

PHP Taint Tool: It Aint a Parser

Luke Welling

luke@omniti.com

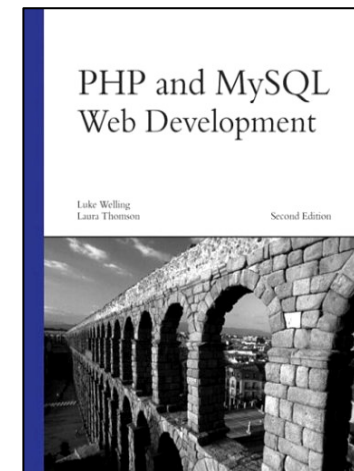
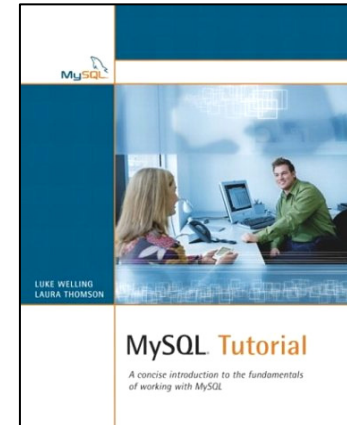
<http://lukewelling.com>



OmniTI

The Speaker

- Used PHP since last century
- More than a decade of web development experience in a range of languages
- Web Architect at OmniTI in Maryland ... but this is not a Maryland Accent



The Disclaimer



- This tool is fragile and not ready to be called alpha quality
- It is definitely not ready to be useful on large programs
- We will release it under an OSI license ... soon

The Problem

- Auditing large PHP codebases for potential security weaknesses is very time consuming,
- Really, only some lines in the application **need** individual attention.
- You want to see where untrusted input can propagate taint within the application.
- In complex logic that might mean chasing many possible execution paths.

What I Want To See

```
$message = $_GET['message'];  
...  
$entity = new Entity($message);  
...  
$this->_message = $message;  
...  
<p><b> {  
    $entity.message | escape: "html"  
} </b></p>
```

Branching increases complexity

```
$message = $_GET['message'];
```



```
$entity = new Entity($message);
```



```
$this->_message = $message;
```



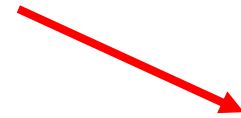
```
if($result)
```



```
$entity->appendMessage($result);
```



```
$this->_message .= $message;
```



```
<p><b>{ $entity.message|escape:"html" }</b></p>
```

2^n ?

- One conditional doubled the number of paths we have to examine
- 2^{50} is 1,125,899,906,842,624
- How many LOC do you need to find 50 conditionals?

Static Analysis



- Using an automatic tool to try to follow execution paths without running all possible input variations is called static analysis.

Previous PHP Work

- Pixy
 - <http://pixybox.seclab.tuwien.ac.at/pixy/>
- PHP-sat
 - <http://www.program-transformation.org/PHP/PhpSat>
- IBM
 - <http://ibm.com/developerworks/wikis/download/attachments/5888/PHP2006-v7.pdf>



A Different Approach



- Normally for static analysis, you take the language's specification and build a parser
- There is already a perfectly good PHP parser ... we don't really have an urge to write one, we just want the output from it

The PHP Engine



1. Lexing - Your source code is broken down into tokens.
2. Parsing – Tokens are grouped into expressions.
3. Compilation - Expressions are translated into opcodes
4. Execution - One opcode at a time

Parsekit

- Sara Goleman wrote a PECL extension to deliver compiled opcodes
- <http://www.php.net/ref.parsekit>
- <http://blog.libssh2.org/index.php?/archives/92-Understanding-Opcodes.html>

A Simple Example

```
<?php
$a = $_GET['a'];
$b = 42;
$c = $_GET['c'];
echo $a;
echo $b;
echo $c;
echo $d;
?>
```

(partial) Parsekit Output 1

```
array(13) {
  [0]=>
  array(7) {
    ["opcode"]=>
    int(80)
    ["opcode_name"]=>
    string(12) "ZEND_FETCH_R"
    ["flags"]=>
    int(2228994)
    ["result"]=>
    array(3) {
      ["type"]=>
      int(4)
      ["type_name"]=>
      string(6) "IS_VAR"
      ["var"]=>
      int(0)
    }
  }
}
```

```
["op1"]=>
array(3) {
  ["type"]=>
  int(1)
  ["type_name"]=>
  string(8) "IS_CONST"
  ["constant"]=>
  &string(4) "_GET"
}
["op2"]=>
array(4) {
  ["type"]=>
  int(8)
  ["type_name"]=>
  string(9) "IS_UNUSED"
  ["var"]=>
  int(0)
  ["EA.type"]=>
  int(0)
}
["lineno"]=>
int(2)
}
```

(partial) Parsekit Output 2

```
[1]=>
array(8) {
  ["opcode"]=>
  int(81)
  ["opcode_name"]=>
  string(16) "ZEND_FETCH_DIM_R"
  ["flags"]=>
  int(16974594)
  ["result"]=>
  array(3) {
    ["type"]=>
    int(4)
    ["type_name"]=>
    string(6) "IS_VAR"
    ["var"]=>
    int(1)
  }
  ["op1"]=>
  array(3) {
    ["type"]=>
    int(4)
    ["type_name"]=>
    string(6) "IS_VAR"
    ["var"]=>
    int(0)
  }
  ["op2"]=>
  array(3) {
    ["type"]=>
    int(1)
    ["type_name"]=>
    string(8) "IS_CONST"
    ["constant"]=>
    &string(1) "a"
  }
  ["extended_value"]=>
  int(0)
  ["lineno"]=>
  int(2)
}
```

(partial) Parsekit Output 3



```
[2]=>
array(7) {
  ["opcode"]=>
  int(38)
  ["opcode_name"]=>
  string(11) "ZEND_ASSIGN"
  ["flags"]=>
  int(197410)
  ["result"]=>
  array(4) {
    ["type"]=>
    int(4)
    ["type_name"]=>
    string(6) "IS_VAR"
    ["var"]=>
    int(2)
    ["EA.type"]=>
    int(1)
  }
}
```

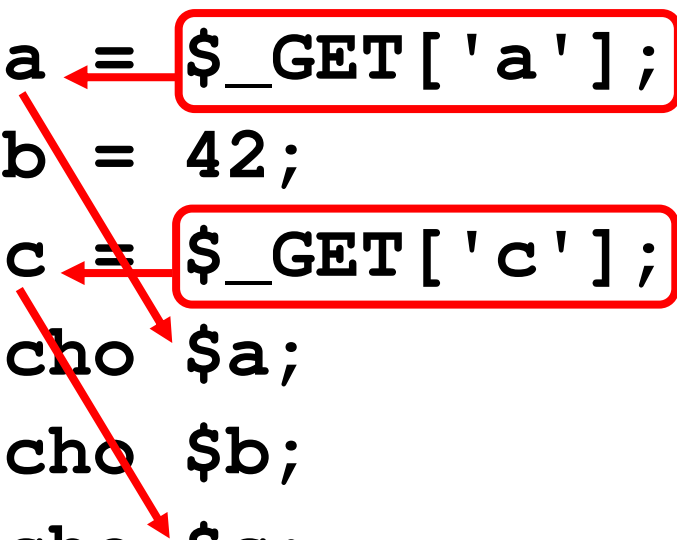
```
["op1"]=>
array(4) {
  ["type"]=>
  int(16)
  ["type_name"]=>
  string(5) "IS_CV"
  ["var"]=>
  int(0)
  ["varname"]=>
  string(1) "a"
}
["op2"]=>
array(3) {
  ["type"]=>
  int(4)
  ["type_name"]=>
  string(6) "IS_VAR"
  ["var"]=>
  int(1)
}
["lineno"]=>
int(2)
}
```

Making it Readable

```
1: <?php
2: $a = $_GET['a'];
   0: ZEND_FETCH_R( IS_CONST("_GET") IS_UNUSED(0, "0")) result(IS_VAR(0, ""))
   1: ZEND_FETCH_DIM_R( IS_VAR(0, "") IS_CONST("a")) result(IS_VAR(1, ""))
   2: ZEND_ASSIGN( IS_CV(0, "a") IS_VAR(1, "")) result(IS_VAR(2, "1"))
3: $b = 42;
   3: ZEND_ASSIGN( IS_CV(1, "b") IS_CONST("42")) result(IS_VAR(3, "1"))
4: $c = $_GET['c'];
   4: ZEND_FETCH_R( IS_CONST("_GET") IS_UNUSED(0, "0")) result(IS_VAR(4, ""))
   5: ZEND_FETCH_DIM_R( IS_VAR(4, "") IS_CONST("c")) result(IS_VAR(5, ""))
   6: ZEND_ASSIGN( IS_CV(2, "c") IS_VAR(5, "")) result(IS_VAR(6, "1"))
5: echo $a;
   7: ZEND_ECHO( IS_CV(0, "a"))
6: echo $b;
   8: ZEND_ECHO( IS_CV(1, "b"))
7: echo $c;
   9: ZEND_ECHO( IS_CV(2, "c"))
8: echo $d;
  10: ZEND_ECHO( IS_CV(3, "d"))
9: ?>
10:
  11: ZEND_RETURN( IS_CONST("1"))
  12: ZEND_HANDLE_EXCEPTION()
```

Simple To Follow by Eye

```
<?php
$a = $_GET['a'];
$b = 42;
$c = $_GET['c'];
echo $a;
echo $b;
echo $c;
echo $d;
?>
```



Tracing the Opcodes

code/vsimple.php

```
1: <?php
2: $a = $_GET['a'];
   0: ZEND_FETCH_R( IS_CONST("_GET") IS_UNUSED(0, "0") result(IS_VAR(0, ""))
   1: ZEND_FETCH_DIM_R( IS_VAR(0, "") IS_CONST("a") result(IS_VAR(1, ""))
   2: ZEND_ASSIGN( IS_CV(0, "a") IS_VAR(1, "") result(IS_VAR(2, "1"))
3: $b = 42;
   3: ZEND_ASSIGN( IS_CV(1, "b") IS_CONST("42") result(IS_VAR(3, "1"))
4: $c = $_GET['c'];
   4: ZEND_FETCH_R( IS_CONST("_GET") IS_UNUSED(0, "0") result(IS_VAR(4, ""))
   5: ZEND_FETCH_DIM_R( IS_VAR(4, "") IS_CONST("c") result(IS_VAR(5, ""))
   6: ZEND_ASSIGN( IS_CV(2, "c") IS_VAR(5, "") result(IS_VAR(6, "1"))
5: echo $a;
   7: ZEND_ECHO( IS_CV(0, "a"))
6: echo $b;
   8: ZEND_ECHO( IS_CV(1, "b"))
7: echo $c;
   9: ZEND_ECHO( IS_CV(2, "c"))
8: echo $d;
  10: ZEND_ECHO( IS_CV(3, "d"))
9: ?>
10:
  11: ZEND_RETURN( IS_CONST("1"))
  12: ZEND_HANDLE_EXCEPTION()
```

What I Actually Want

```
1: <?php
2: $a = $_GET['a'];
3: $b = 42;
4: $c = $_GET['c'];
5: echo $a;
6: echo $b;
7: echo $c;
8: echo $d;
9: ?>
10:
```

Presenting SNAP



- SNAP is Not A Parser
- A slightly modified version of ParseKit
- Some PHP to present the output readably
- Some PHP to reimplement as little logic as possible
- A store of manually classified built in functions
- Because it is reverse engineered, there will be errors or approximations so aim for false positives

Distractions



- I am mostly interested in making code security audits more efficient, but readable opcodes could tell you other things
- They might help with debugging
- They might help with fine grained optimization
- They might help you settle arguments at the pub

String Quoting

```
<?php
    echo "Hello World";
    echo 'Hello World';
    echo <<<EOT
Hello World
EOT;
?>
```

More String Quoting

```
<?php
    $world = "World";

    echo "Hello $world";
    echo 'Hello ' . $world;

?>
```

Preincrement vs Postincrement

```
<?php
    for ($i=0; $i<10000; $i++)
        echo $i;

    for ($i=0; $i<10000; ++$i)
        echo $i;

?>
```

Back to Audits



- Need to be able to follow taint being passed by opcodes from variable to variable
- Need to handle file includes
- Need to handle arrays
- Need to handle built in functions
- Need to handle user functions
- Need to handle classes and inheritance
- @^%# Scope
- Aliases, and Pass By Reference

Opcodes Passing Taint



- If op1 or op2 is tainted, then so is the result
 - ZEND_CONCAT
- If op2 is tainted then taint op1 AND if op2 is not tainted, then untaint op1
 - ZEND_ASSIGN

Safe Opcodes



- ZEND_INIT_STRING
- ZEND_UNSET_VAR
- ZEND_SUB
- ZEND_MUL
- ZEND_DIV
- ZEND_MOD
- ZEND_BOOL
- ZEND_BW_OR
- ZEND_BW_AND
- ZEND_BOOL_NOT
- ZEND_BOOL_XOR
- ZEND_BW_XOR
- ZEND_BW_NOT

Some Opcodes Can Be Ignored

- 'ZEND_FETCH_DIM_UNSET this is used to unset PARTS of multi dimensional arrays, so CANNOT be relied on to make the whole array clean
- ZEND_END_SILENCE end @ error suppression operator
- ZEND_NOP
- and some I ignore because they are part of a multi opcode process that I handle

Compromise: Arrays

```
<?php
    $a[] = $_GET['foo'];
    $a[] = 42;
    var_dump($a);
    $a[0] = 'bar';
    var_dump($a);
    unset($a);
    var_dump($a);
?>
```

Flow of Control Opcodes



- Need following to retest code with new taint
 - ZEND_JMP
 - ZEND_JMPZ
 - ZEND_JMPZNZ
 - ZEND_JMPNZ
 - ZEND_JMPNZ_EX
 - ZEND_JMPZ_EX

Flow of Control Example

```
<?php
for($i = 0; $i<5; $i++)
{
    echo $var;    // note only safe once

    $var = $_GET["foo"];
}
?>
```

Output Opcodes - Special Case

- If op1 is tainted, then we have potentially tainted output
 - ZEND_PRINT
 - ZEND_ECHO
 - ZEND_EXIT
-
- For some dumb* reason print always returns 1, so the result from ZEND_PRINT is always safe

* Using a popular definition of dumb, "Things I don't understand"

Built in Functions



- Built in functions are important parts of programs, but we can't parse or interpret them
- What special classes of function are we interested in?
 - pass on taint from a tainted parameter?
 - create new taint (eg by reading in a file or db record)?
 - create output (eg echo, write file or db record)?
 - known "safe" functions (eg has no side effects and returns an integer)?
 - "special" danger functions, that deserve personal investigation

User Defined Functions

- Need rescanning if they are called with additional taint

```
<?php
function f($a, $b)
{
    echo $a;
    echo $b;
}
f(42, 'foo');
f($_GET['a'], '42');
f('foo', $_GET['b']);
f($_GET['a'], $_GET['b']);
?>
```

Classes and Inheritance

```
class ClassA
{
    function __construct($in)
    {
        echo "I'm $in ur constrctr
            constructin ur stuff";
    }
}
class ClassB extends ClassA
{
}
$a = new ClassA;
$b = new ClassB($_GET["bar"]);
?>
```

Scope in PHP is Evil Anyway

```
<?php
function f($a, $b)
{
    global $c;
    echo $a;
    echo $b;
    echo $c;
}
$a = $_GET['a'];
$b = 42;
$c = $_GET['c'];
f($b, $a);
?>
```

Pass By Reference

```
<?php
$a = 42;
echo $a;
f($a);
echo $a;
function f(&$a)
{
    $a = $_GET['a'];
}
?>
```

Questions?



I Can Haz Kweshtuns?

