

# Using Static Program Analysis to Aid Intrusion Detection

Manuel Egele, Martin Szydlowski, Engin Kirda, and Christopher Kruegel

Secure Systems Lab, Technical University Vienna  
{pizzaman,msz,ek,chris}@seclab.tuwien.ac.at

**Abstract.** The Internet, and in particular the world-wide web, have become part of the everyday life of millions of people. With the growth of the web, the demand for on-line services rapidly increased. Today, whole industry branches rely on the Internet to do business. Unfortunately, the success of the web has recently been overshadowed by frequent reports of security breaches. Attackers have discovered that poorly written web applications are the Achilles heel of many organizations. The reason is that these applications are directly available through firewalls and are often developed by programmers who focus on features and tight schedules instead of security.

In previous work, we developed an anomaly-based intrusion detection system that uses learning techniques to identify attacks against web-based applications. That system focuses on the analysis of the request parameters in client queries, but does not take into account any information about the protected web applications themselves. The result are imprecise models that lead to more false positives and false negatives than necessary.

In this paper, we describe a novel static source code analysis approach for PHP that allows us to incorporate information about a web application into the intrusion detection models. The goal is to obtain a *more precise* characterization of web request parameters by analyzing their usage by the program. This allows us to generate more precise intrusion detection models. In particular, our analysis allows us to determine the names of request parameters expected by a program and provides information about their types, structure, or even concrete value sets. Our experimental evaluation demonstrates that the information derived statically from web applications closely characterizes the parameter values observed in real-world traffic.

## 1 Introduction

Intrusion detection systems (IDSs) are used to detect traces of malicious activities targeted against the network and its resources. These systems have traditionally been classified as either *misuse-based* or *anomaly-based*.

Systems that use misuse-based techniques [1–3] contain a number of attack descriptions, or signatures, that are matched against a stream of audit data to discover evidence that the modeled attacks are occurring. These systems are

usually efficient and generate few erroneous detections, called false positives. The main disadvantage of misuse-based techniques is that they can only detect those attacks that have been modeled. That is, they cannot detect intrusions for which they do not have a signature (i.e., they cannot identify unknown attacks).

Anomaly-based techniques [4–6] follow an approach that is complementary to misuse detection. The detection is based on models of normal user or application behavior, called profiles. Any deviation from an established profile is interpreted as being associated with an attack. The main advantage of anomaly-based techniques is the ability to identify previously unknown attacks. By defining an expected, normal state, any abnormal behavior can be detected, whether it is part of the threat model or not. Unfortunately, the downside of being able to detect previously unknown attacks, is a large number of false positives.

Profiles that describe legitimate program behavior or input can be constructed following one of two approaches. On one hand, the IDS can rely on *a priori* knowledge about the application and its inputs to define specifications that encode legitimate behavior. A problem of such specification-based systems [5, 7–9] is that they exhibit a limited capability for generalizing from the specification. That is, these systems are typically tailored to a particular application. Additional disadvantages of hand-written, specification-based models are the need for human interaction during the training phase and the effort to define a comprehensive specification.

Learning-based approaches are complementary to specification-based techniques and do not rely on any *a priori* assumptions about the applications. Instead, profiles are built by analyzing program traces or input collected during normal program execution. More precisely, a learning-based system has to complete a training phase during which the protected application and its interaction with the environment is monitored. The observed behavior is considered legitimate and captured by appropriate models. Learning-based systems dispose of the appealing property that they can establish profiles of normal behavior in a quick and automated fashion. Thus, it is possible to deploy the IDS for a broad range of applications without the prior need to gain an in-depth understanding of each application’s functionality. The main drawback compared to specification-based techniques is that profiles are often not as precise. This is due to the fact that the legitimate traces observed during the training phase rarely cover the full range of possible application behavior.

In previous work [6], we developed an intrusion detection system that uses anomaly detection techniques to identify attacks against web-based applications. To this end, the system first analyzes client queries that reference server-side programs and then creates models for a wide-range of different features of these queries. Our IDS is following a learning-based approach. That is, the system derives automatically the parameter profiles for different web applications by monitoring their interaction with clients. More precisely, the system observes legitimate web requests and extracts features for all parameters that are used as part of these requests. The assumption is that whenever an attacker attempts to compromise a web application by sending malicious input through one or

more parameters, this malicious input changes some property of the involved parameters and thus, can be detected by the IDS. Clearly, the quality of the detection depends on the quality of the models and their ability to accurately characterize that input that is expected by the web application.

Our original system focuses solely on the monitoring of request parameters and treats each application as a black box that is not taken into account when building models. In this paper, we examine the possibility to incorporate information extracted from the web applications into the model generation process. The key observation is that the web application receives the request parameters as input that is then processed. By analyzing how input is processed by an application, one can draw valuable conclusions about the type and possible values of data that is expected in certain parameters. This information is then used to build more precise models of the input.

We perform light-weight static program analysis to determine how input parameters are handled by an application. In a first step, the type (e.g., integer, boolean, string) of input parameters is inferred. Then, data flow analysis is used to track the use of input parameters in comparison statements or as arguments to sanitization routines. This allows us to determine constraints on parameters (e.g., a parameter must be an integer larger than zero, or a string is not allowed to contain single quotes) or even a set of concrete values that a parameter must hold. A drawback of source code analysis is that one has to select a particular programming language (or languages) that are supported by the analysis. For this work, we decided to work with PHP [10] programs. The reason to choose PHP was that our IDS is aimed at detecting attacks against web applications and PHP is arguably the most popular programming language to create such applications. Note, however, that the idea of extracting information from programs to improve models of their input is independent of the actual programming language used and most concepts can easily be applied to other languages.

The key contributions of this paper are as follows:

- We describe a static source code analysis approach for PHP that allows us to determine the names of request parameters expected by a web application and the exact locations within the program code where they are used.
- We introduce a type inference mechanism and a light-weight data flow analysis to track the use of request parameters in comparison statements and as function arguments. This allows us to identify the type of input parameters or even provides precise expressions (such as regular expressions or sets of concrete values) to characterize parameter values, leading to more precise intrusion detection models.
- We present the results of our experimental evaluation that demonstrate that our techniques closely capture the types and possible values of parameter values observed in real-world traffic.

The paper is structured as follows. In Section 2, we discuss related work. Section 3 provides an overview of our proposed technique, while Section 4 discusses the details. In Section 5, we summarize our experiences with our tool when analyzing real-world PHP applications. Finally, Section 6 briefly concludes.

## 2 Related Work

A large variety of learning-based anomaly detection techniques have been proposed to analyze different event streams. Examples include data mining on network traffic [11], statistical analysis of audit records [12], or monitoring of system call sequences during program run-time [13, 14]. Also, static program analysis techniques have been extensively applied to solve security-related problems, typically for finding bugs and identifying security vulnerabilities. This includes traditional data flow analysis [15–18], model checking [19, 20], or meta-compilation approaches [21, 22].

An important area in which static analysis was previously employed to build more precise anomaly detection models is the monitoring of system call sequences. The first anomaly detection approach [13] used a training phase to learn legitimate system call sequences collected during normal execution traces. This system was improved in [23], where the authors introduced a system that performs static analysis of the application’s source code to extract a model that captures all possible system call sequences that the program can issue. Thus, any deviation observed during run-time is guaranteed to be an attack. The proposed model is realized as a pushdown automaton (PDA) extracted from the control-flow graph of the application. Unfortunately, the run-time operation of this pushdown automaton is prohibitively high for some programs, reaching several tens of minutes per transaction. The major contributing factor to the time and space complexity of the PDA approach was attributed to its severe non-determinism. This problem was addressed in [24], using several optimizations (e.g., the insertion of “null” system calls), and later in [25], where a Dyck model is used to eliminate the non-determinism associated with stack transitions. As a result, a context-sensitive model equivalent to the PDA automaton can be efficiently implemented. A very similar approach, which uses source code analysis, was introduced in [26]. In this work, system call inlining and “notify” calls are used to remove non-determinism. Another system, which is based on a previous gray-box technique [27], uses static analysis to extract an automaton with call stack information [28].

The differences of system-call-based techniques when compared to our approach are twofold. First, previous systems use control flow information, while our system is based on data flow analysis and type inference. The second difference is that system-call-based techniques generate models that directly capture program behavior. We, on the other hand, use static analysis as a means to improve the models that characterize the *input* of monitored applications.

## 3 System Overview

In this section, we briefly explain the goal of our project and the modus operandi of the analysis tool we developed. Then, the following sections explain in detail the different techniques we used to extract meaningful information from PHP source code using static analysis.

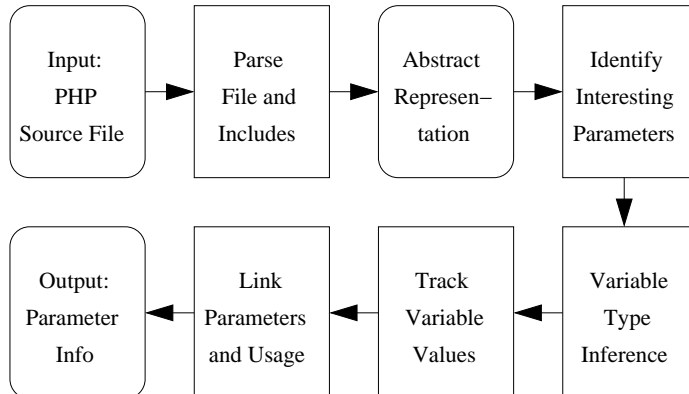
As mentioned previously, PHP [10] is arguably the most popular programming language to create dynamic web sites. One of the designers' motivations to create PHP was to design a programming language that is easier to learn and to use than Perl, while retaining its flexibility. Although PHP has a stand-alone interpreter, its main use is to provide dynamic web contents through either the CGI interface or extensions to web servers (e.g., `mod_php` for Apache). In this paper, we focus in particular on the use of PHP as an implementation language for web applications.

Unfortunately, the ease of use and the popularity of PHP lead to many applications that were created by developers who have little know-how of programming. Furthermore, these developers are often unaware of security issues. This *ad-hoc* web site development often results in applications that contain security flaws. Hence, many PHP-based web applications exist that are vulnerable to attacks such as SQL injection and cross-site scripting (XSS).

The analysis presented in this paper is specific to PHP, however, other programming languages used for the development of web applications (e.g., Python, Perl, or Java) have similar mechanisms of accessing parameters passed by HTTP requests. Since many languages are derived from C/C++, their syntactical constructs also are comparable. Therefore, we do not expect it to be difficult to extend our concepts to these languages. A modular approach is also imaginable, with a parsing module for every language and a common analysis module.

The goal of our analysis is to extract the names, types, and sets of possible values for the parameters that are passed to a PHP web application. The gained knowledge can then be used during the training phase of a learning-based IDS. More precisely, by providing the IDS with knowledge about the types, structures, or even concrete values that can be expected for request parameters, more concise models can be built. This reduces the false negative rate of the system. Moreover, by providing the IDS with information about all the parameter names expected by the application, false positives can be reduced. In particular, a valid parameter that does not appear in the training set is not flagged as anomalous when the IDS knows that the application can process it.

The analysis is performed in two steps (see Figure 1 for an overview of the process). First, the source file is processed using a parser based on the original PHP grammar from the Zend Corporation [10]. During this process, a more convenient, intermediate representation of the PHP file and the files it includes is created in form of an abstract syntax tree. In addition, the discovered variables and functions are stored in hash-tables to ease their retrieval in later steps. For our parser, we decided to use the original grammar provided by the Zend Corporation. Initially, we considered the use of a simplified grammar. However, we soon discovered that this was insufficient to process real-world PHP applications since most language features provided by PHP were frequently used by developers. The main advantage of the original grammar is that we can process almost every valid PHP input file (for a matching, or at least compatible, version of PHP). There are special cases, however, where the original Zend parser handles input outside of the grammar. The parser does not call the flex-generated



**Fig. 1.** Mode of operation

scanner directly but through an intermediate function. This function intercepts certain tokens to handle them separately, returning something different or nothing at all to the parser. One example is the implicit semicolon at the end of PHP input (the `?>` tag). For such input, we had to adapt our parser to mirror the functionality of the one provided by Zend.

The second analysis step uses the abstract syntax tree as a base for the extraction of parameter names as well as variable types and values. Then, connections between the parameters that are passed to a PHP program and the variables that are used within this program are established. Based on these connections and our knowledge of the types and value sets of variables, we can draw conclusions about the structure of the request parameters. To obtain a starting point for the analysis, we need to determine the *locations* within the code where a parameter can “enter” the program. This happens in general through the global `$_GET` and `$_POST` arrays, which hold the names and values of the parameters passed by HTTP GET and HTTP POST requests. However, other ways to access the parameters exist (for a detailed discussion see Section 4.1). With the starting points found, we need to identify which parameters are used. That is, we have to determine the *names* of the parameters that the application expects (Section 4.2). Finally, we try to determine how the values of the parameters are used within the application to extract their *types* (Section 4.3) and possible *value sets* (Section 4.4). Data flow analysis is used to track variables through function calls, expressions, and assignments. The possible values for parameters are in general constants (numbers, strings, boolean) that are found in the source code (and these constants are in some way connected to the parameters). We are also able to observe when parameters are processed by sanitization routines such as `htmlspecialchars`, `urlencode`, `escapeshellcmd` or `preg_match`, which provides insight on what set of possible values a parameter is expected to hold. In the following sections, the different steps of the analysis are discussed in more detail.

## 4 Analysis

### 4.1 Finding Parameter Entry Points

An important goal of the analysis is to identify the names of the CGI parameters that the PHP application expects. To do so, we first have to understand which possibilities a PHP developer has to access these parameters inside her application. That is, we have to find the locations in the code where parameter values can enter the application.

Data that is sent from a client to a PHP application can be transmitted through HTTP GET and HTTP POST requests or cookies. Within a PHP application, this data is accessed through the corresponding superglobal<sup>1</sup>, associative arrays `$_GET`, `$_POST`, and `$_COOKIE`. Additionally, the `$_REQUEST` array holds all parameters contained in the previous three arrays.

The value of a parameter is obtained by indexing the appropriate array with the name of the parameter. This is possible because associative arrays in PHP are very similar to hash tables in other programming languages. That is, they allow an arbitrary string as key for which the corresponding value is returned. In the following example, the value of the parameter `param` is extracted from the GET request.

```
$value = $_GET["param"];
```

Before the `$_{GET, POST}` arrays were introduced with PHP 4.1, alternative mechanisms to access parameters were used. These are still kept for compatibility with legacy applications, although their use is discouraged in the official PHP documentation [10]. One such mechanism is through the global `$HTTP_{POST,GET}_VARS` arrays. The main difference between these arrays and the ones previously mentioned is that `$HTTP_{POST,GET}_VARS` are not superglobal. To access a global variable, which is not superglobal, from within functions and classes, the following two possibilities exist:

- The variable can be explicitly declared to be in the global scope by prefixing its name with `global` at the beginning of the function (example below).

```
1 function foo() {
2     global $HTTP_GET_VARS;
3     ...
4     $value = $HTTP_GET_VARS["param"];
5 }
```

**Listing 1.1.** Use of the `global` keyword

- Since the release of PHP 3.0, through the superglobal `$GLOBALS` array as shown below.

<sup>1</sup> Superglobals in PHP are predefined global variables which are accessible in every scope of the program without the preceding keyword `global`.

```

1 function foo() {
2     ...
3     $value = $GLOBALS["HTTP_GET_VARS"]["param"];
4 }

```

**Listing 1.2.** Use of the `$GLOBALS` array

The most insidious way to access parameters is provided through the *register\_globals* directive, which is a server-side configuration option and defaulted to *on* in all versions of PHP prior to 4.2. This directive automatically promotes request parameters to global variables. For example, the request `GET /mail.php?mailbox=INBOX` would create a variable `$mailbox` with the value `INBOX` that can be accessed from anywhere inside the global scope of `mail.php` and its included files. This creates potentially dangerous situations. Consider the following example. To access sensitive information in the file `secret.php`, authorization is required. This authorization is obtained through some sort of mechanism that sets a global boolean variable `$authorized`. This variable is then queried every time before the sensitive information is displayed. Unfortunately, an attacker could access that information through the simple request `GET /secret.php?authorized=true`. The reason is that this request would create the global variable `$authorized` and set its value to `true`. Now, the protected section of `secret.php` can be entered even if the authorization function fails because of missing credentials.

The bottom line is that using *register\_globals* is risky. However, since this behavior was the default for a long time, many PHP developers are used to it and reluctant to change their existing habits. Furthermore, there are also many legacy application that rely on this feature and were “fixed” to comply with the newer versions by emulating *register\_globals* in software. This is accomplished through using the `import_request_variables` function, available since PHP 4.1.0, or through self-written functions with analogous behavior. The `import_request_variables` function transforms request variables (parameters coming from `GET` or `POST` requests or cookies) into global variables, just as *register\_globals* does. Self-written functions are usually more or less sophisticated variations of the following example, where `$GLOBALS` is the superglobal array holding all global variables. The reason that this works is that global variables can be introduced from within every scope through the `$GLOBALS` array, as shown in line 2 of Listing 1.3.

```

1 foreach ($_GET as $key => $value) {
2     $GLOBALS[$key] = $value;
3 }

```

**Listing 1.3.** Simple variable copying

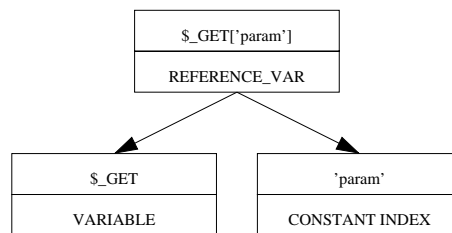
## 4.2 Parameter Name Extraction

To sum up the previous discussion, there are two mechanisms to access request parameters from within PHP:

1. Using the parameter name as an index into a parameter array (e.g., the superglobal `$_GET` array).
2. Using `register_globals` or emulating its behavior.

Our approach handles only the first case. The second possibility, besides being deprecated, brings an unsurmountable obstacle for an automated analysis. The reason is that the names of the parameters are not discernible from regular program variables. Thus, it is impossible to identify parameters that are imported via `register_globals` by looking at the program code alone. To address this problem, one could incorporate information from log files (which contain many valid parameter names), but this is outside the scope of our current analysis.

When considering the first case, the use of a constant parameter name as index into a parameter array is the easiest and most straightforward method to access a parameter in PHP. It also makes finding the parameter name during analysis easy. The names are extracted simply by looking at all interesting reference variables<sup>2</sup> and checking if the index is a constant. A reference variable is considered interesting if it refers to one of the arrays through which parameters can enter the program. For example, the expression `$_GET['param']` is represented in our syntax tree as shown in Figure 2. As can be seen, the name of the extracted parameter is `param`.



**Fig. 2.** Syntax tree for a simple reference variable

Parameter arrays can also be indexed by variables. Our study of real-world PHP applications revealed this to be rather the norm than the exception. Under these circumstances, identifying the correct parameter names within a PHP application is a far more difficult task than simply extracting constant indices. Also, it is common practice, especially in larger PHP applications, to not access these parameters directly where they are used. Instead, the value is retrieved through an intermediate function that takes the name of the parameter as argument. The intermediate function might also perform post-processing before returning the appropriate parameter value to the calling function.

<sup>2</sup> Reference variables are variables which reference an element within an array, e.g., `$a['b']`. The superglobal arrays that store the parameter values are all reference variables.

When dealing with variable indices, we need to employ data flow analysis to determine the possible values of the index variable. In our current system, we use flow-insensitive, inter-procedural data flow analysis to determine possible values of index variables. To determine the value of a variable `$x`, we search *backwards* within the function to find the first assignment statement with `$x` on the left-hand side. When this statement assigns a constant value to `$x`, we have successfully determined its value. This case is shown in the example in Listing 1.4. Here, a constant `param` is first assigned to variable `$x`, which is subsequently used as an index into the `$_GET` array.

```
1  $x = "param";
2  $_GET[$x];
```

**Listing 1.4.** Simple value extraction

Listing 1.5 shows a slightly more complicated case, which is also handled by our analysis. Here, the value of the variable `$y` is not immediately used as an index into the `$_GET` array but through the use of the intermediate variable `$x`. To determine the value of `$x` in this case, we (again) search backwards for the first assignment statement to the variable. This time, however, another variable `$y` is used as the value in the assignment. Thus, we have to continue the backtracking process; this time attempting to identify the value of `$y`. Note that in our current analysis, we only handle constants and variables on the right-hand side of an assignment. When a more complex expression is encountered, the intra-procedural analysis terminates without result.

```
1  $y = "param";
2  $x = $y;
3  $_GET[$x];
```

**Listing 1.5.** Value extraction with intermediate variables

If a variable is identified to be an argument of the enclosing function, the analysis performs an inter-procedural step. To this end, the analysis continues recursively at every call site of this function (that is, at every occurrence of a function call to the function under investigation). For each call site, intra-procedural backtracking analysis is employed to identify all constants that can determine the value of the interesting function argument. This alternation of intra- and inter-procedural analysis steps is then repeated until all relevant values are found.

An example of the interplay between the intra- and inter-procedural analysis steps is shown in Listing 1.6. This example demonstrates how the constant `actionid` is identified to be an index into the `$_GET` array, and thus, a request parameter.

```
1  class Util {
2      function getGet($var, $default = null) {
3          return (isset($_GET[$var]))
4              ? Util::dispelMagicQuotes($_GET[$var])
```

```

5         : $default;
6     }
7
8     function getFormData($arg, $default = null) {
9         return (($val = Util::getPost($arg)) !== null)
10            ? $val
11            : Util::getGet($arg, $default);
12     }
13 }
14
15 $actionID = Util::getFormData('actionid')

```

**Listing 1.6.** Snippet from Horde’s Util class

- First, the parser identifies the use of the `$_GET` (lines 3,4) and flags them as possible parameter entry points. The names of these parameters are undetermined, as `$var` is used as the array index.
- The intra-procedural analysis backtracks and eventually determines that `$var` is an argument of the `getGet` function (line 2). This invokes the inter-procedural step.
- Every call site to `getGet` is examined. In this example, a call is found in `getFormData` (line 11). The argument `$arg` is determined to be the interesting function argument that corresponds to `$var` in the `getGet` function. Again, intra-procedural analysis is invoked, which determines that `$arg` is an argument of the `getFormData` function (line 8).
- All calls to `getFormData` are investigated. In line 15, a call is found, and the constant `actionid` is identified to be the interesting argument. Then, the search terminates as no further calls to `getFormData` are present.

Using the data flow analysis outlined above, we can build a list of parameter names for each file of the PHP application. Note, however, that our flow-insensitive analysis is neither sound nor complete. That is, it might miss certain parameter names. However, the technique works well in practice. In the programs that we examined during the evaluation phase (see Section 5), we were able to detect *all* relevant request parameters, and we expect that our analysis tool is able to perform comparably well with other PHP applications.

### 4.3 Type Inference

The most basic information that we can determine about an input parameter is its type. Knowing a variable’s type allows us to ensure that its value is drawn from the type’s legal value set. For example, we can check that an integer parameter is composed only of number characters and at most one leading dash. Any other value would be flagged as anomalous.

When a parameter is assigned to a variable in the program code, the knowledge of this variable’s type would enable us to draw conclusions about the parameter’s type. In particular, we assume that when a programmer assigns input

to a variable of a certain type, this input is expected to hold a value of the same type. Unfortunately, PHP uses a dynamic type system. That is, no static type qualifiers are used in variable declarations. When variables are used in an operation, their values are cast to the type expected by the operator on the fly. As a result, the type of a variable is not immediately obvious.

To compensate for the lack of static type information, we introduce a type inference process that attempts to identify the types of variables used by the program. Our approach is based on analyzing the operations that are applied to variables. More precisely, type information is gathered by analyzing the types that are possible for the result of an operation. To this end, a *type inference matrix* was generated for each operator. This matrix enables one to determine the type of the result of an operation, given the types of the operands.

Of course, type information is often not available for all source operands, and thus, one cannot immediately retrieve the type of the result from the matrix. However, there are situations when the type of the result can be inferred even without complete knowledge of the operand types. In the easiest case, an operator is encountered that always returns a result of *one* particular type, independent of the types of their arguments (in other words, all entries of the matrix are identical). Here, the type of the variable that receives the result can be immediately identified. For example, the binary logical operators (`&&` `||` `xor`) always return a boolean result, as does the unary not operator (`!`). Another example are the shift operators (`<<` `>>`), which always produce results of type integer. The string concatenation operator (`.`), on the other hand, always produces results of type string. In other situations, even the knowledge of the type of a single operand is sufficient to unambiguously infer the type of the result. This is the case when all entries in the matrix that correspond to the known type of the source operand are identical.

Type information for a certain source operand can also be obtained through other means. One possibility is that an operand is a constant literal in the source code. In this case, the type can be determined statically. Another possibility is the use of a type cast by the programmer to ensure that a variable has a particular type. Finally, type information that has been derived during the analysis process for a particular variable is propagated to all other locations where this variable is used. Thus, whenever the type of a previously undefined variable is identified, all expressions in which this variable appears are revisited. The reason is that the newly derived type information might allow us to resolve the types of other variables.

Deriving the type inference matrices for different operators was complicated by the fact that information on operations' result types is poorly documented in the PHP manual (and sometimes only available by studying the PHP interpreter's source code). For example, the bitwise negation (`~`) fails with an "unsupported operand types" error when used on boolean operands, and automatically rounds floating point operands to the nearest integer. The bitwise logical operators (`&` `|` `^`) always return an integer value, except when both operands are strings, in which case the result has the type string as well. Using an operand that

evaluates to 0 with the modulo operator yields the value *null*, which evaluates to `FALSE` in boolean contexts.

#### 4.4 Value Extraction

After the type of a parameter has been determined, we try to extract sets of possible values this parameter is expected to hold. To this end, we look for string, number, or boolean constants that are compared with this parameter's value. More specifically, we handle three types of comparisons:

1. Direct comparison using the boolean operators `==`, `!=`, `<`, `>`, `...`
2. Indirect comparison through the `switch-case` construct
3. Indirect comparison through sanitization code (e.g., regular expression matching, or built-in functions such as `htmlentities`)

What all the possibilities have in common is the fact that neither the parameter nor the constant that it is compared to have to appear as immediate operands of the comparison operation. The trivial case of such an immediate comparison would look like

```
if ($_GET["param"] == 42)
...
```

where we could immediately add 42 to the list of possible values for the parameter `param`, since the application clearly expects this value and has some mechanism of handling it. Frequently, however, intermediate variables are used, or the values are packed into arrays. Therefore, it is necessary to track the usage of parameters after they have entered the program. To this end, we perform a *forward* reachability analysis to identify those variables that indirectly receive input (i.e., parameter values) through assignment operations. Our analysis is inter-procedural and follows interesting variables into function calls and over return statements. In general, the process is very similar to the backtracking described in Section 4.2, only the direction is reversed.

To see how the forward analysis can be used to extract interesting information about parameters, consider the following (constructed) example (for details on `Util`, refer to Listing 1.6).

```
1 $param = array(
2   "name" => "param",
3   "value" => Util::getFormData("param"),
4   "info" => "something boring");
5 $otherparam = Util::getFormData("otherparam");
6 $thirdparam = do_something($_POST["thirdparam"]);
7
8 $strippedparam = stripslashes($param["value"]);
9 if ($strippedparam == "something")
10 ...
11 switch ($otherparam) {
```

```

12     case "something else":
13         ...
14     }
15     preg_match("/^([0-9]{4}).*", $thirdparam, $number);

```

**Listing 1.7.** Variable tracking examples

From the parameter name extraction (Section 4.2), we already know that the function `Util::getFormData('param')` returns the value of the parameter supplied as argument. Now, we have to determine how this parameter value is used by the program. Therefore, we perform forward tracking to determine those variables that receive the parameter value through assignments, and to examine how these values are used by the program.

Listing 1.7, lines 1-6, has some examples how a parameter value can propagate through the program. The simplest case is shown in line 5, where the parameter value is directly assigned to `$otherparam`. In line 1, the value is inside the `$param` array and can be referenced through `$param["value"]`. Finally, in line 6, the value is used as the first argument of a function (`do_something`). In the last case, the further procedure depends on how much we know about the function `do_something`. If the implementation for this function is part of the application, we can analyze it directly and track the uses of the argument inside the function. Additionally, if the value of the argument is part of the return value, we assume that `$thirdparam` has received the value of the parameter and shall be investigated further. On the other hand, if we do not have the function's code at our disposal, the tracking stops. However, provided that we know more about the function (e.g., by reading its documentation), we can make use of annotations. In this case, we could instruct the analyzer to handle the function in line 6 as if it would simply return its value as argument and continue the tracking with `$thirdparam`.

The next step is to examine the uses of variables that have received program input. We see that `$param["value"]` is used as argument to the PHP built-in function `stripslashes`, and that the result is assigned to `$strippedparam` in line 8. Assume that `stripslashes` is known to return a string that is identical to its function argument, except that all backslashes are removed. This is a typical behavior for a sanitization routine. Then, we can report two things.

1. Backslashes are not desired as part of a value for the parameter `param`.
2. The processed string is assigned to the variable `$strippedparam`, thus, we should examine its uses as well.

In line 9, we note that the value of `$strippedparam` is compared with a string inside an `if`-statement, which leads us to the conclusion that the string is a possible value for `$strippedparam` and, therefore, also for the parameter `param`. The variable `$otherparam` is used in line 11 within a `switch` statement, which is an efficient representation for an `if-elseif` statement. It is compared with every expression after the `case` keyword, so we add all these expressions to the possible values for this parameter. Finally, `$thirdparam` is passed as argument to the (built-in) `preg_match` function. Because the function attempts to match

our variable of interest against a regular expression, we can consider this regular expression as a likely characterization of the parameter.

These three examples illustrate the possibilities that our program has to find interesting uses of request parameters. Experience has shown us that, in most cases, we cover the majority of values that appear in the source code.

## 5 Evaluation

This section is divided into two parts. In Section 5.1, we present the findings of our program when it is run on several real-world PHP web applications. Section 5.2 demonstrates that our findings capture well the real usage of parameters. This is done through the comparison of long-term usage data in log files with our programs results.

### 5.1 Results of Static Analysis

This section presents the results that our program returned on a number of popular PHP web applications.

The first application we examined was the Horde framework (Version 2), which provides a common code-base to its components including libraries and a common user interface, along with its most widely-deployed component - the Internet Messaging Program (IMP Version, 3.1) web mail client. The second choice fell on Squirrelmail, which is another very popular web mail client. Then, the open source bulletin board phpBB (Version 2.0.17) was analyzed, before we turned our attention to a newer version of the Horde/IMP combination (Horde3/IMP4). Finally, we examined PHP iCalendar (Version 2.1), a PHP-based Internet calendaring file viewer to display iCal appointments in a browser.

The results of this analysis are listed in Table 1. In this table, "Parameters found" indicates the number of input parameters that were identified for the given application. Either type or value information about the parameter is considered detailed knowledge, and these sum up to the "Details found" score. The percentage value is simply the fraction of the detailed parameters among all those found. At first glance, the fraction of about 30% of parameters for which detailed information is available appears low. However, one has to take into account that many parameters are treated by the program as opaque data objects that are not processed further. In these cases, no information can be extracted from the code. Also, the provided information is *in addition* to existing models and can be used to improve their precision. In particular, our results showed very precise characterizations for certain parameters that are used directly to influence application logic (and thus, are typically most vulnerable to attacks). For example, we discovered that the `actionID` parameter used in the Horde suite has changed between Horde 2 and Horde 3 in its type and possible value set found by our program. In Version 2, the program returned the following information on `actionID`:

```
actionID: (TYPE_INT):
0,1,101,102,103,104,105,106,107,108,109,110,111,112,113,114, ...
```

whereas, for Version 3, it returned:

```
actionID: (TYPE_STRING):
'add_address', 'add_attachment', 'addchild', 'addchildform', ...
```

This shows that the implementation has changed from using integer values to using more descriptive string representations of the action to perform. Although strings are human readable, the drawback is less precise type information (string instead of integer) that can be used by the IDS. During the analysis of PHP iCalendar, we were able to narrow down the domain for a number of parameters that were checked against a regular expression. The following example provides the technique used by PHP iCalendar to identify date values.

```
getdate (TYPE_STRING:2) Possible Values:
preg_match("/([0-9]{4})([0-9]{2})([0-9]{2})/")
```

Note that, for our experiments, a single annotation for Squirrelmail was necessary. Squirrelmail retrieves parameters via the `sqgetGlobalVar` function, which uses a by-reference argument to return the value of request parameters. Unfortunately, our analysis does not support by-reference arguments, and the annotation was needed to consider this reference argument as the function's return value.

Application	Parameters found	Details found	Percentage
Horde2/IMP3.1	153	47	31%
Squirrelmail 1.4.6-rc1	268	91	34%
phpBB 2.0.17	316	82	26%
Horde3/IMP4.0.2	298	64	21%
PHP iCalendar 2.1	23	15	65%

**Table 1.** Static analysis results

## 5.2 Comparison of Results and Log Files

We gathered log data from live usage of the Horde2/IMP 3.1 and Squirrelmail applications and cross-checked them with the results of our analysis tool. To accomplish this, we compared the set of parameters that are discovered by our tool against the actual parameters stored in the log files. Since only HTTP GET request parameters are logged by the web server, this data is based only on these requests. Note that our analysis discovered many parameters in the application that do not have correspondents in the log files. Nevertheless, the percentage of parameters for which detailed information could be recovered remains roughly unchanged. (Horde: 153 detected, 31% with details; Squirrelmail: 268 detected, 34% with details) This fact gives reason to believe that HTTP POST parameters would be detected with a comparable probability.

**Horde2/IMP 3.1** The log files used for this experiment cover about three months of normal load on a department web server running the Horde2/IMP3.1 combination, which gives about 30,000 hits. Detailed information was extracted

Parameters appearing in log files	37
Parameters appearing in logs found	30 (81%)
Parameters appearing in logs with details found	9 (24%)
Parameters appearing in logs but not found by analysis	7 (21%)

**Table 2.** Horde2/IMP 3.1 comparison

for parameters such as `reason`, which holds a string representation of the reason why a user was logged off the service. We identified the type to be string and the set of possible values was limited to `failed`, `logout` or `session`, and in fact, all occurrences of the parameter `reason` in the log files had exactly one of the before mentioned values.

For parameters such as `to,cc` or `bcc`, the only information that could be derived was that their type is string. However, this is not surprising, as these parameters correspond to their homonymous email header fields which are highly volatile. As shown in Table 2, our analysis failed to detect seven of the parameters that appeared in certain requests recorded in the log files. After examining these parameters and manually studying the source code, we identified all of them as not being used by the program. For example, the parameter `f` appears to be a relict from an older version to provide a filename to the download dialog. In the examined version of Horde, however, this functionality is provided through the extraction of the file’s name from the MIME header. Another example is the parameter `target1`, which holds a copy of the parameter `targetMBox`, but only `targetMBox` is ever read by the application. Finally, one parameter is used by PHP to perform session handling, which is setup by the Horde framework but never used.

**Squirrelmail 1.4.5** About 13,000 hits make up the three weeks of logs for Squirrelmail that were recorded to drive this experiment.

Parameters appearing in log files	26
Parameters appearing in logs found	24 (92%)
Parameters appearing in logs with details found	12 (46%)
Parameters appearing in logs but not found by analysis	2 (7%)

**Table 3.** Squirrelmail 1.4.6-rc1 comparison

A closer look into the program’s output shows that, similar to Horde, no set of possible values can be retrieved for volatile values of search parameters

(**what** or **where**). In a few cases, we were not even able to determine the type of the parameters. This is in contrast to parameters that control the application logic. For example, for the parameter `smaction`, we could identify the type to be string and all the occurrences in the log file have either one of the following values, which we extracted from the program: `draft`, `edit_as_new`, `forward`, `forward_as_attachment`, `reply` or `reply_all`. The two parameters we did not discover in the source code, but which appeared in the log files, are used for hyper-text references requested by a client, but not used by the program.

Our results demonstrate that it is possible to improve intrusion detection by providing *a priori* information about request parameters such as their types or sets of concrete values. In particular, we can improve a number of IDS models presented in previous work [6].

We were able to identify all parameters that are used by the programs under examination. The *parameter presence and absence model* can use this information directly, instead of by learning, where we have no guarantee that all parameters will occur during the training phase. This knowledge alone can help to prevent attacks. For example, we ran our program on phpBB2 (Version 2.0.17), which suffered a mass defacement attack in December 2005. Analyzing the request that contains the exploit<sup>3</sup>, our system observed that a parameter was used that was not reported as an expected parameter for the `profile.php` file. Thus, the attacker's request can be appropriately flagged as anomalous. When considering each parameter that cannot be derived from the program code as potentially malicious, we would have generated nine false positives for the two applications evaluated above (seven for `Horde`, two for `Squirrelmail`). However, given that we analyzed traffic over a period of three weeks, this increase in false positives is very reasonable.

For those parameters for which detailed information was available, the *structural inference* and the *token finder models* can be improved. More precisely, when the type of a parameter is known, we load the *structural inference model* with the appropriate regular expression (e.g., `[0-9][0-9]*` for integer). For our dataset, preparing the *structural inference model* did not lead to the generation of additional false positives.

When our analysis is able to extract a set of concrete values for a parameter, this set is used as input for the *token finder model*. Again, our experiments showed no increase in false positives. That is, our analysis extracted a superset of those parameter values that appeared in the log files. Summing up, the information gathered by our analysis provides better, more accurate models for an existing IDS. This improves the detection rate of actual attacks, but possibly at the cost of more false positives. However, our experimental evaluation shows that the increase in false positives was very moderate for the analyzed data set.

---

<sup>3</sup> The exploit uses requests of the form `profile.php?GLOBALS[...]` to manipulate the contents of the `GLOBALS` array.

## 6 Conclusions

Web applications are prime targets for attackers because they are typically directly available through firewalls and frequently contain vulnerabilities. To mitigate attacks against web applications, we previously developed an anomaly-based intrusion detection system that uses learning techniques to identify attacks [6]. The main problem with this black-box approach is that no information from the web application itself is taken into account.

In this paper, we presented a static analysis technique to extract information from web applications written in PHP. The goal is to determine a more precise characterization of web request parameters by analyzing their use by the program. To this end, we first determine the names of request parameters and their locations in the program. Based on this information, we attempt to identify constraints on the parameters, such as those expressed by the use of the parameter in comparison operations, sanitization routines, or regular expressions.

We tested our prototype implementation on a number of popular, real-world PHP web applications. Our findings demonstrate that using static program analysis on web applications to improve IDS precision is viable. Our tool was capable to retrieve all request parameters that are processed by the analyzed applications and provided detailed information for about a third of these parameters. Using our tool, a mass defacement attack on phpBB2 (Version 2.0.17), launched in December 2005, could have been prevented simply by determining that an unexpected parameter was supplied to the program.

## 7 Acknowledgments

This work has been supported by the Austrian Science Foundation (FWF) under grant P18368-N04.

## References

1. Paxson, V.: Bro: A System for Detecting Network Intruders in Real-Time. In: Usenix Security Symposium. (1998)
2. Lindqvist, U., Porras, P.: Detecting Computer and Network Misuse with the Production-Based Expert System Toolset (P-BEST). In: IEEE Symposium on Security and Privacy. (1999)
3. Vigna, G., Valeur, F., Kemmerer, R.: Designing and Implementing a Family of IDSs. In: 9th European Software Engineering Conference. (2003)
4. Denning, D.: An Intrusion Detection Model. IEEE Transactions on Software Engineering **13**(2) (1987)
5. Ko, C., Ruschitzka, M., Levitt, K.: Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. In: IEEE Symposium on Security and Privacy. (1997)
6. Kruegel, C., Vigna, G.: . In: 10th ACM Conference on Computer and Communications Security (CCS). (2003)

7. Goldberg, I., Wagner, D., Thomas, R., Brewer, E.: A Secure Environment for Untrusted Helper Applications. In: Usenix Security Symposium. (1996)
8. Provos, N.: Improving Host Security with System Call Policies. In: Usenix Security Symposium. (2003)
9. Chari, S., Cheng, P.: BlueBoX: A Policy-driven, Host-Based IDS. In: Symposium on Network and Distributed System Security (NDSS). (2002)
10. Zend Corporation: PHP: Hypertext Preprocessor. <http://www.php.net/> (2006)
11. Lee, W., Stolfo, S., Mok, K.: Mining in a Data-flow Environment: Experience in Network Intrusion Detection. In: ACM International Conference on Knowledge Discovery & Data Mining (KDD). (1999)
12. Javitz, H., Valdes, A.: The SRI IDIES Statistical Anomaly Detector. In: IEEE Symposium on Security and Privacy. (1991)
13. Forrest, S., Hofmeyr, S., Somayaji, A., Longstaff, T.: A Sense of Self for Unix Processes. In: IEEE Symposium on Security and Privacy. (1996)
14. Warrender, C., Forrest, S., Pearlmutter, B.: Detecting Intrusions Using System Calls: Alternative Data Models. In: IEEE Symposium on Security and Privacy. (1999)
15. Ganapathy, V., Jha, S., Chandler, D., Melski, D., Vitek, D.: Buffer overrun detection using linear programming and static analysis. In: ACM Conference on Computer and Communications Security (CCS). (2003)
16. Larochelle, D., Evans, D.: Statically Detecting Likely Buffer Overflow Vulnerabilities. In: Usenix Security Symposium. (2001)
17. Wagner, D., Foster, J., Brewer, E., Aiken, A.: A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In: Network and Distributed System Security (NDSS). (2000)
18. Wagner, D., Dean, D.: Intrusion Detection via Static Analysis. In: IEEE Symposium on Security and Privacy. (2001)
19. Chen, H., Dean, D., Wagner, D.: Model Checking One Million Lines of C Code. In: Network and Distributed System Security (NDSS). (2004)
20. Chen, H., Wagner, D.: MOPS: An infrastructure for examining security properties of software. In: ACM Conference on Computer and Communications Security (CCS). (2002)
21. Ashcraft, K., Engler, D.: Using Programmer-Written Compiler Extensions to Catch Security Holes. In: IEEE Symposium on Security and Privacy. (2002)
22. Engler, D., Chen, D., Hallem, S., Chou, A., Chelf, B.: Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In: ACM Symposium on Operating Systems Principles. (2001)
23. Wagner, D., Dean, D.: Intrusion Detection via Static Analysis. In: IEEE Symposium on Security and Privacy. (2001)
24. Giffin, J., Jha, S., Miller, B.: Detecting Manipulated Remote Call Streams. In: Usenix Security Symposium. (2002)
25. Giffin, J., Jha, S., Miller, B.: Efficient context-sensitive intrusion detection. In: Network and Distributed System Security Symposium (NDSS). (2004)
26. Lam, L., Chiueh, T.: Automatic Extraction of Accurate Application-Specific Sandboxing Policy. In: Symposium on Recent Advances in Intrusion Detection (RAID). (2004)
27. Feng, H., Kolesnikov, O., Fogla, P., Lee, W., Gong, W.: Anomaly Detection using Call Stack Information. In: IEEE Symposium on Security and Privacy. (2003)
28. Feng, H., Giffin, J., Huang, Y., Jha, S., Lee, W., Miller, B.: Formalizing Sensitivity in Static Analysis for Intrusion Detection. In: IEEE Symposium on Security and Privacy. (2004)