Security

Introduction

PHP is a powerful language and the interpreter, whether included in a web server as a module or executed as a separate CGI binary, is able to access files, execute commands and open network connections on the server. These properties make anything run on a web server insecure by default. PHP is designed specifically to be a more secure language for writing CGI programs than Perl or C, and with correct selection of compile-time and runtime configuration options, and proper coding practices, it can give you exactly the combination of freedom and security you need.

As there are many different ways of utilizing PHP, there are many configuration options controlling its behaviour. A large selection of options guarantees you can use PHP for a lot of purposes, but it also means there are combinations of these options and server configurations that result in an insecure setup.

The configuration flexibility of PHP is equally rivalled by the code flexibility. PHP can be used to build complete server applications, with all the power of a shell user, or it can be used for simple server-side includes with little risk in a tightly controlled environment. How you build that environment, and how secure it is, is largely up to the PHP developer.

This chapter starts with some general security advice, explains the different configuration option combinations and the situations they can be safely used, and describes different considerations in coding for different levels of security.

General considerations

A completely secure system is a virtual impossibility, so an approach often used in the security profession is one of balancing risk and usability. If every variable submitted by a user required two forms of biometric validation (such as a retinal scan and a fingerprint), you would have an extremely high level of accountability. It would also take half an hour to fill out a fairly complex form, which would tend to encourage users to find ways of bypassing the security.

The best security is often unobtrusive enough to suit the requirements without the user being prevented from accomplishing their work, or over-burdening the code author with excessive complexity. Indeed, some security attacks are merely exploits of this kind of overly built security, which tends to erode over time.

A phrase worth remembering: A system is only as good as the weakest link in a chain. If all transactions are heavily logged based on time, location, transaction type, etc. but the user is only verified based on a single cookie, the validity of tying the users to the transaction log is severely weakened.

When testing, keep in mind that you will not be able to test all possibilities for even the simplest of pages. The input you may expect will be completely unrelated to the input given by a disgruntled employee, a cracker with months of time on their hands, or a housecat walking across the keyboard. This is why it's best to look at the code from a logical perspective, to discern where unexpected data can be introduced, and then follow how it is modified, reduced, or amplified.

The Internet is filled with people trying to make a name for themselves by breaking your code, crashing your site, posting inappropriate content, and otherwise making your day interesting. It doesn't matter if you have a small or large site, you are a target by simply being online, by having a server that can be connected to. Many cracking programs do not discern by size, they simply trawl massive IP blocks looking for victims. Try not to become one.

Installed as CGI binary

Possible attacks

Using PHP as a CGI binary is an option for setups that for some reason do not wish to integrate PHP as a module into server software (like Apache), or will use PHP with different kinds of CGI wrappers to create safe chroot and setuid environments for scripts. This setup usually involves installing executable PHP binary to the web server cgi-bin directory. CERT advisory » CA-96.11 recommends against placing any interpreters into cgi-bin. Even if the PHP binary can be used as a standalone interpreter, PHP is designed to prevent the attacks this setup makes possible:

- Accessing system files: *http://my.host/cgi-bin/php?/etc/passwd* The query information in a URL after the question mark (?) is passed as command line arguments to the interpreter by the CGI interface. Usually interpreters open and execute the file specified as the first argument on the command line. When invoked as a CGI binary, PHP refuses to interpret the command line arguments.
- Accessing any web document on server: *http://my.host/cgi-bin/php/secret/doc.html* The path information part of the URL after the PHP binary name, */secret/doc.html* is conventionally used to specify the name of the file to be opened and interpreted by the CGI program. Usually some web server configuration directives (Apache: Action) are used to redirect requests to documents like *http://my.host/secret/script.php* to the PHP interpreter. With this setup, the web server first checks the access permissions to the directory */secret*, and after that creates the redirected request *http://my.host/cgi-bin/php/secret/script.php*. Unfortunately, if the request is originally given in this form, no access checks are made by web server for file */secret/script.php*, but only for the */cgi-bin/php* file. This way any user able to access */cgi-bin/php* is able to access any protected document on the web server. In PHP, compile-time configuration option --enable-force-cgi-redirect and runtime configuration directives doc_root and user_dir can be used to prevent this attack, if the server document tree has any directories with access restrictions. See below for full the explanation of the different combinations.

Case 1: only public files served

If your server does not have any content that is not restricted by password or ip based access control, there is no need for these configuration options. If your web server does not allow you to do redirects, or the server does not have a way to communicate to the PHP binary that the request is a safely redirected request, you can specify the option --enable-force-cgi-redirect to the configure script. You still have to make sure your PHP scripts do not rely on one or another way of calling the script, neither by directly *http://my.host/cgi-bin/php/dir/script.php* nor by redirection *http://my.host/dir/script.php*.

Redirection can be configured in Apache by using AddHandler and Action directives (see below).

Case 2: using --enable-force-cgi-redirect

This compile-time option prevents anyone from calling PHP directly with a URL like *http://my.host/cgi-bin/php/secretdir/script.php*. Instead, PHP will only parse in this mode if it has gone through a web server redirect rule.

Usually the redirection in the Apache configuration is done with the following directives:

```
Action php-script /cgi-bin/php
AddHandler php-script .php
```

This option has only been tested with the Apache web server, and relies on Apache to set the non-standard CGI environment variable REDIRECT_STATUS on redirected requests. If your web server does not support any way of telling if the request is direct or redirected, you cannot use this option and you must use one of the other ways of running the CGI version documented here.

Case 3: setting doc_root or user_dir

To include active content, like scripts and executables, in the web server document directories is sometimes considered an insecure practice. If, because of some configuration mistake, the scripts are not executed but displayed as regular HTML documents, this may result in leakage of intellectual property or security information like passwords. Therefore many sysadmins will prefer setting up another directory structure for scripts that are accessible only through the PHP CGI, and therefore always interpreted and not displayed as such.

Also if the method for making sure the requests are not redirected, as described in the previous section, is not available, it is necessary to set up a script doc_root that is different from web document root.

You can set the PHP script document root by the configuration directive doc_root in the configuration file, or you can set the environment variable PHP_DOCUMENT_ROOT. If it is set, the CGI version of PHP will always construct the file name to open with this `doc_root` and the path information in the request, so you can be sure no script is executed outside this directory (except for `user_dir` below).

Another option usable here is user_dir. When user_dir is unset, only thing controlling the opened file name is `doc_root`. Opening a URL like *http://my.host/~user/doc.php* does not result in opening a file under users home directory, but a file called *~user/doc.php* under doc_root (yes, a directory name starting with a tilde [~]).

If user_dir is set to for example *public_php*, a request like *http://my.host/~user/doc.php* will open a file called *doc.php* under the directory named *public_php* under the home directory of the user. If the home of the user is */home/user*, the file executed is */home/user/public_php/doc.php*.

`user_dir` expansion happens regardless of the `doc_root` setting, so you can control the document root and user directory access separately.

Case 4: PHP parser outside of web tree

A very secure option is to put the PHP parser binary somewhere outside of the web tree of files. In */usr/local/bin*, for example. The only real downside to this option is that you will now have to put a line similar to:

```
#!/usr/local/bin/php
```

as the first line of any file containing PHP tags. You will also need to make the file executable. That is, treat it exactly as you would treat any other CGI script written in Perl or sh or any other common scripting language which uses the *#!* shell-escape mechanism for launching itself.

To get PHP to handle PATH_INFO and PATH_TRANSLATED information correctly with this setup, the PHP parser should be compiled with the --enable-discard-path configure option.

Installed as an Apache module

When PHP is used as an Apache module it inherits Apache's user permissions (typically those of the "nobody" user). This has several impacts on security and authorization. For example, if you are using PHP to access a database, unless that database has built-in access control, you will have to make the database accessible to the "nobody" user. This means a malicious script could access and modify the database, even without a username and password. It's entirely possible that a web spider could stumble across a database administrator's web page, and drop all of your databases. You can protect against this with Apache authorization, or you can design your own access model using LDAP,*.htaccess* files, etc. and include that code as part of your PHP scripts.

Often, once security is established to the point where the PHP user (in this case, the apache user) has very little risk attached to it, it is discovered that PHP is now prevented from writing any files to user directories. Or perhaps it has been prevented from accessing or changing databases. It has equally been secured from writing good and bad files, or entering good and bad database transactions.

A frequent security mistake made at this point is to allow apache root permissions, or to escalate apache's abilities in some other way.

Escalating the Apache user's permissions to root is extremely dangerous and may compromise the entire system, so sudo'ing, chroot'ing, or otherwise running as root should not be considered by those who are not security professionals.

There are some simpler solutions. By using open_basedir you can control and restrict what directories are allowed to be used for PHP. You can also set up apache-only areas, to restrict all web based activity to non-user, or non-system, files.

Filesystem Security

PHP is subject to the security built into most server systems with respect to permissions on a file and directory basis. This allows you to control which files in the filesystem may be read. Care should be taken with any files which are world readable to ensure that they are safe for reading by all users who have access to that filesystem.

Since PHP was designed to allow user level access to the filesystem, it's entirely possible to write a PHP script that will allow you to read system files such as /etc/passwd, modify your ethernet connections, send massive printer jobs out, etc. This has some obvious implications, in that you need to ensure that the files that you read from and write to are the appropriate ones.

Consider the following script, where a user indicates that they'd like to delete a file in their home directory. This assumes a situation where a PHP web interface is regularly used for file management, so the Apache user is allowed to delete files in the user home directories.

Example #1 - Poor variable checking leads to....

```
<?php
// remove a file from the user's home directory
$username = $_POST['user_submitted_name'];
$userfile = $_POST['user_submitted_filename'];
$homedir  = "/home/$username";

unlink("$homedir/$userfile");

echo "The file has been deleted!";
?>
```

Since the username and the filename are postable from a user form, they can submit a username and a filename belonging to someone else, and delete it even if they're not supposed to be allowed to do so. In this case, you'd want to use some other form of authentication. Consider what could happen if the variables submitted were "../etc/" and "passwd". The code would then effectively read:

Example #2 -... A filesystem attack

```
<?php
// removes a file from anywhere on the hard drive that
// the PHP user has access to. If PHP has root access:
$username = $_POST['user_submitted_name']; // "../etc"
$userfile = $_POST['user_submitted_filename']; // "passwd"
$homedir  = "/home/$username"; // "/home/../etc"

unlink("$homedir/$userfile"); // "/home/../etc/passwd"

echo "The file has been deleted!";
?>
```

There are two important measures you should take to prevent these issues.

- Only allow limited permissions to the PHP web user binary.
- Check all variables which are submitted.

Here is an improved script:

Example #3 - More secure file name checking

```
<?php
// removes a file from the hard drive that
// the PHP user has access to.
$username = $_SERVER['REMOTE_USER']; // using an authentication mechanisim
$userfile = basename($_POST['user_submitted_filename']);
$homedir  = "/home/$username";

$filepath = "$homedir/$userfile";

if (file_exists($filepath) && unlink($filepath)) {
   $logstring = "Deleted $filepath\n";
} else {
   $logstring = "Failed to delete $filepath\n";
}
$fp = fopen("/home/logging/filedelete.log", "a");
fwrite($fp, $lo gstring);
fclose($fp);

echo htmlentities($logstring, ENT_QUOTES);

?>
```

However, even this is not without its flaws. If your authentication system allowed users to create their own user logins, and a user chose the login "../etc/", the system is once again exposed. For this reason, you may prefer to write a more customized check:

Example #4 - More secure file name checking

```
<?php
$username     = $_SERVER['REMOTE_USER']; // using an authentication
mechanisim
$userfile     = $_POST['user_submitted_filename'];
$homedir      = "/home/$username";

$filepath     = "$homedir/$userfile";

if (!ctype_alnum($username) || !preg_match('/^(?:[a-z0-9_-]|\.(?!\.))+$/iD',
$userfile)) {
   die("Bad username/filename");
}

//etc...
?>
```

Depending on your operating system, there are a wide variety of files which you should be concerned about, including device entries (/dev/ or COM1), configuration files (/etc/ files

and the .ini files), well known file storage areas (/home/, My Documents), etc. For this reason, it's usually easier to create a policy where you forbid everything except for what you explicitly allow.

Null bytes related issues

As PHP uses the underlying C functions for filesystem related operations, it may handle null bytes in a quite unexpected way. As null bytes denote the end of a string in C, strings containing them won't be considered entirely but rather only until a null byte occurs. The following example shows a vulnerable code that demonstrates this problem:

Example #5 - Script vulnerable to null bytes

```
<?php
$file = $_GET['file']; // "../../etc/passwd\0"
if (file_exists('/home/wwwrun/'.$file.'.php')) {
   // file_exists will return true as the file /home/wwwrun/../../etc/passwd
exists
   include '/home/wwwrun/'.$file.'.php';
   // the file /etc/passwd will be included
}
?>
```

Therefore, any tainted string that is used in a filesystem operation should always be validated properly. Here is a better version of the previous example:

Example #6 - Correctly validating the input

```
<?php
$file = $_GET['file'];

// Whitelisting possible values
switch ($file) {
   case 'main':
   case 'foo':
   case 'bar':
       include '/home/wwwrun/include/'.$file.'.php';
       break;
   default:
       include '/home/wwwrun/include/main.php';
}
?>
```

Database Security

Nowadays, databases are cardinal components of any web based application by enabling websites to provide varying dynamic content. Since very sensitive or secret information can be stored in a database, you should strongly consider protecting your databases.

To retrieve or to store any information you need to connect to the database, send a legitimate query, fetch the result, and close the connection. Nowadays, the commonly used query language in this interaction is the Structured Query Language (SQL). See how an attacker can tamper with an SQL query.

As you can surmise, PHP cannot protect your database by itself. The following sections aim to be an introduction into the very basics of how to access and manipulate databases within PHP scripts.

Keep in mind this simple rule: defense in depth. The more places you take action to increase the protection of your database, the less probability of an attacker succeeding in exposing or abusing any stored information. Good design of the database schema and the application deals with your greatest fears.

Designing Databases

The first step is always to create the database, unless you want to use one from a third party. When a database is created, it is assigned to an owner, who executed the creation statement. Usually, only the owner (or a superuser) can do anything with the objects in that database, and in order to allow other users to use it, privileges must be granted.

Applications should never connect to the database as its owner or a superuser, because these users can execute any query at will, for example, modifying the schema (e.g. dropping tables) or deleting its entire content.

You may create different database users for every aspect of your application with very limited rights to database objects. The most required privileges should be granted only, and avoid that the same user can interact with the database in different use cases. This means that if intruders gain access to your database using your applications credentials, they can only effect as many changes as your application can.

You are encouraged not to implement all the business logic in the web application (i.e. your script), instead do it in the database schema using views, triggers or rules. If the system evolves, new ports will be intended to open to the database, and you have to re-implement the logic in each separate database client. Over and above, triggers can be used to transparently and automatically handle fields, which often provides insight when debugging problems with your application or tracing back transactions.

Connecting to Database

You may want to establish the connections over SSL to encrypt client/server

communications for increased security, or you can use ssh to encrypt the network connection between clients and the database server. If either of these is used, then monitoring your traffic and gaining information about your database will be difficult for a would-be attacker.

Encrypted Storage Model

SSL/SSH protects data travelling from the client to the server, SSL/SSH does not protect the persistent data stored in a database. SSL is an on-the-wire protocol.

Once an attacker gains access to your database directly (bypassing the webserver), the stored sensitive data may be exposed or misused, unless the information is protected by the database itself. Encrypting the data is a good way to mitigate this threat, but very few databases offer this type of data encryption.

The easiest way to work around this problem is to first create your own encryption package, and then use it from within your PHP scripts. PHP can assist you in this with several extensions, such as Mcrypt and Mhash, covering a wide variety of encryption algorithms. The script encrypts the data before inserting it into the database, and decrypts it when retrieving. See the references for further examples of how encryption works.

In case of truly hidden data, if its raw representation is not needed (i.e. not be displayed), hashing may also be taken into consideration. The well-known example for the hashing is storing the MD5 hash of a password in a database, instead of the password itself. See also crypt() and md5().

Example #7 - Using hashed password field

```
<?php

// storing password hash
$query  = sprintf("INSERT INTO users(name,pwd) VALUES('%s','%s');",
            pg_escape_string($username), md5($password));
$result = pg_query($connection, $query);

// querying if user submitted the right password
$query = sprintf("SELECT 1 FROM users WHERE name='%s' AND pwd='%s';",
            pg_escape_string($username), md5($password));
$result = pg_query($connection, $query);

if (pg_num_rows($result) > 0) {
   echo 'Welcome, $username!';
} else {
   echo 'Authentication failed for $username.';
}

?>
```

SQL Injection

Many web developers are unaware of how SQL queries can be tampered with, and assume that an SQL query is a trusted command. It means that SQL queries are able to circumvent access controls, thereby bypassing standard authentication and authorization checks, and sometimes SQL queries even may allow access to host operating system level commands.

Direct SQL Command Injection is a technique where an attacker creates or alters existing SQL commands to expose hidden data, or to override valuable ones, or even to execute dangerous system level commands on the database host. This is accomplished by the application taking user input and combining it with static parameters to build a SQL query. The following examples are based on true stories, unfortunately.

Owing to the lack of input validation and connecting to the database on behalf of a superuser or the one who can create users, the attacker may create a superuser in your database.

Example #8 - Splitting the result set into pages ... and making superusers (PostgreSQL)

```
<?php

$offset = $argv[0]; // beware, no input validation!
$query  = "SELECT id, name FROM products ORDER BY name LIMIT 20 OFFSET
$offset;";
$result = pg_query($conn, $query);

?>
```

Normal users click on the 'next', 'prev' links where the *$offset* is encoded into the URL. The script expects that the incoming *$offset* is a decimal number. However, what if someone tries to break in by appending a urlencode() 'd form of the following to the URL

```
0;
insert into pg_shadow(usename,usesysid,usesuper,usecatupd,passwd)
   select 'crack', usesysid, 't','t','crack'
   from pg_shadow where usename='postgres';
--
```

If it happened, then the script would present a superuser access to him. Note that *0;* is to supply a valid offset to the original query and to terminate it.

Note

It is common technique to force the SQL parser to ignore the rest of the query written by the developer with -- which is the comment sign in SQL.

A feasible way to gain passwords is to circumvent your search result pages. The only thing the attacker needs to do is to see if there are any submitted variables used in SQL statements which are not handled properly. These filters can be set commonly in a preceding form to customize *WHERE, ORDER BY, LIMIT* and *OFFSET* clauses in *SELECT* statements. If your database supports the *UNION* construct, the attacker may try to append an entire query to the original one to list passwords from an arbitrary table.

Using encrypted password fields is strongly encouraged.

Example #9 - Listing out articles ... and some passwords (any database server)

```
<?php

$query  = "SELECT id, name, inserted, size FROM products
                WHERE size = '$size'
                ORDER BY $order LIMIT $limit, $offset;";
$result = odbc_exec($conn, $query);

?>
```

The static part of the query can be combined with another *SELECT* statement which reveals all passwords:

```
'
union select '1', concat(uname||'-'||passwd) as name, '1971-01-01', '0' from
usertable;
--
```

If this query (playing with the *'* and --) were assigned to one of the variables used in *$query*, the query beast awakened.

SQL UPDATE's are also susceptible to attack. These queries are also threatened by chopping and appending an entirely new query to it. But the attacker might fiddle with the *SET* clause. In this case some schema information must be possessed to manipulate the query successfully. This can be acquired by examining the form variable names, or just simply brute forcing. There are not so many naming conventions for fields storing passwords or usernames.

Example #10 - From resetting a password ... to gaining more privileges (any database server)

```
<?php
$query = "UPDATE usertable SET pwd='$pwd' WHERE uid='$uid';";
?>
```

But a malicious user sumbits the value *' or uid like'%admin%';* -- to *$uid* to change the admin's password, or simply sets *$pwd* to *"hehehe', admin='yes', trusted=100 "* (with a trailing space) to gain more privileges. Then, the query will be twisted:

```
<?php

// $uid == ' or uid like'%admin%'; --
$query = "UPDATE usertable SET pwd='...' WHERE uid='' or uid like '%admin%';
--";

// $pwd == "hehehe', admin='yes', trusted=100 "
$query = "UPDATE usertable SET pwd='hehehe', admin='yes', trusted=100 WHERE
...;";

?>
```

A frightening example how operating system level commands can be accessed on some database hosts.

Example #11 - Attacking the database hosts operating system (MSSQL Server)

```
<?php

$query  = "SELECT * FROM products WHERE id LIKE '%$prod%'";
$result = mssql_query($query);

?>
```

If attacker submits the value *a%' exec master..xp_cmdshell 'net user test testpass /ADD'* -- to *$prod*, then the *$query* will be:

```
<?php

$query  = "SELECT * FROM products
                  WHERE id LIKE '%a%'
                  exec master..xp_cmdshell 'net user test testpass /ADD'--";
$result = mssql_query($query);

?>
```

MSSQL Server executes the SQL statements in the batch including a command to add a new user to the local accounts database. If this application were running as *sa* and the MSSQLSERVER service is running with sufficient privileges, the attacker would now have an account with which to access this machine.

Note

Some of the examples above is tied to a specific database server. This does not mean that a similar attack is impossible against other products. Your database server may be similarly vulnerable in another manner.

Avoiding techniques

You may plead that the attacker must possess a piece of information about the database schema in most examples. You are right, but you never know when and how it can be taken out, and if it happens, your database may be exposed. If you are using an open source, or publicly available database handling package, which may belong to a content management system or forum, the intruders easily produce a copy of a piece of your code. It may be also a security risk if it is a poorly designed one.

These attacks are mainly based on exploiting the code not being written with security in mind. Never trust any kind of input, especially that which comes from the client side, even though it comes from a select box, a hidden input field or a cookie. The first example shows that such a blameless query can cause disasters.

- Never connect to the database as a superuser or as the database owner. Use always

customized users with very limited privileges.

- Check if the given input has the expected data type. PHP has a wide range of input validating functions, from the simplest ones found in Variable Functions and in Character Type Functions (e.g. is_numeric(), ctype_digit() respectively) and onwards to the Perl compatible Regular Expressions support.
- If the application waits for numerical input, consider verifying data with is_numeric(), or silently change its type using settype(), or use its numeric representation by sprintf().

Example #12 - A more secure way to compose a query for paging

```
<?php

settype($offset, 'integer');
$query = "SELECT id, name FROM products ORDER BY name LIMIT 20 OFFSET
$offset;";

// please note %d in the format string, using %s would be meaningless
$query = sprintf("SELECT id, name FROM products ORDER BY name LIMIT 20
OFFSET %d;",
                 $offset);

?>
```

- Quote each non numeric user supplied value that is passed to the database with the database-specific string escape function (e.g. mysql_real_escape_string(), **sql_escape_string()**, etc.). If a database-specific string escape mechanism is not available, the addslashes() and str_replace() functions may be useful (depending on database type). See the first example. As the example shows, adding quotes to the static part of the query is not enough, making this query easily crackable.
- Do not print out any database specific information, especially about the schema, by fair means or foul. See also Error Reporting and Error Handling and Logging Functions.
- You may use stored procedures and previously defined cursors to abstract data access so that users do not directly access tables or views, but this solution has another impacts.

Besides these, you benefit from logging queries either within your script or by the database itself, if it supports logging. Obviously, the logging is unable to prevent any harmful attempt, but it can be helpful to trace back which application has been circumvented. The log is not useful by itself, but through the information it contains. More detail is generally better than less.

Error Reporting

With PHP security, there are two sides to error reporting. One is beneficial to increasing security, the other is detrimental.

A standard attack tactic involves profiling a system by feeding it improper data, and checking for the kinds, and contexts, of the errors which are returned. This allows the system cracker to probe for information about the server, to determine possible weaknesses. For example, if an attacker had gleaned information about a page based on a prior form submission, they may attempt to override variables, or modify them:

Example #13 - Attacking Variables with a custom HTML page

```
<form method="post"
action="attacktarget?username=badfoo&amp;password=badfoo">
<input type="hidden" name="username" value="badfoo" />
<input type="hidden" name="password" value="badfoo" />
</form>
```

The PHP errors which are normally returned can be quite helpful to a developer who is trying to debug a script, indicating such things as the function or file that failed, the PHP file it failed in, and the line number which the failure occurred in. This is all information that can be exploited. It is not uncommon for a php developer to use show_source(), highlight_string(), or highlight_file() as a debugging measure, but in a live site, this can expose hidden variables, unchecked syntax, and other dangerous information. Especially dangerous is running code from known sources with built-in debugging handlers, or using common debugging techniques. If the attacker can determine what general technique you are using, they may try to brute-force a page, by sending various common debugging strings:

Example #14 - Exploiting common debugging variables

```
<form method="post"
action="attacktarget?errors=Y&amp;showerrors=1&amp;debug=1">
<input type="hidden" name="errors" value="Y" />
<input type="hidden" name="showerrors" value="1" />
<input type="hidden" name="debug" value="1" />
</form>
```

Regardless of the method of error handling, the ability to probe a system for errors leads to providing an attacker with more information.

For example, the very style of a generic PHP error indicates a system is running PHP. If the attacker was looking at an .html page, and wanted to probe for the back-end (to look for known weaknesses in the system), by feeding it the wrong data they may be able to determine that a system was built with PHP.

A function error can indicate whether a system may be running a specific database engine, or give clues as to how a web page or programmed or designed. This allows for deeper investigation into open database ports, or to look for specific bugs or weaknesses in a web page. By feeding different pieces of bad data, for example, an attacker can determine the order of authentication in a script, (from the line number errors) as well as probe for exploits that may be exploited in different locations in the script.

A filesystem or general PHP error can indicate what permissions the web server has, as well as the structure and organization of files on the web server. Developer written error code can aggravate this problem, leading to easy exploitation of formerly "hidden" information.

There are three major solutions to this issue. The first is to scrutinize all functions, and attempt to compensate for the bulk of the errors. The second is to disable error reporting entirely on the running code. The third is to use PHP's custom error handling functions to create your own error handler. Depending on your security policy, you may find all three to be applicable to your situation.

One way of catching this issue ahead of time is to make use of PHP's own error_reporting(), to help you secure your code and find variable usage that may be dangerous. By testing your code, prior to deployment, with E_ALL, you can quickly find areas where your variables may be open to poisoning or modification in other ways. Once you are ready for deployment, you should either disable error reporting completely by setting error_reporting() to 0, or turn off the error display using the *php.ini* option *display_errors*, to insulate your code from probing. If you choose to do the latter, you should also define the path to your log file using the *error_log* ini directive, and turn *log_errors* on.

Example #15 - Finding dangerous variables with E_ALL

```
<?php
if ($username) {  // Not initialized or checked before usage
   $good_login = 1;
}
if ($good_login == 1) { // If above test fails, not initialized or checked
before usage
   readfile ("/highly/sensitive/data/index.html");
}
?>
```

Using Register Globals

Warning

This feature has been *DEPRECATED* and *REMOVED* as of PHP 6.0.0. Relying on this feature is highly discouraged.

Perhaps the most controversial change in PHP is when the default value for the PHP directive register_globals went from ON to OFF in PHP » 4.2.0. Reliance on this directive was quite common and many people didn't even know it existed and assumed it's just how PHP works. This page will explain how one can write insecure code with this directive but keep in mind that the directive itself isn't insecure but rather it's the misuse of it.

When on, register_globals will inject your scripts with all sorts of variables, like request variables from HTML forms. This coupled with the fact that PHP doesn't require variable initialization means writing insecure code is that much easier. It was a difficult decision, but the PHP community decided to disable this directive by default. When on, people use variables yet really don't know for sure where they come from and can only assume. Internal variables that are defined in the script itself get mixed up with request data sent by users and disabling register_globals changes this. Let's demonstrate with an example misuse of register_globals:

Example #16 - Example misuse with register_globals = on

```
<?php
// define $authorized = true only if user is authenticated
if (authenticated_user()) {
   $authorized = true;
}

// Because we didn't first initialize $authorized as false, this might be
// defined through register_globals, like from GET auth.php?authorized=1
// So, anyone can be seen as authenticated!
if ($authorized) {
   include "/highly/sensitive/data.php";
}
?>
```

When register_globals = on, our logic above may be compromised. When off, *$authorized* can't be set via request so it'll be fine, although it really is generally a good programming practice to initialize variables first. For example, in our example above we might have first done *$authorized = false*. Doing this first means our above code would work with register_globals on or off as users by default would be unauthorized.

Another example is that of sessions. When register_globals = on, we could also use *$username* in our example below but again you must realize that *$username* could also

come from other means, such as GET (through the URL).

Example #17 - Example use of sessions with register_globals on or off

```
<?php
// We wouldn't know where $username came from but do know $_SESSION is
// for session data
if (isset($_SESSION['username'])) {

   echo "Hello <b>{$_SESSION['username']}</b>";

} else {

   echo "Hello <b>Guest</b><br />";
   echo "Would you like to login?";

}
?>
```

It's even possible to take preventative measures to warn when forging is being attempted. If you know ahead of time exactly where a variable should be coming from, you can check to see if the submitted data is coming from an inappropriate kind of submission. While it doesn't guarantee that data has not been forged, it does require an attacker to guess the right kind of forging. If you don't care where the request data comes from, you can use *$_REQUEST* as it contains a mix of GET, POST and COOKIE data. See also the manual section on using variables from external sources.

Example #18 - Detecting simple variable poisoning

```
<?php
if (isset($_COOKIE['MAGIC_COOKIE'])) {

   // MAGIC_COOKIE comes from a cookie.
   // Be sure to validate the cookie data!

} elseif (isset($_GET['MAGIC_COOKIE']) || isset($_POST['MAGIC_COOKIE'])) {

  mail("admin@example.com", "Possible breakin attempt",
$_SERVER['REMOTE_ADDR']);
  echo "Security violation, admin has been alerted.";
  exit;

} else {

  // MAGIC_COOKIE isn't set through this REQUEST

}
?>
```

Of course, simply turning off register_globals does not mean your code is secure. For every piece of data that is submitted, it should also be checked in other ways. Always validate your user data and initialize your variables! To check for uninitialized variables you may turn up error_reporting() to show **E_NOTICE** level errors.

For information about emulating register_globals being On or Off, see this FAQ.

Note
Superglobals: availability note Superglobal arrays such as *$_GET*, *$_POST*, and *$_SERVER*, etc. are available as of PHP 4.1.0. For more information, read the manual section on superglobals

User Submitted Data

The greatest weakness in many PHP programs is not inherent in the language itself, but merely an issue of code not being written with security in mind. For this reason, you should always take the time to consider the implications of a given piece of code, to ascertain the possible damage if an unexpected variable is submitted to it.

Example #19 - Dangerous Variable Usage

```
<?php
// remove a file from the user's home directory... or maybe
// somebody else's?
unlink ($evil_var);

// Write logging of their access... or maybe an /etc/passwd entry?
fwrite ($fp, $evil_var);

// Execute something trivial.. or rm -rf *?
system ($evil_var);
exec ($evil_var);

?>
```

You should always carefully examine your code to make sure that any variables being submitted from a web browser are being properly checked, and ask yourself the following questions:

- Will this script only affect the intended files?
- Can unusual or undesirable data be acted upon?
- Can this script be used in unintended ways?
- Can this be used in conjunction with other scripts in a negative manner?
- Will any transactions be adequately logged?

By adequately asking these questions while writing the script, rather than later, you prevent an unfortunate re-write when you need to increase your security. By starting out with this mindset, you won't guarantee the security of your system, but you can help improve it.

You may also want to consider turning off register_globals, magic_quotes, or other convenience settings which may confuse you as to the validity, source, or value of a given variable. Working with PHP in error_reporting(E_ALL) mode can also help warn you about variables being used before they are checked or initialized (so you can prevent unusual data from being operated upon).

Magic Quotes

Warning
This feature has been *DEPRECATED* and *REMOVED* as of PHP 6.0.0. Relying on this feature is highly discouraged.

Magic Quotes is a process that automagically escapes incoming data to the PHP script. It's preferred to code with magic quotes off and to instead escape the data at runtime, as needed.

What are Magic Quotes

When on, all *'* (single-quote), *"* (double quote), \ (backslash) and *NULL* characters are escaped with a backslash automatically. This is identical to what addslashes() does.

There are three magic quote directives:

- magic_quotes_gpc Affects HTTP Request data (GET, POST, and COOKIE). Cannot be set at runtime, and defaults to *on* in PHP. See also get_magic_quotes_gpc().
- magic_quotes_runtime If enabled, most functions that return data from an external source, including databases and text files, will have quotes escaped with a backslash. Can be set at runtime, and defaults to *off* in PHP. See also set_magic_quotes_runtime() and get_magic_quotes_runtime().
- magic_quotes_sybase If enabled, a single-quote is escaped with a single-quote instead of a backslash. If on, it completely overrides magic_quotes_gpc. Having both directives enabled means only single quotes are escaped as *''*. Double quotes, backslashes and NULL's will remain untouched and unescaped. See also ini_get() for retrieving its value.

Why use Magic Quotes

- Useful for beginners Magic quotes are implemented in PHP to help code written by beginners from being dangerous. Although SQL Injection is still possible with magic quotes on, the risk is reduced.
- Convenience For inserting data into a database, magic quotes essentially runs addslashes() on all Get, Post, and Cookie data, and does so automagically.

Why not to use Magic Quotes

- Portability Assuming it to be on, or off, affects portability. Use get_magic_quotes_gpc() to check for this, and code accordingly.
- Performance Because not every piece of escaped data is inserted into a database, there is a performance loss for escaping all this data. Simply calling on the escaping functions (like addslashes()) at runtime is more efficient. Although *php.ini-dist* enables these directives by default, *php.ini-recommended* disables it. This recommendation is mainly due to performance reasons.
- Inconvenience Because not all data needs escaping, it's often annoying to see escaped data where it shouldn't be. For example, emailing from a form, and seeing a bunch of \' within the email. To fix, this may require excessive use of stripslashes().

Disabling Magic Quotes

The magic_quotes_gpc directive may only be disabled at the system level, and not at runtime. In otherwords, use of ini_set() is not an option.

Example #20 - Disabling magic quotes server side

An example that sets the value of these directives to *Off* in *php.ini*. For additional details, read the manual section titled How to change configuration settings.

```
; Magic quotes
;

; Magic quotes for incoming GET/POST/Cookie data.
magic_quotes_gpc = Off

; Magic quotes for runtime-generated data, e.g. data from SQL, from exec(),
etc.
magic_quotes_runtime = Off

; Use Sybase-style magic quotes (escape ' with '' instead of \').
magic_quotes_sybase = Off
```

If access to the server configuration is unavailable, use of*.htaccess* is also an option. For example:

```
php_flag magic_quotes_gpc Off
```

In the interest of writing portable code (code that works in any environment), like if setting at the server level is not possible, here's an example to disable magic_quotes_gpc at runtime. This method is inefficient so it's preferred to instead set the appropriate directives elsewhere.

Example #21 - Disabling magic quotes at runtime

```
<?php
if (get_magic_quotes_gpc()) {
   function stripslashes_deep($value)
   {
       $value = is_array($value) ?
                   array_map('stripslashes_deep', $value) :
                   stripslashes($value);

       return $value;
   }

   $_POST = array_map('stripslashes_deep', $_POST);
   $_GET = array_map('stripslashes_deep', $_GET);
   $_COOKIE = array_map('stripslashes_deep', $_COOKIE);
   $_REQUEST = array_map('stripslashes_deep', $_REQUEST);
}
?>
```

Hiding PHP

In general, security by obscurity is one of the weakest forms of security. But in some cases, every little bit of extra security is desirable.

A few simple techniques can help to hide PHP, possibly slowing down an attacker who is attempting to discover weaknesses in your system. By setting expose_php = off in your *php.ini* file, you reduce the amount of information available to them.

Another tactic is to configure web servers such as apache to parse different filetypes through PHP, either with an*.htaccess* directive, or in the apache configuration file itself. You can then use misleading file extensions:

Example #22 - Hiding PHP as another language

```
# Make PHP code look like other code types
AddType application/x-httpd-php .asp .py .pl
```

Or obscure it completely:

Example #23 - Using unknown types for PHP extensions

```
# Make PHP code look like unknown types
AddType application/x-httpd-php .bop .foo .133t
```

Or hide it as HTML code, which has a slight performance hit because all HTML will be parsed through the PHP engine:

Example #24 - Using HTML types for PHP extensions

```
# Make all PHP code look like HTML
AddType application/x-httpd-php .htm .html
```

For this to work effectively, you must rename your PHP files with the above extensions. While it is a form of security through obscurity, it's a minor preventative measure with few drawbacks.

Keeping Current

PHP, like any other large system, is under constant scrutiny and improvement. Each new version will often include both major and minor changes to enhance security and repair any flaws, configuration mishaps, and other issues that will affect the overall security and stability of your system.

Like other system-level scripting languages and programs, the best approach is to update often, and maintain awareness of the latest versions and their changes.