Automatically Hardening Web Applications Using Precise Tainting

Anh Nguyen-Tuong Salvatore Guarnieri Doug Greene David Evans

University of Virginia Computer Science Technical Report CS-2004-36
December 2004
{nguyen, evans}@cs.virginia.edu

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 (among 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 in 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. We describe our results and prototype implementation on the predominant LAMP (Linux, Apache, MySQL, PHP) platform.

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 have been found in the largest commercial web applications including Gmail [Wei04], eBay [eBay01], Yahoo [Grey04], Hotmail [Grey04] and Microsoft Passport [Eye02].

Several tools have been developed to partially automate aspects of a security review, including static analysis tools that scan code for possible vulnerabilities [Huang04] and automated testing tools that test web sites with inputs designed to expose vulnerabilities [Bene02, Hua03, Ricca01]. 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 security web applications.

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* [Val02] 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.

We propose a completely automated mechanism for preventing two important classes of web application security vulnerabilities: command injection (including SQL injection) and cross-site scripting. 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. A web application developer does not need to do anything to benefit from our technique, as long as 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. VULNERABILITIES

Figure 1 shows a typical web application. 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 field 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) and then to produce an output web page based on the returned results and return it to the client (step 6). If an attacker can access confidential information or corrupt the application state by carefully constructing a malicious input, the application is vulnerable to a command injection attack. If the attacker can arrange for the server to produce a page that will execute a script constructed by the attacker, the application is vulnerable to a cross-site scripting attack.

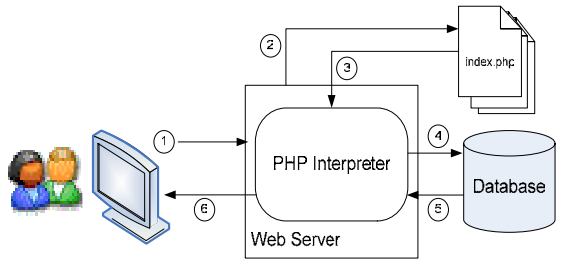

Figure 1. Web application architecture.

2.1 Command Injection

2.1.1 PHP Injection

Scripting languages such as PHP and Perl that drive the generation of dynamic web pages represent the first line of defense against other forms of attacks, i.e, SQL injections and XSS, as well as the first target for attackers. A successful injection at the scripting language level yields control of the web site to attackers, including the ability to execute dangerous OS commands.

Here is a simple example of a PHP injection that was present in phpGedView (version 2.65.1 and earlier), an online viewing system for genealogy information [Jei04]. The attack URL is of the form:

```
http://[target]/[phpGedView-directory]/
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 [Gentoo04, Manip04, Więsek03].

Properly configured, PHP can thwart this type of attack. PHP's safe mode provides the open_basedir and allow_url_fopen settings to control access to the local filesystem or prevent remote files from being read. Unfortunately from a security point of view, the default settings are permissive. The growing popularity of PHP (PHP is used at over 1.3 IP addresses and is installed with 50% of Apache servers [Netcraft04]), coupled with PHP's emphasis on functionality over security, ensures the continued availability of exploitable web sites.

The next example illustrates a recent and sophisticated attack on phpMyAdmin, a popular package for managing databases [Simb04].

phpMyAdmin stores database configuration parameters in the array $cfg. An attack can add an additional server configuration parameters by growing the array through the use of GET variables. The attacker then sets up its own SQL server and waits for a connection. Note that the attacker breaks the implicit and common

assumption of a trusted database server. PhpMyAdmin will then query the attacker's database for a list of table names.

The following code then eventually executes, using a table name (controlled by the attacker) in an argument to eval:

```
$eval_string = '$tablestack[\"
   . implode('\'][\", $_table) . '\'][\'pma_name\'][]
   = \" . str_replace('\", '\\\", $table) . '\';';
eval($eval_string);
```

The authors of phpMyAdmin included code to perform sanitization of the table name before passing it to eval. However, the sanitization routine does not properly handle quotes. A table name of the form

```
\\';exec(\"<Attack Code>\");/*
```

results in a successful attack as eval evaluates the attack string as PHP code. In turn, the PHP exec command will execute the commands in the attack code on the host server.

This multi-step attack illustrates the level of sophistication of attackers and the dangers of ad-hoc sanitization routines.

2.1.2 SQL Injection

Attacking web applications by injecting SQL commands was first described as early as 1998 [Rai98], and remains a common method of attacking web applications [Kost04, Litchfield03, Spett02].

We illustrate a SQL injection attack with a simple example based on a common practice of authenticating web site visitors with a login form requesting a name and password. Suppose the following code fragment is used to construct an SQL query to authenticate users against a database (it is inadvisable to store passwords in cleartext, but many commercial websites do):

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

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.

In fact, this attack would be thwarted by PHP installations that retain 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, the magic quote option is not a panacea in defending against SQL injections. First, many web administrators turn off magic quotes as their use may interfere with applications that explicitly manipulate quotes and backslashes prior to inserting data into a database [Fuecks02]. Second, as the following example illustrates, even if magic quotes are enabled there are SQL injection attacks that succeed without needing to insert a quote character.

This example illustrates a SQL injection vulnerability in PHPnuke, a widely used open-source web portal management system, in which an attacker can acquire the administrator's password hash (which, because of a mistake in the way PHPnuke did authentication with cookies was enough to be able to login without cracking the actual password) [Arm2003]. This is done using a select fish attack in which the attacker guesses the value of a column in a database one character at a time. In this case, the column holds the MD5 hash of the administrator's password.

The full attack string and exploit description is available online [Arm2003]. The attack injects SQL code using a parameter in a URL (+ encodes a space):

```
http://site/modules.php?name=search&query=
   &days=1+or+mid(a.pwd,1,1)=6&type=stories
```

The salient part of the resulting query string is:

```
SELECT s.sid, a.url FROM nuke_stories s,
                         nuke_authors a
  WHERE s.aid=a.aid OR mid(a.pwd,1,1)=6
```

In this example, the attacker guesses that the first character of the password hash is 6. If the attacker guesses correctly the WHERE test is satisfied and the generated output displays a matching record. Since MD5 hashes are 32 characters long with 16 possible values for each character [0-9, A-F], the attacker can obtain the full password hash with a maximum of 512 guesses. A program can easily be written to automate this process and steal the password hash in a matter of minutes.

The authors of PHPnuke provided an ad-hoc fix for this vulnerability in version 6.0 by checking for parenthesis in GET variables. However, they neglected to check POST variables, which permitted a similar exploit to succeed [Arm2003]. This example illustrates the sophistication of attackers, the danger of relying on ad-hoc fixes, and underscores the need for an automated approach to prevent SQL injections.

One solution to SQL injection vulnerabilities is to use SQL prepared statements, supported in MySQL 4.1 [Fisk04] and PHP 5.0 [PHP5]. A prepared statement is a query string with placeholders for variables that are subsequently bound to the statement and type-checked. This clean delineation between application data and logic prevents SQL injections. 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 Cross-Site Scripting

A cross-site scripting vulnerability enables an attacker to insert script code in a web page produced by a site trusted by the victim. 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 the victim an email convincing the 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://www.trusted.com/search.php?
   key=<script src="http://malice.com/attack.js">
   </script>'>
```

If the implementation of search.php uses the key parameter provided in the URL in the generated we page, the malicious script will appear on the resulting page:

```
print "Results for: " . $_GET['key'];
```

A clever hacker can use character encodings to make the malicious script appear nonsensical to a victim who inspects the URL before opening it.

Four years ago, CERT Advisory 2000-02 [CERT00] described the problem of cross-site scripting and advised users to disable scripting languages and web site developers to validate web page output. 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 (for good cause) unwilling to disable them. Even security-conscious web developers frequently produce websites that are vulnerable to cross-site scripting attacks. Hoglund and McGraw [Hog2004] describe several different examples of websites vulnerable to simple cross-site scripting attacks.

An example typical a recent cross-site scripting vulnerabilities was one reported by Rafel Ivgi in microsoft.com in October 2004 [Ivgi04]. An attacker was able to inject scripting code into a web page generated by microsoft.com using a URL parameter:

```
href="http://www.microsoft.com/.../download.asp?
   sTarget=javascript:<attack code>"
```

The attack code would appear in a link on the generated page, and execute if the victim clicked on that link.

A particularly dangerous cross-site scripting vulnerability was found in Microsoft's Hotmail application [Eye02]. Hotmail displays email in HTML format which can be sent to the user from anyone. The site took measures to filter out the most obvious dangerous content in received messages including applets and JavaScript. However, if the

user clicks on a link in an email message to a MSN site, the user's Passport credentials are transmitted to the new site. The http://auctions.msn.com/ site included an ASP script ErrorMsg.asp that could be passed arguments. A URL that links to these scripts with scripting code in the arguments would be interpreted, thus executing a script created by the attacker when the victim clicks on the link. Because this site has access to the victims Passport credentials, the attack script could steal the victim's cookie and send these credentials to the attacker.

The initial fixes to this vulnerability [Eye02] involved filtering script arguments to ignore dangerous tags and common scripting code. The filters, however, could be circumvented by using alternate character encodings. (The ErrorMsg.asp script is no longer supported.) As with PHPNuke, ad hoc approaches rarely fix the whole problem.

Cross-site scripting vulnerabilities continue to be found in widely-used web sites [Endler02]. In the past few weeks, several vulnerabilities have been found in Google [Ley04] and Gmail [Wei04]. The persistence of cross-site scripting vulnerabilities stems from the need to accept user input to provide functionality in web applications, combined with the lack of a clear separation between content and code. Ad hoc solutions can close the vulnerabilities, but often fail to prevent all exploits and are easily overlooked by even security-conscious developers. Hence, we focus on fully automated solutions.

3. PRIOR WORK

Numerous approaches have been proposed for securing web applications including vulnerability testing [Bene02, Hua03, Ricca01] and static analysis. Describing all work on web application security is outside the scope of this paper; instead, we focus here on describing the most relevant work from an information flow perspective.

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 [Sab03]. The earliest work focused on confidentiality, in particular in preventing flows from trusted to untrusted sources [Bell73]. 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 [Biba77].

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 [Eva02, Shan01]. 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 and hybrid techniques.

Information flow policies can be enforced dynamically at any of the steps in Figure 1. Input and output filtering observes traffic between the client and web server. The most closely related work to ours is taint checking.

3.1 Input and Output Filtering

Scott and Sharp [Sco02] developed a system for providing an application-level firewall for preventing malicious input from reaching vulnerable web servers. Their approach required a specification of constraints on different inputs, and compiled those constraints into a checking program. Their approach 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 have been developed including AppShield [Watch04] and InterDo [Kavado04] and Teros-100 APS [Teros04]. These tools provide both input and output filtering, observing web traffic to detect possible attacks. For example, Teros-100 APS can be configured to detect text that appears to be a credit card number in output pages and prevent it from being returned. Without extensive configuration, however, the tools are prone to both false positives and false negatives [Dyck03].

3.2 Taint Checking

Taint checking enforces information flow policies during application execution. Perl and Ruby are popular scripting languages that enforce information flow policies using dynamic checking. Perl's *taint*

mode provides simple rules for preventing the use of untrusted data in critical functions [PerlSec]. The general approach is to mark external data as tainted, and track the taint information through Perl expressions. Perl implements a simple rule: any tainted data in a subexpression taints the entire expression. To untaint variables requires a regular expression pattern match. This assumes that the pattern match provides the needed data validation. In contrast, we have chosen to force developers to untaint variables explicitly. The disadvantage of our approach is that it forces developers to modify their application code if they need to untaint a variable because of a false positive in our PHP injection or XSS detection code. We mitigate this potential problem by keeping track of tainting at the level of a single character in PHP strings. Armed with this information, we can selectively apply our injection detection algorithms, which is the strength and novelty of our precise tainting approach.

Ruby provides several levels of taint modes [Tho04]. Ruby's safe level 1 corresponds roughly to Perl's taint mode and our own use of taint information to prevent PHP injections. Ruby's stricter safe levels define additional restrictions on the use of tainted variables, including the ability to completely sandbox tainted variables from untainted ones. Ruby's additional safe levels are designed for a generic environment, unlike PHP which is primarily targeted for web development where distinguishing between external and trusted input is usually sufficient.

While Perl and Ruby provide a generic mechanism for tainting all variables, we have focused on maintaining taint information for strings only. In practice, we have not seen the need in PHP to taint non-string variables; in PHP external sources of data such as cookies, get, post, database and session values are all represented as strings.

Finally we note that the notion of precise tainting is complementary to the coarse-grain tainting of Perl and Ruby. Our approach for hardening web applications using precise tainting could be applied effectively to Perl, Ruby and other languages as well.

Newsome and Song [New05] used dynamic tainting at the granularity of memory locations to detect and analyze security exploits on binaries. They modified the Valgrind x86 emulator [Net03] to mark all input from untrusted sources as tainted and to keep track of tainting information as a program executes machine instructions. An attack is detected if tainted data is used as a jump target, or passed to certain system calls. As in our approach, using precise taint information enables an automated defense with few false positives. Tracking taint information at the machine level, however, is very expensive and imposes unacceptable overhead for most applications. Since our approach is at the level of programming language semantics, we are able to perform precise tainting with minimal overhead.

Huang et. al developed a hybrid approach to securing web applications [Hua04]. Their WebSSARI tool used a static analysis based on type-based information flow to identify possible vulnerabilities in PHP web applications. Input from external sources is considered tainted and a type-based static analysis is used to detect insecure uses of tainted data. Their type-based approach operates at a coarse-grain: any data derived from tainted input is considered fully tainted. In addition to reporting warnings describing the detected vulnerabilities, WebSSARI would insert calls to sanitization routines. These routines could be programmed by the programmer to filter potentially dangerous content from tainted values before they are passed to security-critical functions. Our precise tainting approach is more automated than WebSSARI; all we require is that the server uses our modified interpreter PHP and all web applications running on the server are protected. Further, our precise tainting and context-specific checking approach limits the number of false positives.

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 resulting web page contains only safe content. Our solution is fully automatic: it prevents a large class of common security vulnerabilities without any direct effort required from web application developer.

Figure 2 illustrates our system architecture. The only change from the standard LAMP architecture is we replace the standard PHP interpreter with a modified interpreter that identifies which data comes from untrusted sources and precisely tracks how that data

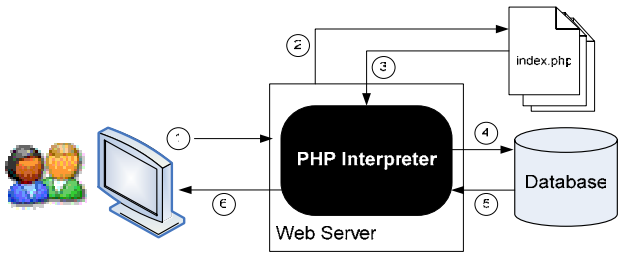

Figure 2. Modified web server architecture.

propagates through PHP code interpretation (Section 4.1), check that parameters to commands do now contain dangerous content derived from user input (Section 4.2), and ensure that generated web pages do not contain scripting code created from untrusted input (Section 4.3). Section 4.4 evaluates the performance of our prototype implementation.

4.1 Taint Marking

We mark an input from untrusted sources including data provided by client requests as tainted (step 1 in Figure 2). In PHP, all external variables are strings. 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 propagate taint information across function calls, variable assignments and composition at the granularity of a single character. We call this fine-grained level of maintaining taint information *precise tainting*. The application of precise tainting enables the prevention of SQL injection attacks and the ability to easily filter output for XSS attacks. If a function uses a tainted variable in a dangerous way, then we can either 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 forget or omit to check other variables that can be set directly by manipulating the HTTP protocol directly. Our approach ensures that *all* such external variables, e.g. hidden form variables, cookies and HTTP header information, are marked as tainted. External variables are accessible in PHP via the associative arrays $_GET, $_POST, $_COOKIE, $_REQUEST and $_SERVER. The values in $_GET, $_POST and $_COOKIE correspond respectively to the parameters passed in HTTP GET and POST requests and transmitted in cookies. The $_REQUEST array aggregates the values in the $_GET, $_POST and $_COOKIE arrays. The $_SERVER array contains variables set by the web server or related to the execution environment of the current script. We only mark as tainted those variables in $_SERVER that are sent by the client: HTTP_ACCEPT, HTTP_ACCEPT-_CHARSET, HTTP_ACCEPT_ENCODING, HTTP_ACCEPT_LANGUAGE, HTTP_CONNECTION, HTTP_HOST, HTTP_REFERER, and HTTP_USER_AGENT.

We also keep track of taint information for session variables and database results. Session variables provide a mechanism for web developers to keep track of application state information without having this information exposed and potentially manipulated by clients (provided that the session id is difficult to guess).

4.1.1 Taint Strings

For each PHP string, we track tainting information for individual characters. This is done by associating a *taint string* with each string value. The taint string can be NULL, indicating that the entire string is untainted, or can be an array of characters representing the taintedness of each character. While this is not an efficient encoding—we do not need the full 8 bit to encode taint information, and typical taint strings have long sequences of the same character—it does make for a simple implementation for our proof-of-concept prototype. (We do not yet handle multi-byte characters, although they could be handled similarly.)

To provide more precise tracking, we maintain the source of tainting in the taint string. The taint markings are:

-	untainted
G	tainted, from a GET method
P	tainted, from a POST method
C	tainted, from a cookie
D	tainted, from a database response
S	tainted, from a session variable

Consider the following code fragment where part of the string $x came from an external input and $y came from a cookie:

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

The values of $_GET['name1'] and $_COOKIE['name2'] are fully tainted (we assume they are Alice and Bob); this tainting propagates through the concatenations. After the final assignment, the value of $x and its taint markings will be:

```
Hello Alice. I am Bob.
------GGGGG-------CCC-
```

This illustrates *precise tainting*: (1) taint information is kept for each character, (2) the source of the taint is maintained, and (3) taint information is kept throughout the lifetime of a PHP script.

4.1.2 Functions

We keep track of taint information across function calls, in particular functions that manipulate and return strings. Whenever feasible we keep track of taint information at the granularity of a single character. For example, consider the substring function (tainted characters are underlined):

```
$original = "precise taint me";
$x = substr($original, 0, 6); // precise
$y = substr($original, 8, 9); // ta
```

The taint markings for the results of the substr call depend on the part of the string they select.

Table 1 lists the functions for which we keep track of fine-grained taint information. The list of functions we obtained from the PHP list of string functions [PHPs].

Type	Function Name
Formatting	sprintf, sscanf (taint information only kept for string variables)
String conversion	strtolower, strtoupper, ucfirst, ucwords, addslashes, addcslashes, stripslashes, stripcslashes
String selection	substr, str_split, strsts, stristr, strchr
White space	chop, ltrim, rtrim, trim
Other string functions	implode, explode, str_ireplace, str_pad str_replace, substr_replace, str_split

Table 1. Taint-preserving string functions.

Table 2 provides several examples of precise tainting through various string functions. For other functions the mapping between input and output characters is less obvious, so it is only feasible to keep track of taint information at a coarse level: the result strings are marked tainted if any of the arguments to the function are tainted. These include the regular expression functions (preg_grep, preg_match, preg_match_all, preg_quote, preg_replace, preg_split) as well as complex string functions like get_headers and get_meta_tags.

Unlike the approach in Perl where a regular pattern expression match is indicative of a sanitization routine and implicitly untaints a variable [PerlSec], we do not assume pattern matching expressions produce untainted results. To allow developers to circumvent the taint markings, we provide the untaint function that explicitly marks its string parameters as untainted. This is more conservative than the Perl approach in which tainting information may be lost accidentally, or an attack permitted through a faulty

Function	Result
sprintf("Bye %s", "John");	Bye John
sscanf("Bye %s","Bye John");	John
strtolower("HELLO");	hello
strtoupper("hello");	HELLO
ltrim(" hello");	hello
str_replace("bonjour", "hello", "bonjour John")	hello John
explode(":", "col1:col2")	col1, col2
implode(':', array("col1", "col2"))	col1:col2

Table 2. Precise tainting examples.

pattern expression. It increases the risk, however, that legitimate web site uses will fail because of overly strict tainting. We mitigate this by providing context sensitive checking of how tainted values are used.

In general we eschew the implicit untainting of variables as potentially dangerous. However cast operations, whether implicit or explicit, from strings to another type will untaint variables. In the following example $s is untainted:

```
$t = $_GET['aNumber']; // $x is tainted
$x = $t + 3; // implicit cast to integer
$y = (float)$t; //explicit cast to float
$s = "x = $x  y = $y"; // $s is untainted
```

4.1.3 Database values

Databases provide another potential venue for attackers to insert malicious values. Three options for dealing with are to (1) disallow tainted data from being stored into the database, (2) maintain tainting information in the database, or (3) consider data received from the database untrustworthy. The first option is too strict and would break the functionality of many web applications. Applications typically use the database to keep track of user input, so it is not reasonable to disallow storing tainted data in the database. A variation could attempt to check stored data for potentially dangerous content; however, as we will see in several examples in Sections 4.2 and 4.3, it is not possible to safely determine if content is dangerous without knowing about the context in which it will be used.

The second option would provide the most precise tainting information, but would require either modifications to the database server or modifications of the application database (for example, keeping a shadow table corresponding to the tainting information for each entry in the data table). Neither implementation option is desirable. Modifying the database server would tie applications to a particular database server. Modifying the application would involve extensive rewriting of all database commands. This could be automated, but the added complexity would increase the likelihood of security vulnerabilities.

Hence, we opt for the third option and 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 for typically web applications. The other advantage of this option, is that if the database is compromised by some other means, the attacker is still not able to use the compromised database to construct a cross-site scripting attack.

We modified the PHP functions that return data from the database (mysql_fetch_array, mysql_fetch_row, mysql_fetch_assoc, mysql_result) to automatically mark all characters in the resulting string values as tainted. While we focus on MySQL, it is straightforward to modify the analogous functions for other databases.

4.1.4 Session variables

The stateless nature of HTTP (HyperText Transfer Protocol), the protocol that underlies the world wide web, requires developers to keep track of application state across client requests. However, exposing session variables to clients would allow attackers to manipulate applications. Thus well-designed web applications will keep session variables on the server only and use a session id to communicate with clients. Several steps are commonly used to protect the session id, including encrypting traffic by using HTTPS (HTTP over Secure Socket Layer), generating session ids that are difficult to predict, and timing out sessions.

By default, PHP stores and retrieves session variables in files. We modified PHP to store and retrieve tainting information as well. When the session variable is read from the file, its taint markings are restored. PHP allows developers to replace the default implementations with their own handlers, for example by storing session state in shared memory or a back-end database. Since this will circumvent our modified handlers, developers using this approach will either lose tainting information or need to modify their handlers to maintain it.

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 allow typical web applications to function without false positives.

4.2.1 PHP Injection

To prevent PHP injection attacks we disallow calls to the functions shown in Table 3 if one of their arguments is tainted. These are similar, though not as extensive, as the functions disallowed by Perl's taint mode and Ruby's taint level 1. Unlike the SQL injection and cross-site scripting checking, checking here is done at a coarse level: any tainted input to these functions is too dangerous to allow.

Type	Functions
Program execution	system, exec, passthru, shell_exec, proc_open
Filesystem	fopen, glob, mkdir, popen, readfile, file
PHP language	eval, include, require, include_once, require_once

Table 3. Taint-checked PHP functions.

Both attacks described in Section 2.1.1 are thwarted as we propagate tainting information from the first GET or POST variables, through string manipulation functions and database operations, to their eventual use in the arguments to require and eval. In the PhpGedView example [Jei04], the argument to require is marked tainted.

In the phpMyAdmin attack example, the database table name $_table is marked as tainted as it comes from the result of a database query (in this case, to the trojan database setup by the attacker). The value of $eval_string is indirectly derived from $_table and is thus partially tainted. Hence, the call to eval is not permitted and the attack is unsuccessful.

4.2.2 SQL Injection

To prevent SQL injection attacks, we modify the PHP functions that can send commands to the database (mysql_query and mysql_unbuffered_query) to check for the presence of special SQL tokens in the tainted portion of the query string before allowing the command to be sent to the database. Our algorithm for detecting SQL injection involves four steps:

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

While step 2 appears simple, its implementation is quite complicated and potentially error prone. It must deal with various complex quoting rules in SQL commands. We adopted the scanner from the Postgres implementation (which appeared easier to extract and modify than the mySQL scanner). We stripped the Postgres scanner of any code related to parsing and kept just the tokenizer.

Our approach would prevent certain sites from working properly, especially those whose application semantic is to explicitly allow users to issue SQL queries, e.g., PhpMyAdmin. For these, developers need to call untaint explicitly.

Using the examples in Section 2.1.2, we show how our techniques defend against SQL injections. Recall the first example:

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

in which the attacker provides a malicious value for $user: ' OR 1 = 1 ; -- '. The resulting query string is tainted as follows:

```
SELECT user FROM users WHERE
  user = '' OR 1 = 1 ; -- ' AND password = 'x'
```

[1] The list of keywords was obtained by merging the keywords found in the MySQL and Postgres scanners. As an additional preventative measure we could also add critical table names to this list or potentially harmful stored procedures. In our inspection of SQL injection exploits to date, we have found that the existing rules suffice.

We detect the injection since OR is tainted and a keyword (step 4 of our algorithm). Note that we would also have detected the injection at the =, ; or -- tokens. After detecting the injection, our modified PHP interpreter returns an error result from mysql_query, instead of sending the command to the database.

In the second example, the query string is:

```
SELECT s.sid, a.url FROM nuke_stories s,
                         nuke_authors a
  WHERE s.aid=a.aid OR mid(a.pwd,1,1)=6
```

Injection is again detected because OR is tainted. In contrast to the ad hoc filtering in the original patch of PhPNuke, our approach stops the attack using not just parenthesis but a comprehensive list of SQL tokens.

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 accomplish this by modifying 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.

Further, our checking involves whitelisting safe content rather than blacklisting dangerous content. Whereas a blacklist attempts to prevent cross-site scripting attacks by identifying known dangerous tags, such as <SCRIPT> and <OBJECT>. This fails to prevent script injection involving other tags. For example, a script can be injected into the apparently harmless <B> (bold) tag using its parameters. The text below would enable an attacker to steal the victim's cookie without using any script tags of apparent calls to JavaScript routines:

```
<b onmouseover=
  'location.href ="http://evil.com/steal.php?"
  + document.cookie'>
```

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.

4.4 Performance

Our current prototype implementation implements precise tainting and performs the checking necessary to prevent command injection and cross-site scripting attacks. We report its performance on both micro-benchmarks designed to stress the modified PHP interpreter, and benchmarks typical of web applications. Our prototype implementation is designed as a simple proof-of-concept, and there would be many opportunities to improve performance in a production implementation. Nonetheless, its performance is adequate for typical web applications.

Table 4 presents results from some micro-benchmark tests. Each micro-benchmark measures the time required to execute 10,000 iterations of a loop body that executes specific PHP functions. This illustrates the worst-case performance of our mechanisms. The highest measures overhead is 77% for the sql.php micro-benchmark which isolates the SQL injection checking. It creates a partially tainted string and passes it to the function that checks SQL commands.

Benchmark	Standard PHP (ms)	Tainting PHP (ms)	Overhead
concat.php	16.57	17.10	3.2%
print.php	117.78	171.56	45.7%
sql.php	32.73	57.88	76.8%

Table 4. Micro-benchmark Results.
Each time is the average for 100 executions. The concat.php benchmark concatenates tainted strings; the print.php benchmark prints partially tainted strings; the sql.php benchmarks checks a partially tainted SQL command (but does not send a request to the database).

We also measured our system overhead by testing typical web application requests: processing a login, entering a message and generating an output page from the contents of a database table. Table 5 summarized the impact of our modified interpreter on the request rate. For all the application benchmarks the measured overhead was below 5%. With some simple optimizations to our implementation, this could be reduced.

Benchmark	Standard PHP (req/s)	Tainting PHP (req/s)	Overhead
login.php	69.7	67.9	2.6%
message.php	32.8	31.8	2.9%
members.php	25.0	23.9	4.9%

Table 5. Application benchmark Results.
Each time is the average for 100 requests. We use Flood [Flood04] to test the application benchmarks by starting the modified and normal server on the same machine using different ports. The login.php benchmark tests a login that involves constructing a SQL query and generating a welcome page. The message.php benchmark posts a message. The members.php benchmark executes several SQL queries and generates a large output page containing 100 entries.

5. CONCLUSION

We have described a fully automated, end-to-end approach for hardening web applications. By exploiting precise tainting 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.

The Internet continues to be riddled with insecure web applications, despite much work on detecting vulnerabilities and hardening web applications. Effective solutions 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.

ACKNOWLEDGEMENTS

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

REFERENCES

[Arm2003] Lucas Armstrong. *PHPNuke SQL Injection*. Bugtraq message, 20 February 2003.

[Bell73] D. E. Bell and L. J. LaPadula. *Secure Computer Systems: Mathematical Foundations*. MITRE Corporation MTR-2547, Vol. 1. 1973.

[Bene02] M. Benedikt, J. Freire and P. Godefroid. *VeriWeb: Automatically Testing Dynamic Web Sites*. WWW 2002. May 2002.

[Biba77] K. J. Biba. *Integrity Considerations for Secure Computer Systems*. USAF Electronic Systems Division ESD-TR-76-372. April 1977.

[CERT00] CERT® Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests. February 2, 2000. http://www.cert.org/advisories/CA-2000-02.html

[Dyck03] Timothy Dyck. *Review: Appshield* and *Review: Teros-100 APS 2.1.1*. eWeek. May 2003. http://www.eweek.com/article2/0,3959,1110435,00.asp

[Endler02] David Endler. *The Evolution of Cross-Site Scripting Attacks*. iDEFENSE Labs. May 2002. http://www.cgisecurity.com/lib/XSS.pdf

[Eva02] David Evans and David Larochelle. *Improving Security Using Extensible Lightweight Static Analysis*. IEEE Software. Jan/Feb 2002.

[Eye02] EyeonSecurity. *Microsoft Passport Account Hijack Attack: Hacking Hotmail and More.* Hacker's Digest, Winter 2002.

[Fisk04] Harrison Fisk. *Prepared Statement*. 2004. http://dev.mysql.com/tech-resources/articles/4.1/prepared-statements.html

[Fuecks02] Harry Fuecks. *Magic Quotes and Add Slashes in PHP*. 2002. http://www.webmasterstop.com/tutorials/magic-quotes.shtml

[Gentoo04] *Gallery PHP Injection*, Gentoo Linux Security Advisory # 200402-04,

Feb 2004. http://www.linuxsecurity.com/advisories/gentoo_advisory-4015.html

[Grey04] GreyMagic Software. *Remotely Exploitable Cross-Site Scripting in Hotmail and Yahoo*. March 2004. http://www.greymagic.com/security/advisories/gm005-mc/

[Hog2004] Greg Hoglund and Gary McGraw. *Exploiting Software: How to Break Code*. Addison-Wesley. 2004.

[Hua03] Yao-Wen Huang, S. K. Huang, T. P. Lin, C. H. Tsai. *Web Application Security Assessment by Fault Injection and Behavior Monitoring.* WWW 2003. May 2003.

[Hua04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D. T. Lee and Sy-Yen Kuo. *Securing Web Application Code by Static Analysis and Runtime Protection.* WWW 2004. May 2004.

[Ivgi04] Rafel Ivgi. *Cross-Site-Scripting Vulnerability in Microsoft.com*. Full Disclosure mailing list. 4 October 2004.

[Jei04] JeiAr, *PhpGedView PHP Injection*, Jan 2004. http://xforce.iss.net/xforce/xfdb/14205

[Kavado04] Kavado, Inc. *InterDo Web Application Firewall*. 2004. http://www.kavado.com/products/interdo.asp

[Kost04] Stephen Kost. *An Introduction To SQL Injection Attacks For Oracle Developer*s. Integrigy Corporation White Paper. January 2004. http://www.net-security.org/dl/articles/IntegrigyIntrotoSQLInjectionAttacks.pdf

[Ley04] Jim Ley. *Simple Google Cross Site Scripting Exploit.* 17 October 2004.

[Litchfield03] David Litchfield. *SQL Server Security*. McGraw-Hill Osborne Media. August 2003.

[Manip04] Manip. *Centre 1.0 PHP Injection*. July 2004. http://seclists.org/lists/fulldisclosure/2004/Jul/0088.html

[Netcraft04] Netcraft Survey. November 2004. http://news.netcraft.com/

[Net03] Nicholas Nethercote and Julian Seward. *Valgrind: a program supervision framework.* Workshop on Runtime Verification. July 2003.

[New05] James Newsome and Dawn Song. *Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software*. To appear in NDSS 2005. February 2005.

[PerlSec] *Perl 5.6 Documentation: Perl Security*. http://www.perldoc.com/perl5.6/pod/perlsec.html

[PHP5] *Improved MySQL Extensions*. http://www.php.net/manual/en/ref.mysqli.php

[PHPs] *PHP Manual: String Functions*. http://www.php.net/manual/en/ref.strings.php

[PHPsec] *PHP Manual: Security*. http://us2.php.net/manual/en/security.php

[Rai98] Rain.forest.puppy. *NT Web Technology Vulnerabilities*. Phrack Magazine. Volume 8, Issue 54. December 1998.

[Ricca01] F. Ricca, P. Tonella. *Analysis and Testing of Web Applications.* IEEE International Conference on Software Engineering. May 2001.

[Sab03] Andrei Sabelfeld and Andrew C. Myers. *Language-Based Information-Flow Security*. IEEE Journal on Selected Areas in Communications. January 2003.

[Sco02] David Scott and Richard Sharp. *Abstraction Application-Level Web Security.* WWW 2002. May 2002.

[Shan01] Umesh Shankar, Kunal Talwar, Jeffrey Foster and David Wagner. *Detecting format-string vulnerabilities with type qualifiers.* USENIX Security Symposium 2001.

[Simb04] Nasir Simbolon. *PHPmyAdmin critical bug*. http://xforce.iss.net/xforce/xfdb/16542

[Spett02] Kevin Spett. *SQL Injection: Are your web applications vulnerable?* SPI Labs White Paper. 2002. http://www.spidynamics.com/whitepapers/WhitepaperSQLInjection.pdf

[Teros04] Teros, Inc. *Teros-100 Application Protection System.* 2004. http://www.teros.com/products/aps100/aps.shtml

[Tho04] Dave Thomas with Chad Fowler and Andy Hunt. *Programming Ruby: The Pragmatic Programmer's Guide, Second Edition.* Pragmatic Programmers. 2004.

[Val02] Janet Valade. *PHP & MySQL for Dummies*. Wiley Publishing. 2002.

[Watch04] Watchfire Corporation. *AppShield 4.5 Technical Overview*. 2004. http://www.watchfire.com/resources/appshield-overview.pdf

[Wei04] Nitzan Weidenfeld. *Security hole found in Gmail.* Nana NetLife Magazine. 27 October 2004. http://net.nana.co.il/Article/?ArticleID=155025&sid=10

[Więsek03] Karol Więsek, *GOnicus System Administrator PHP Injection*. Feb 2003. http://lists.netsys.com/pipermail/full-disclosure/2003-February/003932.html