Empir Software Eng (2010) 15:556–576
DOI 10.1007/s10664-010-9131-y

An empirical investigation into open source web applications' implementation vulnerabilities

Toan Huynh · James Miller

Published online: 22 May 2010

Editor: Bojan Cukic

Abstract Current web applications have many inherent vulnerabilities; in fact, in 2008, over 63% of all documented vulnerabilities are for web applications. While many approaches have been proposed to address various web application vulnerability issues, there has not been a study to investigate whether these vulnerabilities share any common properties. In this paper, we use an approach similar to the Goal-Question-Metric approach to empirically investigate four questions regarding open source web applications vulnerabilities: What proportion of security vulnerabilities in web applications can be considered as implementation vulnerabilities? Are these vulnerabilities the result of interactions between web applications and external systems? What is the proportion of vulnerable lines of code within a web application? Are implementation vulnerabilities caused by implicit or explicit data flows? The results from the investigation show that implementation vulnerabilities dominate. They are caused through interactions between web applications and external systems. Furthermore, these vulnerabilities only contain explicit data flows, and are limited to relatively small sections of the source code.

Keywords Empirical evaluation · Web applications · Security · Vulnerability · Injection · Classification of vulnerabilities

1 Introduction

The Laws of Vulnerabilities 2.0[1] states that "80% of vulnerability exploits are now available within single digit days after the vulnerability's public release". The 2008 Internet Security Threat Report[2] from Symantec notes that web applications contain 63% of all documented

[1]http://www.qualys.com/research/rnd/vulnlaws/, last accessed August 16, 2009

[2]http://www4.symantec.com/Vrt/wl?tu_id=gCGG123913789453640802, last accessed January 29, 2010

T. Huynh · J. Miller (✉)
Department of Electrical and Computer Engineering, Electrical and Computer Engineering Research Facility, University of Alberta, Edmonton, AB T6G 2V4, Canada
e-mail: jm@ece.ualberta.ca

T. Huynh
e-mail: huynh@ece.ualberta.ca

vulnerabilities. Insecure applications can be extremely costly. For example, ChoicePoint, after exposing 145,000 customer accounts, reported $11.4 million in charges directly related to the incident (Rapid7 2005). Immediately after the incident was disclosed, ChoicePoint's total market capitalization dropped by $720 million. Meanwhile, CardSystems is barred from accepting Visa and American Express cards after compromising 40 million accounts due to a SQL Injection vulnerability. Hence, security is a prominent non-functional requirement for modern web applications.

Web applications have short release cycles and development time (Baskerville and Pries-Heje 2004). Many new features, enhancements and bug fixes are continually added during these cycles. Every change made to the system can introduce new security vulnerabilities. Using an approach similar to the Goal Question Metric approach (Basili et al. 1994), our goal is to help researchers improve the security posture of web applications by performing an empirical analysis of discovered vulnerabilities in 20 web applications to uncover any similarities in this sample.

Given the relative newness of this topic, limited factual or empirical information exists; hence, we principally rely upon our previous experience with, and observations of, web applications. This previous research has led us to construct some tentative questions with regard to the vulnerabilities that exist within a wide cross-section of web applications; these questions are used to achieve the stated goal:

1. What proportion of security vulnerabilities in web applications can be considered as implementation vulnerabilities? The metric we use to answer this question is the percentage of implementation vulnerabilities versus other types for the 20 applications under examination.
2. Are these vulnerabilities the result of interactions between web applications and external systems? The metric we use to answer this question is the percentage of function calls to external systems that exist in the vulnerabilities.
3. What is the proportion of vulnerable LOC within a web application? That is, what is the vulnerability density? The metric we use to answer this question is the number of vulnerable LOC versus the systems' total LOC.
4. Are implementation vulnerabilities caused by implicit or explicit data flows? The metric we use to answer this question is the number of vulnerable code blocks (which are defined in Section 4.4) with implicit data flow and the number of variables assigned from an input.

Given the lack of solid causal theory utilized to derive the questions, we believe that these questions should be viewed as an initial attempt in hypothesis formulation rather than an exercise in hypothesis confirmation or refutation. The remaining sections of this paper are organized as follows. Section 2 introduces the terminology used in this paper. Section 3 explains the survey and its procedure. Section 4 contains the metrics obtained for the four questions. Section 5 provides an overview of current techniques for detecting and eliminating web vulnerabilities. Finally, Section 6 presents our conclusions.

2 Terminology

We define several terms for the reader's convenience:

- **External Systems**—These are systems that the web application depends upon for its operation. For example, a shopping cart web application retrieves its product information from a Database Management System (DBMS), the external system.

```
1. $email = get_input();
2. if ($email != RFC2822) {
3.   print “invalid email address”;
4.   exit;
5. }
6. $sql = “SELECT phone FROM users WHERE email =’”+$email+”’”;
7. $phone = query($sql);
8. print $phone;
```

Fig. 1 Example program

- **EIV**—External Interaction Vulnerabilities. These vulnerabilities allow attackers to use vulnerable web applications as a vessel to transmit malicious code to an external system that can interact with the web application. The malicious code will modify the syntactic content of the information sent to the external application. In other words, EIVs allow attackers to target external systems that interact with the web application, rather than the actual web application itself.

  Popular EIVs include SQL injections and cross-site scripting vulnerabilities. Any vulnerability is classified as an EIV if it has the following properties:

- A malicious input is required to initiate the attack.
- The malicious input is transmitted from the web application to an external system.
- The malicious input does not exploit the web application directly. For example, all buffer overflow vulnerabilities are not be classified as an EIV because they attack the application's input buffer directly without interacting with an external system.

- **SQL Injection Vulnerabilities** (Scambray et al. 2006)—These vulnerabilities allow attackers to inject and execute SQL statements through the web application. For example, Fig. 1 displays the pseudocode for a web application that asks the user for an email address stored in a database and displays the phone number associated with that email to the browser.

  Statement 1 retrieves the email address from the input. Statements 2–5 parses the input for a valid email address based on the RFC 2822,[3] which defines the standard format of an email address. Statement 6 builds a dynamic SQL statement based on the input retrieved. Statement 7 then instructs the DBMS to execute the SQL statement. Statement 8 prints the phone number retrieved from the email address entered. RFC 2822 allows many characters to be part of an email address which allow names with single quotes such as "O'Reilly" to be used in an email. Hence the user using a specially crafted address, which meets the specification, such as:

```
hi"' OR 1=1 --"@example.com
```

[3] http://www.ietf.org/rfc/rfc2822.txt, last accessed July 25, 2009

can embed a SQL statement. Using this email address, the expanded SQL statement becomes:

```
SELECT phone FROM users WHERE email ='hi"' OR 1=1 --"@example.com'
```

Hence, the SQL statement is successfully injected.

- **Cross-site Scripting (XSS) Vulnerabilities** (Scambray et al. 2006)—These vulnerabilities allow an attacker to inject JavaScript/HTML code that other visitors to the website will execute. For example, an attacker can create a link to a vulnerable web application, such as

  http://www.site.com/?<script src=http://hacker.com/getcookie.js></script>

  When users click on this link, they are taken to the actual www.site.com website (not a website that the attacker controls), which then allows the attacker to retrieve the users' cookie data for that website.
- **Code Injection Vulnerabilities** (Scambray et al. 2006)—These vulnerabilities allow an attacker to inject and execute programming statements in the same language as the web application. This vulnerability is extremely dangerous as it allows the attacker to become a programmer for the vulnerable application. For example, an attacker named John, can exploit a vulnerable application to write and execute statements such as

  ```
  eval("setUserLevel('John','Admin')");
  ```

  which allows the attacker to become an administrator for that application.
- **Command Execution (Injection) Vulnerabilities** (Scambray et al. 2006)—These vulnerabilities allow an attacker to run various system commands ("`cd`", "`ls`", "`dir`", "`cat`", etc .) through the vulnerable system. An attacker, for example, exploiting this vulnerability can perform DoS (Denial of Service) attacks on the system by removing files essential to the application. Other system commands can be used to retrieve information or even alter the application's configuration settings.
- **Privilege Escalation Vulnerabilities** (Scambray et al. 2006)—These vulnerabilities allow an attacker to bypass the authentication system or escalate their privileges without using an injection attack. A typical vulnerable application would allow an attacker to access restricted sections without being identified as a valid user. For example, a web application can use a flag to identify administrators from normal users. This flag is stored in a hidden form field. The attacker, with knowledge of this flag, can manipulate it and escalate their account to gain additional (administrative) functions.
- **Information Disclosure (Leakage) Vulnerabilities** (Scambray et al. 2006)—These vulnerabilities allow an attacker, without using an injection attack, to access information not available to a normal user. Information disclosure differs from authentication bypass because authentication bypass allows an attacker to perform tasks and retrieve information not available to them; whereas, information disclosure only allows the attacker to retrieve restricted information. For example, instead of displaying a generic error message when encountering an error, the web application can display the entire call stack which contains detailed information on the internal structure of the web application.

3 Survey

For this survey, we examined 20 different applications implemented using six popular languages (PHP, ASP-VBscript, ASP.NET – C#, Java-JSP, Perl, and Python). The survey is

Table 1 Number of vulnerabilities in the OSVDB

Total vulnerabilities	19,173
Products	5,175
Total web related vulnerabilities	7,290
Total web applications	2,695

explicitly limited to web applications; and hence several common languages (such as C) and vulnerability types (such as buffer overflows) are relatively uncommon within this domain (OWASP 2007).

3.1 Vulnerability Databases

We used two popular vulnerability databases (VDB), the Open Source Vulnerability Database[4] (OSVDB) and the Bugtraq mailing list[5] to identify the vulnerabilities for these applications. These two databases provide information on known vulnerabilities for open source and proprietary products. Unfortunately, the survey requires detailed analysis of the source code, which is unavailable for proprietary systems; and hence the investigation is limited to open source systems. Although we cannot perform the complete survey for proprietary systems, we briefly examined the vulnerability types of 20 proprietary systems to determine whether they are similar to the vulnerability types found in open source systems. The results, discussed in Section 4.2.1, show that these proprietary systems have a similar distribution of vulnerability types.

Although the two databases have different maintainers, they are far from independent; in fact, Bugtraq can be viewed as a subset of OSVDB. OSVDB effectively collates information from all of the other major open-source vulnerability databases including: The National (U.S.) Vulnerability Database,[6] US-CERT Vulnerability Notes[7]; Internet Security Systems—X-Force Database[8]; CERIAS Vulnerability Database[9], and the LWN security vulnerabilities database.[10] Hence, OSVDB can be considered as being a meta-source of information on this topic; and therefore, it is utilized as the basis of the selection procedure. Having said this, Bugtraq (due to its message board format) tends to include a more extended description of vulnerabilities than OSVDB, and hence this information source was always used, when it was available, to increase the understanding of the vulnerabilities.

3.2 Survey Procedure

The survey, for purposes of sampling, extracted vulnerability information covering the period between January 1, 2002 to May 31, 2007 from the OSVDB resulting in the records shown in Table 1.

OSVDB requires that all vulnerabilities be inspected to increase accuracy; unfortunately, Bugtraq has no such screening process. The survey worked with the vulnerabilities from OSVDB; the reliability of Bugtraq's vulnerability information was validated by comparing

[4] http://www.osvdb.org/, last accessed July 22, 2009
[5] http://www.securityfocus.com/archive/1, last accessed July 22, 2009
[6] http://nvd.nist.gov/statistics.cfm, last accessed July 31, 2009
[7] http://www.kb.cert.org/vuls/, last accessed July 31, 2009
[8] http://xforce.iss.net/, last accessed July 31, 2009
[9] http://www.cerias.purdue.edu/about/history/coast/projects/vdb.html, last accessed July 31, 2009
[10] http://lwn.net/Vulnerabilities/, last accessed July 31, 2009

it with the corresponding entry from OSVDB. In addition, both databases encourage a product's developers to refute any vulnerabilities that they believe are incorrect, providing a further crosscheck of validity. None of the systems in our survey contained any disputed vulnerability. Our analysis suggests that the quality of data in both systems is extremely high.

Our sampling procedure was to select randomly 20 open source web applications from the OSVDB database. However, these 20 web applications were required to meet certain criteria:

- They must have more than one update released.
- They must be larger than three KLOC.
- They must have vulnerabilities that are exploitable.
- They can be commercial systems, but the source code has to be available.

Table 1 shows that the selected web applications represent only a small fraction of the total number of web applications listed within the database. The results of the sampling process are shown in Table 2. Once the products were selected, the following steps were performed, on each product, to gather the necessary data for the analysis:

1. We downloaded the source code for all applications. This includes downloading older source code that contained the vulnerabilities of interest. Our analysis requires us to trace, in detail, paths through the source code.
2. We used a source code counting tool (Practiline Source Code Line Counter[11]) to count the LOC for each application. Only files containing program statements were counted. The reported LOC does not include empty lines and comments.
3. We retrieved vulnerabilities for the applications from the VDBs.
4. For each vulnerability, we traced the source code to the statements causing the actual vulnerability. Nested function calls are traced and stopped at calls to standard library functions.

Due to the different programming languages involved, different designs associated with each application and over 330 KLOC to examine, the entire process required about 1 year of effort. One week was required to study the OSVDB's relational diagram and to import OSVDB's data into a local database for faster access. One week was used to create a tool to query the database. Twelve weeks were used to study the programming languages. One week was used to install, configure, and deploy the web applications in a test environment. Ten weeks were used to study the web applications and the associated source code; 4 weeks were used to examine all the vulnerabilities associated with each application. Twenty-six weeks were used to independently repeat the manual operations. This "verification" task was believed to be important as any manual task of this "length" is clearly error-prone and this approach is believed to have resolved any inconsistencies in the process.

3.3 Chosen Applications

Table 2 displays the examined applications and the number of vulnerabilities identified.

3.4 Tracing the Source Code

To determine the number of vulnerable LOC and how deep these statements are within the call stack, we traced the source code for each known vulnerability. Program slicing was first introduced by Weiser (1984) as a method of automatically decomposing applications. A

[11] http://sourcecount.com/, last accessed July 29, 2009

Table 2 Applications examined

Application	Description	Vulnerabilities	Language
A-CART	A commercial fully-featured shopping cart developed on the ASP platform using VBScript	8	ASP (VB)
AWStats	A popular open source log file analyzer for web/streaming/ftp/mail servers	5	Perl
Bonsai	An open source web-based querying front-end for CVS from the Mozilla Foundation	8	Perl
BugZilla[a]	An open source bug tracking system from the Mozilla Foundation	25	Perl
BugTracker.NET	A web-based bug tracker system that is currently used by thousands of development teams.	4	ASP.NET (C#)
Calcium	A commercial web calendar system by Brown Bear Software.	1	Perl
Daffodil CRM	A commercial open source customer relationship management system by Daffodil Software Ltd.	1	Java (JSP)
DEV web management system	A content management system for web portals.	5	PHP
FileLister	A file system indexing tool	2	Java (JSP)
JSPWiki	An open source JSP-based WikiWiki engine	1	Java (JSP)
Mantis[b]	An open source tracking system	12	PHP
Neomail	A web-based email system; thousands of servers utilize the system.	1	Perl
PDF Directory	An open source software that generates a printable directory listing for any organization.	12	PHP
phpBB[c]	An open source popular message board system written in PHP that's being used on millions of websites.	23	PHP
ProjectApp	A commercial web-based project and task management system used for team communication by Iatek Corporation.	5	ASP (VB)
osCommerce	An open source e-commerce system, by osCommerce, currently being installed and utilized by 10,942 online stores.	15	PHP
Roundup	A full featured bug tracking system.	4	Python
sBlog	An open source blog system.	2	PHP
SkunkWeb	A robust, open source web application server.	2	Python
ViewVC	A browser interface for CVS and Subversion control repository.	2	Python
Total		138	

[a] Due to the numerous vulnerabilities reports available for BugZilla, we limited the versions of the vulnerable systems to 2.16.0 or higher

[b] Due to the numerous vulnerabilities reports available for Mantis, we limited the versions of the vulnerable systems to 1.0.0a1 or higher

[c] Due to the numerous vulnerabilities reports available for phpBB, we limited the versions of the vulnerable systems to 2.0.7 or higher

slice of a program is a reduced, executable segment of the original program. A slice can be produced dynamically or statically. Static slicing techniques do not require input values whereas dynamic slicing techniques rely on some specific input to produce a slice (Tip 1995). Due to the lack of slicing tools for the languages examined, in this survey, we used a technique similar to dynamic slicing (Agrawal and Horgan 1990; Tip 1995) to produce contamination graphs (CGs) of the systems examined. The CG is not a SDG (system dependency graph), but rather a def-use graph that follows the malicious input from the entry point to the exit point of the system. While the technique used is similar to slicing, it does not produce complete slices of the system (hence, cannot be considered a slicing technique) and the graphs produced by the algorithm do not take into account object-oriented programming features such as inheritance and polymorphism; however, they contain sufficient information for this survey. More formally, a CG is a directed graph $G=<N,E_c,E_d>$, where N is a set of vertices corresponding to statements and control predicates, and E_c and E_d are the set of edges corresponding to the def-use data dependencies. The slicing criterion is C=(v, i, X*), where v is a variable in the system, i is an input value for v and X is a set of statements in the program. For this survey, v and i consist of variables and values that exploit the known vulnerabilities, while X* ($\subseteq$ X) consists of program statements where it is possible to export the vulnerability to an external system; and X is the entire set of statements in the program. The following algorithm is used to produce a CG for each v and i of interest (Fig. 2)

An example of a CG using C = (keyword, “`<script>alert('hello')</script>`”,{`query, echo, print`}) for an application examined, sBlog, is shown in Fig. 3. The source code for this example is approximately 7,800 lines of PHP. Dotted directed edges on this graph represent DEF dependences (definition of a contaminated variable), while the solid edges represent USE

```
1. DEF(w) is a definition of the variable w
2. USE(w) is a use of the variable w
3. Let V be a set of v
4. Let F be a set of statements; F ⊆ X; f_j be the statement at location j.
5. Let curloc be the program's current statement's location
6. Initialize V := {}; F := {}; prevloc := 0; prevDEFloc := 0;
7. Locate the first DEF(v) where v := malicious input
8. G := G + <curloc,{},{}>
9. prevloc := curloc
10.      prevDEFloc := curloc
11.      V := v ∪ V
12.      Execute program until ∃ v∈V such that USE(v)
13.      If DEF(w) := USE(v) then
      a. G:= G + <curloc, prevloc→curloc, prevDEFloc→curloc>
      b. V := w ∪ V
      c. prevDEFloc := curloc
   Else
      a. G := G + <curloc, prevloc→curloc,{}>
14.      prevloc := curloc
15.      If f_curloc ∈ X* then F := f_curloc ∪ F
16.      Go to 12 unless F - X* = {} ∨ curloc = EOF ∨ program encounters an
   error due to a successful exploit.
```

Fig. 2 The algorithm used to generate the CG

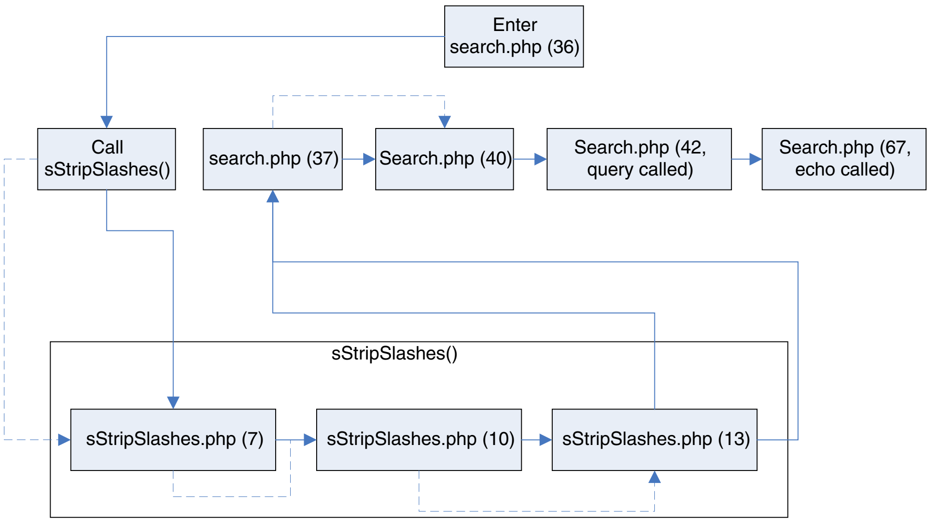


Fig. 3 CG for sBlog

dependences (usage of a contaminated variable). Each node is labeled with the source code's filename and the line where the statement can be found (in parenthesis). If a node represents a function call then it is labeled as "call 'function name'". System calls are also placed within the parenthesis. The graph above shows that the malicious input entered the system at line 36 of the search.php file. The solid edges show the transition between each USE statement. Nine lines of code use the malicious input (number of nodes) with five variables defined based on the malicious input (the number of doted edges).

4 Results

This section contains the results from our survey. These results answer the four questions raised in the introduction and can be used to help our goal which is to improve the security posture of web applications by uncovering similarities between vulnerabilities.

4.1 Question 1

Question: What proportion of security vulnerabilities in web applications can be considered as implementation vulnerabilities?

Metric: The percentage of implementation vulnerabilities versus other types for the 20 applications under examination.

To answer Question 1, we characterized the known vulnerabilities into three categories based on Swidersky and Snyder's categorization (Swiderski and Snyder 2004):

- Architecture vulnerability: A vulnerability that is caused by a design flaw. For example, if the session ID generated by an application is easily guessable because the specification for a secure session management system does not have requirements on how IDs will be generated, such as a specific cryptographically hash routine, then the issue is considered architectural in nature.

Table 3 Vulnerability category distribution

	number of vulnerabilities	% of vulnerabilities found in sample	standard error[a] (%)
Implementation	101	73.2	3.77
Architecture	30	21.7	3.51
Configuration	7	5.1	1.87

[a] In this context, the margin of error in the survey is approximately twice the standard error. Specifically, assuming a 95% confidence level, it is 1.96*the standard error

- Implementation vulnerability: A vulnerability that is the result of an insecure coding practice. Using the same example as above, if the session ID is easily guessable because the cryptographically secure hash routine used to generate session IDs is written incorrectly then the issue is considered implementation in nature.
- Configuration vulnerability: A vulnerability that is caused by an incorrect configuration of the application; hence, if the vulnerability ceases to exist after an application is reconfigured, the vulnerability is classified as a configuration vulnerability. For example, the "register_globals" issue with PHP is considered a configuration vulnerability. This is a setting in the configuration file to instruct PHP to create global variables from the EGPCS (Environment, GET, POST, Cookie, Server) variables. When enabled, attackers can use the feature to define many global variables.

Table 3 shows the vulnerabilities and their distribution within the three categories defined. The standard error in the table is used to show the uncertainty of the value for each category. The equation for the standard error is:

$$\text{standard error} = \sqrt{\frac{p(1-p)}{n}} \tag{1}$$

Where p is the probability of the sample belonging in a certain category and n is the sample size. This assumes that: n is small relative to the population size, the samples are selected from a simple random sampling process, and the sampling distribution of p is the binomial distribution.[12] Each category is treated independently from each other. For example, the first row of the table examines the implementation vulnerability. Hence, p is the probability of a vulnerability being an implementation vulnerability, and 1-p is the probability of it not being an implementation vulnerability.

This table answers Question 1 by showing that implementation vulnerabilities dominate; hence, addressing vulnerabilities within this category would allow a significant reduction in the number of vulnerabilities.

4.2 Question 2

Question: Are these vulnerabilities the result of interactions between web applications and external systems?

Metric: The percentage of function calls to external systems that exist in the vulnerabilities.

[12] Clearly, this is a simplification of the situation. However, the study has insufficient data to allow the evaluation of more complex models.

Usually, these implementation vulnerabilities can be traced through a dynamic string, constructed from an input, being used in a function or method that allows the string to be passed to another system. We begin the answer to Question 2 by examining the types of vulnerabilities within the implementation category. This examination reveals six different types of vulnerabilities are commonly discovered within web applications: SQL Injection, SQL Injection, XSS, Code Injection, Command Execution, Privilege Escalation, and Information Disclosure.

Table 4 displays the vulnerability types discovered during the survey. Close examination reveals that the majority of these types occur due to an interaction with an external system. These types of implementation vulnerabilities, bolded in Table 4, account for 95 of the 101 implementation vulnerabilities. While information disclosure may also be caused due to an interaction between the web application and the file system, this interaction is not obvious from Table 4, and the actual statements causing the vulnerability have to be examined to determine the exact cause.

4.2.1 Vulnerability Types for Proprietary Systems

Since the vulnerability databases used also include proprietary systems, 20 of these systems were selected and examined to provide some level of comparison with the results found in the survey. Like their open source counterparts, these 20 applications were also randomly selected from the OSVDB. Table 5 shows the 20 applications examined.

These 20 applications are commercial applications that either do not have their source code available or they require a developer's license to be purchased before the source code can be obtained. This table shows that ASP and ASP.NET is used for sixteen of the 20 web applications. Two out of 20 applications are powered by ColdFusion, which is the only scripting language that supports source code encryption without additional plug-ins or extensions. The remaining applications are created using Java technology.

Table 6 displays the vulnerabilities encountered for these 20 applications versus the vulnerabilities encountered for the 20 open source systems. This table shows that the top two vulnerabilities encountered on both types of system are XSS and SQL Injection, respectively. Code injection is less frequently encountered in proprietary systems, which can be attributed to the fact that PHP remote file inclusion does not occur in these systems because these 20 systems do not use PHP. "Other" contains vulnerabilities that cannot be classified due to limited information provided for these vulnerabilities.

This table reveals that:

- The two types of systems agree that XSS and SQL injection (in that order) are the most numerous types of vulnerabilities experienced by web applications. Furthermore, the

Table 4 Implementation vulnerability types

	number of vulnerabilities	% vulnerabilities	standard error (%)
XSS	56	55.4	4.23
SQL Injection	30	29.7	3.89
Code Injection	6	5.9	2.01
Command Execution	3	3.0	1.86
Information Disclosure	5	5.0	1.45
Privilege Escalation	1	1.0	0.85

Table 5 Proprietary systems

Application	Description	Vulnerabilities	Language
Active Auction House	A web based auction software designed for online auctions (ex. ubid.com, ebay.com).	7	ASP (VB)
AliveSites Forum	A component (COM) object tool that allow collaboration among members and users of a company or organization though the internet or intranet.	4	ASP[a]
ampleShop	A complete e-commerce system.	4	ColdFusion
AspDotNetStorefront	An ASP.NET shopping cart used by over 5,000 customers.	3	ASP.NET (C# and VB.NET)
ASPRunner	A web-based database management tool that provides administration for many popular databases.	7	ASP
Baseline CMS	A web-based content management system.	2	ASP
Bugzero	A web-based bug tracking, defect tracking, issue tracking, and change management system.	5	Java
Cisco CallManager Web Interface	The web-based interface for the Cisco Unified CallManager system.	3	ASP
couponZONE	A web-based system that provides online e-coupons.	2	ColdFusion
DUPortal Pro	An ASP-based Web Portal application.	11	ASP
E-School Management System	A web-based School Management Software designed to allow easy communication between students, teachers, parents & management.	1	ASP.NET
iCMS	A content management system.	2	ASP
Mall23 eCommerce	An e-commerce solution for Web Development and Hosting companies.	3	ASP
NetAuctionHelp	An ASP-based online auctioning system.	1	ASP
OneWorldStore	An e-commerce system that can be integrated to existing websites.	10	ASP
Revize CMS	A content management system.	5	Java
SCOOP!	Another web content management system for users without HTML knowledge.	7	ASP
SmarterMail	An advanced email and collaboration server.	5	ASP.NET
uStore	A dynamic storefront application for e-commerce websites.	3	ASP
Web Quiz	An easy application that for online test creations and assessments.	2	ASP

[a] ASP and ASP.NET applications can be created using many programming languages. Due to the proprietary nature of the applications, the exact programming language used is unknown

Table 6 Proprietary versus open source

	Proprietary		Open Source	
	% vulnerabilities	Standard Error (%)	% vulnerabilities	Standard Error (%)
XSS	48.8	4.26	55.4	4.23
SQL Injection	36.0	4.09	29.7	3.89
Code Injection	2.3	1.28	5.9	2.01
Command Execution	1.2	2.17	3.0	1.85
Information Disclosure	7.0	0.93	5.0	1.45
Privilege Escalation	1.2	0.93	1.0	0.85
Other	3.5	1.56	0	0

injection type vulnerabilities (SQL, XSS, code, command execution) combined to be the most popular vulnerability for web applications. This suggests that researchers interested in security problems associated with web applications should concentrate their efforts on these types of vulnerabilities. Clearly, this suggestion assumes that all vulnerabilities have a similar (negative) economic value.

- The two types of systems experience code injection problems at differing percentages. However, care needs to be exercised when considering this conclusion given the relatively low volume of these types of defects.

4.2.2 Mapping Vulnerabilities Down to Code Statements

Table 7 displays the statement types that cause the 95 external interaction vulnerabilities (EIVs). Several functions sharing the same properties are grouped into one family. For example, output statements such as `print`, `echo`, and `write` all send data to the browser, and hence they are grouped in the "`print`" family. Statements querying the DBMS such as `executeQuery`, `mysql_query`, `db.execute` are grouped in the "`query`" family.

Table 7 Statement usage

Statement Type	Number of Occurrences	Occurrence Percent (%)
"copy" file	1	1
dir	1	1
eval	11	12
file	1	1
open	2	2
preg_replace	2	2
"print" family	47	49
"query" family	27	29
require	1	1
system	1	1
wrong operator	1	1
Total	95	100

For every vulnerability, a CG was created using the technique discussed in Section 3.4. Statements resulting in the vulnerabilities can be located from these graphs. Table 7 highlights the statements used to call standard library functions which are the majority of the statements (99%). The statements listed in the table have the following behavior:

- "`copy`" file—a function that allows programmers to copy an existing file.
- `dir`—a function that lists all files within a directory.
- `eval`—a function that accepts a string parameter and executes that string as a programming statement.
- `file`—a function that opens and reads a file based on a provided filename.
- `open`—a function that opens a file, pipe, or file descriptor.
- `preg_replace`—a function that will evaluate a provided string as a program statement if a special character is used (PHP only).
- "`print`" family—a group of functions that allows the application to send output to a browser.
- "`query`" family—a group of functions accepts a string containing one or more valid SQL statements and sends it to the underlying DBMS.
- `require` (PHP only)—a function that accepts a string parameter containing a filename (which contains programming statements), reads the file, then evaluates all the programming statements within that file. Similar functions include include and include_once (PHP only).
- `system`—a function that accepts a string parameter containing a system's command, then creates a new process and executes the command.
- typographical error—this is a statement where the programmer used the wrong operator for a conditional branch. For example, instead of using the < operator in an `if` statement the programmer used the <= operator. This operator does not enable an interaction and is the exception to our general rule.

Based on Table 7, the implementation vulnerabilities can be divided into two categories:

1. Interaction with external systems (EIV).
2. Wrong statement usage.

Table 7 shows that 99% of the implementation vulnerabilities are EIVs; this answers Q2. This answer means developers should concentrate on the data flow between the web application and other systems because this is where most of the vulnerabilities occur.

4.3 Question 3

Question: What is the proportion of vulnerable LOC within a web application? That is, what is the vulnerability density?

Metric: The number of vulnerable LOC versus the systems' total LOC.

Alhazmi et al. (2007) have explored the vulnerability density for Operating Systems and discovered that the density is very low. In this paper, we explore the vulnerability density for web applications. We believe implementation vulnerabilities are also limited to relatively small portions of the entire web application. That is, the number of vulnerable LOC is significantly smaller than the total LOC of a web application.

To answer Question 3, we traced each implementation vulnerability using the method outlined in Section 3.4. We generated 101 graphs for the 20 applications. To determine the complexity of the vulnerable code, we examined the number of nodes per graph and contaminated variables per graph. Figures 4 and 5 show that the majority of the graphs have

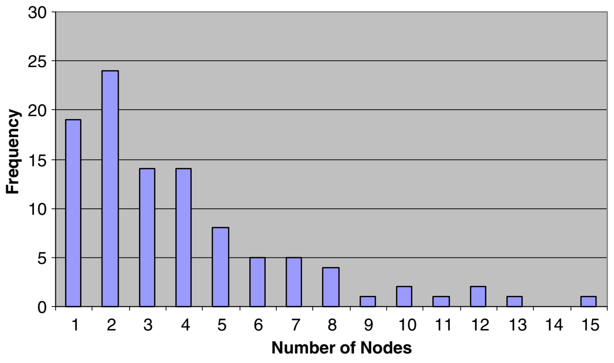


Fig. 4 Histogram of nodes

less than five nodes and four contaminated variables. In fact, 70% of the CGs contain less than five nodes and 93% of the CGs contain three or less contaminated variables. Hence, the majority of the vulnerabilities can be viewed as "small and manageable". In fact, even the largest number of statements and contaminated variables associated with a vulnerability (15 and 12 respectively) is quite small when compared to the overall size of the system.

Once the CGs are obtained, we counted the vulnerable LOC contained within each CG. Table 8 further demonstrates that the number of vulnerable LOC for the known vulnerabilities is significantly smaller than the overall LOC. The results from Figs. 3 to 4 and Table 8 provide the answer to Question 3 which is that vulnerability density is small. Since Figs. 3 and 4 and Table 8 show that the number of vulnerable LOC is small compared to the overall size of the system, it can be beneficial to introduce a solution to solve implementation vulnerabilities by concentrating on the CGs with vulnerable LOC.

4.4 Question 4

Question: Are implementation vulnerabilities caused by implicit or explicit data flows?

Metric: The number of vulnerable code blocks with implicit data flow and the number of variables assigned from an input.

Implicit data flows are information flows via the control structure of the program (Denning and Denning 1997). For example, the statement "`if (y == true) then x:='a'; else x:='b'`" shows that variable `y`

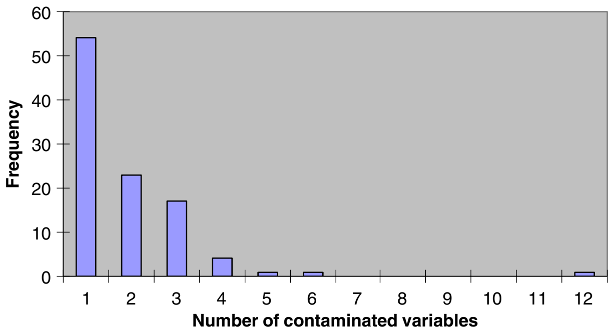


Fig. 5 Histogram of contaminated variables

Table 8 Vulnerable LOC versus total LOC

Application	Total LOC	Vulnerabilities	Vulnerable LOC
A-CART	4,067	8	8
AWStats	26,688	5	9
Bonsai	6,980	8	42
BugTracker.NET	18,101	4	4
BugZilla	9,306	4	39
Calcium	39,348	1	2
Daffodil CRM	25,221	1	4
DEV web management system	11,434	5	5
FileLister	9,139	2	12
JSPWiki	21,231	1	4
Mantis	25,295	12	50
Neomail	1,438	1	5
osCommerce	38,833	15	34
PDF Directory	9,451	12	38
phpBB	29,812	23	100
ProjectApp	11,444	5	11
Roundup	27,061	4	8
sBlog	7,844	2	12
SkunkWeb	6,554	2	4
ViewVC	7,549	2	2

implicitly defines the value of variable x. Hence, there is an implicit data flow from variable y to variable x. To obtain implicit flow information, we manually examined conditional branching statements for all the nodes from the CGs generated in Section 4.3. We discovered 56 statements with conditional branching from the 101 CGs. We then inspected the code blocks for each of these statements for any implicit data flows. A code block is defined as a block of code that is part of the conditional branch. For example, the following conditional statement would contain two code blocks with the first code block containing an implicit data flow:

```
if (x=1)
  y := 2;
  print y;
else
  call func(x);
end
```

The above example shows that if a CG has statements like those in the first code block, the CG would contain an implicit data flow. Table 9 shows the results of the code block investigation. The 56 statements with conditional branching lead to 83 code blocks. We discovered that 29 of these code blocks do have implicit data flow. However, none of these code blocks with implicit data flows are part of the CGs obtained in Section 4.3.

Table 9 shows that the CGs do not contain any implicit data flow statements. To determine if the code blocks containing implicit data flows can lead to potential vulnerabilities, we performed a further investigation of the 29 code blocks. Thirty-six variable assignments were discovered in these code blocks. We found that the variable assignments are either from constants or pre-existing variables. The two example code blocks below shows two possible methods for the variables to be assigned. The first code block shows that the variable is assigned from a constant. The second code block shows the variable being assigned from an existing variable.

```
if (isset($_GET['admin'])
  $admin_mode = 1;
end
```

```
if (strlen($_POST['msg']) < 20)
  $error = $too_short;
  $print($error);
end
```

Although it is clear that constants are generally safe from implementation vulnerabilities, the pre-existing variables need to be examined to determine the original source of the data. We performed a back-trace for each variable assigned from an existing variable. If the variable can be traced to an input, then the potential for vulnerabilities exists. The results can be seen in Table 10.

The results from Tables 9 and 10 provide an answer to Question 4. That is, implicit data flows do not lead to any vulnerabilities in the systems examined. Hence, without further evidence, efforts on eliminating implementation vulnerabilities can focus on explicit data flows.

5 Background

Although studies on web application vulnerabilities properties currently do not exist, many techniques and approaches to detect, or mitigate against, web vulnerabilities have been proposed. In this section, these techniques are briefly presented and discussed.

SQLrand (Boyd and Keromytis 2004), AMNESIA (Halfond and Orso 2005), SQL-Guard (Buehrer et al. 2005), SQLCheck (Su and Wassermann 2006), CSSE (Pietraszek and Berghe 2005), WASP (Halfond et al. 2006, 2008), SQLProb (Liu et al. 2009) are all approaches aimed at addressing SQL injection vulnerabilities. SQLRand inserts random tokens into SQL statements and uses a proxy server to translate these tokens. An incorrect query can be detected if the SQL query does not contain the correct tokens. AMNESIA, SQLGuard and SQLCheck are all model-based approaches. AMNESIA uses static analysis and runtime monitoring to detect for SQL injection vulnerabilities. Static analysis is used to build models of the SQL statements, while the runtime engine detects whether the query

Table 9 Code blocks

Number of CGs containing conditional statements	56
Number of code blocks inspected	83
Number of code blocks with implicit data flows	29
Number of CGs containing code blocks with implicit data flows	0

Table 10 Variable assignments

Number of variables being assigned from a constant	9
Number of variables being assigned from an existing variable	27
Number of existing variables initialized from a constant	27
Number of existing variables initialized from an input	0

strings matches the models. SQLGuard requires the developers to call special functions to build a model of the SQL query to be used. SQLCheck uses a formal definition of a SQL injection vulnerability; and identifies SQL injection attacks based on the formal definition. CSSE and WASP are dynamic approaches designed to address SQL injection vulnerabilities using taint analysis. These approaches attempt to mark negative tainting (CSSE) or positive tainting (WASP) to identify malicious query statements, before they are passed onto the DBMS. Both approaches involve modification to either the runtime engine or usage of a specialized API. SQLProb uses a proxy to identify SQL injection attacks before they reach the web application.

Other approaches to applications' security have also been proposed which address all types of web application vulnerabilities. Security Gateway proposed by Scott and Sharp (2002) is an application firewall that filters out all malicious inputs before they reach the web application. Nguyen-Tuong et al. (2005) proposed a dynamic approach to detect attacks through taint analysis. Martin et al. (2005) proposes PQL (Program Query Language) that enables programmers to specify a sequence of events between objects. Cova et al. (2007) presents a static analysis approach capable of detecting both workflow attacks and data-flow attacks. WebSSARI (Huang et al. 2004) combines static analysis with a runtime component to check on the static model. Pixy (Jovanovic et al. 2006) is currently the one of the more advanced static taint analysis tool available for PHP. Shankar et al. (2001) proposed a static approach that can detect format-string vulnerabilities commonly found in C-based applications. The method defines two extended data types, tainted and untainted, which help reduce the amount of false positives generally associated with static analysis methods. Zhang et al. (2002) and Johnson and Wagner (2004) further extend the approach by using it to assess security issues with the Linux Security Modules framework and user/kernel pointers successfully. These approaches are designed to detect vulnerabilities in C-based applications, and hence, their effectiveness with scripting languages used to develop web applications such as PHP, Ruby, and Python remain unknown.

Scanning tools also exist to help developers and system administrators identify vulnerabilities also exist. QED (Martin and Lam 2008) and Ardilla (Kiezun et al. 2008) attempt to generate SQL Injection and XSS attacks automatically. Secubat (Kals et al. 2006) and other commercial web scanners, such as Acunetix Web Vulnerability[13] Scanner, extend bypass testing by creating tools that provide automatic penetration testing for web applications without using the web applications' target clients. Lin and Chen (2006) extend traditional black-box testing techniques with elements of static analysis by including a tool to automatically inject guards at input points found through the crawling component.

[13] http://www.acunetix.com/, last accessed Feb. 7, 2006

6 Conclusions

In this paper, a research goal of determining whether web application vulnerabilities have any common properties was raised. To reach this goal, four questions were examined.

1. What proportion of security vulnerabilities in web applications can be considered as implementation vulnerabilities? Section 4.1 shows that the majority of the known vulnerabilities are of this type. This means researchers should continue to concentrate on implementation vulnerabilities as it will have the most impact on the security of web applications.
2. Are these vulnerabilities the result of interactions between web applications and external systems? The results from Section 4.2 show that dynamically created strings passed to functions that allow interactions between the application and an external system cause nearly all of the vulnerabilities in this survey. Hence, developers should be careful when allowing data to flow between the web application and other systems.
3. What is the proportion of vulnerable LOC within a web application? That is, what is the vulnerability density? We discovered that the percentage of vulnerable LOC for a web application is extremely small; therefore it can be beneficial to introduce a solution to solve implementation vulnerabilities by concentrating on the CGs with vulnerable LOC.
4. Are implementation vulnerabilities caused by implicit or explicit data flows? Tables 9 and 10, from Section 4.4, show that implementation vulnerabilities for web applications are not caused by implicit data flows. This means efforts on eliminating implementation vulnerabilities can focus on explicit data flows.

References

Agrawal H, Horgan JR (1990) Dynamic program slicing. Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, New York, USA, pp 246–256

Alhazmi OH, Malaiya YK, Ray I (2007) Measuring, analyzing and predicting security vulnerabilities in software systems. Comput Secur J 26(3):219–228

Basili V, Caldeira G, Rombach HD (1994) The goal question metric approach. Encyclopedia of Software Engineering, Wiley

Baskerville R, Pries-Heje J (2004) Short cycle time systems development. Inf Syst J 14(3):237–264

Boyd SW, Keromytis AD (2004) SQLrand: preventing SQL injection attacks. In Proc. of the 2nd Applied Cryptography and Network Security Conf. (ACNS '04), Yellow Mountain, China pp 292–302

Buehrer GT, Weide BW, Sivilotti PAG (2005) Using parse tree validation to prevent SQL injection attacks. In Proc. of the 5th Intl. Workshop on Software Engineering and Middleware (SEM '05), Lisbon, Portugal, pp 106–113

Cova M, Balzarotti D, Felmetsger V, Vigna G (2007) Swaddler: an approach for the anomaly-based detection of State violations in web applications, Recent Advance in Intrusion Detection (RAID), pp 63–86

Denning DE, Denning PJ (1997) Certification of programs for secure information flow. Commun ACM 20:504–513, New York, USA, ACM

Halfond WG, Orso A (2005) AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks. In Proceedings of 20th ACM International Conference on Automated Software Engineering (ASE), Long Beach, CA, USA, pp 174–183

Halfond WG, Orso A, Manolios P (2006) Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In Proceedings of the 14th ACM SIGSOFT international Symposium on Foundations of Software Engineering, Portland, Oregon, USA, pp 175–185

Halfond WGJ, Orso A, Manolios P (2008) WASP: protecting web applications using positive tainting and syntax-aware evaluation. IEEE Trans Softw Eng 34(1):65–81

Huang YW, Yu F, Hang C, Tsai CH, Lee DT, Kuo SY (2004) Securing web application code by static analysis and runtime protection, in WWW '04: Proceedings of the 13th International Conference on World Wide Web. New York, NY, USA: ACM Press, pp 40–52

Liu A, Yuan Y, Wijesekera D, Stavrou A (2009) SQLProb: a proxy-based architecture towards preventing SQL injection attacks. Proceedings of the 2009 ACM symposium on Applied Computing, Honolulu, Hawaii, pp 2054–2061

Johnson R, Wagner D (2004) Finding user/kernel pointer bugs with type inference. In Proceedings of the 2004 Usenix Security Conference, San Diego, CA, USA, pp 119–134

Jovanovic N, Kruegel C, Kirda E (2006) Pixy: a static analysis tool for detecting web application vulnerabilities. In 2006 IEEE Symposium on Security and Privacy, Berkeley/Oakland, CA, USA, pp 258–263

Kals S, Kirda E, Kruegel C, Jovanovic N (2006) SecuBat: a web vulnerability scanner. The 15th International World Wide Web Conference (WWW 2006), Edinburgh, Scotland, pp 247–256

Kiezun A, Guo PJ, Jayaraman K, Ernst MD (2008) Automatic creation of SQL injection and cross-site scripting attacks. Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, Vancouver, British Columbia, Canada, pp 199–209

Lin J-C, Chen J-M (2006) An automatic revised tool for anti-malicious injection. Sixth IEEE International Conference on Computer and Information Technology (CIT'06), Seoul, South Korea, pp 164–170

Martin M, Lam M (2008) Automatic generation of XSS and SQL injection attacks with goal-directed model checking. Proceedings of the 17th conference on Security symposium, San Jose, CA, pp 31–43

Martin M, Livshits B, Lam MS (2005) Finding application errors and security flaws using PQL: a program query language. In OOPSLA '05: Proc. of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, San Diego, CA, USA, pp 365–383

Nguyen-Tuong A, Guarnieri S, Greene D, Shirley J, Evans D (2005) Automatically hardening web applications using precise tainting. In Proceedings of the 20th IFIP International Information Security Conference, Chiba, Japan, pp 372–382

OWASP (2007) Top 10 2007. http://www.owasp.org/index.php/Top_10_2007, last accessed June 29, 2009

Pietraszek T, Berghe CV (2005) Defending against injection attacks through context-sensitive string evaluation. In Proceedings of Recent Advances in Intrusion Detection (RAID2005), Seattle, Washington, USA, pp 124–145

Rapid7 (2005) Vulnerability management trends. (2)1–9

Scambray J, Shema M, Sima C (2006) Hacking exposed: web applications second edition. McGraw-Hill, San Francisco

Scott D, Sharp R (2002) Abstracting application-level web security. In Proc. of the 11th Intl. Conference on the World Wide Web (WWW 2002), Honolulu, Hawaii, USA, pp 396–407

Shankar U, Talwar K, Foster JS, Wagner D (2001) Detecting format string vulnerabilities with type qualifiers. In 10th USENIX Security Symposium, Washington, D.C., pp 201–220

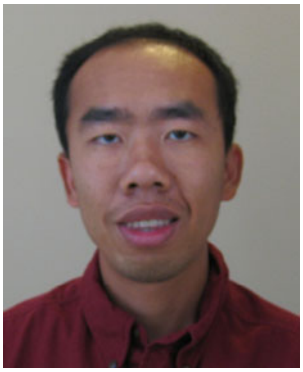
Su Z, Wassermann G (2006) The essence of command injection attacks in web applications. In The 33rd Annual Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, pp 372–382

Swiderski F, Snyder W (2004) Threat modeling. Microsoft Press, Redmond

Tip F (1995) A survey of program slicing techniques. J Program Lang 3(3):121–189

Weiser M (1984) Program slicing. IEEE Trans Softw Eng SE-10(4):352–357

Zhang X, Edwards A, Jaeger T (2002) Using CQual for static analysis of authorization hook placement. In the Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA, pp 33–48



Toan Huynh received a B.Sc. degree in Computer Engineering and Ph.D. degree in Software Engineering from the University of Alberta, Canada. His research interests include: web systems, e-commerce, software testing, vulnerabilities and defect management, and software approaches to the production of secure systems.

James Miller received the B.Sc. and Ph.D. degrees in Computer Science from the University of Strathclyde, Scotland. Subsequently, he worked at the United Kingdom's National Electronic Research Initiative on Pattern Recognition as a Principal Scientist, before returning to the University of Strathclyde to accept a lectureship, and subsequently a senior lectureship in Computer Science. Initially during this period his research interests were in Computer Vision; since 1993, his research interests have been in Software and Systems Engineering. In 2000, he joined the Department of Electrical and Computer Engineering at the University of Alberta as a full professor and in 2003 became an adjunct professor at the Department of Electrical and Computer Engineering at the University of Calgary. He has published over one hundred refereed journal and conference papers on Software and Systems Engineering (see www.steam.ualberta.ca for details on recent directions); and currently serves on the program committee for the IEEE International Symposium on Empirical Software Engineering and Measurement; and sits on the editiorial board of the Journal of Empirical Software Engineering.