A hybrid analysis framework for detecting web application vulnerabilities[*]

Mattia Monga, Roberto Paleari, Emanuele Passerini
Università degli Studi di Milano
Milano, Italy
{monga,roberto,ema}@security.dico.unimi.it

Abstract

Increasingly, web applications handle sensitive data and interface with critical back-end components, but are often written by poorly experienced programmers with low security skills. The majority of vulnerabilities that affect web applications can be ascribed to the lack of proper validation of user's input, before it is used as argument of an output function. Several program analysis techniques were proposed to automatically spot these vulnerabilities. One particularly effective is dynamic taint analysis. Unfortunately, this approach introduces a significant run-time penalty.

In this paper, we present a hybrid analysis framework that blends together the strengths of static and dynamic approaches for the detection of vulnerabilities in web applications: a static analysis, performed just once, is used to reduce the run-time overhead of the dynamic monitoring phase.

We designed and implemented a tool, called Phan, *that is able to statically analyze PHP bytecode searching for dangerous code statements; then, only these statements are monitored during the dynamic analysis phase.*

1 Introduction

A *web application* is an application developed by adopting the web paradigm. Computation is performed via a client-server model, where the client is a web browser, the server is a web server augmented by some extension modules which enable the execution of server side code, and the communication between client and server relies on the HTTP protocol.

Today web applications are employed in a wide variety of different contexts. Common examples of web applications include web mails, forums, blogs, social networking websites, online stores, and so on. Unfortunately, the insecurity of these applications is a well-known problem. According to a recent analysis [17], roughly 60% of the software vulnerabilities annually reported are specific to web applications. Moreover, the majority of these vulnerabilities can be ascribed to the same root cause: the lack of proper validation of user's input. The most common web application attacks that exploit such vulnerabilities are *cross-site scripting* (XSS [4]) and *SQL injection* (SQLI [8]). In the first case, an attacker supplies an input that contains malicious Javascript code, that is later sent to an unaware client without a proper sanitization. Because of an implicit trust relationship between servers and clients, the malicious Javascript code will be interpreted by the client's browser, thus leading to possible privacy violations (e.g., cookies stealing). Similarly, in a SQLI attack user-supplied data is used for building a SQL query string that is sent to a DBMS, without being properly validated against the presence of special characters (e.g., quotes or other SQL tokens) that could alter the semantics of the query, as it was intended by the programmer.

In both these examples, the root cause of the vulnerability is that the application programmer did not correctly validate the user-supplied input. To prevent these security problems, web languages offer native sanitization primitives that a developer can use to validate input data. For example, PHP provides the mysql_real_escape_string() function that can be used to escape SQL special characters inserted in a given string. However, in order to avoid introducing security vulnerabilities in their applications, programmers must be aware of these security problems and properly sanitize *each* user-supplied input before *any* possible use as the argument of an output function. Un-

[*]This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

fortunately, nowadays web applications are often written by developers with low programming and security skills, that sometimes ignore programming good practice. Moreover, most applications are produced by assembling scripts coming from different developers, and it is not always feasible to review all the code base. As web applications are getting more and more complex, it is becoming quite difficult to be able to assert any elementary property about their code.

Several solutions have been proposed that aim at finding automatically security vulnerabilities in web applications [5]. Existing solutions can typically be classified into *static* and *dynamic* approaches. Static analyzers [12, 11, 20, 10] consider the source code without actually executing it; their strength is that they can reason over all possible program paths, but they are often overly conservative, since they normally reports properties weaker than the ones that actually hold in a specific execution. On the other side, dynamic approaches [16, 7] focus on an actual execution of the target application; they consider only a limited number of program paths (i.e., those that have been covered in the observed executions), but they can provide more accurate results. Unfortunately, dynamic tools introduce a significant run-time overhead in the application being analyzed.

In this paper we present a system for analyzing web applications based on a hybrid approach. Our solution blends together the strengths of static and dynamic approaches [6]. It has been implemented in an experimental prototype code named Phan (**P**HP **H**ybrid **An**alyzer). Phan currently targets PHP applications, but can be easily extended to other environments. First, Phan statically analyzes the target application, searching for dangerous statements; afterwards, only those statements that have been found to be dangerous will be monitored dynamically, thus reducing the run-time overhead. It is worth noting that Phan does not work on PHP source code, but directly at the *bytecode level.* In fact, PHP applications are first compiled into a low-level and poorly documented bytecode language that is then interpreted by Zend, the PHP underlying virtual machine. Even if dynamic approaches targeting Zend bytecode already exist [16, 7, 13], this is, to the best of our knowledge, the first time static techniques are directly applied to Zend bytecode.

The contributions of this paper can be summarized as follows:

- we present a hybrid program analysis framework for detecting input-driven security vulnerabilities in web applications. Our solution relies on a static preprocessing technique to reduce the run-time overhead of the subsequent dynamic analyses.
- We describe how we instrumented Zend in order to implement a prototype of Phan, our experimental tool that performs static and dynamic analysis of PHP applications, at the Zend bytecode level.

This paper is structured as follows. In Section 2 we motivate our work and we give a high-level view of a hybrid analysis framework for web applications. Section 3 describes the architecture of Phan, our hybrid analyzer for PHP applications. Section 4 presents some technical details of the experimental prototype we built. In Section 5 we discuss the limitations of our analysis framework, we present some preliminary experimental results and we outline some directions for the future. Finally, in Section 6 we discuss some related work, while Section 7 briefly concludes this paper.

2 Hybrid analysis of web applications

The goal of our approach is to monitor the execution of a given web application and to intercept *injection attacks*, i.e., those attacks that exploit the improper validation of user-supplied input data, before it is used as argument of an output function.

Our approach is made up of two distinct logic steps: (I) a static analysis of the application, that identifies dangerous statements and (II) run-time monitoring of the identified statements.

Initially, we generate a static model of the whole application. The application code is translated into an intermediate form. The rationale behind the choice of our intermediate representation language was to reduce the number of instruction and expression classes, in order to ease the subsequent analyses, while still being able to precisely capture the semantics of the application code.

Then, for each program function, we build its control flow graph (CFG) [2]. Such CFGs are connected together, thus obtaining an interprocedural CFG (iCFG), that is analyzed to individuate all possible code paths from a user input source (e.g., $_GET, $_POST and $_COOKIE arrays in PHP) to a *sensitive sink.* By sensitive sink we mean any function that could lead to security problems when executed over unsanitized user-supplied data (e.g., **mysql_query**() and **echo**(), in PHP).

Finally, from all statements appearing in these paths, we extract only those that might affect the input arguments of a sensitive sink. We do this by calculating, for each sensitive sink, the backward slice [19] over its input arguments. All these statements are marked as "*dangerous*". During the subsequent dynamic phase, only dangerous statements need to be monitored.

The resulting static model is overly conservative, mainly due to limited support for aliasing and class constructs; currently, we address these limitations by extending the number of program statements to be monitored dynamically. In other words, we try to preserve soundness at the cost of greater run-time overhead.

Dangerous statements will be used to perform an efficient dynamic taint analysis, since most of the statements have been filtered out by the static preprocessing. Data that originate or derive from an untrusted source are marked as *tainted*: we start by marking input data as tainted, and then we dynamically keep track of how the tainted attribute propagates to other data. Initially, only a minimal set of program variables are considered to be tainted (e.g., all PHP global arrays that contain user-supplied data). As the execution goes on, other program variables become tainted. When we detect that tainted data containing malicious characters has reached a sensitive sink, we can choose either to block the execution or to sanitize tainted data before allowing the application to continue.

It is worth pointing out that tainted values cannot be sanitized as soon as they are read from input sources; in fact, at this point, we are not sure whether untrusted variables will eventually be sanitized by the application, nor if they will ever reach a sensitive sink. For these reasons, the preemptive sanitization of tainted variables could alter the original semantics of the target application.

On-line monitoring can be very effective, but, in order to minimize the run-time penalty, it should be constrained to a limited number of dangerous statements. However, the identification of these code paths requires a priori knowledge of the structure of the program, that, without the initial off-line phase, would be almost incomplete.

In Figure 1 we report a simple PHP script[1] that checks whether the user has supplied a product ID as a GET parameter of the HTTP request (line 7); in this case, a SQL query is built to extract the information about the specified product from the underlying MySQL database (line 2). This script contains a SQLI vulnerability: since the user input is not properly sanitized, an attacker could manipulate the query submitted to the DBMS (line 4). This example will be used in the following to illustrate our approach.

[1]Phan deals with *Zend bytecode*, however, for the sake of clarity, the example is shown in its source code form.

```
 1  function get_product($id) {
 2    $q = "SELECT ... WHERE id=$id";
 3    mysql_connect(...);
 4    $res = mysql_query($q);
 5  }
 6
 7  if(isset($_GET['product_id'])) {
 8    $a = $_GET['product_id'];
 9    get_product($a);
10  } else {
11    $msg = 'Invalid request';
12    echo $msg;
13  }
```

Figure 1. Sample PHP code fragment with a SQLI vulnerability.

3 Phan: a hybrid analyzer for PHP applications

In this section we describe how the high-level solution introduced in Section 2 can be applied to PHP code. To this end, we present Phan, our hybrid analysis framework for PHP applications. Phan does not require any modification to the source code of the target application, nor any interaction with web application developers. All the static and dynamic analyses performed by Phan are carried out directly on Zend bytecode. We adopted this strategy in order to avoid the intricacies of parsing PHP: the bytecode has ~ 150 opcodes, and it is pretty stable among PHP releases.

Phan is organized into the two following main components:

1. an *off-line analysis engine*, that translates Zend bytecode into an intermediate form, constructs a control flow graph for each program function, merges together the CFGs into a single interprocedural CFG, and finally identifies dangerous program statements;

2. an *on-line monitoring engine*, that performs dynamic taint tracking on Zend bytecode, and reacts properly when tainted user-supplied data reaches a sensitive sink.

Each of these components is described into more details in the following sections.

3.1 Off-line analysis

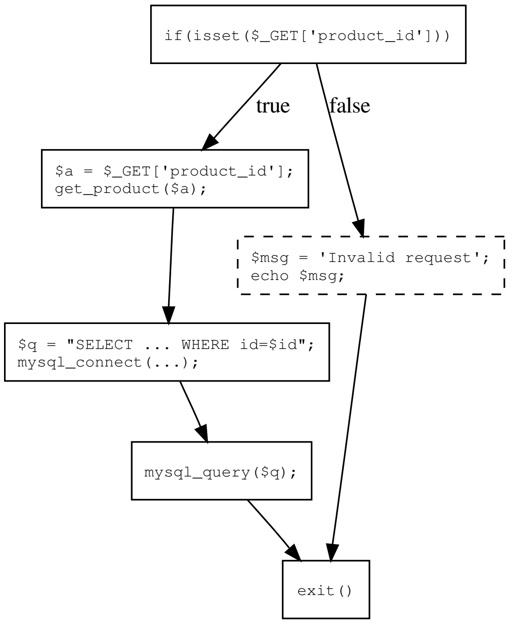
The goal of this phase is to provide a conservative view of the whole application, that will be used to drive the on-line analysis to a limited number of program statements. It is worth pointing out that the off-line analysis has to be done just once for each application script, and has not to be repeated every time a user requires the execution of a PHP script. In this section, we describe the steps involved in the off-line analysis of a single application script. The same steps have to be carried out for each script in the target application.

Translation into intermediate representation. First of all we had to instrument Zend, in order intercept the compilation of PHP scripts. In this way, we are able to obtain a bytecode representation of each application script. To ease the analysis, bytecode instructions are translated into a simple intermediate representation (IR). Our IR resembles a RISC-like assembly language, including just 5 instruction types (`Assignment`, `Call`, `Ret`, `Jump` and `Nop`) and 4 expression types (`Constant`, `Variable`, `Array`, and `CompoundExpression`; the last one is used to model unary, binary and ternary expressions). For each intermediate instruction, we also compute the set of used and defined variables [1]. The translation of Zend bytecode into IR language has required a significant engineering effort: each opcode supported by the Zend virtual machine has to be precisely modeled using our RISC-like language, in order to capture the exact semantics of the application being analyzed. If an application script includes additional modules, each of them is recursively compiled and translated into IR language. In this way, we obtain a complete and self-contained (except for PHP native functions) view of the PHP script.

The example in Figure 1, when compiled by Zend, includes 24 opcodes, and is then translated into 33 intermediate instructions.

CFG construction. We briefly recall that a control flow graph (CFG) is a directed graph $C = (B, E)$, where B is a set of nodes and $E \subseteq B \times B$ is a set of edges [2]. In our context, CFG nodes represent *basic blocks*, i.e., sequences of intermediate instructions with a single entry point and a single exit point. Each graph edge $(b_i, b_j) \in B$ indicates that the execution can flow from basic block b_i to b_j; we say that b_j is a successor of b_i, and b_i is a predecessor of b_j.

Let $S = \{P_1, P_2, \ldots, P_n\}$ be a PHP application script, where each P_i is a program procedure, and P_1 is the "main" procedure of S (i.e., P_1 is the first code sequence that gets executed when S is invoked). We build the CFG of each procedure $P_i \in S$ using standard techniques [1]. We inspect each CFG searching for indirect control transfer instructions. Indirect control transfers are handled using constraint propagation and reaching definition analysis [14]. We have now a set of CFGs $\mathcal{C} = \{C_{P_1}, C_{P_2}, \ldots, C_{P_n}\}$, where C_{P_i} is the control flow graph for program procedure P_i. Then, in order to generate an interprocedural CFG (iCFG) of S, we combine together the CFGs in $\mathcal{C}$ in the following way. For each instruction $i \in S$, if i is a call to a user-defined function P_i, let b_i be the basic block i belongs to, and let b_j be the successor of b_i; then, we remove from the iCFG the control flow edge (b_i, b_j) and we add two edges $(b_i, b_{\text{entry}}), (b_{\text{exit}}, b_j)$, where b_{entry} and b_{exit} are the entry and exit points of C_{P_i}, respectively. Similarly, if i is an inclusion statement (i.e., **include**(), **include_once**(), **require**(), or **require_once**()) that includes the PHP script S', then we replace the control flow edge (b_i, b_j) with two edges $(b_i, b'_{\text{entry}}), (b'_{\text{exit}}, b_j)$, where b'_{entry} is the entry point of $C_{P'_1}$ and b'_{exit} is its exit point.



if(isset($_GET['product_id']))
true
false
$a = $_GET['product_id'];
get_product($a);
$msg = 'Invalid request';
echo $msg;
$q = "SELECT ... WHERE id=$id";
mysql_connect(...);
mysql_query($q);
exit()

Figure 2. Interprocedural control flow graph for the PHP code fragment in Figure 1.

In Figure 2 we show the interprocedural CFG of the example described in Figure 1. To build the iCFG, we merged together the CFG of the "main" procedure with the CFG of get_product(): the basic block containing the function call to get_product() is connected with the entry point of the called procedure.

Identification of dangerous statements. The off-line analysis terminates with the identification of dangerous code statements. We analyze the iCFG and we identify all input sources and sensitive sinks. Input sources correspond to those PHP superglobal variables[2] that allow an application developer to read user-supplied data (e.g., \$_GET, \$_POST, \$_COOKIE and \$_REQUEST). In order to prevent second-order injection attacks, we should also consider the output arguments of functions that read data from a database or the filesystem as sensitive data sources. However, not all data coming from these sources was originally supplied by the user. For this reason, we excluded this feature from our current implementation, as we have not investigated yet how to handle the false positives that could arise from this design choice.

Sensitive sinks correspond to those functions that might send malicious data back to the user (e.g., **echo**() and **print**()) or to the underlying DBMS (e.g., **mysql_query**()). Through the application of standard data-flow analyses on the iCFG, we are able to ignore those sink function calls that are guaranteed to receive as input only constant arguments. Then, we use a graph traversal algorithm to identify all possible code paths from an input source to a sensitive sink. Dangerous statements are identified by extracting from these paths those statements that might affect an input argument of a sensitive sink. Let i denote a sensitive sink, and let W be the set of program variables that represent the input arguments of i. We identify dangerous statements by computing over the iCFG the backward slice for the slicing criterion (i, W). We first compute the set $H = \bigcup_{w \in W} srd(w, i)$, where $srd(w, i)$ represents the static reaching definitions for variable w at program point i. Then, as described in [9], we can reduce our problem to the computation of the set L of program statements that are reachable in the data dependence graph of the analyzed program, starting from a statement in H. Dangerous statements are all those statements included in L that also appear in a code path that connects an input source with a sensitive sink.

In the example reported in Figure 1 the only input source is represented by the \$_GET array, used at lines 7 and 8. We have two different sensitive functions: **mysql_query**() (line 4) and **echo**() (line 12); however, constant propagation analysis reveals that the only input argument of **echo**() is a constant value, thus this function is not considered as a sentive sink. In Figure 2 we depict with a solid border the basic blocks that appear in a code path that connects an input source with a sensitive sink. The backward slice for the slicing criterion $(4, \{\$q\})$ includes only source lines $\{8, 9, 2, 4\}$; these are the dangerous statements whose corresponding Zend opcodes will be monitored in the on-line phase.

3.2 On-line analysis

During the on-line analysis phase we perform a dynamic taint analysis on Zend bytecode. Initially, only the input sources are marked as tainted. We modified the Zend virtual machine to guarantee the correct propagation of taint information during program execution; only dangerous code statements are dynamically monitored. When a function corresponding to a sensitive sink is invoked over tainted malicious characters, we can choose either to abort the execution or to sanitize the input before allowing the function to continue.

Taint meta-information. We modified the Zend virtual machine to keep track of the taint meta-information connected to string variables. Zend associates to each variable x a **zval** structure, updated during the execution to reflect the current value of x. We augmented the **zval** structure by including taint meta-information. In particular a list of (`index`, `labels`) pairs is associated to each string variable, where `index` denotes a specific string character, while `labels` is a bit vector that specifies which taint labels are associated to that element. Taint labels allow us to precisely track which input sources affect a tainted program variable. In our architecture, taint meta-information is protected from unauthorized modifications by the isolation provided by the Zend virtual machine: as long as an attacker cannot tamper the virtual machine, taint meta-information cannot be altered.

Propagation of taint meta-information. To ensure the correct propagation of taint meta-information, we had to modify the implementation of string-related functions inside the Zend virtual machine. We also instrumented Zend's internal functions that manipulate **zval** operands, propagating taint information from source operands to destination.

Phan is able to perform fine-grained tracking of tainted meta-information: as taint propagation is performed with character-level granularity, we can precisely handle also program statements that directly manipulate strings as character arrays.

Detection of injection attacks. When program execution reaches a sensitive sink, we check whether the sink function is going to be invoked over tainted input

[2]PHP superglobals are built-in global variables that are always available in all scopes.

arguments. To each sink function, we associate an "oracle" procedure that determines if a particular tainted string exploits the vulnerability associated to that specific sink. In order to detect exploitation attempts, our current implementation of the oracle functions leverages well-known attack techniques. As an example, the oracle associated to the **mysql_query**() procedure performs limited syntactical analysis of the SQL query that is going to be sent to the database, searching for tainted characters in unsafe positions (i.e., we search for tainted characters that could alter the original semantics of the query statement).

4 Implementation details

We have implemented Phan in an experimental prototype that extends PHP 5.2.6. The off-line analysis module consists of $\sim$ 6000 lines of Python code and $\sim$ 1500 lines of C code for interfacing with the Zend virtual machine. The on-line engine consists of $\sim$ 1000 lines of C code.

The off-line analyzer has been realized as a PHP extension module that hooks Zend's compilation routine. After Zend has successfully compiled a source file, the extension sends its bytecode representation to the Python module, that translates it into the intermediate language and performs the analyses described in Section 3.1. The final outcome is the set of dangerous opcode statements that have to be monitored at runtime. Our current implementation is still not complete, as we currently supports 93 out of 150 Zend opcodes.

For performance reasons, the on-line analyzer is entirely written in C. We had to install a limited number of hooks inside the Zend virtual machine, but the majority of the taint propagation code is encapsulated into a self-contained module. By limiting the number of modifications to Zend's source code, we tried to minimize the burden of work required for porting the on-line engine to different versions of PHP.

5 Discussion

Limitations and future work. The current version of Phan has some limitations, that we briefly summarize in this paragraph together with possible directions for future work.

The off-line engine can be significantly improved by integrating a static taint analysis module [11], that could further reduce the number of program statements to be monitored dynamically. Moreover, the current static analysis engine has limited support for aliasing and class constructs. In the current implementation, we address these limitations by dynamically monitoring all those code regions that we are not able to analyze statically. Finally, Phan assumes the output of a sanitization routine to be untainted, without even considering that the sanitization process implemented by the application developer could be incorrect or incomplete. At this end, we could use the approach described in [3] to verify the correctness of the input sanitization process.

Preliminary evaluation. Table 1 presents some preliminary experiments we accomplished over a set of open-source PHP applications. For each application, we report the vulnerability type, a reference to the vulnerability description, the total number of Zend opcodes (i.e., Zend bytecode statements) in the monitored application script, the total number of Zend opcodes that appear along code paths that connect sources to sinks, and the number of dangerous opcodes. In the last column, we report the percentage of dangerous opcode with respect to path opcodes. As path opcodes represent a lower bound to the number of opcodes monitored by a fully dynamic approach, this percentage is a good approximation of the performance gain coming from a hybrid analysis solution. Moreover, we believe the improvements sketched out in the previous paragraph might further decrease the number of dangerous opcodes, and thus reduce the run-time overhead of the dynamic phase.

6 Related work

Existing solutions for the automatic detection of security vulnerabilities in web applications can be classified into two broad categories: *static* and *dynamic* approaches.

Static approaches. Many different approaches have been proposed for statically detecting security vulnerabilities in web applications. Huang *et al.* proposed WebSSARI [10], a lattice-based static analysis algorithm for the intra-procedural analysis of PHP programs. WebSSARI is derived from type systems and typestate, and it does not track the value of string variables; this can lead to a high false positives rate. In [12, 11], Jovanovic *et al.* present Pixy, a static analysis tool that performs flow-sensitive, interprocedural and context-sensitive data-flow analysis on PHP applications. Pixy is quite efficient and precise, with a low false positives rate. Finally, in [18] the author propose a fully automated static technique for detecting SQLI vulnerabilities in PHP programs. Their approach consists in approximating possible queries that the appli-

Application	Type	Reference	Opcodes	Path opcodes	Dangerous opcodes
Clean CMS 1.5	SQLI	CVE-2008-5290	221	104	56 (53.85%)
Goople CMS 1.8.2	SQLI	Bugtraq ID 33135	62	58	17 (29.31%)
MyForum 1.3	SQLI	Bugtraq ID 31926	1102	651	141 (21.66%)
Pizzis CMS 1.5.1	SQLI	Bugtraq ID 33173	91	38	11 (28.95%)
W2B phpGreetCards	XSS	Bugtraq ID 33001	1078	814	221 (27.15%)
WordPress	XSS	CVE-2008-5278	612	26	10 (38.46%)

Table 1. Preliminary evaluation.

cation could submit to the database using context free grammars; then, they track how input coming from the user can influence these grammars. However, both these approaches do not support several features of the PHP language, most notably classes and dynamically generated code. In Phan, we address these limitations with our on-line analysis engine.

Dynamic approaches. Probably, the first approach that employed dynamic techniques for the taint analysis of applications is Perl taint-mode [15]: the interpreter prevents the use of user-supplied data that has not been explicitly sanitized.

The works presented in [16, 13] are very close to our on-line engine. Both solutions protect PHP application against injection attacks using taint analysis, with an average run-time overhead of $\sim 10\%$. Unfortunately, both these works propose a fully dynamic analysis solution; we believe a hybrid approach like the one discussed in this paper can further reduce their run-time overhead. Moreover, the taint propagation performed by CSSE [16] is too coarse-grained, as it is not able to propagate taint meta-information when character-level string operations are performed; Phan offers a more fine-grained taint tracking solution.

7 Conclusions

In this paper, we presented an approach for detecting injection vulnerabilities in web applications through hybrid analysis techniques. Our proposal blends together the strengths of static and dynamic approaches: the preliminary static analysis phase helps reducing the run-time overhead connected with dynamic monitoring. We described the design and implementation of Phan, a hybrid analyzer for PHP applications that works directly at the Zend bytecode level.

The preliminary results indicate that the improvement with respect to a taint analysis entirely dynamic is significant. Thus, we plan to further increase the accuracy of our analysis in order to evaluate our solution in extended examples.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] F. E. Allen. Control Flow Analysis. *SIGPLAN Notices*, 5, 1970.

[3] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[4] CERT. Advisory CA-2000-02: Malicious HTML Tags Embedded in Client Web Requests, 2002.

[5] M. Cova, V. Felmetsger, and G. Vigna. Vulnerability Analysis of Web Applications. In L. Baresi and E. Di Nitto, editors, *Testing and Analysis of Web Services*. Springer, 2007.

[6] M. D. Ernst. Static and Dynamic Analysis: Synergy and Duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, Portland, OR, May 9, 2003.

[7] A. Futoransky, E. Gutesman, and A. Waissbein. A dynamic technique for enhancing the security and privacy of web applications. In *Black Hat USA*, 2007.

[8] W. G. Halfond, J. Viegas, and A. Orso. A Classification of SQL-Injection Attacks and Countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Arlington, VA, USA, March 2006.

[9] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, New York, NY, USA, 1988. ACM Press.

[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 *Proceedings of the 13th international conference on World Wide Web*. ACM, 2004.

[11] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 workshop on*

Programming languages and analysis for security, New York, NY, USA, 2006. ACM.

[12] N. Jovanovic, C. Krügel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2006.

[13] A. Nguyen-tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *In 20th IFIP International Information Security Conference*, 2005.

[14] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[15] Perl documentation. perlsec. `http://perldoc.perl.org/perlsec.html`.

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

[17] Symantec Inc. Symantec internet security threat report: Volume XIII. Technical report, Symantec Inc., apr 2008.

[18] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, 2007. ACM.

[19] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th International Conference on Software engineering*, Piscataway, NJ, USA, 1981. IEEE Press.

[20] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.