Analyzing PHP
An introduction to PHP-Sat *

Eric Bouwers
embouwer@cs.uu.nl

Center for Software Technology
Universiteit Utrecht, The Netherlands

ABSTRACT
Mastering all subtleties of the PHP language is a process that can easily take years. Because of this there exist many projects that contain code that is either insecure or incorrect. Finding this kind of code manually is unfeasible because of the size of the existing code base. By defining patterns of common misconceptions a static analysis of PHP-source code allows the isolation of code-blocks which have a high probability of being incorrect or insecure. In this paper we present PHP-Sat, a tool that is capable of performing such an automatic analysis. We show examples of common misconceptions and explain an algorithm for finding security vulnerabilities.

1. INTRODUCTION
The number of programmers that use PHP[1] to develop web-applications has been growing rapidly the last year. Statistics from August 2006 show that PHP is running on almost 20 million domains and over 1.3 million ip-addresses [2]. The reason for this popularity might be because PHP has a very low learning curve. It comes with a massive function library that supports string-, file-, image- and database manipulation, together with native support for shell-access and text-output. Most of the constructs within PHP can be written in several alternative ways. This flaccidity, together with the loosely typed nature of PHP, creates an environment in which it is very easy to develop powerful applications without losing flexibility.

This flexible and pragmatic approach is also the Achilles' heel of PHP. The library is a useful resource, but lacks internal consistency in naming and parameters. Consider the functions `strpos($haystack, $needle)` and `in_array($needle, $haystack)`, which both look for a needle in a haystack. Furthermore, a misspelled variable name can introduce logical errors that are hard to find because the dynamic type system allows the use of undeclared variables. Interaction with databases or the shell and the generation of output can be done in several ways, which makes it hard to keep track of the origin of data. There is a large amount of code-examples available on the web, but the overall quality of this code is not very high. This gives the impression that PHP is being developed in a open community without serious support for checking the correctness of an application.

But correctness might not be the biggest problem for web-applications that are written in PHP. A recent report[3] from Symantec states that they documented 1,896 new vulnerabilities in six months. Vulnerabilities in web applications made up 69 percent of this number. The same report mentions the first web-worms[4, 5, 6], two of these worms attack applications that are implemented in PHP. While correctness checks are needed for programmers implementing applications, security checks are needed to protect the users of the applications.

The need for tools that perform correctness- or security-checks is not specific to PHP and many researchers have addressed this problem. Static analysis is the most widely used solution to find suspicious constructs in source code. There are numerous tools for C, C++, Java and C# that can statically find such constructs[7], compilers are also part of this set.

1.1 Correctness checkers
The native PHP executable offers a syntactical check on source files. This behaviour is documented as (lint) although it can hardly be considered a lint check. Zend studios[1] code analyzer offers functionality that resembles lint-like behaviour more accurately. Checks covered in the analyzer include usage of variables before they are declared and dead code identification. The problem with this analyzer is that it is intra-procedural and does not analyze control-flow, which limits its use and generates (many) false positives.

OcPortal's *Code quality checker*[2], allows the detection of some bug-patterns such as wrong parameter count and calling an undefined function. The downside of this checker is that it comes with a list of 23 language features that it does not support. This list includes alternative syntax, static function calls and embedded variables in double-quote strings. These limitations have a major impact on the usefulness of this checker.

PMD's *Copy/Paste Detector* (CPD) [3] and the Rough Auditing Tool for Security (RATS)[4] are more generic quality checkers. These checkers support multiple languages in a

*This paper is written in the context of the Software Technology Colloquium at the Universiteit Utrecht. The PHP-Sat project was started during the Google Summer of Code™.

[1] `http://www.zend.com/products/zend_studio/`
[2] `http://ocportal.com/site/pg/downloads/entry/7/index.php&root=1`
[3] `http://pmd.sourceforge.net/cpd.html`
[4] `http://www.securesoftware.com/resources/download_rats.html`

high-level nature. The generic nature of the tools is reflected by the quality of the parsers which are used. Both parsers accept syntactical incorrect PHP-sources as input without generating warnings or errors. The output of the CPD is a pair of similar pieces of code, something that is to be expected from such a tool. The output of RATS is based on a configuration file that can match certain function calls. This results in messages similar to:

> Arguments 1, 2, 4 and 5 of this function may be passed to an external program (usually sendmail). If these values are derived from user input, make sure they are properly formatted and contain no unexpected characters or extra data.

The problem with these kind of tools is that they are unable to perform any kind of semantic analysis. This results in many false positives and generic output which does not really help a programmer to identify misconceptions.

1.2 Security checkers

There exists very few approaches that deal with the static security checking of PHP applications. Huang et al.[8] were the first ones who targeted this issue. In their first approach they use a lattice of security-levels to track the security state of variables through a program. Their second approach[9] uses techniques from bounded-model checking to find security breaches and automatically patch the source code. The results of the experiments performed by Huang et al. are very promising, but there are several (technical) issues that are not addressed. The reports do not mention how references or the accessing of arrays is handled. Files are not included automatically and a substantial number of files can not be parsed. Details about these issues could not be attained from the authors because of the fact that the tool has been commercialized into WebSSARI.

Pixy[10] is a Java-based tool which uses a combination of three types of analysis to produce promising results. The basis of this all is a context-sensitive, inter-procedural *data-flow analysis*. This analysis is combined with *alias analysis* to deal with references and accessing arrays. The last analysis tracks whether a variable is *tainted* or *safe* and is called *taint analysis*. Pixy will not analyze PHP directly but operated on the control flow graph of a program. Where Huang et al. included files manually, Pixy deals with file inclusion automatically. However, there are still issues that have a negative impact on the practical usage of the tool. One of the issues is that the output of Pixy uses the internal representation of the source code. This leads to messages referring to variable which do not exists in the original source code. Another issue is that Pixy uses a safe/unsafe state for a specific type of vulnerabilities. So detecting a different kind of vulnerability requires a partial reimplementation of the tool.

Both WebSSARI and Pixy are unable to deal with the object oriented features of PHP. This is unfortunate because new PHP code is mostly developed with this paradigm in mind. Another thing that is lacking in both tools is the ability to adapt the tool to the needs of specific projects.

1.3 Contributions

This paper will present *PHP-Sat*, a tool that closes the gap between the research that has been done and the practical needs of an average PHP programmer. We will give examples of bug patterns, explain why these indicate logical errors, and point out a number of bugs which are already found in a well tested library. Furthermore, we will explain an algorithm that is used to find vulnerable places in PHP applications. This algorithm can be adapted to specific projects in order to deal with the dynamic nature of PHP.

1.4 Organization

The rest of the paper is organized as follows. Section 2 gives some more details about PHP and the environment in which it runs. Section 3 will show examples of common misconceptions and how to find them. Section 4 will discuss the security algorithm together with several considerations. The implementation of the tool will be discussed briefly in Section 5 after which we conclude in Section 6.

2. PHP ENVIRONMENT

This section will introduce PHP and the environment it runs in by example. Readers that are already familiar with PHP can skip this section without missing crucial information.

PHP is a general-purpose scripting language. It is capable of performing a wide range of tasks ranging from basic arithmetic to image-, database- and file-manipulation. Most of this functionality is offered through the native library which contains over 4000 functions. PHP is a scripting language which means that it is interpreted rather then compiled. There exists some solutions that are able to compile a PHP-script to an executable, but the use of these solutions is rare. The most popular application of PHP is as a module in a web-server to handle the server-side logic of web-applications through scripts.

An example of such a script is given in Figure 1 which implements a simple guestbook. The output of this script after posting three messages is shown in figure 2. The form that is shown is generated by the first five lines of the script. Notice that this is just plain HTML which is passed directly to the output. PHP code can be embedded into HTML by special open- and close tags as shown in, for example, line two of the script. The code between these tags is evaluated by the PHP interpreter. In this case the code consist of the built-in construct `echo` which prints its argument(s) to the output. The current argument is the *global variable* `$PHP_SELF` which contains the path to the current script. There exists several of these pre-filled variables that are available anywhere in a script.

One such global variable is the `$_GET`-array on line seven. Arrays in PHP are ordered maps that use either a string or an integer as the key. The values of the `$_GET`-array consist of the parameters passed to the script by the request of the users. Other examples of such arrays include `$_POST`, `$_COOKIE` and `$_FILES` which all represent a different part of the request. The mother of all global variables is the `$GLOBALS`-array (notice the missing underscore). It holds all the global variables including itself.

```
01:My first guestbook: <br />
02:  <form action="<?php echo $PHP_SELF; ?>">
03:    <input type="text" name="message">
04:    <input type="submit" value="go">
05:  </form>
06:   <?php
07:     if($_GET['message']){
08:       $fp  = fopen('./messages.txt', 'a');
09:       $msg = htmlentities($_GET['message']);
10:        fwrite($fp, "$msg");
11:        fclose($fp);
12:      }
13:     readfile('./messages.txt');
14:   ?>
```

Figure 1: MyFirstGuestBook.php

Figure 2: Guestbook output

Arguments passed to a script are always strings, but the code on line seven uses the variable `$_GET['message']` as a boolean. This can be done because PHP uses a *dynamic type-system*. It will cast the value of a variable to the type that is needed in a given situation. The code on line seven relies on this behavior to check for a non-empty message because an empty or non-existing string will result in a boolean value of `false`.

When the script receives a message it will store it in a file called `messages.txt`. Line eight opens this file, line ten writes the message to the file with the function `fwrite` and line eleven will close the file again. The second argument given to the function `fwrite` consists of a double-quoted string with a variable. PHP will expand this variable to the value before evaluating the function.

Before the message is written to the file it is sanitised by the function `htmlentities`. This function transforms special characters to their HTML-encodings. The function `readline` on line thirteen eventually reads the file and passes it directly to the output. These exotic, HTML-specific functions show that PHP specifically supports web-development.

The discussed example shows a small program that is easily captured in a single file. But other projects, for example bulletin boards or administration systems, can grow to several thousand lines of code. Such projects need functions or objects to group functionality in order to keep an understandable code-base. PHP allows the definition of functions since the earliest versions and the definition of simple objects since version 4. The support for objects is extended in the current version 5 by access-modifiers and the definition of interfaces.

Another way of coping with large code-bases is the definition of certain parts of the script in different files. These files can contain function- or class-definition as well as normal code. When a script needs these parts it can `include` or `require` the files at runtime. The code of the included file replaces the `include`-statement and inherits the current scope. This is a very powerful mechanism which is used frequently in most projects.

```
<?php
  if($_GET['key']){
     //show details of item
  } else {
    //show list of items
  }
?>
```

Figure 3: C000.php

This section only covered the basic features of PHP, many subtleties have been ignored. The important thing is to keep in mind that the combination of dynamic-typing, runtime code-inclusion and the existence of pre-filled variables makes the PHP environment very dynamic and non-trivial to analyze.

3. BUG PATTERNS

This section shows several examples of *bug patterns* that are likely to be the cause of problems. These bug patterns do not indicate syntactical problems, but problems at the semantic level that can be the cause of time-consuming debug-sessions. Each bug pattern targets a probable misconception of a programmer. Finding and flagging these misconceptions will not only reduce development-time, but will also help a programmer to better understand the semantics of PHP. A more detailed explanation of the usefulness of bug patterns in general is given by Hovemeyer and Pugh [11].

3.1 If-variable (C000)

Many applications will store items in a database indexed by a unique key. This key is generated when the item is inserted and is used to retrieve the item at a later time. The details of an item are shown when the user visits an URL that for example looks like: `www.example.com/C000.php?key=1`. If there is no GET-parameter the list of all items will be displayed. If there is a GET-parameter, it will be available in the `$_GET`-superglobal, in this specific case in the entry `$_GET['key']`. The code in `C000.php` might look like the code in Figure 3.

```
<?php
  function foo($bar){
    echo $bar;
  }
?>
```

Figure 4: c002_function.php

```
<?php
  include 'c002_function.php';

  foo("param1", "param2");
?>
```

Figure 5: c002.php

The condition here should check whether or not `$_GET['key']` has a value, i.e. is not `NULL`. The programmer probably expects that, due to the implicit type-conversion within PHP, a `NULL`-value is type-juggled to `False`, which is correct, and that any String-value is type-juggled to `True`, which is a misconception. Almost all String-values that are type-juggled to Boolean result in a `True`, except for the empty string and the string `"0"`. This behaviour will make the condition fail when the key of an item is `0`, making it impossible to show the details of this item. The solution here is to use the function `isset` to determine whether a variable is set. The usage of this function will make the intention of the programmer explicit, even for people with only modest programming skills.

Notice that this bug pattern should only be reported for variables that do not contain Boolean value. The current implementation of this bug pattern only matches conditional statements that have a super-global as condition.

3.2 Too-many-parameters (C002)

Most programming languages allow the definition of a function and the main entry point of a program to be in a separate file. An example of such a function definition is shown in Figure 4. This code defines a function `foo` with a single parameter. When a file with a function definition is included in another script the function becomes available in the global scope. Figure 5 shows an example of this scenario in which the file with the function-definition of `foo` is included and the function is immediately called. Notice that the function-call passes two parameters to the function, a clear example of a misconception of the programmer. When this situation occurs one would expect the interpreter to either stop the execution, or at least issue a warning.

However, PHP will parse and interpret the file without complaints and will deliver the string `param1` as output. This shows that PHP will just ignore the extra parameters given to a function when the function definition defines fewer parameters. So the output is to be expected considering the function definition, but the function call clearly shows a misconception of the programmer.

```
function sendMailsInQueue( ... ){
  initialization
 while ( more mail ){
    $result = sendmail( ... );
   if (!PEAR::isError($result)){
      set result as send
    } else{
     PEAR::raiseError( ... );
    }
  }
 return true;
}
```

Figure 6: Queue.php

The algorithm of finding this misconception is captured in the following straight-forward pseudo-code:

```
at every function call:
 1: Count the number of parameters given
 2: Retrieve the function definition
    from the environment
 3: Count the number of parameters
 4: Flag function call when the count in
    1 is larger then the count in 3
```

The relationship between this algorithm and the implementation will be shown in section 5.

3.3 Ignored return value (C006)

When a return value is ignored the programmer makes the assumption that this value is not interesting. This is a valid assumption when return values are static, but when return values are dynamic the programmer ignores valid execution paths. This can lead to the unwanted behavior as the example extracted from PEAR:Mail_Queue[5] shown in Figure 6. This example was found after running PHP-Sat on the PEAR code-repository and examining the flagged code.

The function which is declared in Figure 6 is used to send all the mail-messages that are in the current queue one-by-one. The comment of this function states that the return value will be `true` on success and an error-object when an error-occurs. A normal programmer that uses this function will check for these values and handle them in the right manner.

However, the return value of this function will always be `true`. There is no statement within the `while`-loop that breaks the normal execution-path. After a message is send the code checks for errors and generates an error-object by calling `PEAR::raiseError( ... )`, but the resulting error-object is ignored. Programmers who rely on the documented behavior will never find out that a message was not send correctly.

The example shows that ignoring return values can cause high-level errors that are very hard to spot directly. This pattern is not PHP-specific but can also be applied to most

[5]PEAR is an initiative that aims to provide a set well-tested and high-quality modules. See `http://pear.php.net`

```
class Net_Geo {
   class variables
  function Net_Geo( ... ){
      initialization
      return true;
   }
   more functionality
}
```

Figure 7: Geo.php

other languages. There is one issue with this bug pattern that holds for all languages, namely the risk for a high false positive rate. With a naive implementation the pattern will flag all ignored return values, including the once that are constant. Since constant values do not introduce alternative execution paths it is safe to ignore them. So when this situation is not taken into account the resulting output can contain many false positives, making it harder to find the real problematic situations. This actually happened in our experiments with the PEAR-library. One of the base classes of this library contains a function which will always return the value `true`. Most of the calls to this function ignored this value and where flagged by PHP-Sat. The results from PHP-Sat needed to be filtered again, a time-consuming and annoying task.

3.4 Return from constructor (C007)

Section 2 already mentioned that the support for Object Oriented (OO) features is relatively new in PHP. Understanding all of the concepts behind OO-programming takes time and practice. This bug-pattern is an example of a common misconception of (novice) programmers related to OO-programming.

The basic concept in OO-programming is the object itself. An object may have multiple or no *methods* and *fields*. The methods within an object define the behavior and the fields define the internal state of an object. When an object is instantiated by the `new`-keyword a *constructor*-method is called. This method allows the initialization of the fields prior to using the object. After calling the constructor the object will be the return value of the `new`-expression. An example of this is `$bar = new Foo();` in which an object `Foo` is created and assigned to the variable `$bar`.

Objects are defined in `class`-declaration which have a list of field- and method-statements. The constructor method is the method named `__construct` in PHP5 and the method with the same name as the class in PHP4. An example of the latter is given in Figure 7 which defines a class `Net_Geo`, showing the bug-pattern in its full glory. The comment of the function declares that the function will return `true`, which is a clear misconception. A novice programmer might read the comments and assume that the comment is right, resulting in a spread of the misconception.

An analysis of the PEAR-repository has resulted in 13 separate cases with `return`-statements in constructors. One of them was already reported in 2004 but quickly fixed after our report.

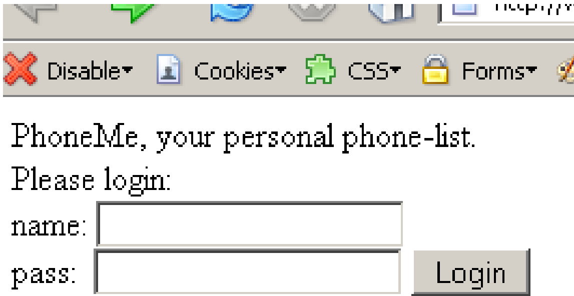

Figure 8: PhoneMe Login

```
1:<?php
2:  .. Initialization ..
3:  $query =
4:     "SELECT * FROM users
5:      WHERE name='{$_POST['name']}'
6:        AND pass='{$_POST['pass']}'";
7:  .. Use $query as query ...
8:?>
```

Figure 9: PhoneMe source code

3.5 Conclusion

The bug-patterns explained above are not difficult to understand or fix, but can be the cause of high-level errors. Indicating which parts of the code are based on possible misconceptions can help a reviewer to focus on specific parts of a large code base. The results which are mentioned in the text already show that simple misconceptions even occur in code that is well-tested and reviewed and used by many.

4. SECURITY PATTERN

Section 3 describes bug-patterns that either limit the possibilities in the program, by ignoring valid execution paths, or do not change the observable behaviour from a users perspective. In both cases the user will not be able to perform any action that is not permitted by the programmer. In contrast, this section will introduce a security pattern that find places in a program where a user might be able to perform unauthorized actions. We will describe the source of such security vulnerabilities, an algorithm to (partially) prevent them and several considerations that need to be taken into account for the implementation of the pattern.

```
1:<?php
2:  .. Initialization ..
3:  $query =
4:     "SELECT * FROM users
5:      WHERE name='".addslashes($_POST['name'])."'
6:        AND pass='".addslashes($_POST['pass'])."'";
7:  .. Use $query as query ...
8:?>
```

Figure 10: PhoneMe source code secured

4.1 Vulnerability types

To identify the source of many security vulnerabilities we will explain two different types of vulnerabilities by example, *SQL injections* and *Cross Site Scripting*

SQL injections allow an attacker to gain control over a query send to a database. An example of such an attack is described by a scenario in which *Alice* wants to gain access to *Bob* his phone-numbers. Bob is subscribed to an online telephone book in which he can store all of his phone-numbers. In order to see his list Bob has to fill in his username and password in a form similar to the one in figure 8.

The PHP script which processes this form is shown in figure 9. The values from the form are used in lines five and six to construct the SQL-query. So after filling in `bob` as user and `pass` as password the script constructs the following query-string:

```
SELECT * FROM users
  WHERE name='bob' AND pass = 'pass'.
```

When there is a result from the database that matches both these conditions the script will allow access to the phone-numbers.

Alice finds out that the query is constructed in this way and a hack is initiated. The documentation of SQL mentions that anything behind a double-dash is considered to be comment. When Alice supplies `bob'--` as a user and `useless` as a password, the query that is constructed by these values looks like:

```
SELECT * FROM users
  WHERE name='bob'--' AND pass = 'pass'.
```

Since the double-dashes render the `pass` condition useless the database returns every user with the name `bob`. In this case Alice is given access to all of the phone-numbers of Bob, but the same flaw can also be used to remove, for example, all the data in the database.

Protecting against this kind of vulnerability is a matter of escaping the input from the user. Figure 10 shows the secured source code for the PhoneMe-example by using the library function *addslashes*. The library of PHP also contains escape-functions for specific databases which escape all the special characters for a particular database implementation.

Cross Site Scripting (XSS) injections allow an attacker to gain control over the output served to a regular user. An example of such an injection is given by the following scenario in which *Bob* wants to obtain the creditcard information of *Alice*. For this scenario Alice will use a website called *TrustMe* to shop on-line A probable, yet very basic, interface of the TrustMe web-shop is shown in Figure 11. The user has to type in a description of the product after which an order form for this product will be generated. The (partial) source of the TrustMe-website is shown in Figure 12. When Bob examines the source code he notices the assignment on line two. The value of the variable `$words` is instantiated

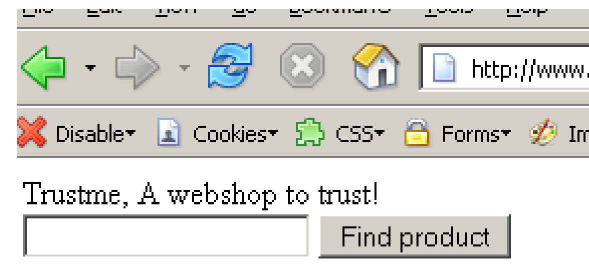

Figure 11: TrustMe main screen

```
01: <?php
02:   $words = $_GET['product'];
03:     ... Obtain $result ...
04:   echo 'You searched for: ';
05:   echo $words , '<br />';
06:   if($results){
07:       ... Output results ...
08:   } else {
09:     echo 'Nothing found :(';
10:   }
11:?>
```

Figure 12: TrustMe source

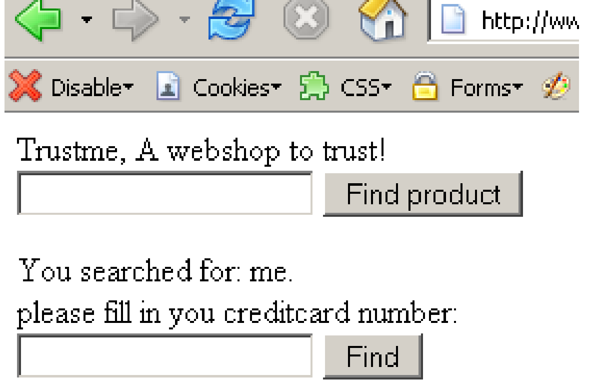

Figure 13: TrustMe hacked screen

```
01: <?php
02:   $words = $_GET['product'];
03:     ... Obtain ($)result ...
04:   echo 'You searched for: ';
05:   echo htmlentities($words) , '<br />';
06:   if($results){
07:       ... Output results ...
08:   } else {
09:     echo 'Nothing found :(';
10:   }
11:?>
```

Figure 14: TrustMe source secured

with direct user-input and displayed on line five. This allows Bob to control the output of the page by passing the right parameter to the URL. Figure 13 shows an example of output that could be generated by passing HTML-code to the TrustMe-application. The information that is supplied in this form is not send to the TrustMe-application, but to a server which is controlled by Bob. The URL needed for this task can look like

```
http://www.example.com/?search= ↩
  <script src= "http://evil.org/trustme.js"> ↩
  </script>
```

After creating the right URL Bob needs to trick Alice into clicking the crafted link. This can for example be done by creating an e-mail that seems to originate from the TrustMe-company asking Alice to fill in some missing information. If Alice follows the URL the content still originates from `http://www.example.com` and is assumed to be safe. Bob misuses the trust of Alice and obtains all the information he wants.

Protecting against XSS-injections is also a matter of escaping the input from the user. The secured code for the TrustMe-shop is listed in figure 14 and differs only on line five. Instead of directly displaying the variable all the HTML-characters are escaped. This will make sure that the attack of Bob cannot occur.

4.2 Security algorithm

The examples from the previous show that the source of many SQL- and XSS-injections, but also command-line injections, can be found in user-input that is not escaped properly. Unfortunately, due to the overwhelming number of ways in which input can occur within PHP, it is not trivial to recognize all user-input. Even when a value is recognized as user-supplied it can be very hard to track every usage of the value within a large program. Another difficulty is that different injections need different escaping mechanisms. A variable that is safe to use in a SQL-query might not be safe to use as HTML-output. The examples from Section 4.1 show that it takes only a single function call to prevent injections, but finding the places where the function calls should take place manually is impossible in large applications. Instead, we should aim at an automated approach.

A naive approach would simply look for all places where data leaves the program, the so-called *sensitive sinks*, and escapes all the parameters passed to these sinks. This approach is not feasible because of the different escaping mechanism needed for different type of injections. There is simply no generic way to escape data for every type of injection at once. Furthermore, escaping data automatically can result in the escaping of already escaped data. This results in strange or even invalid data.

Fortunately, a more adaptive approach exists. This approach is based on typestate[12] and the work of Huang et al.[8]. The basis for this algorithm is formed by the set of *safety-types*, Γ, which follow the properties of the model proposed by Denning[13]. Figure 15 shows the safety-types used within PHP-Sat. These safety-types form a complete lattice with *safe* as lower bound and *unsafe* as upper bound. This means that $\forall \gamma \in \Gamma, \gamma \cup \bot = \bot$ and $\forall \gamma \in \Gamma, \gamma \cap \top = \top$. The set of safety types is ordered with the normal $\supset$ from top to bottom. More concretely this means that anything that needs the `Raw Input`-safety-type will accept something of safety-type `float`, but not vice versa.

Safety types	
Unsafe	$\top$
Raw Input	
Escaped HTML , Escaped Shell , Escaped Slashes	
Formatted String, Encoded String	
Known String , Matched String	
Integer , Null , Object , Array , Float	
Safe	$\bot$

Figure 15: Γ, set of safety types

```
01:  $foo = $_GET['foo'];
02:  if ($condition){
03:    $bar = htmlentities($foo);
04:  } else
05:    $bar = $foo;
06:
07:  $bar;
```

Figure 16: Control flow example

Every variable within a PHP script is assigned one of the listed safety-types on its first encounter. The safety-type with which a variable starts depends on the context in which it is used. When a variable is declared with a compile-time constant the safety-type of that value is used. For example, `$foo` will get the safety-type `Integer` in the expression `$foo = 15`. Variables can also have a pre-defined safety-type. For example, entries from the `$_GET`-array will generally get the safety-type `Raw Input`. But there exist cases in which the safety-type of a variable is simply unknown. This can happen when the safety-type of a source is not defined or when variables are used in the right hand side of an expression without being declared earlier. To prevent a high false-positive rate the default safety-type of these variables is defined as `Safe`. This choice implies that the algorithm will not find every vulnerability in a program because unknown values might be unsafe.

The safety-types which are assigned to variables propagate through the control-flow of a script. In the easiest case the control-flow is linear, the safety-type will only change with new assignments. More difficult cases occur at control-flow branches, for example at an `if`-statement. Consider the code in figure 16, the variable `$bar` will get a safety-type of `Escaped HTML` within the if-branch. Within the else-branch

```
01:  $foo = $_GET['foo'];
02:  echo htmlentities($foo);
03:
04:  // parameter does not meet pre-condition
05:  echo  $foo;
```

Figure 17: Sensitive sinks

magic_quotes	register_globals	output
Off	Off	What's the config?
Off	On	What's the config?What's the config?
On	Off	What\'s the config?
On	On	What\'s the config?What\'s the config?

Figure 18: Output under different configurations

```
1:<?php
2:  echo $_GET['msg'];
3:  echo $msg;
4:?>
```

Figure 19: Code for INI-example

```
1:<?php
2:  $foo =  'foo';
3:  $bar =& $foo;
4:  // $foo and $bar reference the same value
5:?>
```

Figure 20: References in PHP

the same variable `$bar` will get a safety-type of `Raw Input`. The question arises which safety-type should be assigned to the variable `$bar` on line seven. The answer is that the safety-type should be the least upper bound of the both types, or in other words, the biggest set. In this case `$bar` would get the safety-type `Raw Input`. Notice that this choice implies that the algorithm can report false positives because all branches are taken into account, even the ones that never can be reached.

In the same manner that every variable is equipped with a safety-type, each sensitive sink in the program must receive a safety-type for its arguments. These safety-types function as a pre-condition for variables passed as arguments to the sensitive sink. This term implies that the safety-type of a variable should be a subset of the pre-condition of a sensitive sink. When this property is not met the sensitive sink will be flagged. An example of both a normal and a flagged sensitive sink is given in figure 17. In this example the pre-condition of the `echo`-statement would be `Escaped HTML`.

4.3 Considerations

Although the algorithm above might miss certain vulnerabilities, or report false positives, it will still find vulnerabilities which where previously undetected. But the quality of the algorithm greatly depends on the configuration of the pre-conditions and the safety-types of user-input. When pre-conditions are too restrictive the algorithm will generate many false positives, but when they are too loose not a single vulnerability will be found.

The optimal configuration of the algorithm can even differ across projects. A project that makes a one-user blog might allow posts to incorporate HTML-tags, but a multi-user wiki will usually disallow such a thing. Another influence on the quality of the algorithm is the configuration of the PHP executable itself. There is a large number of

```
[tainted sources]
 array: _GET     level: raw_input
 array: _REQUEST level: raw_input
[function result]
 function: htmlentities level: escaped-html
[sensitive sinks]
 construct: echo  (escaped-html && escaped-slashes)
 function:  fopen (known-string , known-string)
```

Figure 21: Example configuration of PHP-Sat

configuration settings that control the behavior of PHP on system, user or even script-level. Two of the settings that have a major impact on the safety-type a variable starts with are `magic_quotes` and `register_globals`. Figure 18 shows the output of the code from figure 19 when it is evaluated under different configurations. The URL that is used is:

`www.example.com/echo.php?msg=what's%20the%20config?`

To allow programmers to adapt PHP-Sat to their specific needs the algorithm is configurable through a configuration file. An example of such a file is shown in Figure 21. This file can be used to specify influence the algorithm in three ways. First, the safety-type with which certain global variables start can be defined. Second, the safety-type which is returned by a function, both built-in and user-defined, can be listed. Third, the pre-condition for any sensitive sink can be defined on a per-argument basis. The configuration also allows the combination of safety-types with conjunction (`&&`) and disjunction (`||`).

An optimal configuration can only be useful when the algorithm follows the semantics of PHP as closely as possible. Since PHP allows the inclusion of files during run-time this will also need to be supported by PHP-Sat. This task is almost trivial when programmers use the complete path- and file-name to include files. But the reality is that larger applications will use expression such as

`include $include_path.'name.php'`

to include files. I.e. these applications use a static configuration variable to include files relative to the location of other source-files. The main reason for this is that the installation of a project can easily be moved without the need to adapt the code. In order to make these includes constant PHP-Sat will perform *constant-propagation* before the security analysis takes place. When PHP-Sat is unable to statically detect the value of an argument which is passed to

the *include* statement a warning will be issued. In this case the argument can probably be adapted by a user which imposes a security-threat. Performing constant-propagation prior to the analysis also improves the algorithm because there is more knowledge available about static values in the program.

The concept of constant-propagation is also part of the algorithm of Piyx[10]. The research on this prototype also mentions the need for alias analysis. Reason for this need is that an assignment to a variable can influence the value of other variables as well. Figure 20 shows an example of two variables `$foo` and `$bar` which are referencing the same value. When an assignment is made to `$foo` the safety-type of `$bar` should be updated and vice-versa. The concepts and techniques for performing a precise alias analysis are well described by Jovanovic et al. in [14]. These techniques are also part of the implementation of PHP-Sat.

5. IMPLEMENTATION

PHP-Sat is implemented in Stratego/XT, a modular language for the specification of programming transformations. We will explain some of the basic principles of this language and show an intuitive example of the implementation of a bug pattern. A full explanation of the constructs of Stratego/XT can be found in [15, 16, 17].

5.1 Stratego/XT

A program written in Stratego/XT will transform *terms* using *strategies*. These terms are described in the Annotated Term Format (ATerm), which is comparable to XML. For example, the ATerm representation of the expression $1+2$ is `Plus(LNumber(1),LNumber(2))`. It is possible to annotate terms with information, for example the value of an expression, like this: `Plus(LNumber(1),LNumber(2)){Result(3)}`.

Transforming ATerms can be done by applying *rewrite rules* which are described by the format `name: p1 → p2`. Rules try to match the current term to the pattern `p1` and build the pattern `p2` with variables instantiated from `p1`. When the current term cannot be matched against `p1` the application of a rule fails. So if we define the rule `swop: Plus(a,b) → Plus(b,a)` and apply if to the term above the result would be `Plus(LNumber(2),LNumber(1))`.

Rules can be grouped together into *strategies* by various combinators. In the examples only the Sequential combinator will be used. This combinator will apply a strategy `s1` and `s2` after each other when defined like this (`s1 ; s2`). In order to deal with failing rules the strategy `Try(s)` will be used. Application of this strategy results in the new term when the given strategy `s` succeeds, or the old term when the strategy `s` fails. The `Try(s)`-strategy is one example of the large number of generic strategies available in the Stratego-library. This library also contains generic-traversal strategies which for example can be used to traverse an ATerm `bottomup(s)` or `topdown(s)`.

5.2 PHP-Front

In order to operate on the ATerm representation of a PHP-program the sources need to be parsed. The library that is responsible for this is PHP-Front. This library is able to parse PHP sources written for version 4 and 5 and can automatically include files after parsing. After an analysis is done the (possibly) transformed sources can be pretty-printed to normal source code. Figure 23 gives a visual representation of this process.

One of the unique features of PHP is the possibility to reflect over user-defined class- and function-definitions. On any given moment the ATerm representation of a function definition can be obtained when the name is known. PHP-Front also offers a range of generic traversal strategies which are tailored to PHP. An example of this is the strategy `php-topdown-with-inclusion(s)` which will perform a top-down traversal of the current source, descending into included files where available.

5.3 PHP-SAT

The implementation of PHP-Sat is divided into several modules which all target a different kind of patterns. Section 3 already gave examples from patterns of the module *correctness* (C), but others deal with *style*(S) or *optimization*(O). The algorithm described in section 4 is also captured in a separate module called *malicious code vulnerability* (MVC).

The MVC module is currently work in progress. Currently, the implementation is capable of reading a configuration file, assigning the safety-type to a variable which is first used and checking the pre-condition of sensitive sinks. Control-flow is not yet supported, but this is a matter of implementation. The techniques that are needed for the algorithm are well defined by Bravenboer at al. [18].

Other modules in PHP-Sat contain fully implemented bug patterns. All of the patterns are basically a traversal of the ATerm representation in a top-down fashion with specific rules. An example of such a rule is given in figure 22. This rule describes the essence of pattern C002 which was already explained in section 3.2 and is applied by calling the strategy `php-topdown-with-inclusion(ManyParameters)`.

The rule in figure 22 will match all function-calls according to the pattern on line three. When such an expression is encountered it is stored in the variable `t`. After this, lines four to six will retrieve the current environment, the called function-representation and the number of parameters for the called function. Line seven will apply the strategy `has-too-many-parameters` and when this succeeds a special annotation will be added to the current term on line eight. When the number of parameters is correct, so the strategy `has-too-many-parameters` has failed, the result will be the current term. The implementation of the strategy `has-too-many-parameters` is shown in line fourteen to sixteen and just checks the length of the parameter-list against the correct number of parameters.

6. CONCLUSION

There exists a huge amount of research that indicates that static analysis of source code can be used to find semantic misconceptions or security vulnerabilities. Furthermore, a vast amount of static analysis tools are available for widely used languages such as Java (FindBugs, JLint, Bandera) and C (Lint, BLAST, Purify). It is therefore stunning to see that, considering the usage-statistics and availability of

PHP, the number of solutions for PHP is limited. The available tools either reject important features of the language or are not suitable to be used in production environments.

PHP-Sat offers the first open-source combination of bug-patterns and security analysis for PHP. The currently implemented bug-patterns already found bugs in well-tested and widely used code. Although the implementation of the security analysis is still work in progress, we believe that PHP-Sat is already a useful addition to the workbench of the average PHP-programmer.

7. REFERENCES

[1] The PHP Group. Php: Hypertext preprocessor, 2006. http://www.php.net, [Online; accessed 02-October-2006].

[2] The PHP Group. Php: Php usage stats, 2006. http://www.php.net/usage.php, [Online; accessed 02-October-2006].

[3] S. Entwisle et all D. Turner. Trends for july 05 december 05. *Symantec Internet Security Threat Report*, Volume IX, March 2006.

[4] J. Mun. Perl.santy, security response, 2004. http://www.symantec.com/security_response/writeup.jsp?docid=2004-122109-4444-99, [Online; accessed 02-October-2006].

[5] N. Mook. Cross-site scripting worm hits myspace, 2005. http://www.betanews.com/article/CrossSite_Scripting_Worm_Hits_MySpace/1129232391, [Online; accessed 02-October-2006].

[6] UNIRAS. Malicious software report linux/elxbot, 2005. http://www.uniras.gov.uk/niscc/docs/br-20051206-01073.html?lang=en, [Online; accessed 02-October-2006].

[7] Wikipedia. List of tools for static code analysis — wikipedia, the free encyclopedia, 2006. http://en.wikipedia.org/w/index.php?title=List_of_tools_for_static_code_analysis&oldid=78306553, [Online; accessed 2-October-2006].

[8] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. 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 Press.

[9] C. Hang Y. Huang, F. Yu. Verifying web applications using bounded model checking. *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, 2004.

[10] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. *2006 IEEE Symposium on Security and Privacy*, May 2006.

[11] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, New York, NY, USA, 2004. ACM Press.

[12] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[13] D. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[14] N. Jovanovic, C. Kruegel, and E. 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 Press.

[15] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[16] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/xt 0.16: components for transformation systems. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 95–99, New York, NY, USA, 2006. ACM Press.

[17] E. Visser, Z. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 13–26, New York, NY, USA, 1998. ACM Press.

[18] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules, 2005.

```
01: rules:
02: ManyParameters :
03:  t@Expr(FunctionCall(FunctionName(name),params)) -> t'
04:    where  environment := <get-php-environment>
05:         ; function    := <get-function(|name)> environment
06:         ; paramcount  := <get-param-count> function
07:         ; if <has-too-many-parameters> (function,params)
08:           then t' := <add-correctness-annotation(|C002())> t
09:           else t' := t
10:           end
11:
12: strategies
13: has-too-many-parameters =
14:   ?(function,params)
15:  ; paramcount  := <get-param-count> function
16:  ; not( <geq> (paramcount, <length> params))
```

Figure 22: C002 Stratego code

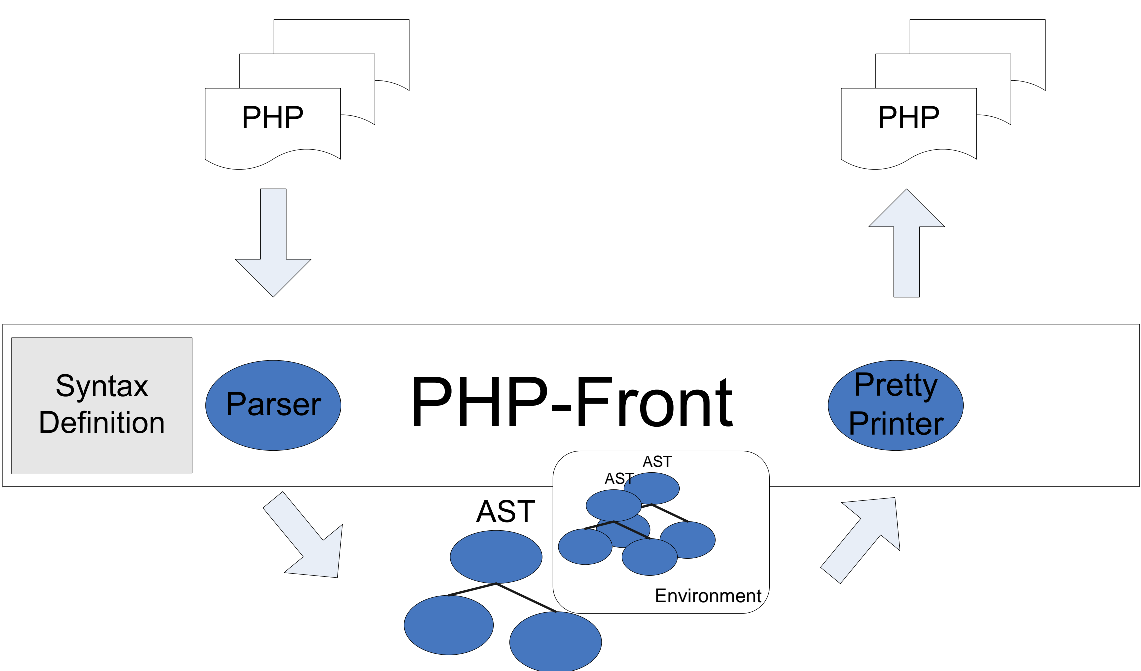

Figure 23: PHP-Front transformation framework