# AUTOMATICALLY HARDENING WEB APPLICATIONS USING PRECISE TAINTING

Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, David Evans
*Department of Computer Science, University of Virginia, 151 Engineer's Way, Charlottesville, VA 22904-4740, USA* [⊥]

Abstract: Most web applications contain security vulnerabilities. The simple and natural ways of creating a web application are prone to SQL injection attacks and cross-site scripting attacks as well as other less common vulnerabilities. In response, many tools have been developed for detecting or mitigating common web application vulnerabilities. Existing techniques either require effort from the site developer or are prone to false positives. This paper presents a fully automated approach to securely hardening web applications. It is based on precisely tracking taintedness of data and checking specifically for dangerous content only in parts of commands and output that came from untrustworthy sources. Unlike previous work in which everything that is derived from tainted input is tainted, our approach precisely tracks taintedness within data values.

Key words: web security; web vulnerabilities; SQL injection; PHP; cross-site scripting attacks; precise tainting; information flow

## 1. INTRODUCTION

Nearly all web applications are security critical, but only a small fraction of deployed web applications can afford a detailed security review. Even when such a review is possible, it is tedious and can overlook subtle security vulnerabilities. Serious security vulnerabilities are regularly found in the most prominent commercial web applications including Gmail[1], eBay[2],

[⊥] This work was funded in part by DARPA (SRS FA8750-04-2-0246) and the National Science Foundation (NSF CAREER CCR-0092945, SCI-0426972).

Yahoo[3], Hotmail[3] and Microsoft Passport[4]. Section 2 provides background on common web application vulnerabilities.

Several tools have been developed to partially automate aspects of a security review, including static analysis tools that scan code for possible vulnerabilities[5] and automated testing tools that test web sites with inputs designed to expose vulnerabilities[5-7]. Taint analysis identifies inputs that come from untrustworthy sources (including user input) and tracks all data that is affected by those input values. An error is reported if tainted data is passed as a security-critical parameter, such as the command passed to an exec command. Taint analysis can be done statically or dynamically. Section 3 describes previous work on securing web applications, including taint analysis.

For an approach to be effective for the vast majority of web applications, it needs to be fully automated. Many people build websites that accept user input without any understanding of security issues. For example, *PHP & MySQL for Dummies*[8] provides inexperienced programmers with the knowledge they need to set up a database-backed web application. Although the book does include some warnings about security (for example, p. 213 warns readers about malicious input and advises them to check correct format, and p. 261 warns about <script> tags in user input), many of the examples in the book that accept user input contain security vulnerabilities (e.g., Listings 11-3 and 12-2 allow SQL injection, and Listing 12-4 allows cross-site scripting). This is typical of most introductory books on web site development.

In Section 4 we propose a completely automated mechanism for preventing two important classes of web application security vulnerabilities: command injection (including script and SQL injection) and cross-site scripting (XSS). Our solution involves replacing the standard PHP interpreter with a modified interpreter that precisely tracks taintedness and checks for dangerous content in uses of tainted data. All that is required to benefit from our approach is that the hosting server uses our modified version of PHP.

The main contribution of our work is the development of precise tainting in which taint information is maintained at a fine level of granularity and checked in a context-sensitive way. This enables us to design and implement fully-automated defense mechanisms against both command injection attacks, including SQL injection, and cross-site scripting attacks. Next, we describe common web application vulnerabilities. Section 3 reviews prior work on securing web applications. Section 4 describes our design and implementation, and explains how we prevent exploits of web application vulnerabilities.

## 2. WEB APPLICATION VULNERABILITIES

Figure 1 depicts a typical web application. For clarity, we focus on web applications implemented using PHP, which is currently one of the most popular language for implementing web applications (PHP is used at approximately 1.3M IP addresses, 18M domains, and is installed on 50% of Apache servers[9]). Most issues and architectural properties are similar for other web application languages.

A client sends input to the web server in the form of an HTTP request (step 1 in Figure 1). `GET` and `POST` are the most common requests. The request encodes data created by the user in HTTP header fields including file names and parameters included in the requested URI. If the URI is a PHP file, the HTTP server will load the requested file from the file system (step 2) and execute the requested file in the PHP interpreter (step 3). The parameters are visible to the PHP code through predefined global variable arrays (including `$_GET` and `$_POST`).

The PHP code may use these values to construct commands that are sent to PHP functions such as a SQL query that is sent to the database (steps 4 and 5), or to make calls to PHP API functions that call system APIs to manipulate system state (steps 6 and 7). The PHP code produces an output web page based on the returned results and returns it to the client (step 8).

We assume a client can interact with the web server only by sending HTTP requests to the HTTP server. In particular, the only way an attacker can interact with system resources, including the database and file system, is by constructing appropriate web requests. We divide attacks into two general classes of attacks: *injection attacks* attempt to construct requests to the web server that corrupt its state or reveal confidential information; *output attacks* (e.g., cross-site scripting) attempt to send requests to the web server that cause it to generate responses that produce malicious behavior on clients.

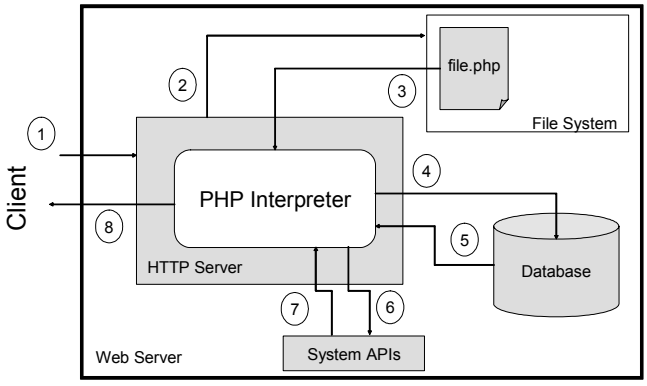


*Figure 1*. Typical web application architecture

## 2.1 Command injection attacks

In a command injection attack an attacker attempts to access confidential information or corrupt the application state by constructing an input that allows the attacker to inject malicious control logic into the web application. With the system architecture shown in Figure 1, an attack could attempt to inject PHP code that will be executed by the PHP interpreter, SQL commands that will be executed by the database, or native machine code that will be executed by the web server host directly. We consider only the first two cases. Web application vulnerabilities are far more common than vulnerabilities in the underlying server or operating system since there are far more different web applications than there are servers and operating systems, and developers of web applications tend to be far less sophisticated from a security perspective than developers of operating systems and web servers.

**PHP injection.** In a PHP injection attack, the attacker attempts to inject PHP code that will be interpreter by the server. If an attacker can inject arbitrary code, the attacker can do everything PHP can and has effectively complete control over the server. Here is a simple example of a PHP injection in phpGedView, an online viewing system for genealogy information[10]. The attack URL is of the form:

```
http://[target]/[...]/editconfig_gedcom.php?gedcom_config=../../../../../../etc/passwd
```

The vulnerable PHP code uses the gedcom_config value as a filename: require($gedcom_config);. The semantics of require is to load the file and either interpret it as PHP code (if the PHP tags are found) or display the content. Thus this code leaks the content of the password file. Abuse of require and its related functions is a commonly reported occurrence[11,12], despite the fact that, properly configured, PHP is impervious to this basic attack. However, additional defenses are needed for more sophisticated injection attacks such as the recently released Santy Worm[13] and the phpMyAdmin attack[14].

**SQL injection.** Attacking web applications by injecting SQL commands is a common method of attacking web applications[15,16]. We illustrate a simple SQL injection that is representative of actual vulnerabilities. Suppose the following is used to construct an SQL query to authenticate users against a database:

```
$cmd="SELECT user FROM users WHERE user = ' " . $user
        . "' AND password = ' " . $pwd . " ' ";
```

The value of $user comes from $_POST['user'], a value provided by the client using the login form. A malicious client can enter the value: ' OR 1 = 1 ; --' (-- begins a comment in SQL which continues to the end of the line). The resulting SQL query will be: SELECT user FROM users WHERE user = ' ' OR 1 = 1 ; -- ' AND password = 'x'. The injected command closes the quote and

comments out the AND part of the query. Hence, it will always succeed regardless of the entered password.

The main problem here is that the single quote provided by the attacker closes the open quote, and the remainder of the user-provided string is passed to the database as part of the SQL command. This attack would be thwarted by PHP installations that use the default magic quotes option. When enabled, magic quotes automatically sanitize input data by adding a backslash to all strings submitted via web forms or cookies. However, magic quotes do not suffice for attacks that do not use quotes[17].

One solution to prevent SQL injections is to use prepared statements[18]. A prepared statement is a query string with placeholders for variables that are subsequently bound to the statement and type-checked. However, this depends on programmers changing development practices and replacing legacy code. Dynamic generation of queries using regular queries will continue to be prevalent for the foreseeable future.

## 2.2 Output attacks

Output attacks send a request to a web application that causes it to produce an output page designed by the attacker to achieve some malicious goal. The most dangerous kind of output attack is a *cross-site scripting* attack, in which the web server produces an output page containing script code generated by the attacker. The script code can steal the victim's cookies or capture data the victim unsuspectingly enters into the web site. This is especially effective in phishing attacks in which the attacker sends potential victims emails convincing them victim to visit a URL. The URL may be a trusted domain, but because of a cross-site scripting vulnerability the attacker can construct parameters to the URL that cause the trusted site to create a page containing a form that sends data back to the attacker. For example, the attacker constructs a link like this:

```
<a href='http://bad.com/go.php?val=<script src="http://bad.com/attack.js"></script>'>
```

If the implementation of go.php uses the val parameter in the generated web page output (for example, by doing print "Results for: " . $_GET['val'];), the malicious script will appear on the resulting page. A clever attacker can use character encodings to make the malicious script appear nonsensical to a victim who inspects the URL before opening it.

Five years ago, CERT Advisory 2000-02 described the problem of cross-site scripting and advised users to disable scripting languages and web site developers to validate web page output[19]. Nevertheless, cross-site scripting problems remain a serious problem today. Far too much functionality of the web depends on scripting languages, so most users are unwilling to disable

them. Even security-conscious web developers frequently produce websites that are vulnerable to cross-site scripting attacks[1,4,20-22]. As with SQL injection, ad hoc fixes often fail to solve discovered problems correctly—the initial filters develop to fix the Hotmail vulnerability could be circumvented by using alternate character encodings[4]. Hence, we focus on fully automated solutions.

## 3. RELATED WORK

Several approaches have been developed for securing web applications including filtering input and output that appears dangerous, automated testing and diversity defenses. The approaches most similar to our proposed approach involve analyzing information flow.

**Input and Output Filtering.** Scott and Sharp developed a system for providing an application-level firewall to prevent malicious input from reaching vulnerable web servers[23]. Their approach required a specification of constraints on different inputs, and compiled those constraints into a checking program. This requires a programmer to provide a correct security policy specific to their application, so is ill-suited to protecting typical web developers. Several commercial web application firewalls provide input and output filtering to detect possible attacks[24,25]. However, these tools are prone to both false positives and negatives[26].

**Automated Testing.** There are several web application security testing tools designed specifically to find vulnerabilities[5,27,28]. The problem with these tools is that they have to guess the exploit data in order to expose the vulnerability. For well-known generic classes of vulnerabilities, such as SQL injection, this may be possible. But for novel or complex vulnerabilities, it is unlikely the scanner will guess the right inputs to expose the vulnerability.

**Diversity Defenses.** Instruction-Set Randomization is a form of diversity in which defenders modify the instruction set used to run applications[29]. Thus, code-injection attacks that rely on knowledge of the original language are detected and thwarted easily. This approach has been advocated for general scripting languages[29] and for protection against SQL injections[30]. There are two main problems with ISR: (1) it is effective only against code injection attacks and incomplete by itself (it does not handle cross-site scripting attacks), and (2), the deployment of ISR is not transparent to developers and requires the transformation of application code.

**Information Flow.** All of the web vulnerabilities described in Section 2 stem from insecure information flow: data from untrusted sources is used in a trusted way. The security community has studied information flow extensively[31]. The earliest work focused on confidentiality, in particular in

preventing flows from trusted to untrusted sources[32]. In our case, we are primarily concerned with integrity. Biba showed that information flow can also be used to provide integrity by considering flows from untrusted to trusted sources[33].

Information flow policies can be enforced statically, dynamically or by a combination of static and dynamic techniques. Static taint analysis has been used to detect security vulnerabilities in C programs[34,35]. Static approaches have the advantage of increased precision, no run-time overhead and the ability to detect and correct errors before deployment. However, they require substantial effort from the programmer. Since we are focused on solutions that will be practically deployed in typical web development scenarios, we focus on dynamic techniques.

Huang et. al developed WebSSARI, a hybrid approach to securing web applications[36]. The WebSSARI tool uses a static analysis based on type-based information flow to identify possible vulnerabilities in PHP web applications. Their type-based approach operates at a coarse-grain: any data derived from tainted input is considered fully tainted. WebSSARI can insert calls to sanitization routines that filter potentially dangerous content from tainted values before they are passed to security-critical functions. Because we propose techniques for tracking taintedness at a much finer granularity, our system can be more automated than WebSSARI: all we require is that the server uses our modified interpreter PHP to protect all web applications running on the server.

## 4. AUTOMATIC WEB HARDENING

Our design is based on maintaining precise information about what data is tainted through the processing of a request, and checking that user input sent to an external command or output to a web page contains only safe content. Our automated solution prevents a large class of common security vulnerabilities without any direct effort required from web developers.

The only change from the standard web architecture in Figure 1 is that we replace the standard PHP interpreter with a modified interpreter that identifies which data comes from untrusted sources and precisely tracks how that data propagates through PHP code interpretation (Section 4.1), checks that parameters to commands do not contain dangerous content derived from user input (Section 4.2), and ensures that generated web pages do not contain scripting code created from untrusted input (Section 4.3).

## 4.1 Keeping track of precise taint information

We mark an input from untrusted sources including data provided by client requests as tainted. We modified the PHP interpreter's implementation of the string datatype to include tainting information for string values at the granularity of individual characters. We then propagate taint information across function calls, assignments and composition at the granularity of a single character, hence *precise tainting*. The application of precise tainting enables the prevention of injection attacks and the ability to easily filter output for XSS attacks. If a function uses a tainted variable in a dangerous way, we can reject the call to the function (as is done with SQL queries or PHP system functions) or sanitize the variable values (as is done for preventing cross-site scripting attacks).

Web application developers often remember to sanitize inputs from GET and POSTs, but will omit to check other variables that can be manipulated by clients. Our approach ensures that *all* such external variables, e.g. hidden form variables, cookies and HTTP header information, are marked as tainted. We also keep track of taint information for session variables and database results.

### 4.1.1 Taint strings

For each PHP string, we track tainting information for individual characters. Consider the following code fragment where part of the string $x comes from a web form and the other from a cookie:

```
$x = "Hello " . $_GET['name1'] . ". I am " . $_COOKIE['name2'];
```

The values of $_GET['name1'] and $_COOKIE['name2'] are fully tainted (we assume they are Alice and Bob). After the concatenation, the values of $x and its taint markings (underlined) are: `Hello Alice. I am Bob.`

### 4.1.2 Functions

We keep track of taint information across function calls, in particular functions that manipulate and return strings. The general algorithm is to mark strings returned from function as tainted if any of the input arguments are tainted. Whenever feasible, we exploit the semantics of functions and keep track of taintedness precisely. For example, consider the substring function in which taint markings for the result of the substr call depend on the part of the string they select: substr("precise taint me", 2, 10); // ecise tai

### 4.1.3 Database values and session variables

Databases provide another potential venue for attackers to insert malicious values. We treat strings that are returned from database queries as untrusted and mark them as tainted. While this approach may appear overly restrictive, in the sense that legitimate uses may be prevented, we show in Section 4.3 how precise tainting and our approach to checking for cross-site scripting mitigates this potential problem. Further, if the database is compromised by some other means, the attacker is still unable to use the compromised database to construct a cross-site scripting attack.

The stateless nature of HTTP requires developers to keep track of application state across client requests. However, exposing session variables to clients would allow attackers to manipulate applications. Well-designed web applications keep session variables on the server only and use a session id to communicate with clients. We modified PHP to store taint information with session variables.

## 4.2 Preventing command injection

The tainting information is used to determine whether or not calls to security-critical functions are safe. To prevent command injection attacks, we check that the tainted information passed to a command is safe. The actual checking depends on the command, and is designed to be precise enough to prevent all command injection attacks from succeeding while allowing typical web applications to function normally when they are not under attack.

### 4.2.1 PHP injection

To prevent PHP injection attacks we disallow calls to potentially dangerous functions if any one of their arguments is tainted. The list of functions checked is similar to those disallowed by Perl and Ruby's taint mode[37,38] and consists of functions that treat input strings as PHP code or manipulate the system state such as system calls, I/O functions, and calls that are directly evaluated.

### 4.2.2 SQL injection

Preventing SQL injections requires taking advantage of precise taint information. Before sending commands to the database, e.g. `mysql_query`, we run the following algorithm to check for injections:

1. Tokenize the query string; preserve taint markings with tokens.
2. Scan each token for identifiers and operator symbols (ignore literals, i.e., strings, numbers, boolean values).
3. Detect an injection if an operator symbol is marked as tainted. Operator symbols are ,()[].;:+-*/\%^<>=~!?@#&|`
4. Detect an injection if an identifier is tainted and a keyword. Example keywords include UNION, DROP, WHERE, OR, AND.

Using the example from Section 2.1:

```
$cmd="SELECT user FROM users WHERE user = ' " . $user
     . "' AND password = ' " . $password . " ' ";
```

The resulting query string (with $user set to ' OR 1 = 1 ; -- ') is tainted as follows: SELECT user FROM users WHERE user = ' ' OR 1 = 1 ; -- ' AND password = 'x'. We detect an injection since OR is both tainted and a keyword.

## 4.3 Preventing cross-site scripting

Our approach to preventing cross-site scripting relies on checking generated output. Any potentially dangerous content in generated HTML pages must contain only untainted data. We modify the PHP output functions (print, echo, printf and other printing functions) with functions that check for tainted output containing dangerous content. The replacement functions output untainted text normally, but keep track of the state of the output stream as necessary for checking. For a contrived example, consider an application that opens a script and then prints tainted output: print "<script>document.write ($user)</script>";

An attacker can inject JavaScript code by setting the value of $user to a value that closes the parenthesis and executes arbitrary code: " me");alert("yo". Note that the opening script tag could be divided across multiple print commands. Hence, our modified output functions need to keep track of open and partially open tags in the output. We do not need to parse the output HTML completely (and it would be unadvisable to do so, since many web applications generate ungrammatical HTML).

Checking output instead of input avoids many of the common problems with ad hoc filtering approaches. Since we are looking at the generated output any tricks involving separating attacks into multiple input variables or using character encodings can be handled systematically. Our checking involves whitelisting safe content whereas blacklisting attempts to prevent cross-site scripting attacks by identifying known dangerous tags, such as <script> and <object>. The latter fails to prevent script injection involving other tags. For example, a script can be injected into the apparently harmless <b> (bold) tag using parameters such as onmouseover.

Our defense takes advantage of precise tainting information to identify web page output generated from untrusted sources. Any tainted text that could be dangerous is either removed from the output or altered to prevent it being interpreted (for example, replacing < in unknown tags with <). Our conservative assumptions mean that some safe content may be inadvertently suppressed; however, because of the precise tainting information, this is limited to content that is generated from untrusted sources.

## 5. CONCLUSION

We have described a fully automated, end-to-end approach for hardening web applications. By exploiting precise tainting in a way that takes advantage of program language semantics and performing context-dependent checking, we are able to prevent a large class of web application exploits without requiring any effort from the web developer. Initial measurements indicate that the performance overhead incurred by using our modified intepreter is less than 10%.

Effective solutions for protecting web applications need to balance the need for precision with the limited time and effort most web developers will spend on security. Fully automated solutions, such as the one described in this paper, provide an important point in this design space.

## 6. REFERENCES

1. N. Weidenfeld, *Security Hole Found in Gmail*, (27 October 2004); http://net.nana.co.il/Article/?ArticleID=155025&sid=10.
2. *Report of Ebay Cross-Site Scripting Attack*; http://securityfocus.com/archive/82/246275.
3. *Remotely Exploitable Cross-Site Scripting in Hotmail and Yahoo*, (March 2004); http://www.greymagic.com/security/advisories/gm005-mc/.
4. EyeonSecurity, Microsoft Passport Account Hijack Attack: Hacking Hotmail and More, *Hacker's Digest*.
5. Y.-W. Huang *et al.*, Web Application Security Assessment by Fault Injection and Behavior Monitoring, *Proc. of the World Wide Web Conference (WWW 2003)*, (May 2003).
6. M. Benedikt *et al.*, Veriweb: Automatically Testing Dynamic Web Sites, *Proc. of the World Wide Web Conference*, (May 2002).
7. F. Ricca, and P. Tonella, Analysis and Testing of Web Applications, *Proc. of the IEEE International Conference on Software Engineering*, (May 2001).
8. J. Valade, *Php & Mysql for Dummies*, (Wiley Publishing, 2002).
9. *Netcraft Survey*, (January 2005); http://news.netcraft.com/.
10. JeiAr, *Phpgedview Php Injection*, (Jan 2004); http://xforce.iss.net/xforce/xfdb/14205.
11. Gentoo, *Gallery Php Injection*, (February 2004); http://www.linuxsecurity.com/advisories/gentoo_advisory-4015.html.

12. K. Więsek, Gonicus System Administrator Php Injection, (February 2003).
13. *Santy Worm Used Google to Spread*, (23 December 2004); http://newsfromrussia.com/world/2004/12/23/57537.html.
14. N. Symbolon, *Phpmyadmin Critical Bug*; http://xforce.iss.net/xforce/xfdb/16542.
15. D. Litchfield, *Sql Server Security*, (McGraw-Hill Osborne Media, 2003).
16. K. Spett, "Sql Injection: Are Your Web Applications Vulnerable?" (SPI Labs White Paper, 2002).
17. L. Armstrong, *Phpnuke Sql Injection*, (20 February 2003).
18. *Improved Mysql Extensions*; http://www.php.net/manual/en/ref.mysqli.php.
19. *Malicious Html Tags Embedded in Client Web Requests*, (February 2, 2000); http://www.cert.org/advisories/CA-2000-02.html.
20. G. Hoglund, and G. McGraw, *Exploiting Software: How to Break Code*, (Addison-Wesley, 2004).
21. R. Ivgi, *Cross-Site-Scripting Vulnerability in Microsoft.Com*, (4 October 2004).
22. J. Ley, *Simple Google Cross Site Scripting Exploit*, (17 October 2004).
23. D. Scott, and R. Sharp, Abstraction Application-Level Web Security, *Proc. of the WWW*, (May 2002).
24. *Interdo Web Application Firewall*; http://www.kavado.com/products/interdo.asp.
25. Teros, Inc., *Teros-100 Application Protection System*, (2004); http://www.teros.com/products/aps100/aps.shtml.
26. T. Dyck, *Review: Appshield and Review: Teros-100 Aps 2.1.1*, (May 2003); http://www.eweek.com/article2/0,3959,1110435,00.asp.
27. Tenable Network Security, *Nessus Open Source Vulnerability Scanner Project*, (2005); http://www.nessus.org.
28. J. Offutt *et al.*, Bypass Testing of Web Applications., *Proc. of the IEEE International Symposium on Software Reliability Engineering*, (November 2004).
29. G. S. Kc *et al.*, Countering Code-Injection Attacks with Instruction-Set Randomization., *Proc. of the ACM Computer and Communication Security (CCS)*, (October 2003).
30. S. W. Boyd, and A. D. Keromytis, Sqlrand: Preventing Sql Injection Attacks, *Proc. of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, (June 2004).
31. A. Sabelfeld, and A. C. Myers, Language-Based Information-Flow Security, *IEEE Journal on Selected Areas in Communications* (January 2003).
32. D. E. Bell, and L. J. LaPadula, *Secure Computer Systems: Mathematical Foundations* Mtr-2547, (MITRE Corporation, 1973).
33. K. J. Biba, *Integrity Considerations for Secure Computer Systems* Esd-Tr-76-372, (USAF Electronic Systems Division, 1977).
34. U. Shankar *et al.*, Detecting Format-String Vulnerabilities with Type Qualifiers, *Proc. of the USENIX Security Symposium*.
35. D. Evans, and D. Larochelle, Improving Security Using Extensible Lightweight Static Analysis, *IEEE Software* (January/February 2002).
36. Y.-W. Huang *et al.*, Securing Web Application Code by Static Analysis and Runtime Protection, *Proc. of the World Wide Web Conference*, (May 2004).
37. *Perl 5.6 Documentation: Perl Security*; http://www.perldoc.com/perl5.6/pod/perlsec.html.
38. D. Thomas *et al.*, *Programming Ruby: The Pragmatic Programmer's Guide*, (Pragmatic Programmers, ed. Second, 2004).