PHP: Hackers Paradise

by Nathan Wallace
http://www.e-gineer.com/articles/php-hackers-paradise.phtml

1 Introduction

PHP (http://www.php.net) is a powerful server side web scripting solution. It has quickly grown in popularity and according to the 1999 January Netcraft Web Server Survey PHP is installed on 12.8% of all web sites. Much of its syntax is borrowed from C, Java and Perl with a couple of unique PHP-specific features thrown in. The goal of the language is to allow web developers to write dynamically generated pages quickly.

Being a good PHP hacker isn't just about writing single line solutions to complex problems. For example, web gurus know that speed of coding is much more important than speed of code. In this article we'll look at techniques that can help you become a better PHP hacker. We'll assume that you have a basic knowledge of PHP and databases.

If nothing else, you should leave here with the 3 key ideals for PHP hackers:

- Laziness is a Virtue
- Chameleon Coding
- Speed of Coding, Not Speed of Code

2 Laziness is a Virtue

2.1.1 Introduction

It seems strange to think of a web programmer as lazy. Most of us work one hundred-hour week's in our quest to join the gold rush. In fact, we need to be lazy because we are so busy.

There are two key ways to be lazy. Firstly always use existing code when it is available, just integrate it into your standards and project. The second technique is to develop a library of helpful functions that let you be lazy in the future.

2.1.2 Use Other People's Code

We need to use laziness to our advantage and PHP is the perfect tool. PHP was born and raised in an open source environment. The community holds open source ideals close to its heart. As a result there are thousands of people on the mailing list willing to share their knowledge and code. There are also many open source PHP projects that you can tap into.
I'm not suggesting that you spend all day asking people to write code for you. But through clever use of the knowledge base, mailing list archives and PHP projects you can save yourself a lot of time.

PHP Knowledge Base – http://php.faqts.com

PHP Mailing List Archive - http://www.progressive-comp.com/Lists/?l=php3-general&r=1&w=2

2.1.3 Helpful Functions and Classes

2.1.3.1 Introduction

In this section we will work at developing a library of PHP code which will aid us in future development. A small amount of work now let's us be lazy in the future.

Some of this code has been taken from open source PHP projects. Other parts from the mailing list archives. In fact, all the work I really needed to do was structure the code into a coherent library of functions.

2.1.3.2 Database Abstraction

One of the features / problems with PHP is that it does not have a uniform method for accessing databases. There are specialized functions for each database PHP is able to connect to. This is a

feature because it allows you to optimize your database code. It is a problem because it makes your code less portable and increases the learning curve for newcomers.

A number of database wrapper classes have been developed to solve this problem. They provide a uniform set of functions for accessing any database. Personally I like them because I find it much easier to remember a few simple functions like query and next_record than having to think about database handles, connections and so on.

The most commonly used (and defacto standard) is PHPLib - http://phplib.netuse.de/

Metabase - http://phpclasses.UpperDesign.com/browse.html/package/20

There is also PHPDB - http://phpdb.linuxbox.com/

2.1.3.3 Session Management

The main purpose of PHPLib is session management. This allows you to associate data with visitors to your site for the duration of their stay. This can be useful for remembering options and so on.

PHP4 has session management features built into the PHP internal function library.

2.1.3.4 Debugging Variables

There is limited debugging support for PHP. This is no Smalltalk environment where you can browse objects and perform methods on them. Instead we need to make creative use of the old, reliable echo statement.

The first thing we need to be able to do is look at the value of variables. The loose typing of PHP lets us use most variables directly in strings. This is great for numbers and so on, but falls down when we are dealing with arrays and objects.

The other problem with debugging is that sometimes I'm not even sure what a variable is likely to contain. If I was, there be no need to debug.

So, lets be smart now and lazy for the rest of time. We can write a function that shows us the type and value of any variable.

```
function ss_array_as_string (&$array, $column = 0) {
    $str = "Array(<BR>\n";
    while(list($var, $val) = each($array)){
        for ($i = 0; $i < $column+1; $i++){
            $str .= "    ";
        }
        $str .= $var.' ==> ';
        $str .= ss_as_string($val, $column+1)."<BR>\n";
    }
    for ($i = 0; $i < $column; $i++){
        $str .= "    ";
    }
    return $str.')';
}

function ss_object_as_string (&$object, $column = 0) {
    if (empty($object->classname)) {
        return "$object";
    }
    else {
        $str = $object->classname."(<BR>\n";
        while (list(,$var) = each($object->persistent_slots)) {
            for ($i = 0; $i < $column; $i++){
                $str .= "    ";
```

```
                }
                global $$var;
                $str .= $var.' ==> ';
                $str .= ss_as_string($$var, column+1)."<BR>\n";
            }
            for ($i = 0; $i < $column; $i++){
                $str .= "    ";
            }
            return $str.')';
        }
    }

    function ss_as_string (&$thing, $column = 0) {
        if (is_object($thing)) {
            return ss_object_as_string($thing, $column);
        }
        elseif (is_array($thing)) {
            return ss_array_as_string($thing, $column);
        }
        elseif (is_double($thing)) {
            return "Double(".$thing.")";
        }
        elseif (is_long($thing)) {
            return "Long(".$thing.")";
        }
        elseif (is_string($thing)) {
            return "String(".$thing.")";
        }
        else {
            return "Unknown(".$thing.")";
        }
    }
```

Note that these functions work together to correctly print, format and indent arrays. They are also able to print objects when they have been defined with the PHPLIB standard variables classname (the name of the class) and persistent_slots (an array of the variable names we care about).

Now we can see the state of any variable by just doing:

```
echo ss_as_string($my_variable);
```

We can see the value of all variables currently defined in the PHP namespace with:

```
echo ss_as_string($GLOBALS);
```

2.1.3.5 Log Functions

A great way to debug is through logging. It's even easier if you can leave the log messages through your code and turn them on and off with a single command. To facilitate this we will create a number of logging functions.

```
    $ss_log_level = 0;
    $ss_log_filename = '/tmp/ss-log';
    $ss_log_levels = array(
        NONE  => 0,
        ERROR => 1,
        INFO  => 2,
        DEBUG => 3);

    function ss_log_set_level ($level = ERROR) {
        global $ss_log_level;
```

```
    $ss_log_level = $level;
}

function ss_log ($level, $message) {
    global $ss_log_level, $ss_log_filename;
    if ($ss_log_levels[$ss_log_level] < $ss_log_levels[$level])
{
        // no logging to be done
        return false;
    }
    $fd = fopen($ss_log_filename, "a+");
    fputs($fd, $level.' - ['.ss_timestamp_pretty().'] -
'.$message."\n");
    fclose($fd);
    return true;
}

function ss_log_reset () {
    global $ss_log_filename;
    @unlink($ss_log_filename);
}
```

There are 4 logging levels available. Log messages will only be displayed if they are at a level less verbose than that currently set. So, we can turn on logging with the following command:

```
ss_log_set_level(INFO);
```

Now any log messages from the levels ERROR or INFO will be recorded. DEBUG messages will be ignored. We can have as many log entries as we like. They take the form:

```
ss_log(ERROR, "testing level ERROR");
ss_log(INFO, "testing level INFO");
ss_log(DEBUG, "testing level DEBUG");
```

This will add the following entries to the log:

```
ERROR - [Feb 10, 2000 20:58:17] - testing level ERROR
INFO - [Feb 10, 2000 20:58:17] - testing level INFO
```

You can empty the log at any time with:

```
ss_log_reset();
```

2.1.3.6 Optimization

We need a way to test the execution speed of our code before we can easily perform optimizations. A set of timing functions that utilize microtime() is the easiest method:

```
function ss_timing_start ($name = 'default') {
    global $ss_timing_start_times;
    $ss_timing_start_times[$name] = explode(' ', microtime());
}

function ss_timing_stop ($name = 'default') {
    global $ss_timing_stop_times;
    $ss_timing_stop_times[$name] = explode(' ', microtime());
}

function ss_timing_current ($name = 'default') {
    global $ss_timing_start_times, $ss_timing_stop_times;
    if (!isset($ss_timing_start_times[$name])) {
```

```
        return 0;
    }
    if (!isset($ss_timing_stop_times[$name])) {
        $stop_time = explode(' ', microtime());
    }
    else {
        $stop_time = $ss_timing_stop_times[$name];
    }
    // do the big numbers first so the small ones aren't lost
    $current = $stop_time[1] -
$ss_timing_start_times[$name][1];
    $current += $stop_time[0] -
$ss_timing_start_times[$name][0];
    return $current;
}
```

Now we can check the execution time of any code very easily. We can even run a number of execution time checks simultaneously because we have established named timers.

See the optimizations section below for the examination of echo versus inline coding for an example of the use of these functions.

2.1.3.7 Debugging and Optimizing Database Operations

The best way to gauge the stress you are placing on the database with your pages is through observation. We will combine the logging and timing code above to assist us in this process.

We will alter the query() function in PHPLib, adding debugging and optimizing capabilities that we can enable and disable easily.

```
function query($Query_String, $halt_on_error = 1) {
    $this->connect();
    ss_timing_start();
    $this->Query_ID = @mysql_query($Query_String,$this-
>Link_ID);
    ss_timing_stop();
    ss_log(INFO, ss_timing_current().' Secs - '.$Query_String);
    $this->Row   = 0;
    $this->Errno = mysql_errno();
    $this->Error = mysql_error();
    if ($halt_on_error && !$this->Query_ID) {
      $this->halt("Invalid SQL: ".$Query_String);
    }
    return $this->Query_ID;
}
```

3 Chameleon Coding

3.1.1 Introduction

A chameleon is a lizard that is well known for its ability to change skin color. This is a useful metaphor for web programming as it highlights the importance of separating well structured and stable backend code from the dynamic web pages it supports.

PHP is the perfect language for chameleon coding as it supports both structured classes and simple web scripting.

3.1.2 Structuring your PHP Code

3.1.2.1 Introduction

When writing PHP code we need to make a clear distinction between the code which does the principal work of the application and the code which is used to display that work to the user.

The backend code does the difficult tasks like talking to the database, logging, and performing calculations.

The pages that display the interface to these operations are part of the front end.

3.1.2.2 Dynamic, Hackable Frontend Code

Mixing programming code in with HTML is messy. We can talk about ways to format the code or structure your pages, but the end result will still be quite complicated.

We need to move as much of the code away from the HTML as possible. But, we need to do this so that we don't get lost in the interaction between our application and the user interface.

A web site is a dynamic target. It is continually evolving, improving and changing. We need to keep our HTML pages simple so that these changes can be made quickly and easily. The best way to do that is by making all calls to PHP code simple and their results obvious.

We shouldn't worry too much about the structure of the PHP code contained in the front end, it will change soon anyway.

That means that we need to remove all structured code from the actual pages into the supporting include files. All common operations should be encapsulated into functions contained in the backend.

3.1.2.3 Stable, Structured Backend Code

In complete contrast to the web pages your backend code should be well designed, documented and structured. All the time you invest here is well spent, next time you need a page quickly hacked together all the hard parts will be already done waiting for you in backend functions.

Your backend code should be arranged into a set of include files. These should be either included dynamically when required, or automatically included in all pages through the use of the php3_auto_prepend_file directive.

If you need to include HTML in your backend code it should be as generic as possible. All presentation and layout should really be contained in the front end code. Exceptions to this rule are obvious when they arise, for example, the creation of select boxes for a date selection form.

PHP is flexible enough to let you design your code using classes and or functions. My object oriented background means that I like to create a class to represent each facet of the application. All database queries are encapsulated in these classes, hidden from the front end pages completely. This helps by keeping all database code in a single location and simplifying the PHP code contained in pages.

3.1.3 Coding Techniques

3.1.3.1 Include Files

If we are building these function libraries we need to work out a scheme for including them in our pages. There are a couple of different approaches to this.

We can either include all our library files all the time, or include them conditionally as required.

As part of my speed of coding philosophy I prefer to just include all the files and never think about it again. When the Zend optimizing engine becomes available to pre-parse this code the performance hit will not be significant.

I have about 10,000 lines of code in PHP libraries for my site. A quick check using the timing functions will tell us the damage:

```
<?php
require('timing.inc');
ss_timing_start();
// include other library files here
ss_timing_stop();
echo '<h1>'.ss_timing_current().'</h1>';
?>
```

It seems to take about 0.6 seconds to parse all my function libraries. My sites do not receive millions of hits so this penalty is not important enough to worry about yet.

One drawback of including all libraries all the time is that it makes it difficult to work on them. One mistake in any of those files will bring down every page on the entire site. Be very, very careful.

If you are not as lazy as me then perhaps you'd prefer the conditional include technique. It's simple to use and implement. Just structure all of your library files like the example below:

```
<?php // liba.inc

if ( defined( '__LIBA_INC' ) ) return;
define( '__LIBA_INC', 1 );

/*
 * Library code here.
 */

?>
```

Then you just need to include this library in any script where it is used. Libraries may also need to include other libraries. Your include statements look the same as normal:

```
include('liba.inc');
```

This way, the calling scripts don't have to do any of the work. Unfortunately return won't work from require()d files in PHP4 anymore. So, you will need to use include() instead. You can still use require() in PHP3.

3.1.3.2 Design Patterns for Web Programming

Some of the best web programming techniques are captured in the Web Programming Design Patterns. They are high level descriptions of the best solutions to common web programming problems. You can read more about these here:

```
http://www.e-gineer.com/articles/design-patterns-in-web-
programming.phtml
```

4 Speed of Coding, Not Speed of Code

4.1.1 Introduction

The hardest thing for me to learn as a web programmer was to change the way I wrote code. Coming from a product development and university background the emphasis is on doing it the right way. Products have to be as close to perfect as possible before release. School assignments need to be perfect.

The web is different. Here it is more important to finish a project as soon as possible than it is to get it perfect first time. Web sites are evolutionary, there is no freeze date after which it is difficult to make changes.

I like to think of my web sites as prototypes. Everyday they get a little closer to being finished. I can throw together 3 pages in the time it would take to do one perfectly. It's usually better on the web to release all three and then decide where your priorities lie. Speed is all important.

So, everything you do as a programmer should be focused on the speed at which you are producing code (pages).

4.1.2 Optimizations to Satisfy Your Hacker Instinct

4.1.2.1 Introduction

This section describes some tricks you can use to speed up your PHP code. Most of them make very little difference when compared to the time taken for parsing, database queries and sending data down a modem.

They are useful to know both so you can feel you are optimizing your code and to aid your understanding of certain PHP concepts.

4.1.2.2 Use Inline Tags Instead of echo

The PHP interpreter gets invoked once for each page. Whatever is not contained in PHP tags like <? ?> is just echoed back out by the interpreter.

As a result it is faster to use lots of little in-line tags than it is to build massive strings or use echo statements.

Let's use the timing functions we developed above to run a quick test.

```
<h2>Test Inline Tags vs echo</h2>
<p>

<?php ss_timing_start('echo'); ?>
<?php
for ($i=0; $i<1000; $i++) {
    echo $i."<br>";
}
?>
<?php ss_timing_stop('echo'); ?>

<p>

<?php ss_timing_start(str); ?>
<?php
$str = '';
for ($i=0; $i<1000; $i++) {
    $str .= $i."<br>";
}
echo $str;
?>
<?php ss_timing_stop(str); ?>

<p>

<?php ss_timing_start(inline); ?>
<?php
```

```
for ($i=0; $i<1000; $i++) {
?>
123<br>
<?php
}
?>
<?php ss_timing_stop(inline); ?>

<p>
<br>

<h2>Results</h2>

echo - <?php echo ss_timing_current('echo') ?>

<p>

str - <?php echo ss_timing_current(str) ?>

<p>
inline - <?php echo ss_timing_current(inline) ?>
```

The results of this test averaged out to be:

```
echo   - 0.063347 secs
str    - 0.083996 secs
inline - 0.035276 secs
```

We can see that inline is clearly the fastest technique. But, when we consider that we only save 0.03 milliseconds each time we use it, the method you use to echo your values is pretty much irrelevant. A moral victory at best…

4.1.2.3 str_replace vs ereg_replace

It's predictable that the simple str_replace() will be significantly faster than ereg_replace. A quick test also reveals the time difference when we introduce a simple pattern match into the ereg_replace.

```
<h2>Test str_replace vs ereg</h2>
<p>

<?php $string = 'Testing with <i>emphasis</i>'; ?>

<?php ss_timing_start('str_replace'); ?>
<?php
for ($i=0; $i<1000; $i++) {
    str_replace('i>', 'b>', $string).'<br>';
}
?>
<?php ss_timing_stop('str_replace'); ?>

<p>

<?php ss_timing_start(ereg); ?>
<?php
for ($i=0; $i<1000; $i++) {
    ereg_replace('i>', 'b>', $string).'<br>';
}
?>
<?php ss_timing_stop(ereg); ?>

<p>
```

```
<?php ss_timing_start(ereg_pattern); ?>
<?php
for ($i=0; $i<1000; $i++) {
    ereg_replace('<([/]*)i>', '<\1b>', $string).'<br>';
}
?>
<?php ss_timing_stop(ereg_pattern); ?>

<p>
<br>

<h2>Results</h2>

str_replace - <?php echo ss_timing_current(str_replace) ?>

<p>

ereg - <?php echo ss_timing_current(ereg) ?>

<p>
ereg_pattern - <?php echo ss_timing_current(ereg_pattern) ?>
```

Here are the results. Notice how using the simple pattern in ereg_replace has almost doubled the execution time.

```
str_replace - 0.089757
ereg - 0.149406
ereg_pattern - 0.248881
```

Again, the difference of these functions relative to one another is noticable but in the context of returning a web page basically irrelevant.

4.1.2.4 Quoted Strings

PHP parses double quoted strings to look for variables. Any variable contained in a double quoted string will be resolved and inserted into the string at that location.

Single quoted strings are printed exactly as they appear. They are not parsed.

So, you should use single quoted strings where possible to reduce the work to be done by the parser.

4.1.3 Optimizations that Really Make a Difference

4.1.3.1 Reduce queries

Accessing the database is expensive. Persistent connections reduce a lot of the overhead by removing the need to connect with each request, but performing queries is still a high cost exercise compared with the execution of PHP code.

This is particularly true due to locking issues in the database. In testing you might see that individual queries to the database are actually quite fast. In production you will see the database get overloaded with many small queries as it struggles to satisfy a single large query.

4.1.3.2 Optimize your Queries

The type of queries you make to the database will have a dramatic effect on the speed of your application. Making smart use of column indexes is essential. Small changes to your SQL can result in dramatic time savings.

4.1.3.3 Avoid joins

Joins are expensive. The minute you do a join the size of the resulting table becomes the multiple of the tables being joined.

Lets look at some quick statistics to give you a feel for the cost of joins. I have created two tables, foo and big_foo. Foo contains a single column with the numbers 1-1000. Big_foo contains a single column with the numbers 1-1,000,000. So, big_foo is equivalent in size to the join of foo with itself.

```
$db->query("select * from foo");
0.032273 secs

$db->next_record();
0.00048999999999999 secs
$db->query("insert into foo values (NULL)");
0.019506 secs

$db->query("select * from foo as a, foo as b");
17.280596 secs

$db->query("select * from foo as a, foo as b where a.id >
b.id");
14.645251 secs

$db->query("select * from foo as a, foo as b where a.id =
b.id");
0.041269 secs

$db->query("select * from big_foo");
25.393672 secs
```

We can see from the results above that selecting all rows from the join of a 1000 row table is only marginally quicker than selecting all rows from a 1,000,000 row table.

It is worth noting that a join that returns a small number of rows is still very fast.

4.1.3.4 Make your pages smaller

Let's consider a typical user on a 56Kbps modem. On a good connection, they can download pages at approximately 6kBps. We were looking at optimizations above that save approximately 0.15 seconds on an extremely complex page. Reducing the size of your page by about 900 bytes will give you an equivalent saving.

Usually response times for the user are gained most easily through examination of your HTML and optimizing the use of images.

4.1.4 Gotchas

4.1.4.1 Introduction

This section will be quite small since most PHP developers are now making the switch to PHP4. The parsing engine has been completely rewritten and has removed a lot of the annoying quirks that can bring a lot of grief to both newcomers and experienced PHP programmers.

4.1.4.2 Arrays of Objects

PHP3 does not handle arrays of objects very well. The following code will NOT parse correctly:

```
$a[$i]->foo();
```

PHP does not like the object reference after the array index brackets. Instead you need to use a temporary variable:

```
$tmp = $a[$i];
$tmp->foo();
```

4.1.4.3 Calling overridden methods

PHP3 has support for classes and inheritence. You can even override functions in subclasses. Problems occur when you need to call the overridden function in the parent class. Unfortunately this is quite common as you may want to define the function in the subclass as being the original function plus some extra work. If that explanation has made you completely confused take a look at the example below.

There is a (hacky) work around. The basic idea is to define a unique method name in each class for the same method. Then the extended class can reference directly to the unique method name in its parent.

To achieve the appearance of polymorphism when using the class you just create a method with the desired name in every class definition that calls the unique method name in that class. An example will explain it better:

```
class A {
  function A() { }

  function A_dspTwo() {
    echo "A: Two<br>";
  }

  function dspTwo() {
    return $this->A_dspTwo();  // call the class A dspTwo
method
  }
}

class B extends A {
  function B() {
    $this->A(); // call the parent constructor.
  }

  function B_dspTwo() {
    $this->A_dspTwo();
    echo "B: Two<br>";
  }

  function dspTwo() {
    return $this->B_dspTwo();
  }
}

$object = new B();
$object->dspTwo();
```

This is supported by the Zend engine and will thus be supported in PHP 4.0.

4.1.4.4 Trouble with Types

PHP is a loosely typed language. That means that the variables actually do have types, but in general you do not need to worry about them. PHP will automatically convert variables between types when required.

Unfortunately there are some cases where you need to manually convert the type of variables. This can lead to confusion because they are very rare. Below is an example page to highlight how rare these cases can be:

```
<h2>Test String Integer Comparisons</h2>

<?php
$a = 1;
$b = '2';
if ($a < $b) {
    echo ss_as_string($a).' < '.ss_as_string($b);
}
else {
    echo ss_as_string($a).' >= '.ss_as_string($b);
}
?>

<p>

<?php
$a = 2;
$b = '2';
if ($a == $b) {
    echo ss_as_string($a).' == '.ss_as_string($b);
}
else {
    echo ss_as_string($a).' != '.ss_as_string($b);
}
?>

<p>

<?php
$a = array(2, '1');
if ($a[0] > $a[1]) {
    echo ss_as_string($a[0]).' > '.ss_as_string($a[1]);
}
else {
    echo ss_as_string($a[0]).' <= '.ss_as_string($a[1]);
}
?>

<p>

<?
$a = array('2', '1');
echo ss_as_string($a).'<br>sorts to<br>';
sort($a);
echo ss_as_string($a);
?>

<p>

<?
$a = array(2, 1);
echo ss_as_string($a).'<br>sorts to<br>';
sort($a);
echo ss_as_string($a);
?>

<p>
```

```
<?
$a = array('2', 1);
echo ss_as_string($a).'<br>sorts to<br>';
sort($a);
echo ss_as_string($a);
?>

<p>

<?
$a = array(2, '1');
echo ss_as_string($a).'<br>sorts to<br>';
sort($a);
echo ss_as_string($a);
?>
```

Here is the output from these tests. Notice that all the tests work correctly except for the last one, sorting array(2, '1'). We can even sort array('2', 1) without problems. The error occurs when we have multiple types in an array passed to the sort function with the order number then string.

```
Long(1) < String(2)

Long(2) == String(2)

Long(2) > String(1)

Array(
0 ==> String(2)
1 ==> String(1)
)
sorts to
Array(
0 ==> String(1)
1 ==> String(2)
)

Array(
0 ==> Long(2)
1 ==> Long(1)
)
sorts to
Array(
0 ==> Long(1)
1 ==> Long(2)
)

Array(
0 ==> String(2)
1 ==> Long(1)
)
sorts to
Array(
0 ==> Long(1)
1 ==> String(2)
)

Array(
0 ==> Long(2)
1 ==> String(1)
)
```

```
sorts to
Array(
0 ==> Long(2)
1 ==> String(1)
)
```

4.1.5 Tricky Concepts

4.1.5.1 Include vs Require

Include() and require() are slightly different. Basically, include is conditional and require is not.

This would include 'somefile' if $something is true:

```
if($something){
    include("somefile");
}
```

This would include 'somefile' unconditionally

```
if($something){
    require("somefile");
}
```

This would have VERY strange effects if somefile looked like:

```
} echo "Ha!  I'm here regardless of something:
$something<br>\n";
if (false) {
```

Another interesting example is to consider what will happen if you use include() or require() inside a loop.

```
$i = 1;
while ($i < 3) {
    require("somefile.$i");
    $i++;
}
```

Using require() as above will cause the same file to be used every single iteration. Clearly this is not the intention since the file name should be changing in each iteration of the loop. We need to use include() as below. Include() will be evaluated at each iteration of the loop including somefile.0, somefile.1, etc as expected.

```
$i = 1;
while ($i < 3) {
    include("somefile.$i");
    $i++;
}
```

The only interesting question that remains is what file will be required above. It turns out that PHP uses the value of $i when it reads the require() statement for the first time. So, the require() loop above will include something.1 two times. The include() loop includes something.1 and something.2.

4.1.5.2 Echo vs Print

There is a difference between the two, but speed-wise it should be irrelevant which one you use. print() behaves like a function in that you can do:

```
$ret = print "Hello World";
```

and $ret will be 1.

That means that print can be used as part of a more complex expression where echo cannot. print is also part of the precedence table which it needs to be if it is to be used
within a complex expression. It is just about at the bottom of the precendence list though. Only "," AND, OR and XOR are lower.

echo is marginally faster since it doesn't set a return value if you really want to get down to the nitty gritty.

If the grammar is:

```
echo expression [, expression[, expression] ... ]
```

Then

```
echo ( expression, expression )
```

is not valid. (expression) reduces to just an expression so this would be valid:

```
echo ("howdy"),("partner");
```

but you would simply write this as:

```
echo "howdy","partner";
```

if you wanted to use two expressions. Putting the brackets in there serves no purpose since there is no operator precendence issue with a single expression like that.

4.1.6 Scripting with PHP

It's easy to forget that PHP is a complete programming language that can be used for more than just generating web pages. I was once writing a script to receive emails and place them in a database. I was fumbling around in Perl and shell scripts until it dawned on me to install PHP for scripting. 30 minutes later the emails were churning in.

Installing PHP for scripting on unix is easy. Just remove the –with-apache directive from your configure options. This will create the PHP binary that can be used to run scripts directly from the command line. There are complete instructions for installing PHP for scripting here:

```
http://www.e-gineer.com/instructions
```

You can then write your script like any other shell script. Here is an example:

```
#!/usr/local/bin/php -q
<?php
// your php code here
?>
```

Once you start scripting with PHP the possibilities are endless. It's a fully featured language, you can do anything you would normally do in a shell script.

4.1.7 Extreme Programming

We are getting a little off topic here, but I believe programming techniques are an important part of being a good programmer.

My working style is based on the ideas of Extreme Programming. >From the Extreme Programming web site:

XP improves a software project in four essential ways; communication, simplicity, feedback, and courage. XP programmers communicate with their customers and fellow programmers. They keep their design simple and clean. They get feedback by testing their software starting on day one. They deliver the system to the customers as early as possible and implement changes as suggested. With this foundation XP programmers are able to courageously respond to changing requirements and technology.

The focus on speed and change is what makes Extreme Programming so suitable for web projects.

You can learn more about Extreme Programming here:

`http://www.extremeprogramming.org`

5 Getting Help

There are a number of resources available for PHP help. The PHP community is generous with its time and assistance. Make use of their contributions and use the time you save to help others.

The PHP Knowledge Base is a growing collection of PHP related information. It captures the knowledge from the mailing list into a complete collection of searchable, correct answers. Of course, I may be a little biased.

`http://php.faqts.com`

The PHP manual is a great reference point for information on functions or language constructs.

`http://www.php.net/manual`

If you can't find the relevant information in the PHP Knowledge Base your next stop should be the mailing list archives. There are thousands of questions on the mailing list every month so you can be almost certain your question has been asked before. Prepare to do some wading.

`http://www.progressive-comp.com/Lists/?l=php3-general&r=1&w=2`

If all that searching fails to help, try asking on the mailing list. A lot of PHP gurus reside there.

`php3@lists.php.net`

If all these on-line resources aren't enough or you hate reading from a computer screen, you might be interested in one of the many PHP books that are now available.

`http://www.php.net/books.php3`