and password system in, 101
Update, Windows, 168–177, 187
Updated alerts, 46, 144
Upgrades, 144, 213
Uploads
creating form for, 90
identifying points of failure, 270
opening local files, 70–71
patching application to allow image
files, 88–90
securing application against file, 73–74
User accounts
creating in Zend, 210–211
securing MySQL by deleting default,
164–165
User agent verification, 132
User database table
adding encryption to, 126
adding to guestbook application, 118–119
storing authentication information in,
114–115
User input
identifying points of failure, 270–271
preventing XSS attacks, 138–139
sanitizing variables, 46
as source of data, 48
validating, 32
User instances, enabling in SQL Server, 194
Usernames and passwords
accessing vulnerability of, 117
configuring Web file authentication,
111–114
configuring Windows file authentica-
tion, 114–115
encrypting, 115
overview of, 97–99
password encryption, 125
password strength, 116–117
placing .htaccess text file, 101
securing MySQL, 163–164
setting up sandboxes for Web sites, 182
storing information in user database, 114–115, 118–119
as "what you know" authentication, 95–96

Users. See also Administrative users;
Anonymous users
building error-handling mechanism, 19–23
configuring Web file authentication, 111–114
configuring Windows file authentication, 104–110
creating for each application in Apache, 149–151
designing security for data, 260–267
UTF-8 encoding, 42–44, 292

V
validateUsernamePassword( ) function, 119–120
Validation
creating authentication API, 119–120
input. See Input validation
preventing XSS attacks, 138–139
Variable sanitation
checking, 51–52
creating authentication API, 119–120
to prevent buffer overflows, 46–49
preventing XSS attacks, 138–139
securing existing applications, 277
using regular expressions for, 65–67
Variables
initializing, 33
session, 129
Verification
file upload, 74–75
IP address, 133
preventing remote filesystem attacks with, 72–73
token, 133
user agent, 132
of Windows Updates, 175

Version control system, 275–276
Versions
Apache, hiding information on, 151
Apache, using latest, 147–149
MySQL, using latest, 159–163
PHP, finding latest stable, 212–213
PHP, using latest, 207–208
SQL Server, using latest, 187–200
UNIX/Linux/MAC OS X, using latest, 145–146
verifying latest stable, 49–50
Windows, finding latest, 185
Windows, using latest, 167

Virtual directories, setting permissions on, 110

Visitors. See Anonymous users
Visual impairment, accessibility issues, 100
Visual SourceSafe, 275
VPN tokens, 96
Vulnerabilities
alerts notifying of, 46
application hardening checklist, 276–277
automated scanning of, 247–254
PowerFuzzer report on, 233

W
Web Authors group, 179
Web file access, 111–114
Web hosts, secure, 144
Web root
creating Web sites in IIS Manager, 179–180
Web root (continued)
  enabling only needed Web services, 185–187
  setting up on nonsystem drive, 179
  setting up sandboxes for each site, 181–184
Web servers, inherent insecurity of, 143–144
Web Service Extensions folder, 185–187
Web Site Creation Wizard, 180
"What you are" authentication, 96
"What you have" authentication, 96
"What you know" authentication, 96
White-box testing, 277, 292
Windows Explorer, securing Web root, 179
Windows file permissions, changing,
  77–87
  configuring authentication, 102–110
  explicitly selecting, 85–87
  granularity of, 77–79
  setting using GUI, 83–85
  use of inheritance, 79–82
Windows Update, 168–177, 187
Windows Web server, 167, 168–177
Workflow diagram, 260–261, 272
Write permissions, 76
X
XOR bit manipulation, 124
XSS Attacks tab, CAL9000 toolkit,
  236–237
XSS (cross-site scripting)
  defined, 137
  fuzz testing for, 227
  patching application to prevent, 138–139
  reflected, 137–138
  stored, 138
Z
Zend, 208–211
  extending PHP, 207–208
  Framework and Optimizer, 208–211