Easy, Powerful Code Security Techniques for Every PHP Developer

SECURING PHP WEB APPLICATIONS

TRICIA BALLAD
WILLIAM BALLAD

Library of Congress Cataloging-in-Publication Data

Ballad, Tricia.
Securing PHP web applications / Tricia Ballad, William Ballad.
p. cm.
Includes index.
ISBN 978-0-321-53434-7 (pbk. : alk. paper)
1. PHP (Computer program language) 2. Web services—Security measures. 3. Internet—Computer programs—Security measures. 4. Application software—Development. I. Ballad, Bill. II. Title.

QA76.73.P224B35 2009
005.8—dc22

2008042783

ISBN-13: 978-0-321-53434-7
ISBN-10: 0-321-53434-4
Text printed in the United States on recycled paper at Donnelley in Crawfordsville, Indiana
First printing, December 2008

8 Encryption

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.

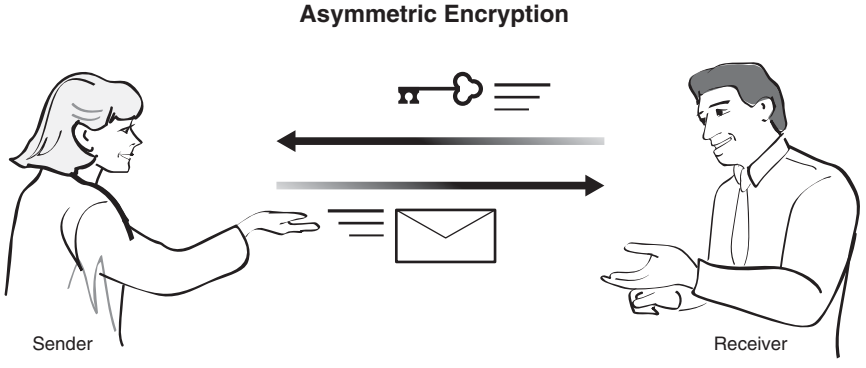

Figure 8.1 Diagram of asymmetric encryption.

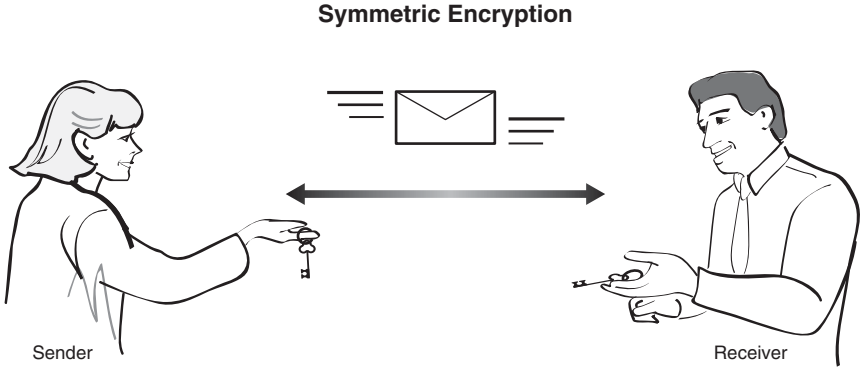

Figure 8.2 Diagram of symmetric encryption.

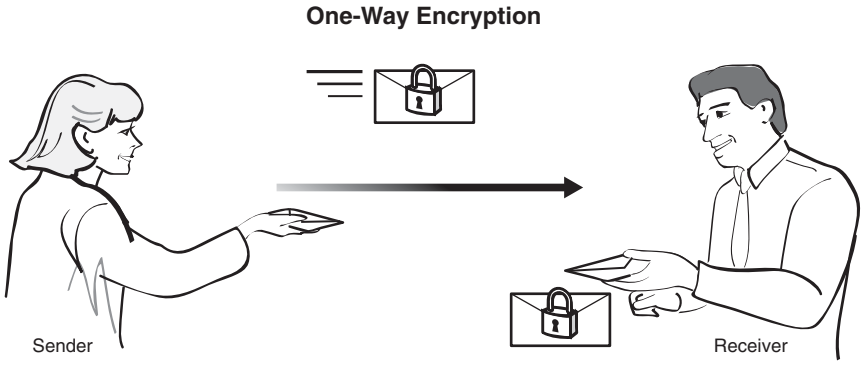

Figure 8.3 Diagram of one-way encryption.

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 8.1 Characteristics of the Random Number Field

Column Name	Type	NULL?	Default Value
`salt`	Varchar(30)	No	` `

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.

```
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."

9 Session Security

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."

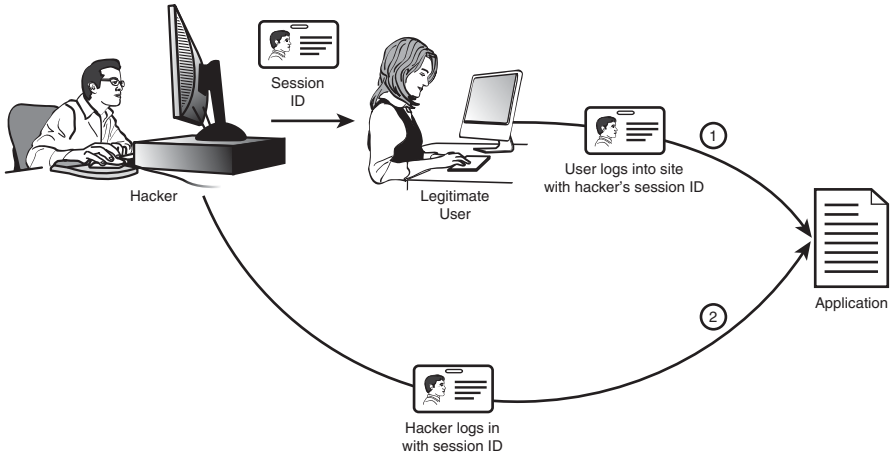

Figure 9.1 Diagram of a session fixation attack.

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

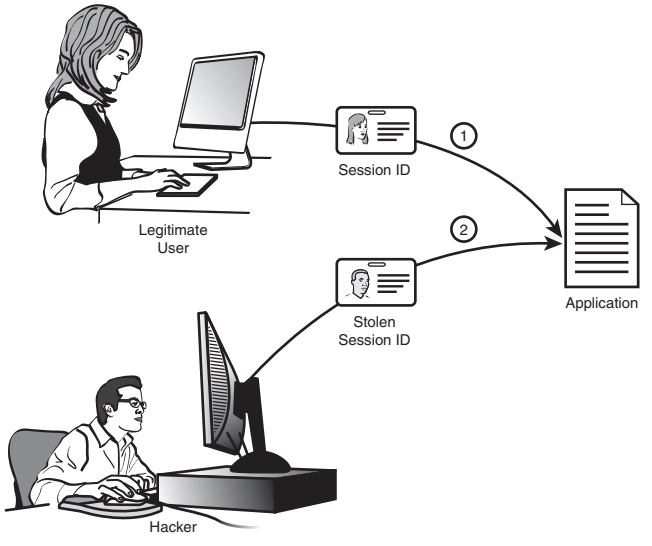

Figure 9.2 Diagram of a session hijacking attack.

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()`:

```
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:

```
        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) {
```

```
                $this->_generateToken(); // Keep the window of opportunity
                // as small as possible
                return TRUE;
        }
        return FALSE;
}
```

Finally, we retrofit the `login()` and `logout()` functions to create or destroy both the session and the token.

```
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";
```

```
        $result = mysql_query($sql, $dbh);
        $success = mysql_affected_rows($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.

10 Cross-Site Scripting

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 redisplays 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

D

E

F

G

H

I

K

L

M

R

S

T

U

X

Z