Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper)

Nenad Jovanovic, Christopher Kruegel, and Engin Kirda
Technical University of Vienna
Secure Systems Lab
{enji,chris,ek}@seclab.tuwien.ac.at

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.

1. 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 [14], 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, 21], and Perl's taint mode 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 [20] code by means of data flow analysis. We chose PHP as target language since it is widely used for designing Web applications [23], 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 2). 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.

- 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.

2. Taint-Style Vulnerabilities

The presented work is targeted at the detection of taint-style vulnerabilities. *Tainted* data denotes data that originates from possibly malicious users and that can possibly 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].

2.1. Cross-Site Scripting (XSS)

One of the main purposes of XSS attacks is to steal the credentials (e.g., the cookie) of an authenticated user by means of malicious JavaScript code. Due to 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 that have access to the desired cookies. More details on this attack can be found in [4]. 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.
- Sanitation 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().

2.2. 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.

3. 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. 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*[1] 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. Hence, we first construct a parse tree of the PHP input file using the Java lexical analyzer JFlex [12] and the Java parser Cup [5]. The parse tree is then transformed into a linearized form resembling three-address code [1], and kept as a control flow graph for each encountered function.

A straightforward approach to solving the problem of detecting taint-style vulnerabilities would be to immediately conduct a *taint analysis* on the intermediate three-address code 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, we also perform an *alias analysis* for providing information about alias relationships. 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. This task is performed by *literal analysis*. Currently, we use the information collected by literal analysis to evaluate branch conditions and ignore program paths that cannot be executed at runtime (a technique called *path pruning*). Other potential uses of literals information would be the resolution of non-literal include statements, variable variables, variable array indices, and variable function calls.

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 analy-

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

sis 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 an unpublished paper by Xie and Aiken [25], 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. More details on these and other aspects of the applied analyses can be found in our full-length technical report [13].

3.1. 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. 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.

4. 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 [13]. 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 1 and 2 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.

4.1. 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 our technical report [13]. False positives arising from harmless but unmodeled functions can easily be eliminated by providing a specification of the function's true behavior. In Figure 1, a simplified version of the vulnerable code shows that the second parameter of function postcomment is echoed on Line 3. Originally, this warning was issued because the function urldecode 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 2 shows a simplified version of the file preview_static_cgi.php in Simple PHP Blog. The sensitive sink

```
function postcomment($id, $title) {
   $title = urldecode($title);
   echo $title;
}
```

Figure 1. PhpNuke vulnerability (simplified).

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

Figure 2. Simple PHP Blog vulnerability (simplified).

on Line 7 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 3), 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.

4.2. 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 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.

5. Related Work

Currently, there exist only few approaches that deal with static detection of web application vulnerabilities. Huang et al. [10] 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 [11]. 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 3.1. 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 [25] addresses the problem of statically detecting SQL injection

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 1. 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 2. Unknown vulnerabilities discovered by Pixy.

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 [24] 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. [22] to detect format string vulnerabilities in C code. However, it remains questionable whether this technique can be applied to untyped scripting languages.

6. Conclusions

Web applications have become a popular and widespread 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.

7. 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*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 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*, 2001.

[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*, 1999.

[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*, 2003.

[10] 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*, 2004.

[11] 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.

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

[13] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting XSS vulnerabilities. http://www.seclab.tuwien.ac.at/projects/pixy/, 2006.

[14] 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)*.

[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.

[16] Y. Minamide. Static approximation of dynamically generated web pages. In *WWW '05: Proceedings of the 14th International Conference on World Wide Web*, 2005.

[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*. Springer-Verlag New York, Inc., 1999.

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

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

[22] 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.

[23] Stephen Shankland. Andreessen: PHP succeeding where Java isn't. http://www.zdnet.com.au, 2005.

[24] 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*, 2004.

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