# Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Technical Report)

Nenad Jovanovic, Christopher Kruegel, Engin Kirda
Secure Systems Lab
Vienna University of Technology

**Abstract**

The number and the importance of Web applications have increased rapidly over the last years. At the same time, the quantity and impact of security vulnerabilities in such applications have grown as well. Since manual code reviews are time-consuming, error-prone and costly, the need for automated solutions has become evident.

In this paper, we address the problem of vulnerable Web applications by means of static source code analysis. More precisely, we use flow-sensitive, interprocedural and context-sensitive data flow analysis to discover vulnerable points in a program. In addition, alias and literal analysis are employed to improve the correctness and precision of the results. The presented concepts are targeted at the general class of taint-style vulnerabilities and can be applied to the detection of vulnerability types such as SQL injection, cross-site scripting, or command injection.

Pixy, the open source prototype implementation of our concepts, is targeted at detecting cross-site scripting vulnerabilities in PHP scripts. Using our tool, we discovered and reported 15 previously unknown vulnerabilities in three web applications, and reconstructed 36 known vulnerabilities in three other web applications. The observed false positive rate is at around 50% (i.e., one false positive for each vulnerability) and therefore, low enough to permit effective security audits.

## I. Introduction

Web applications have become one of the most important communication channels between various kinds of service providers and clients. Along with the increased importance of Web applications, the negative impact of security flaws in such applications has grown as well. Vulnerabilities that may lead to the compromise of sensitive information are being reported continuously, and the costs of the resulting damages are increasing. The main reasons for this phenomenon are time and financial constraints, limited programming skills, or lack of security awareness on part of the developers.

The existing approaches for mitigating threats to Web applications can be divided into client-side and server-side solutions. The only client-side tool known to the authors is Noxes [13], an application-level firewall offering protection in case of suspected *cross-site scripting* (XSS) attacks that attempt to steal a user's credentials. Server-side solutions have the advantage of being able to discover a larger range of vulnerabilities, and the benefit of a security flaw fixed by the service provider is instantly propagated to all its clients. These server-side techniques can be further classified into dynamic and static approaches. Dynamic tools (e.g., [9], [18], [22], and Perl's taint mode [26]) try to detect attacks while executing the audited program, whereas static analyzers ([10], [11], [15], [16]) scan the Web application's source code for vulnerabilities.

In this paper, we present Pixy, the first open source tool for statically detecting XSS vulnerabilities in PHP 4 [21] code by means of data flow analysis. We chose PHP as target language since it is widely used for designing Web applications [25], and a substantial number of security advisories refer to PHP programs [3]. Although our prototype is aimed at the detection of XSS flaws, it can be equally applied to other *taint-style* vulnerabilities such as SQL injection or command injection (see Section II). The main contributions of this paper are as follows:

- A flow-sensitive, interprocedural, and context-sensitive data flow analysis for PHP, targeted at detecting taint-style vulnerabilities. This analysis process had to overcome significant conceptual challenges due to the untyped nature of PHP.
- Additional literal analysis and alias analysis steps that lead to more comprehensive and precise results than those provided by previous approaches.

```
echo "Here is what you wrote: $_GET['content']";
```

Fig. 1. Very simple PHP script vulnerable to reflected XSS.

- Pixy, a system that implements our proposed analysis technique, written in Java and licensed under the GPL.
- Experimental validation of Pixy's ability to detect unknown vulnerabilities with a low false positive rate.

The paper is structured as follows. First, Section II presents the types of vulnerabilities detectable by Pixy. In Section III, we give a short introduction to the theory of data flow analysis. Section IV describes the front-end responsible for compiling the PHP input into a form suitable for the following data flow analysis, which is detailed in Sections V to IX. Section X summarizes our empirical results. Related work is discussed in Section XI. Finally, Section XII briefly concludes.

## II. Taint-Style Vulnerabilities

The presented work is targeted at the detection of taint-style vulnerabilities. *Tainted* data denotes data that originates from potentially malicious users and thus, can cause security problems at vulnerable points in the program (called *sensitive sinks*). Tainted data may enter the program at specific places, and can spread across the program via assignments and similar constructs. Using a set of suitable operations, tainted data can be *untainted* (*sanitized*), removing its harmful properties. Many important types of vulnerabilities (e.g., cross-site scripting or SQL injection) can be seen as instances of this general class of *taint-style vulnerabilities*. An overview of these vulnerabilities is given in [15].

### A. Cross-Site Scripting (XSS)

One of the main purposes of XSS attacks [4] is to steal the credentials (e.g., the cookie) of an authenticated user. Every web request that contains an authentication cookie is treated by the server as a request of the corresponding user as long as she does not explicitly log out. Thus, everyone who manages to steal the cookie is able to personate its owner for the current session. The browser automatically sends a cookie only to the Web site that created it, but with JavaScript, a cookie can be sent to arbitrary locations. Fortunately, the access rights of JavaScript programs are restricted by the *sandbox model*: JavaScript has access only to cookies that belong to the site from which the JavaScript originated.

XSS attacks circumvent the sandbox model by injecting malicious JavaScript into the output of vulnerable applications. For example, consider the simple PHP script in Figure 1, where a user's posting to a message board is displayed after submitting it. The posting's content is retrieved from a GET parameter, therefore, it can also be supplied in a specifically crafted URL such as the following, which results in the user's cookie being sent to "evilserver.com":

```
http://vulnerable.com/post.php?
content=<script>document.location='evilserver.com/steal.php?'+document.cookie</script>
```

All that the attacker has to do is to trick a user into clicking this link, for example, by sending it to the victim via email. Attacks using such mechanisms are called *reflected* cross-site scripting attacks. Even more dangerous are *stored* cross-site scripting attacks, where no direct user interaction is necessary. In the example above, the malicious JavaScript code might be stored directly on the server and displayed to all users on a message board, leading to the same result.

In general, an XSS vulnerability is present in a Web application if malicious content (e.g., JavaScript) received by the application is not properly stripped from the output sent back to a user. When speaking in terms of the sketched class of taint-style vulnerabilities, XSS can be roughly described by the following properties:

- Entry Points into the program: GET, POST and COOKIE arrays.
- Sanitization Routines: htmlentities(), htmlspecialchars(), and type casts that destroy potentially malicious characters or transform them into harmless ones (such as casts to integer).
- Sensitive Sinks: All routines that return data to the browser, such as echo(), print() and printf().

Note that the number of entry points grows when the PHP configuration flag "register_globals" is active. Moreover, the identification of entry points and sensitive sinks is influenced by the treatment of databases, depending on whether the programmer considers the database to be potentially holding tainted data or not.

### B. Other Vulnerabilities

Although our current prototype focuses on XSS vulnerabilities, other vulnerabilities such as SQL injection and command injection have been identified as belonging to the general class of taint-style vulnerabilities and differ only with respect to the concrete values of a few parameters. The presented concepts are targeted at their underlying general characteristics, and adjusting Pixy to the detection of other instances only requires some engineering effort.

## III. Data Flow Analysis

The goal of our analysis is to determine whether it is possible that tainted data reaches sensitive sinks without being properly sanitized. To this end, we have to identify the taint value[1] of variables that are used in these sinks. For this, we apply the technique of data flow analysis, which is a well-understood topic in computer science and has been used in compiler optimizations for decades ([1], [17], [19]). In a general sense, the purpose of data flow analysis is to statically compute certain information for every single program point (or for coarser units such as functions). For instance, the classical *constant analysis*[2] computes, for each program point, the literal values that variables may hold. Data flow analysis operates on the control flow graph (CFG) of a program. In our implementation, the nodes of this CFG represent atomic statements of the program, and not basic blocks.

To illustrate how data flow analysis is used to perform literal analysis, imagine a fictitious programming language that uses only one variable ($\mathrm{v}$) and two literals (the integers 3 and 4). Figure 2 shows the CFG for a simple example program, with "skip" nodes representing empty instructions. Assume further that the condition of the "if" branch cannot be resolved statically, for example, because it depends on the value of an environment variable. We use the symbol $\Omega$ for the *unknown literal*, which indicates that the exact value of the literal cannot be determined. This is the case on program entry, since in our programming language, a variable contains random garbage before being initialized. After performing literal analysis for this program, each CFG node is associated with information about which literal is mapped to variable $\mathrm{v}$ *before* executing that node. Note that the exact value for variable $\mathrm{v}$ after the "if" construct is also unknown because the analysis cannot determine which branch will be taken at runtime. Inside the branches, however, precise information is available.

An important concept used in the theory of data flow analysis is that of a *carrier lattice*, which is used to represent the type of information that is to be collected. Every information that could ever be associated with a CFG node by the analysis must be contained as an element of the used lattice. The lattice for our previous example is depicted in Figure 3. Note that an additional *bottom element* $\bot$ is required by the analysis algorithm for marking nodes as "not visited yet" at the beginning. Each line in Figure 3 indicates an ordering between the elements at its end points with regard to precision. For instance, the element ($\mathrm{v}$:3) is more precise (less conservative) than ($\mathrm{v}$:$\Omega$), written as ($\mathrm{v}$:3) $\sqsubset$ ($\mathrm{v}$:$\Omega$). In this sense, the bottom element is defined as being smaller than all other elements. The *least upper bound* $\sqcup$ of two elements is the smallest element that is greater than or equal to both of the elements. For example, ($\mathrm{v}$:3) $\sqcup$ ($\mathrm{v}$:4) = ($\mathrm{v}$:$\Omega$), and ($\mathrm{v}$:3) $\sqcup$ ($\bot$) = ($\mathrm{v}$:3). The $\sqcup$ operator is used for conservatively combining the information of merging paths (e.g., after "if" branches), which can also be seen at the end node in Figure 2.

Another important ingredient for data flow analyses are *transfer functions*. Each CFG node is associated with such a transfer function, which takes a lattice element as input and returns a lattice element as output. Their purpose is to model the semantics of their corresponding node with respect to the collected information. In our example in Figure 2, all assignment nodes possess a transfer function that adjusts the literal mapping of the assigned variable accordingly.

After specifying a data flow analysis with its carrier lattice and transfer functions, an iterative algorithm for computing the results is initiated. Beginning at the program's entry node, this algorithm propagates analysis information through the program by applying transfer functions and combining information at merge points. As

[1]The taint value of a variable determines whether the variable is tainted or not.

[2]Note that we use the name "literal analysis" instead of the classical term "constant analysis" in order to prevent confusion with PHP's constants.

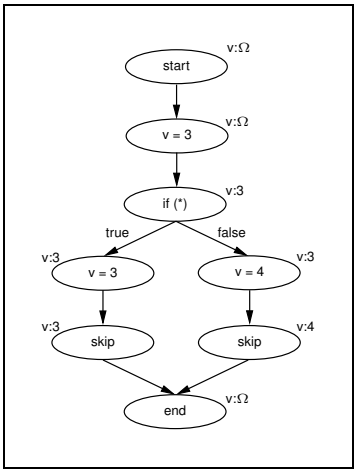


Fig. 2. Example CFG with associated analysis information.

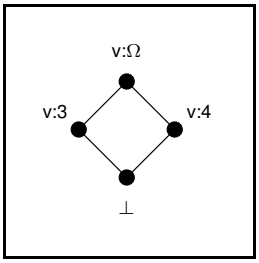


Fig. 3. A simple lattice.

soon as a fixed point is reached, where additional computations do not lead to changes anymore, the algorithm terminates, and the analysis is finished.

A *flow-sensitive* analysis considers the ordering of program instructions, whereas a *flow-insensitive* analysis does not (i.e., reordering instructions inside a function does not affect its results). *Interprocedural* analyses take function calls into account, while *intraprocedural* analyses operate only inside a single function. *Context-sensitive* interprocedural analyses are able to distinguish between different call sites to a function, *context-insensitive* interprocedural analyses are not, and therefore mix up the information computed for different calls to the same function. Hence, the highest precision can be achieved by performing an analysis that is flow-sensitive, interprocedural, and context-sensitive. Our presented scanner is equipped with a combination of all these desirable features. To be more precise, we apply the *functional approach* described in [24] for performing a context-sensitive analysis, with a number of extensions for handling local function variables and call-by-reference parameters.

An analysis is *sound* if the information it generates is a safe approximation of the real information. In the context of vulnerability scans, a sound analysis must include all vulnerabilities in its report. *Completeness* denotes the opposite notion, meaning that a complete analysis must not generate any false alarms (*false positives*).

Note that the theory of *abstract interpretation* is closely related to data flow analysis, although it is formulated on a higher level. A comprehensive overview of abstract interpretation, data flow analysis and other program analyses is given in [19], and a vivid introduction to the topic can be found at [20].

## IV. PHP Front-End

In order to conduct static analysis, the input program has to be parsed and transformed into a form that makes the following analysis as easy as possible. This first step includes the linearization of arbitrarily deep expressions in the original language as well as the reduction of various loop and branch constructs such as “foreach” and “switch” to combinations of “if”s and “goto”s. The resulting intermediate representation, *P-Tac*, resembles the classical *three-address code* (TAC) presented in [1]. TAC is an assembly-like language characterized by statements with at most three operands and the general form “x = y op z”. For example, the input statement “a = 1 + b + c” would be translated into the corresponding TAC sequence “t1 = 1 + b; t2 = t1 + c; a = t2”. The variables t1 and t2 are temporaries introduced in the course of the translation, and do not appear elsewhere in the program.

### A. Parse Tree Construction

As the first step towards the desired intermediate representation, the input PHP code is parsed and stored as a parse tree for further processing. PhpParser, our tool for generating parse trees for PHP code, is a combination of the Java lexical analyzer JFlex [12], the Java parser Cup [5], and the Flex and Bison specification files from the sources of the PHP interpreter [21]. Due to a few incompatibilities of Flex and Bison with their Java counterparts, we carefully modified JFlex and Cup in order to accept the original specification files and to permit the convenient definition

| CFG Node | Shape / Description |
|---|---|
| Simple Assignment | {var} = {place} |
| Unary Assignment | {var} = {op} {place} |
| Binary Assignment | {var} = {place} {op} {place} |
| Array Assignment | {var} = array() |
| Reference Assignment | {var} &= {var} |
| Unset | unset({var}) |
| Global | global {var} |
| Call Preparation | A call node's predecessor. |
| Call | Represents a function call. |
| Call Return | A call node's successor. |

TABLE I
MAIN TYPES OF CFG NODES IN P-TAC.

of Java actions for constructing the parse tree. The PhpParser package contains a more detailed documentation of these modifications.

### B. Intermediate Representation: P-Tac

In the next step, the constructed parse tree is transformed into *P-Tac*, a linearized form of the original PHP script resembling three-address code [1], and kept as a control flow graph for each encountered function. The code in the global scope (external to all user-defined functions) is moved into a special "main" function, which represents the starting point of the PHP script. All loop constructs (while, for, switch) are replaced by equivalent "if"-branches in order to prevent unnecessary redundancies in the following analysis. *Constants* in PHP are specified with the built-in "define" function, whereas values such as 77 or "foo" will be termed as *literals*. Table I gives an overview of the most important CFG nodes created by the P-Tac converter. The *place* abstraction denotes variables, constants, and literals and was introduced to permit more concise specifications. Function calls are represented by three CFG nodes to make the design of interprocedural transfer functions easier (a call preparation node, the actual call node, and a call return node). Calls to functions for which no definition was found are replaced by calls to the special *unknown function*.

## V. ANALYSIS BACK-END

A straightforward approach to solving the problem of detecting taint-style vulnerabilities would be to immediately conduct a *taint analysis* on the intermediate representation generated by the front-end. This taint analysis would identify points where tainted data can enter the program, propagate taint values along assignments and similar constructs, and inform the user of every sensitive sink that receives tainted input. However, to enable the analysis to produce correct and precise results, significant preparatory work is required. For instance, whenever a variable is assigned a tainted value, this taint value must not be propagated only to the variable itself, but also to all its aliases (variables pointing to the same memory location). Hence, information about alias relationships has to be provided by a preceding *alias analysis*. Moreover, it would be very beneficial for the taint analysis to know about the literal values that variables and constants may hold at each program point, since it allows a number of improvements in precision presented later in this paper. This information is collected by *literal analysis*. These three components (alias analysis, literal analysis, and taint analysis) will be discussed in the following sections separately for better clarity. Our prototype implementation, however, conducts all three analyses simultaneously, i.e., it operates on a single unified carrier lattice. The differences between these two approaches are restricted to the level of engineering and have no effect on the presented concepts.

One of the key features of our analysis is its high precision, since it is flow-sensitive, interprocedural, and context-sensitive. Moreover, we are the first to perform alias analysis for an untyped, reference-based scripting language such as PHP. Although there exists a rich literature on C pointer analysis, it is questionable whether these techniques can be directly applied to the semantically different problem of alias analysis for PHP references. As mentioned in a recent technical report by Xie and Aiken [29], static analysis of scripting languages is regarded as a

difficult problem and has not achieved much attention so far. In this context, even apparently trivial issues such as the simulation of the effects of a simple assignment require careful considerations. For instance, multi-dimensional arrays can contain elements that are neither explicitly addressed nor declared. To correctly handle the assignment of such a multi-dimensional array to another array variable, these hidden elements must be taken into account. These issues will be discussed in the following sections.

## VI. Literal Analysis: Basics

The purpose of literal analysis is to determine, for each program point, the literal that a variable or a constant can hold. This information can be used for improving the precision of the overall analysis in various ways. For instance, our prototype attempts to evaluate branch conditions (resulting from "if" statements) based on this information, and ignores program paths that cannot be executed at runtime (a technique called *path pruning*). Other potential uses of literals information are the resolution of non-literal include statements, variable variables, variable array indices, and variable function calls.

### A. Carrier Lattice Definition

As mentioned in Section III, an important building block for data flow analyses is the carrier lattice. The lattice used for literal analysis basically resembles the simple lattice for our toy programming language that was introduced in Figure 3, with two differences. First, the literal analysis lattice does not provide mappings for a single variable, but for all variables and constants that appear in the scanned program. Second, it is able to describe the mapping to any possible literal, and not just to the literals 3 and 4. Since the number of possible literals is infinite, this means that the "breadth" of the lattice (when shown as Hasse diagram [28]) is infinite as well. Figure 4 shows a fragment of a carrier lattice for a program with two variables ($a and $b) and one constant (CONST) to provide an intuitive feeling for the ordering among lattice elements. Note that the lower two elements differ only in the value that the variable $a is holding. Hence, the least upper bound of these two elements is identical except that it maps $a to $\Omega$. The top element of the whole lattice (which is not depicted in Figure 4) maps all variables and constants to the unknown literal $\Omega$, meaning that we know "absolutely nothing". As in the previous simple lattice, the bottom element (not depicted either) is just a special placeholder element needed by the analysis algorithm.

### B. Transfer Functions Definition

After defining the carrier lattice for the analysis, each CFG node has to be associated with a transfer function. These transfer functions determine how the analyzed information is affected when control flows through the corresponding CFG node. That is, it takes the lattice element entering the node as input, and returns a lattice element reflecting the node's semantics as output. The most straightforward example for literal analysis is a node of the form "simple_variable = literal", with the term *simple variable* denoting a variable that is neither an array nor an array element. For such a node, the transfer function only has to adjust the literal mapping of that variable. For example, the statement "$a = 5" assigns the literal value 5 to the variable $a in the carrier lattice. For CFG nodes like "simple_variable = variable", the assigned literal is not immediately available, but has to be extracted from the incoming lattice element by inspecting the mapping of the variable on the right side. Nodes like "simple_variable = constant" can be treated analogously to "simple_variable = variable".

Additional complexity arises when taking arrays, array elements and non-literal array indices into account. In the course of our work, we found that the problem space can be divided into four cases. These cases depend only on characteristics of the variable on the *left* side of the assignment. An overview of these cases is given in Table II, which also contains information about taint analysis and aliases. These topics will be dealt with later in this paper and can be ignored for the moment. In the rest of this section, we will discuss the four cases in order of increasing complexity.

Before proceeding with the first case, we have to be aware that PHP is an untyped language without explicit type declarations. It provides no explicit information about whether a variable is an array or not. Hence, the analysis considers a variable to be an array if it is indexed at some point in the program. For instance, if the expression "$a[2]" appears somewhere in the program, then variable "$a" certainly is an array. Unfortunately, the absence of such expressions does not necessarily imply that a variable is not an array. This is demonstrated in Figure 5. Here,

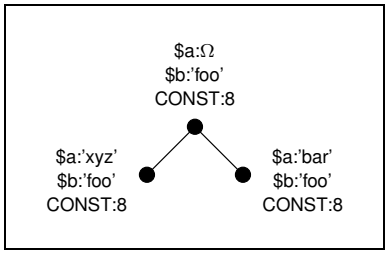


Fig. 4. Fragment of a literal analysis lattice.

```
1: $a[1] = 7;   // $a obviously is an array
2: $b = $a;     // $b is a hidden array:
3:              // it is not indexed anywhere
4: $c = $b;
5: echo $c[1];  // $c obviously is an array
```

Fig. 5. Arrays can be hidden.

$b is never indexed, but is still an array due to the assignment of array $a. Therefore, to assert the correctness of the statement "variable x is not an array", additional techniques would be required. This means that the intuitive introductory examples involving "simple_variable" need to be extended to deal with this issue, as we have no guarantee that a variable is really simple. Array elements do not suffer from this uncertainty: While $a[1] surely is an array element, $a is not.

The simplest, first case applies when the left variable is not an array element (which can be easily decided) and not known as array[3]. Here, the analysis proceeds just as in the introductory "simple_variable" example, without taking into account whether the variable on the right side might have array elements or not. This punctual overwrite operation is called *strong update*. Note that when using this simple approach, precision might suffer. For instance, in Figure 5, the analysis cannot determine that $c[1] is mapped to the literal 7 at the end of the program. However, this issue did not lead to significant losses in precision during our experiments.

The second of the four cases that need to be distinguished is when the left variable is an array, but not an array element. A useful concept in this respect is that of an *array tree*, which describes an array and its contents as a tree. The array variable itself is the tree's root, the array's elements are interior nodes and leaves, and its indices are edge labels. For example, Figure 6 shows the tree for a two-dimensional array with the elements $a[$i][2] and $a[3][4]. Literals can be associated with each node to represent the current mappings. Intuitively, the analysis has to "overlap" the array tree of the left variable with the array tree of the right variable such that literals for matching nodes are overwritten. For instance, in the course of the assignment of array $b to array $a ($a = $b), the literal of $b[1] must overwrite the literal of $a[1]. The literals of nodes on the left side for which there is no matching node on the right side generally have to be set to Ω because it is uncertain whether there *really* is no matching node (due to the possibility that the array might contain hidden elements). In Figure 5, for example, $c[1] is set to Ω after the assignment in line 4 because the analysis cannot determine whether there is an element $b[1] or not. An exception to this rule can be made if there is a literal or a constant on the right side of the assignment. Since literals and constants can never be arrays, the analysis is free to set the literals of all array elements on the left side to NULL[4]. A recursive algorithm for performing all these operations is given in Figure 7, where the term *direct element* denotes an array element that is retrieved by adding a single index (the *direct index*) to its enclosing array.

Cases three and four require the understanding that array elements can have one or more non-literal indices. An example would be $a[1][$i][2], which has one non-literal and two literal indices. Such *non-literal array elements* have to be treated in a special way because they can represent multiple variables (such as $a[1][8][2] or $a[1][9][2] for the given example). Our analysis handles them in a pessimistic way in that they are permanently mapped to Ω. Otherwise, it would be necessary to track the non-literal indices of these elements and recompute the taint value of a non-literal array element whenever one of its indices changes. For example, if the variable $i changes at some point in the program, the analysis would have to recompute the values for all array elements with at least one index containing $i (such as $a[$i] or $b[$a[$i]]). Our experiments did not suggest that this additional complexity would lead to a significant gain in precision.

In the third case, the left variable is an array element without non-literal indices. This variable may also be an array (i.e., it does not matter whether it is known as array or not). This case is handled in the same way as case

[3]Note that we use the phrase "not known as array" to include the possibility that a variable might be a "hidden" array

[4]This corresponds to the actual semantics in PHP.

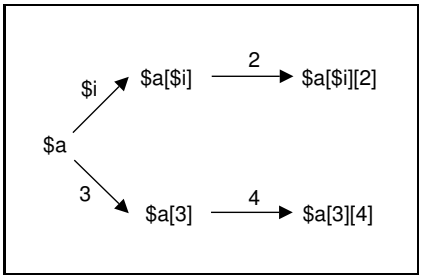


Fig. 6. Array Tree Example.

```
strongOverlap(Variable target, Place source) {
  if source known as array
    if target known as array
      for all direct elements of target
        if there is a direct element of source with the same direct index
          strongOverlap(target element, source element)
        else
          set the array tree of the direct element to Omega
  else if source is literal or constant
    set the array tree below target to NULL
  else
    set the array tree below target to Omega

  set the target literal to the source literal
}
```

Fig. 7. Strong Overlap Algorithm.

two, using the strong overlap algorithm. Note, however, that taint analysis (discussed in Section IX) will perform different operations for these two situations.

In the fourth and last case, the left variable is an array element with non-literal indices and maybe an array. As already mentioned, this case has to be treated separately because it is not certain which array element is actually meant by the non-literal array element. For instance, when assigning the literal 3 to $a[$i], the analysis has to consider that this might affect $a[8], $a[9], or any other element. So, instead of overwriting the literals of the possibly affected variables with the literal 3, we have to conservatively replace them with the least upper bound of the old and the new literal. For example, if the old literal of $a[8] was 7, it becomes $\Omega$. If it was 3, it remains 3. This approach can be formulated in a succinct way using the terms *MI variables* and *weak overlap*. The *MI variables* of a non-literal array element are all variables that are *maybe identical* to this array element. For example, $a[8] and $a[9] are MI variables of $a[$i]. These are the variables that might be affected by an assignment, and hence, have to be considered by the analysis. The *weak overlap* algorithm is analogous to the strong overlap algorithm, with the difference that all overwrite operations are replaced by least upper bound operations. This way, the analysis can handle assignments of the fourth type by performing a weak overlap for all MI variables.

### C. Dependence on Alias Analysis

In the explanations given so far, we omitted an important problem. PHP allows the creation of aliases, which means that two or more variables can point to the same memory location. If one of these variables is assigned a new value, it also affects the aliases of the variable. Ignoring this issue would prevent literal analysis from producing correct results in a number of cases. Hence, the following Section VII gives a more detailed introduction to PHP references and describes an alias analysis that collects the alias information required by literal analysis. Section VIII revisits literal analysis such that this information is taken into account, and presents transfer functions for the remaining CFG nodes.

## VII. Alias Analysis

The basic literal analysis introduced in the previous section is required to improve the precision of taint analysis, which is responsible for detecting vulnerabilities in the input program. As previously mentioned, a dependency exists between literal analysis and alias analysis. To understand why literal analysis can produce correct results only when having access to alias information, we give a short introduction to PHP references. After this problem definition, we specify the workings of our alias analysis.

### A. Aliases in PHP

An *alias* of a variable is a variable that refers to the same memory location. Assigning a value to a variable results in this value being written to the variable's memory location, and therefore, also affects all aliases of this variable. In this sense, aliases in PHP resemble those in Java. Note that PHP references are different from pointers in C, which makes our work distinct from existing C pointer analysis literature. Figure 8 shows an example for creating an alias relationship between variables $a and $b through the use of the *reference* operator "&". Further details on PHP references are discussed in the PHP Manual [21].

The last assignment in Figure 8 demonstrates why literal analysis requires access to analysis information. Without this information, literal analysis would not be able to decide that the last assignment to $a also affects $b. Instead, the value of $b would remain unchanged and hence, be incorrect. Therefore, alias analysis has to be performed to collect the necessary alias information.

### B. Carrier Lattice Definition

Analogous to literal analysis, the first step towards creating an alias analysis is the definition of an appropriate carrier lattice, which encodes the information to be collected. For our purposes, we represent alias information in the following manner. Let an *alias group* denote a group of variables that are referencing the same memory location. If some variable is the only one to reference a specific memory location, it is the only member of its alias group. An intuitive way to describe alias information is through sets of alias groups (*alias group sets*). In such an alias group set, each variable appears in exactly one alias group, together with all its aliases. Unfortunately, this approach makes it impossible to model the effects of merging program paths, for instance, when two program paths meet at the end of an "if"-construct. Figure 9 shows an example for such a case. Inside the "true" branch, the variable $a is aliased with $b, but if the condition indicated by ".." evaluates to false, $a is aliased with $c. Under the assumption that the condition is determined by dynamic factors (such as environment variables or user input), static analysis is not able to decide which path the program will take. Thus, we have to take both possibilities into account, and the information after the "if"-construct has to specify that "$a could alias $b, but it could also alias $c". This can be achieved by modeling alias information through *sets of alias group sets*. In Figure 9, an alias group is enclosed by round braces, whereas an alias group set is enclosed by curly braces. When the analysis has finished, each program point is associated with a set of alias group sets, which is an element of the carrier lattice.

An important characteristic of the lattice elements described in the last paragraph is that they permit the computation of must-aliases and may-aliases of a variable. Intuitively, the *must-aliases* of a variable are all variables that certainly reference the same memory location as the variable. Given the lattice element "{(a,b) (c)}", $b is a must-alias of $a because there is no other alias group set, and hence no other alias group containing $a. The *may-aliases* of a variable are all variables that may reference the same memory location. In the lattice element "{(a,c) (b)} {(a,b) (c)}", $c and $b are may-aliases of $a.

The order among lattice elements is simply defined as subset inclusion. Figure 10 illustrates a fragment of an alias analysis lattice for three variables. The element "{(a,b) (c)}" obviously is a subset of the element "{(a,c) (b)} {(a,b) (c)}". Higher elements inside the lattice are less precise than lower ones: As already mentioned, element "{(a,b) (c)}" denotes that $a must be an alias of $b. This information is certainly more precise than that of the element "{(a,c) (b)} {(a,b) (c)}", which indicates that $a may alias either $b or $c.

### C. Transfer Functions Definition

In the following description of transfer functions for alias analysis, we refer to CFG nodes by the names given in Table I. All nodes that are not mentioned have no effect on alias information and are associated with the identity function.

```
1: $a = 1;
2: $b = 2;
3: $b =& $a;  // $b == 1
4: $a = 3;    // $a == 3, $b == 3
```

Fig. 8. Aliasing example.

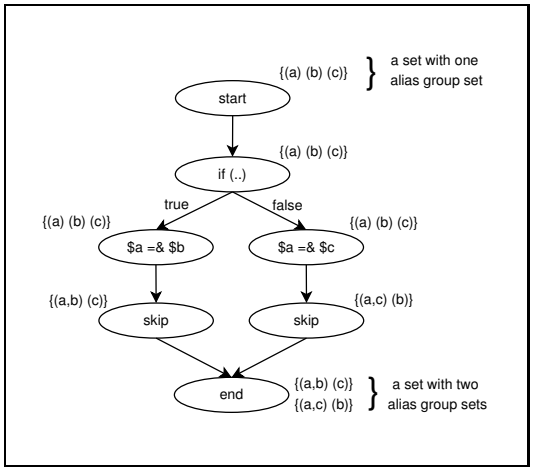


Fig. 9. Merging paths and alias analysis lattice elements.

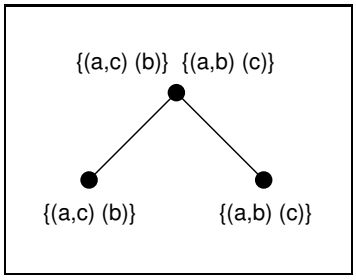


Fig. 10. Fragment of an alias analysis lattice.

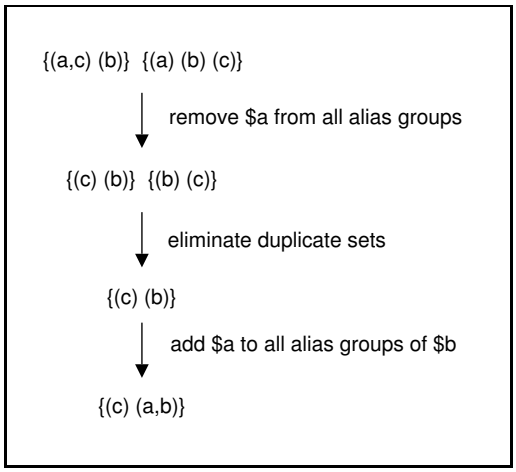


Fig. 11. Modification of alias group sets for $a =& $b.

Nodes of the form "$a =& $b" have already appeared in previous examples. For such **reference assignment nodes**, variable $a (the redirected variable) is first removed from its alias group in each alias group set. Then, it is added to $b's alias group in all alias group sets. This reflects that $a and $b are now aliases. If any duplicate sets are created through these operations, they have to be removed (see Figure 11 for an example).

In PHP, the "unset" statement has the effect that the operand variable is associated with a fresh memory location holding the value NULL. Hence, for the **unset node**, the analysis removes the operand variable from its alias group and puts it into its own one-element alias group for each alias group set. The "global" keyword in PHP is used inside functions to get access to variables from the global scope (which is outside all functions). Hence, declaring a variable as "global" with a **global node** is equivalent to a reference assignment with this variable on the left side and an equally-named variable from the global scope on the right side.

Note that alias information about constants does not have to be maintained at all, since PHP does not permit reference statements to have constant operands. Apart from this, we ignore reference statements with operands that are arrays or array elements. The reasons for this restriction are the complexity that such statements would add to our analysis and their rare usage in practice.

The only transfer functions that remain to be defined are those for function calls (*interprocedural* transfer functions). These are required for function calls and responsible for handling parameters, clearing and restoring local variables, and passing back the return value. A general problem of all interprocedural analyses is that it might be impossible to determine the depth of a recursive call chain. Consider a function that contains a conditional recursive call to itself. If the condition in which this recursive call is enclosed depends on dynamic information (such as user

```
 1: function f1() {
 2:   // when entering this function, the local variables $a and $b
 3:   // do NOT point to the same memory location
 4:   $a; $b;
 5:
 6:   // after the following statement, $a and $b DO point to the same memory location,
 7:   // but this must not affect $a and $b in other incarnations of this function
 8:   $a =& $b;
 9:
10:   if (..) f1();
11: }
```

Fig. 12. Example PHP function highlighting interprocedural issues.

input), it is impossible to determine the call depth statically. For non-terminating programs, the call depth would even be infinite. This particular problem is addressed by the techniques described in [24], which are beyond the scope of this paper. We adapted one of these techniques (the *functional approach*, as opposed to the *call-string approach*) for our purposes. An important problem is to prevent data flow information for one *incarnation* of a local function variable to interfere with the information of another incarnation of this variable, as demonstrated in Figure 12. In this example, variable $b points to the same location as variable $a after the reference assignment in Line 8. However, after the function f1 is called recursively in Line 10, a new incarnation of local variables is needed where $a and $b do *not* reference each other. Interference between variables could occur because the underlying carrier lattice does not distinguish between different variable incarnations. To address this problem, the transfer function at the **call preparation node** stores the alias information for the local variables of the calling function, and resets it to its default (initial) value. On function return (i.e., at the **call return node**), the alias information for local variables of the callee is reset to its default, while the caller's locals are restored again. Note that formal parameters of a function also belong to the function's local variables and can be treated analogously. The interprocedural flow of values along reference parameters is handled by the extended literal analysis described in Section VIII.

## VIII. Literal Analysis Revisited

Using the information collected by alias analysis, we can extend the basic concepts for literal analysis presented in Section VI to correctly include alias relationships. For a simple assignment of the form "$a = $b", the aliases of the variable on the left side are relevant for literal analysis. The reason is that all aliases of variable $a are also affected by the assignment. Of the four cases that literal analysis had to distinguish in Section VI, all but the first involved an array or array element on the left side. Since we only consider references to simple variables, there cannot exist aliases for the variable on the left side in these cases. Hence, no further extensions are necessary. In the first case, however, the left variable is not an array element (and not known as array), and therefore, might possess aliases. Must-aliases of this variable are treated by the transfer function with a strong update, i.e., their literals are simply overwritten with the literal read from the right side of the assignment. As there is no sufficient certainty for may-aliases, they must be handled conservatively by a weak update. That is, all aliases of the variable on the left-hand side are assigned the least upper bound of their literal and the literal from the right side. An overview of the cases and actions for simple assignment nodes is given in Table II.

Now that the transfer function definition for simple assignment nodes is completed, the transfer functions for the other relevant CFG nodes can be introduced. For **unary assignment nodes** such as "$a = -$b", the literal resulting from the application of the operator on the right side is computed first, and then the presented technique for simple assignment nodes is used. For example, if $b evaluates to 3, $a is assigned -3. Special care was taken to reflect PHP's implicit type conversion mechanisms, which are beyond the scope of this paper. The treatment of **binary assignment nodes** such as "$a = $b + $c" is analogous to unary assignments, with the sole difference that the operator on the right side takes two input arguments instead of one. Note that when we do not dispose of specific information for an operand (i.e., if it is $\Omega$), the result of the operation is also $\Omega$. For **reference assignment nodes**

| Left Variable | Literal Analysis | Taint Analysis |
|---|---|---|
| Not an array element and not known as array (“normal variable”). | strong update for must-aliases, weak update for may-aliases | strong update (taint, CA flag) for must-aliases, weak update (taint, CA flag) for may-aliases |
| Array, but not an array element. | strong overlap | target.caFlag = source.caFlag; strong overlap (taint) |
| Array element (and maybe an array) without non-literal indices. | strong overlap | target.root.caFlag $\sqcup$= source.caFlag; strong overlap (taint) |
| Array element (and maybe an array) with non-literal indices. | weak overlap for all MI variables | target.root.caFlag $\sqcup$= source.caFlag; weak overlap (taint) for all MI variables |

TABLE II

ACTIONS PERFORMED BY LITERAL ANALYSIS AND TAINT ANALYSIS FOR SIMPLE ASSIGNMENT NODES DEPENDING ON THE LEFT-HAND VARIABLE.

such as “$a =& $b”, it is sufficient to overwrite the literal of $a with the literal of $b, since reference statements have been restricted to simple variables. **Global nodes** can be handled as normal assignments, with the operand variable on the left side and an equally-named variable from the global scope on the right side.

analogously, as was already done for alias analysis. In the case of **unset nodes**, the unset variable and all its literal array elements are set to the NULL literal, which represents the PHP null value. As far as literal analysis is concerned, **array assignment nodes** have the same semantic effects as unset nodes, and can be treated in the same way.

What remains to be specified are the interprocedural transfer functions. For **call preparation nodes**, all value and reference parameters of the called function are treated like assignments with the shape “formal parameter = actual parameter”. Local variables are reset to their initial (default) values, analogously to alias analysis (as described in Section VII-C). At the **call return node**, the function’s locals have to be restored again. Next, the effect of values flowing through reference parameters has to be handled. For this purpose, the analysis performs assignments of the shape “actual parameter = formal parameter” for each parameter that was called by reference. This technique relies on the assumption that reference parameters are not redirected to another variable inside the callee. A straightforward way to handle deviations from this assumption in a sound way would be to note the violating formal parameters for each function and to conservatively set the corresponding actual parameters to $\Omega$ at the return node. Before resetting the callee’s locals and its return value, the return value is “saved” by assigning it to the temporary variable provided by the P-Tac conversion for representing the function’s expression value.

Functions that are built into PHP are conservatively modeled as returning $\Omega$, since the increased precision is expected to be rather small compared to the required work necessary for simulating these functions. For a safe approach, the analysis can additionally set all reference parameters to $\Omega$ as well. The only built-in function that is modeled precisely is the frequently used *define*, which is needed for the definition of constants.

## IX. TAINT ANALYSIS

With the information from literal analysis and alias analysis at its disposal, taint analysis is finally able to produce satisfactory results. Taint analysis strongly resembles literal analysis. Its purpose is to determine, for each program point, the taint value (instead of the literal) of a variable or constant. Once these results have been computed, it is possible to inspect whether any sensitive sink in the program is receiving malicious data, and hence, to detect vulnerabilities.

### A. Carrier Lattice Definition

A variable or constant is said to be tainted if it can hold a malicious[5], not yet sanitized (checked) value originating from user input. Since literals can never hold user input, they are always untainted. The basic carrier lattice for taint analysis resembles that of literal analysis, with the difference that it does not map to literals and $\Omega$, but to the taint

[5]Malicious with respect to cross-site scripting for our prototype implementation, but the presented concepts are able to cover all taint-style vulnerabilities.

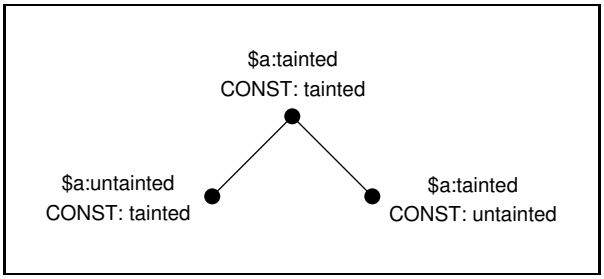


Fig. 13. Fragment of a basic taint analysis lattice.

```
1: $a = <user input>;
2: // $a[1] can be controlled by an attacker
3:
4: $a = array();
5: // now $a[1] is no longer controlled by
6: // an attacker
```

Fig. 14. Untainting with "array".

values *tainted* and *untainted*. Note that we take a conservative (safe) approach in that a variable being mapped to tainted means "this variable *might* be tainted", whereas a mapping to untainted means "this variable *is* untainted". Hence, whenever the analysis cannot determine whether a variable is tainted or not, it is conservatively assumed to be tainted. In the context of data flow analysis, being tainted is therefore less precise than being untainted, which is illustrated by the carrier lattice fragment in Figure 13. Just like for the previous analyses, less precise lattice elements are located above more precise elements.

Recall that literal analysis treats non-literal array elements (such as $a[$i]) in a pessimistic way in that they are always mapped to $\Omega$. For the same reasons, taint analysis maps non-literal array elements to tainted. In the context of taint analysis, however, this leads to undesirable false positives in certain cases. In particular, declaring a variable with the built-in "array" function clears all its content, including all values possibly injected by an attacker (see Figure 14). Whenever an array is cleared in such a manner, we would like to treat its content as untainted, even though taint analysis might yield a different result. This is why we track an additional *clean array flag* (CA flag) for each variable that is not an array element. An active CA flag overrides the taint information for the whole array tree, meaning that all its elements (including those with non-literal indices) are untainted. The CA flag of an array element is implicitly considered to be equal to the CA flag of its enclosing array.

### *B. Transfer Functions Definition*

Due to its similarity to literal analysis, the transfer functions for taint analysis sound familiar. For the **simple assignment node**, the same distinction of cases with regard to the left variable has to be made, and the operations affecting the taint values are completely analogous to those for literal values. The only difference is that taint analysis also has to consider the "CA flag" extension to its lattice. An overview of the necessary operations is given in the rightmost column of Table II. The "root" field used in Table II denotes the variable itself if it is not an array element (e.g., $a.root == $a) and the root node of the corresponding array tree otherwise (e.g., $b[1][2].root == $b).

As mentioned in Section VIII, special care was taken to handle PHP's implicit type conversion mechanisms for **unary assignment nodes**. Doing this in the context of taint analysis has the desirable effect that sanitization through type casting is handled correctly. For instance, implicitly casting a tainted variable into an integer (with unary operators such as +, -, and (int)) untaints this variable, since cross-site scripting attacks require to display more data than just simple integers in order to work properly. The same holds for **binary assignment nodes**.

For **reference assignment nodes** such as "$a =& $b", we can adapt the literal analysis transfer function in a straightforward fashion: It is sufficient to overwrite the taint value and CA flag of $a with the taint value and CA flag of $b. At **unset nodes**, the operand variable and all literal array elements are set to untainted. If the operand variable is not an array element, its CA flag is set to "clean" (recall that CA flags are not tracked explicitly for array elements). **Array assignment nodes** and **global nodes** are treated as usual, i.e., analogous to unset nodes and reference assignment nodes, respectively.

The interprocedural operations necessary for taint analysis are completely analogous to literal analysis, with the differences for handling CA flags already discussed.

In contrast to literal analysis, it is important for taint analysis to correctly model built-in PHP functions in order to reduce the number of false positives. For this purpose, Pixy processes a specification file on startup which contains (currently about 30) abstracted versions of built-in functions in PHP syntax. For modeling taint values, the special

placeholder variables $_TAINTED and $_UNTAINTED are used. For instance, the effect of the built-in function "htmlentities", which is very effective in sanitizing input against cross-site scripting attacks, is implemented by simply letting it return $_UNTAINTED.

### C. Using the Analysis Results

Generating warnings that point the developer to possible cross-site scripting vulnerabilities at the end of the analysis is straightforward. The analysis information for each sensitive sink (such as calls to "echo" and "print") is searched for tainted input variables, and a warning message indicating the corresponding line is issued if such a violation is discovered.

### D. Limitations

Currently, Pixy does not support object-oriented features of PHP. Each use of object member variables and methods is treated in an optimistic way, meaning that malicious data can never arise from such constructs. In addition, files included with "include" and similar keywords are not scanned automatically. In our experiments, we frequently observed false positives stemming from these lacking file inclusions, which we eliminated through manual inclusion. Unfortunately, automation of this manual procedure is not straightforward because file inclusions in PHP are dynamic, in contrast to the static preprocessor includes in C and imports in Java. This means that the names of the files to be included can be constructed at run-time, recursive and conditional inclusions are permitted, and included files can even return values. In this sense, the inclusion mechanism of PHP strongly resembles that of function calls, with a number of differences concerning variable scoping.

## X. Empirical Results

We performed a series of experiments with our prototype implementation to demonstrate its ability to detect previously known cross-site scripting vulnerabilities, as well as new ones. To this end, Pixy was run on six popular, open source PHP programs. The program files on which Pixy was evaluated and our prototype itself can be obtained from our website [14]. Since Pixy does not automatically continue its analysis into included files yet, we manually resolved include relationships for the scanned files. More precisely, we simply provided missing function definitions and static definitions of global variables, which took less than an hour for each application. We are currently working on a straightforward extension that automatically inlines included files, which would eliminate this manual task. Each file was analyzed in less than a minute using a 3.0 GHz Pentium 4 processor with 1GB RAM, even though our prototype still presents many opportunities for performance tuning.

Tables III and IV summarize the results of our experiments. In three applications, we reconstructed 36 known vulnerabilities with 27 false positives (FP's). In three other applications, we discovered 15 previously unknown vulnerabilities with 16 false positives. In these cases, we informed the authors about the issues and posted security advisories to the BugTraq mailing list [3]. Pixy also reported a few programming bugs not relevant for security, such as function calls with too many arguments. Note that since these bugs have no influence on a program's security properties, they were counted neither as vulnerabilities nor as false positives. These results clearly show that our analysis is capable of finding novel vulnerabilities in real-world applications.

### A. Case Studies

Detailed descriptions of the discovered vulnerabilities can be found in the corresponding BugTraq postings. In this section, we will take a closer look at two interesting vulnerabilities that demonstrate the requirement to perform an analysis that is able to track data flows throughout the program.

The Reviews Module of PhpNuke contains an interesting flaw related to the use of a superficially harmless-looking built-in function. Our analyzer makes sure that all built-in functions are considered to return tainted values by default. This way, no vulnerabilities can be missed due to built-in functions that have not been modeled explicitly to return untainted values. Explicit modeling is performed by providing a short specification in a configuration file that is processed at start-up. A list of currently modeled built-in functions can be found in the appendix. False positives arising from harmless but unmodeled functions can easily be eliminated by providing a specification of the function's true behavior. In Figure 15, a simplified version of the vulnerable code shows that the second parameter of function postcomment is echoed on Line 5. Originally, this warning was issued because the function urldecode

```
1: function postcomment($id, $title) {
2:     ...
3:     $title = urldecode($title);
4:     ...
5:     echo $title;
6:     ...
7: }
```

Fig. 15. PhpNuke vulnerability (simplified).

```
 1: if (...) {
 2:     $entry = $_GET['entry'];
 3:     ...
 4:     $temp_file_name = $entry;
 5:     ...
 6: } else {
 7:     ...
 8:     $temp_file_name =
           stripslashes($_POST['file_name']);
 9:     ...
10: }
11: ...
12: echo($temp_file_name);
```

Fig. 16. Simple PHP Blog vulnerability (simplified).

was unmodeled, and hence, returned a tainted value. However, a look into the PHP manual revealed that urldecode has to be handled with care, since it is able to transform benign character sequences into dangerous ones, such as transforming %3c into <. This is why even explicit sanitization prior to the call of function postcomment fails, which was reported in BugTraq posting 10493.

Figure 16 shows a simplified version of the file preview_static_cgi.php in Simple PHP Blog. The sensitive sink on Line 12 receives the variable $temp_file_name, which is initialized with a tainted value on *both* program paths of the "if"-construct on Line 1. If the guarding condition is true, the variable is initialized with $entry (on Line 4), which was assigned a tainted value from the GET array before (Line 2). Inside the second branch, $temp_file_name is tainted by a POST variable indirectly over a call to the built-in function stripslashes. This function returns the taint value of its parameter and has been modeled to do so. Note that Pixy would also have correctly detected a vulnerability if $temp_file_name were assigned an untainted value on just one of the two branches.

### *B. False Positives*

Among the 47 false positives that Pixy reported, 14 were caused by global variables that are initialized dynamically (e.g., through a database read) inside an included file. As mentioned previously, we only considered static initializations during the manual preprocessing step. Since uninitialized globals are conservatively treated as tainted, warnings were issued at the program points where these variables are sent back to the user. We are confident, however, that these false positives will be eliminated when include files are automatically processed as well.

The second largest group of false positives contains 13 warnings that can be traced back to file reads. In our analysis, we conservatively regarded values originating from files as being tainted. In these 13 cases, it turned out that an attacker is actually not able to inject malicious content into the files that were read. However, our conservative approach led to the detection of two previously unknown vulnerabilities. The ratio between false positives and vulnerabilities for this problem could be improved by tracking the files into which an attacker may be able to inject tainted values.

Since our alias analysis does not cover aliasing relationships for arrays and array elements, a global array and its content cannot be untainted by statements that are located inside functions. In seven cases, a global array element is untainted inside a sanitization function, followed by an output statement that contains the (incorrectly tainted) global.

An interesting kind of false positive with six warnings arose while scanning PhpNuke. In the YourAccount module, values originating from the user are embedded into the output as attributes of HTML tags. Although these values were not thoroughly sanitized prior to their use, the existing sanitization is sufficient because it makes sure that they do not contain double quotes. But since the attribute fields are delimited by double quotes, the attacker's input is "trapped" inside these attributes where it is not able to do any harm.

Custom sanitization using regular expressions is a dangerous practice. It is easy to miss dangerous characters, especially when the cases get more complex and when the implementor lacks the necessary expertise. Therefore, Pixy does not consider the use of such methods as sanitization. In two cases, values that have undergone such a

| Program | File | LOC | Variables | Vulnerabilities | FP's | Advisories |
|---|---|---|---|---|---|---|
| PhpNuke 6.9 | Reviews Module | 8409 | 3113 | 15 | 5 | BugTraq: 10493, 10524, 365368 |
| | YourAccount Module | 9070 | 3452 | 9 | 25 | BugTraq: 13007, 394971, 394867, 321324 |
| PhpMyAdmin 2.6.0-pl2 | select_server.lib.php | 89 | 23 | 9 | 0 | PMASA-2005-01 |
| Gallery 1.3.3 | search.php | 1810 | 530 | 2 | 1 | BugTraq: 348514 |
| | login.php | 1719 | 488 | 1 | 0 | BugTraq: 8039 |
| | **Totals** | 21097 | 7606 | 36 | 31 | |

TABLE III

KNOWN VULNERABILITIES DISCOVERED BY PIXY.

| Program | File | LOC | Variables | Vulnerabilities | FP's | Advisories |
|---|---|---|---|---|---|---|
| Simple PHP Blog 0.4.5 | preview_cgi.php | 6938 | 2342 | 3 | 5 | TUVSA-0511-001, BugTraq 415463 |
| | preview_static_cgi.php | 6883 | 2316 | 4 | 4 | |
| | colors.php | 6971 | 2313 | 1 | 6 | |
| Serendipity 0.8.4 | personal.inc.php | 6588 | 2305 | 2 | 1 | TUVSA-0509-001, BugTraq 412023 |
| Yapig 0.95b | view.php | 5128 | 1302 | 5 | 0 | TUVSA-0510-001, BugTraq 413255 |
| | **Totals** | 29508 | 10578 | 15 | 16 | |

TABLE IV

UNKNOWN VULNERABILITIES DISCOVERED BY PIXY.

custom sanitization were reported as tainted. Manual inspection, however, did not reveal any ways for circumventing the protection.

The remaining five false positives were due to more or less complex "if"-constructs that are responsible for untainting a critical variable. Under certain conditions, it might be possible that none of the branches of the construct is taken, leaving the variable tainted. However, we did not find a way to induce such a bypassing condition.

## XI. RELATED WORK

Currently, there exist only few approaches that deal with static detection of web application vulnerabilities. Huang et al. [11] were the first to address this issue in the context of PHP applications. They used a lattice-based analysis algorithm derived from type systems and typestate, and compared it to a technique based on bounded model checking in their follow-up paper [10]. A substantial fraction of PHP files (8% in their experiments) is rejected due to problems with the applied parser. In contrast, we are able to parse the full PHP language. Moreover, Huang et al.'s work leaves out important issues such as the handling of references, array elements, or any of the limitations that we addressed in Section IX-D. Unfortunately, comparing their results to ours was not possible due to the limited detail of their reports (no version numbers or advisory ID's are given). After requesting a copy of their tool, the authors informed us of their plans to commercialize it, which prevents them to share it with other researchers.

A recent, unpublished paper by Xie and Aiken [29] addresses the problem of statically detecting SQL injection vulnerabilities in PHP scripts. By applying a custom, three-tier architecture instead of using full-fledged data flow analysis techniques, they operate on a less ambitious conceptual level than we do. For instance, recursive function calls are simply ignored instead of being handled correctly. Moreover, alias analysis is not performed at all, which further lowers the correctness of their approach. Multi-dimensional arrays also appear to be unsupported. They apply a heuristic for resolving simple cases of include statements that seems to yield good results in practice. It should be easy to incorporate this approach into our prototype.

Livshits and Lam [15] applied an analysis supported by binary decision diagrams presented in [27] for finding security vulnerabilities in Java applications. Their work differs from ours in the underlying analysis, which is flow-

insensitive for the most part, and the target language Java, which is a typed language. This considerably eases the challenges faced by static analysis.

In [16], a technique for approximating the string output of PHP programs with a context-free grammar is presented. While primarily targeted at the validation of HTML output, the author claims that it can also be used for the detection of cross-site scripting vulnerabilities. However, without any taint information or additional checks, it appears to be difficult to distinguish between malicious and benign output. Only one discovered XSS vulnerability is reported, and the observed false positive rate is not mentioned. Moreover, the presented tool currently supports only "basic features" of PHP, excluding references.

Engler et al. have published various static analysis approaches to finding vulnerabilities and programming bugs in the context of C programs. For example, in [6], the authors describe a system that translates simple rules into automata-based compiler extensions that check whether a program adheres to these rules or not. An extension to this work is given in [7], where the authors present techniques for the automatic extraction of such rules from a given program. In [2], tainting analysis is used to identify vulnerabilities in operating system code where user supplied integer and pointer values are used without proper checking.

An alternative approach aiming at the detection of taint-style vulnerabilities introduces special type qualifiers to the analyzed programming language. One of the most prominent tools that applies this concept is CQual [8], which has been, among other things, used by Shankar et al. [23] to detect format string vulnerabilities in C code. However, it remains questionable whether this technique can be applied to untyped scripting languages.

## XII. Conclusions

Web applications have become a popular and wide-spread interaction medium in our daily lives. At the same time, vulnerabilities that endanger the personal data of users are discovered regularly. Manual security audits targeted at these vulnerabilities are labor-intensive, costly, and error-prone. Therefore, we propose a static analysis technique that is able to detect taint-style vulnerabilities automatically. This broad class includes many types of common vulnerabilities such as SQL injection or cross-site scripting. Our analysis is based on data flow analysis, a well-understood and established technique in computer science. To improve the correctness and precision of our taint analysis, we conducted a supplementary alias analysis as well as literal analysis. All our analyses are interprocedural, context-sensitive and flow-sensitive for providing a high degree of precision and keeping the number of false positives low, making our tool useful for real-world applications.

We implemented our concepts in Pixy, an open-source Java tool able to detect cross-site scripting flaws in PHP scripts. In the course of our experimental validation, we discovered and reported 15 previously unknown vulnerabilities and reconstructed 36 known vulnerabilities, while observing a moderate false positive rate of around 50% (i.e., one false positive for each vulnerability on average).

There is an urgent need for automated vulnerability detection in Web application development, especially because Web applications are growing into large and complex systems. We believe that our presented concepts provide an effective solution to this problem, therefore offering benefits to both users and providers of Web applications.

## XIII. Acknowledgments

This work has been supported by the Austrian Science Foundation (FWF) under grant P18368-N04. We would like to thank our shepherd for his guidance in preparing the camera-ready version of the paper, and Markus Schordan for insightful discussions on the theory of data flow analysis and abstract interpretation.

## References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[2] K. Ashcraft and D. Engler, "Using programmer-written compiler extensions to catch security holes," in *IEEE Symposium on Security and Privacy*, 2002.

[3] BugTraq, "BugTraq Mailing List Archive," http://www.securityfocus.com/archive/1, 2005.

[4] CERT, "CERT Advisory CA-2000-02: Malicious HTML Tags Embedded in Client Web Requests," http://www.cert.org/advisories/CA-2000-02.html, 2005.

[5] CUP, "CUP: LALR Parser Generator in Java," http://www2.cs.tum.edu/projects/cup/, 2005.

[6] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *OSDI 2000*, 2000.

[7] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *SOSP '01: Proceedings of the 18th ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM Press, 2001, pp. 57–72.

[8] J. S. Foster, M. Faehndrich, and A. Aiken, "A theory of type qualifiers," in *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM Press, 1999, pp. 192–203.

[9] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai, "Web application security assessment by fault injection and behavior monitoring," in *WWW '03: Proceedings of the 12th International Conference on World Wide Web*. New York, NY, USA: ACM Press, 2003, pp. 148–159.

[10] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo, "Verifying web applications using bounded model checking." in *DSN*, 2004, pp. 199–208.

[11] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *WWW '04: Proceedings of the 13th International Conference on World Wide Web*. New York, NY, USA: ACM Press, 2004, pp. 40–52.

[12] JFlex, "JFlex: The Fast Scanner Generator for Java," http://jflex.de, 2005.

[13] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: A client-side solution for mitigating cross-site scripting attacks," in *The 21st ACM Symposium on Applied Computing (SAC 2006)*, 2006.

[14] S. S. Lab, "Secure Systems Lab, Technical University of Vienna," http://www.seclab.tuwien.ac.at, 2006.

[15] V. B. Livshits and M. S. Lam, "Finding security errors in Java programs with static analysis," in *Proceedings of the 14th Usenix Security Symposium*, Aug. 2005, pp. 271 – 286.

[16] Y. Minamide, "Static approximation of dynamically generated web pages," in *WWW '05: Proceedings of the 14th International Conference on World Wide Web*. New York, NY, USA: ACM Press, 2005, pp. 432–441.

[17] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[18] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *IFIP Security 2005*, 2005.

[19] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999.

[20] PAG, "PAG/WWW: Static Program Analysis," http://www.program-analysis.com, 2005.

[21] PHP, "PHP: Hypertext Preprocessor," http://www.php.net, 2005.

[22] T. Pietraszek and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *Recent Advances in Intrusion Detection 2005 (RAID)*, 2005.

[23] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting format string vulnerabilities with type qualifiers," in *Proceedings of the 10th USENIX Security Symposium*, 2001.

[24] M. Sharir and A. Pnueli, *Two approaches to interprocedural data flow analysis*. Prentice-Hall, 1981, ch. 7.

[25] Stephen Shankland, "Andreessen: PHP succeeding where Java isn't," http://www.zdnet.com.au/news/software/soa/Andreessen_PHP_succeeding_where_Java_isn_t/0,2000061733,39218171,00.htm, 2005.

[26] L. Wall, T. Christiansen, R. L. Schwartz, and S. Potter, *Programming Perl (2nd ed.)*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.

[27] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM Press, 2004, pp. 131–144.

[28] Wikipedia, "Hasse diagram," http://en.wikipedia.org/wiki/Hasse_diagram, 2005.

[29] Y. Xie and A. Aiken, "Static Detection of Security Vulnerabilities in Scripting Languages," http://glide.stanford.edu/yichen/research/sec.ps, 2006.

## APPENDIX

| | |
|---|---|
| base64_decode | mt_rand |
| basename | mysql_num_rows |
| count | nl2br |
| date | phpversion |
| define | rawurlencode |
| dirname | round |
| each | sizeof |
| ereg_replace | split |
| explode | strftime |
| gmdate | stripslashes |
| htmlentities | strip_tags |
| htmlspecialchars | str_replace |
| implode | urldecode |
| intval | urlencode |
| microtime | |

TABLE V
EXPLICITLY MODELED PHP FUNCTIONS.