Confeagle: Automated Analysis of Configuration Vulnerabilities in Web Applications

Conference Paper · June 2013

DOI: 10.1109/SERE.2013.30

4 authors, including:

Birhanu Eshete
University of Michigan, Dearborn, United States
29 PUBLICATIONS **381** CITATIONS

SEE PROFILE

Komminist Weldemariam
IBM Research, Africa
99 PUBLICATIONS **587** CITATIONS

SEE PROFILE

2013 7th International Conference on Software Security and Reliability

Confeagle: Automated Analysis of Configuration Vulnerabilities in Web Applications

Birhanu Eshete, Adolfo Villafiorita
Fondazione Bruno Kessler
Trento, Italy
{*eshete, adolfo.villafiorita*}*@fbk.eu*

Komminist Weldemariam, Mohammad Zulkernine
Queen's University
Kingston, Ontario Canada
{*weldemar, mzulker*}*@cs.queensu.ca*

***Abstract*—Web applications and server environments hosting them rely on configuration settings that influence their security, usability, and performance. Misconfiguration results in severe security vulnerabilities. Recent trends show that misconfiguration is among the top critical risks in web applications. While effective at uncovering numerous classes of vulnerabilities, generic web application vulnerability scanners are limited in identifying configuration vulnerabilities. In this paper, we present an approach that effectively combines hierarchical configuration scanning and preliminary source code analysis of web applications to pinpoint potential configuration vulnerabilities, quantify the degree of severity based on standard metrics, and facilitate fixing of vulnerabilities found therein. We implemented our approach in a tool called `Confeagle` and evaluated it on 14 widely deployed PHP web applications. Unlike generic web vulnerability scanners, on the subject applications, `Confeagle` detected potential configuration vulnerabilities that could result in information disclosure, denial-of-service, and session hijacking attacks on the applications.**

***Keywords*- security configuration, web applications, vulnerability analysis, Common Configuration Scoring System (CCSS).**

I. Introduction

Web applications are mainstream in online service delivery. They consist of components that rely on client-side and server-side technologies. While main features are developed in-house, a significant number of third-party components are used to complete functionalities of web applications.

A combination of security-unaware development practices and reliance on third-party components have made web applications vulnerable to a number of attacks [1]–[5]. Among the vulnerabilities, security misconfiguration is one of the most prominent. In the recent past, security misconfiguration has become a real concern for the security of web applications. Misconfiguration may happen at different layers of the web architecture while misconfiguration at environment and application levels are the common ones.

To detect web application vulnerabilities, a number of approaches have been proposed. One common technique is to use black-box vulnerability analysis to scan the application by simulating an attack. In this regard, a number of vulnerability scanners are proposed [6]–[8]. A complementary technique is white-box vulnerability analysis that involves automatic and/or manual inspection of application source code to pinpoint vulnerabilities before the application is deployed [9], [10]. A gray-box approach combines the strengths of both to conduct vulnerability analysis.

While existing vulnerability analysis techniques are effective at detecting critical vulnerabilities, little attention has been paid to configuration vulnerability in spite of its criticality. An attack vector coverage based comparison of black-box web vulnerability scanners by Bua et al. [8] confirmed that configuration vulnerability is among the least covered (2.5%) vulnerabilities compared to other vulnerabilities such as XSS (20%), SQL Injection (10%), and Information Leakage (47%).

On the other hand, configuration vulnerability consistently ranks among the top 10 web application security risks [11]. In the case when configuration vulnerability is considered, it is mostly limited to the deployment environment [12]–[14]. This is a practical bottleneck because an invulnerable deployment environment does not guarantee an invulnerable application from configuration perspective. The main reason is that a number of security critical configuration settings can be overridden by applications statically or at runtime. For instance, in PHP web applications, the global configuration settings of Apache and PHP could be redefined either at the level of directories (e.g., using *htaccess* files) or at the source code level in scripts (e.g., using the *ini_set()* function).

In this paper, we present a novel approach to supplement existing web vulnerability analysis techniques by automatically scanning web application directory hierarchy and application source code to detect, quantify, and fix configuration vulnerabilities before deployment. A standard metrics based on the Common Configuration Scoring System (CCSS) [15] is used to quantify the severity of configuration vulnerabilities based on which finer-grained vulnerability reports are generated. By doing so, we ensure that an invulnerable application is deployed on an invulnerable environment.

We have implemented our approach in a tool called `Confeagle` and evaluated it on 14 widely deployed open source PHP applications. The results of running our tool on default installations of these applications suggest that 10 out of the 14 have one or more configuration vulnerabilities. Using the results of our approach, we associate the vulnerabilities with potential attacks they are likely to result in (such as information disclosure and session hijacking attacks).

978-0-7695-5021-3/13 $26.00 © 2013 IEEE
DOI 10.1109/SERE.2013.30

We compared Confeagle with 3 popular web vulnerability scanners for detecting configuration vulnerabilities. According to the results, while the scanners are precise at pinpointing configuration vulnerabilities at the environment level, they fell short of identifying the equivalent vulnerabilities at the application level, which our tool was able to report.

Our approach benefits Web Developers and Web Masters. For Web Developers, it supplements test-driven and security-aware development of web applications while for Web Masters it guarantees that they are not going to deploy web applications with configuration vulnerabilities. In a nutshell, the contributions of this paper are the following:

- We proposed a novel approach that reinforces existing web vulnerability analysis techniques by automatic scanning of web application directory hierarchy and application source code to detect, quantify, and fix configuration vulnerabilities.
- We put into practice the CCSS to quantify vulnerability scores by introducing additional metric variables relevant to configuration in web applications.
- We implemented and evaluated our approach on 14 most widely adopted open source PHP applications and compared its effectiveness with 3 popular web vulnerability scanners.

This paper is organized as follows. Section II discusses exemplified background on configuration vulnerability in web applications. In Section III, we first highlight our approach and then we present technical details focusing on the novel aspects. In Section IV, we describe the subject applications and the experimental setup. We discuss evaluation results in Section V. In Section VI, we compare our approach with related work. Finally, Section VII concludes the paper.

II. Configuration Vulnerabilities in Web Applications

In this section, we discuss background on configuration vulnerabilities in web applications by focusing on configuration vulnerabilities that may arise from customization of web applications at configuration and functionality level. While our discussion applies to applications developed using other technologies (e.g., Java, Python), for the sake of illustration, we use PHP applications here.

A. Configuration in Web Applications

Among other configuration settings at different layers, a typical PHP application is governed by configurations at the deployment environment and at the application level. Configuration at the environment level usually involves manipulating configuration directives in the *httpd.conf* and *php.ini* configuration files of Apache HTTP server and PHP interpreter, respectively. It is also a common practice to define configuration settings of MySQL server in the *php.ini*.

When deploying web applications, ensuring configuration invulnerability of the deployment environment is sufficient in a situation where application specific configurations are absent. However, practice shows that most applications override global configuration settings at the level of directories or scripts within the application. For instance, in PHP, directory level configuration settings may be overridden using the *htaccess* file and script specific configuration settings may programmatically be redefined at runtime using the *ini_set()* function. In such a scenario, configuration vulnerability has to cover not only the environment but also the application. Otherwise, configuration overriding may leave the application vulnerable to attacks. In addition to customizing global configuration settings, popular web applications (e.g., Joomla, WordPress) undergo customization that may introduce further configuration vulnerabilities.

As part of ensuring safe configuration, installation scripts of some web applications (e.g., Joomla, phpBB, Moodle) include selective configuration setting analysis for the deployment environment. While this kind of pre-installation configuration analysis guarantees partial safety, it does not guarantee configuration invulnerability neither at the environment nor at the application level. In the next section, we illustrate possible exploit scenarios that could happen due to silly mistakes in configuration and/or application customization using Joomla and WordPress as examples.

B. Exploit Scenarios

To illustrate how a few flips in configuration may lead to vulnerabilities, we installed Joomla 2.5 and modified the *php.ini* with *register_globals=on*, *allow_url_fopen=on*, and *allow_url_include=on*. In fact, Joomla installation includes checks for certain configuration settings in *php.ini* and suggests the recommended values. Nevertheless, these values could be changed anytime before the application is deployed.

Joomla allows adding custom components to extend its features. To simulate a customization, we created a new component named *com_badone* under the *components/* directory. Under each component is a PHP script named after the component name (*badone.php* in this case) which is used to invoke the component. Now, suppose that in *badone.php*, we have the following piece of code:

```
if (isset($_REQUEST["theFile"])){
    $file_name = $_REQUEST["theFile"];
}
require_once($file_name);
```

Listing 1: Example of a vulnerability in a customized Joomla application due to a little customization fault.

As can be seen from Listing 1, the *badone.php* script takes a file name as parameter (through *theFile*) and then includes it in the web page. Then, the index page is accessed using the request

http://host/Joomla/index.php?option=com_badone&theFile=http://evilsite.com. This request loads the http://evilsite.com page within the http://host/Joomla/index.php. Such an attack is called Remote File Inclusion caused by enabling the *allow_url_fopen* (which allows treating URLs as files) and *allow_url_include* (which uses *require_once* to open URLs as files). Similarly, by changing the value of *theFile* to the path of an easily guessable path on the server, such as */etc/passwd* in Unix, the file containing all the hash values of passwords is revealed resulting in Information Disclosure attack.

Another common attack that may result from exploiting configuration vulnerabilities in web applications is Cross-Site Scripting (XSS). We show an example of this attack in WordPress. In particular, we installed WordPress 3.0 on top of PHP Version 5.4.4. For demonstration, we enabled the *display_errors* directive using the PHP runtime configuration function (i.e., *ini_set*) and inserted the *error_reporting* function with error report level of *E_ALL* to report all errors.

```
if (isset($_GET['message']) && (int) $_GET['
    message'] ) {
    $message = $messages[$_GET['message']];
    $_SERVER['REQUEST_URI'] =
        remove_query_arg(array('message'),
        $_SERVER['REQUEST_URI']);
}
```

Listing 2: Example of vulnerable code pattern in WordPress.

A potential vulnerable code pattern in the *upload.php* script of WordPress is shown in Listing 2. In this code, the *message* field serves as an index for the *messages* variable, that holds an array of messages. When the *upload.php* page is loaded, an appropriate text is displayed depending on the value of the *message* field that is passed through the *$_GET* variable. For example, we accessed the upload page with http://host/WordPress/wp-admin/upload.php?message=1, and *"Media attachment updated."* text (i.e., *$messages[1]*) is displayed. A notice text is generated when the *message* field is supplied with a value outside the range 0 to 5. The generated output notices are not fully sanitized. By taking advantage of this, we modified the previous page request as http://host/WordPress/wp-admin/upload.php?message=1;<script>alert(7-1)</script>. In effect, the request successfully rendered a pop-up alert that displays the result of 7-1. This is an evidence for other potential XSS attacks. Thus through the XSS flaw on the *upload.php* page and by controlling part of the notice message, an attacker can craft and inject arbitrary HTML and JavaScript code into the page.

Most real life web applications are customized based on popular web applications. Customization usually involves modifying: some configuration settings, look and feel, and server-side code related to crucial features such as authentication and search. Unless carefully done, little mistakes in customization result in potential security vulnerabilities such as the ones we just demonstrated. From the examples discussed so far, it is clear that it requires a little configuration fault in the customization of popular web applications (such as Joomla and WordPress) to expose vulnerabilities which could be exploited by an attacker. As we shall show in our evaluation in Section V, let alone customized versions, even default installations of these popular web applications suffer from configuration specific and other vulnerabilities.

III. Approach to Analyze Configuration Vulnerabilities

Given a web application and a deployment environment, our approach scans configuration of the environment and the application (through exhaustive traversal of application directory and preliminary source code analysis), analyzes configuration vulnerabilities using a standard vulnerability scoring metrics, generates fine-grained vulnerability report, and facilitates automated fixing of configuration vulnerabilities.

Figure 1 shows workflow of our approach with three phases: configuration parsing, configuration vulnerability analysis, and configuration fixing. The configuration parsing phase accepts a web application and deployment environment to parse the respective configuration details to generate structured list of configuration directives. The configuration vulnerability analysis phase is the core of the approach whereby parsed configuration values are asserted against recommended configuration. A fine-grained configuration vulnerability analysis report is generated after the analysis with vulnerability scores based on a standard metrics. The configuration fixing phase automatically reconfigures the environment and the application based on the vulnerability report and the recommended configuration. In the remainder of this section, we describe these phases in detail.

A. Configuration Parsing

The goal of this phase is to collect configuration artifcats. More specifically, given a web application and deployment environment, we parse security sensitive configuration settings of the environment and the application. For the application, two classes of configuration settings have to be parsed. These are the directory specific configurations (set using *htaccess* files) and those settings executed at run time (located in the various scripts in the application). Similarly, the environment specific configuration settings are collected from Apache and PHP configuration files.

All the three parsers in Figure 1 scan the respective locations to generate Key-Value pairs, where "Key" is the name of configuration directive and "Value" is its value. The

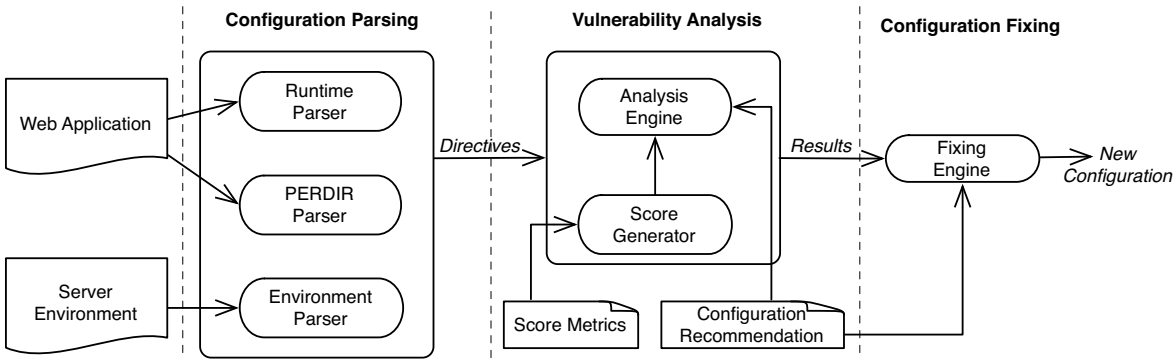

Figure 1: Workflow of the proposed approach.

`Environment Parser` consumes environment configuration files (e.g., *http.conf*, *php.ini*) and additional parameters to collect configuration values. The `PERDIR Parser` recursively locates directory specific configuration files (e.g., *htaccess*) in the directory hierarchy of the application. The `Runtime Parser` also recursively performs preliminary source code analysis to locate lines of code where configuration directives are assigned a value at runtime (e.g., in PHP applications *ini_set()* is used to set directive values at runtime).

In addition to the key-value pairs collected for each configuration directive, context information about directives is also collected. For example, context may indicate whether a directive is a server configuration, directory-level configuration, or script-level configuration.

The parsing outputs of application's configuration directives (i.e., results from the `Runtime` and `PERDIR` parsers) are organized inline with the application's directory structure along with additional metadata on how often each directive appears in each directory and in each script.

B. Configuration Vulnerability Analysis

The goal of this phase is to analyze the values of each configuration directive against a "Gold Standard" of configuration recommendations, quantify potential configuration vulnerability based on a standard configuration vulnerability scoring system, and generate not only fine-grained but also intelligible vulnerability report so as to facilitate automatic fixing of configuration vulnerabilities.

1) Analysis Engine: More formally, suppose that n is the number of configuration directives and the set of directives is denoted as $C = \{c_1, c_2, c_3, \ldots, c_{n-1}, c_n\}$, where each c_i stands for a tuple with $<directive\_name, recommended\_value, platform, interpreter\_version, web\_server\_version, db\_server\_version, description, remark>$. For each c_i, we determine whether or not c_i is in conformance with the recommended value in the "Gold Standard" which we compiled based on a thorough survey of configuration best practices [16]–[19], official documentations [20]–[22], and expert opinions [23], [24]. When a configuration directive has a value recommended by the "Gold Standard", we say that it is *safely-set*, otherwise, we deem it as *unsafely-set*.

Notice that the task of configuration vulnerability analysis is way beyond a mere value comparison. The different types of configuration directives require different sub-analysis steps to verify their conformance to the "Gold Standard". Depending on the purpose of configuration directives, values may come in different forms such as flags (e.g., *expose_php=On*), exact values (e.g., *variables_order=EGPCS*), limits (e.g., *upload_max_filesize=20M*), enumerated list (e.g., *disable_functions=exec,show_source*), and file paths (e.g., *session.save_path=/home/sessions/*). For configuration directive values indicating file paths, checking the presence of a path is not enough to confirm configuration safety. Attributes of file paths should further be checked for their conformance with the "Gold Standard" (e.g., path should not be world-writable). Some configuration recommendations vary across platforms (e.g., operating systems) and across different versions of web servers, script interpreters, and database servers. Dealing with such differences in platforms and versions while analyzing configuration vulnerabilities is also an integral part of this phase.

2) Configuration Vulnerability Scoring: For the sake of vulnerability scoring, we adopted the Common Configuration Scoring System (CCSS) of NIST [15], [25] which suggests combining exploitability metrics with impact metrics to quantify vulnerability score of a certain configuration issue. In our approach, a vulnerability issue is a configuration directive that is *unsafely_set* and exploitability is estimated against the degree to which unsafe configuration value makes the application (environment) vulnerable to attacks. As a reference heuristic, we defined a metric based on rigorous examination of exploitability and impact of configuration directives taking the "Gold Standard" as a reference. The heuristic may be modified whenever the need arises without affecting the configuration vulnerability analysis task.

Exploitability (Exp) is composed of three sub-metrics: Access Vector (AV), Authentication (Au), and Access Complexity (AC). Based on the access required to exploit the configuration directive, AV is rated as Local[1] (0.395), Adjacent Network (0.646), or Network (1.0). For instance, if the current value of a configuration directive allows exploiting the application remotely, the AV is assigned "Network" which equates to the highest level of severity. Depending on the number of required authentications to exploit the application, Au is rated as None (0.704), Single (0.56), or Multiple (0.45). The AC is rated as Low (0.71), Medium (0.61), or High (0.35) depending on the level of difficulty of actions to exploit the application.

Impact (Imp) is estimated by combining confidentiality, integrity, and availability. The Confidentiality Impact (CI), Integrity Impact (I), and Availability Impact (AI) are all rated as None (0.0), Partial (0.275), or Complete (0.660) pertinent to the degree to which each of these security requirements is compromised due to a certain configuration issue. The vulnerability score for a security directive which we call impact coefficient (φ) in our approach is computed as:

$$\varphi = ((0.6 \times Imp) + (0.4 \times Exp) - 1.5) \times f(Imp) \quad (1)$$

where:

$$Imp = 10.41 \times (1 - (1 - CI) \times (1 - I) \times (1 - AI)) \quad (2)$$

$$Exp = 20 \times AV \times Au \times AC \quad (3)$$

$$f(Imp) = \begin{cases} 0 & \text{if } Imp \text{ is } 0 \\ 1.176 & \text{Otherwise} \end{cases} \quad (4)$$

The value of the impact coefficient φ is rounded to give severity scores ranging 0 to 10.

For example, consider the directive *allow_url_fopen* with value *On* in the default deployment of Drupal. In a nutshell, enabling this directive means allowing opening of URLs as files (using functions like *fopen()*, *file_get_contents()*) which, combined with bad input filtering, could result in code injection attacks. This configuration issue could be exploited from a remote location for which AV is 1.0 (Network) and Au is 0.704 (None) as it does not require authentication to initiate an attack based on this issue. The access complexity is moderate for which AC is 0.610 (Medium). With regards to impacts, an exploit that takes advantage of this issue could lead to severe impact on: confidentiality (e.g., injected code reveals customer records), integrity (e.g., injected code changes records in a database), and availability (e.g., injected code disables crucial features). As a result, the values of CI, I, and AI is set to the most severe value, 0.66, meaning the impact is potentially complete compromise of confidentiality, integrity, and availability. Applying Equation 1, the φ value we get is 9 which correlates to the level of severity of the vulnerability. Similarly, we computed φ for all security sensitive directives of Apache, MySQL and PHP.

[1]In the CCSS standard of NIST, all the ratings are assigned metric values to quantify scoring of configuration vulnerabilities.

Configuration vulnerability score at the level of an application is computed as follows. Let $D = \{d_1, d_2, d_3, \ldots, d_m\}$ be the set of m distinct directories in a given application. Based on the impact coefficient in Equation 1, we compute the weighted aggregate of vulnerability score $V(d_k)$ for a directory $d_k \in D$ in a web application as follows:

$$V(d_k) = \frac{\sum_{i=1}^{n}[\varphi_{c_i} \times (f_{c_i} \times \sigma_{c_i})]}{\sum_{i=1}^{n}(f_{c_i} \times \sigma_{c_i})} \quad (5)$$

In Equation 5, n is the number of distinct directives in directory D_k, φ_{c_i} is the impact coefficient of directive c_i, f_{c_i} is the frequency of directive c_i in directory d_k, and σ_{c_i} is the dependency coefficient of directive c_i, which precisely refers to a magnitude of influence of a directive on other directives. A dependency coefficient σ_{c_i} value of k ($k \geq 2$) for directive c_i means that there exist k distinct directives that depend on c_i. A special value of $\sigma_{c_i} = 1$ is assigned to a directive with no dependents.

The weighted average is used to compensate the presence of any bias since we might expect some variations in the relative contribution of certain directives to the overall vulnerability score of a directory in the application under analysis. Namely, $f_{c_i} \times \sigma_{c_i}$ values with larger weights contribute more to the weighted average and $f_{c_i} \times \sigma_{c_i}$ values with smaller weights contribute less to the weighted average.

Table I: Example: Vulnerability report for the *UpgradeWizard* directory of the SugarCE application. The numbers in bracket indicate number of occurrences a directive is unsafely used within that file. The aggregate vulnerability score for the directory is computed using Equation 5.

Directory : SugarCE/modules/UpgradeWizard		
Directive Name	**Vulnerability Score [0-10]**	**Filename**
display_errors	8	setup.php (2)
memory_limit	8	setuplib.php (1)
display_errors	8	installlib.php (1)
Aggregate Vulnerability Score for this directory: 8		

Finally, the configuration vulnerability report is generated at the level of the deployment environment and the application. For both cases, vulnerability scores are color coded to easily spot the most critical ones. Color coding is based on three vulnerability score ranges: low (green), medium (yellow), and red (severe) —see Table I. In addition to color coding, the vulnerability report is broken down into the level of directories to give finer-grained details of potential vulnerabilities under individual directories.

C. Configuration Vulnerability Fixing

Once the reported configuration vulnerabilities are evaluated, then vulnerabilities are patched by applying the

Table II: Subject web applications used in our experiment.

Application	Version	Type	#PHP files	# htaccess files
Joomla	2.5	Content management	1459	0
Drupal	7.18	Content management	115	1
WordPress	3.5	Blogging platform	427	0
phpBB	3.0.11	Forum engine	333	6
phpMyAdmin	3.5.5	DBMS front-end	340	3
SquirrelMail	1.4.22	Webmail package	294	11
osCommerce	2.3.3	Online shop engine	545	11
Gallery	3.0.4	Photo album organizer	509	4
Elgg	1.8.11	Social networking engine	1203	1
Moodle	2.4	e-learning platform	5488	0
SugarCRM	6.5.8	Customer relations management	294	11
myBB	1.6.9	Forum engine	318	0
MediaWiki	1.20.2	Wiki platform	1523	10
TestLink	1.9.5	Test management platform	1055	10

necessary fixes with respect to recommended configurations. Notice that fixing those vulnerable directives manually is a tedious process and often requires extensive knowledge of the environment and the application structure. Thus, the `Configuration Fixer` component has at its disposal the "Gold Standard" (supplied as XML representation) along with the metadata collected at the configuration parsing phase —to automatically fix the identified configuration vulnerabilities.

Fixing configuration vulnerabilities is way more complex than replacing *unsafe* values with *safe* values. This is because automatic fixing of configuration vulnerabilities requires sub-fixing phases such as changing (to safe values) of permissions and the creation of unavailable files whenever possible. In fact, absolute automation may not be achieved in this phase because of platform-specific issues (e.g., file ownership). In such a case, a very minimal manual intervention is affordable to complete the configuration vulnerability fixing task. Finally, the configuration analysis component is run to verify that the vulnerabilities are gone and the application is functioning as expected.

IV. Subject Applications and Experimental Setup

In this section, we describe subject applications and experimental procedure used to evaluate our approach.

A. Subject Applications

We used the most widely adopted open-source PHP web applications based on which millions of customized web applications are developed and deployed on the Web. To balance the mix of application types, we considered 12 distinct classes of web applications (see the "Type" column of Table II).

The number of *htaccess* files shown in the last column of Table II confirms that configuration is overridden at different levels of the application hierarchy. While not all configuration overriding affects security, a significant number of security critical directives could be redefined in the application within *htaccess* files or in scripts at runtime.

B. Experimental Methodology

First, we installed the latest versions (shown in the "Version" column of Table II) of the subject web applications on a common deployment environment. We used MAMP Version 2.0 with PHP 5.4.4, Apache 2.2.22, and MySQL 5.5.25 on MacOS X 10.6.4. We also made all application directories and files recursively readable so as to ensure entire coverage of the applications. Then, we configured our configuration vulnerability analysis tool, implemented in PHP, to parse the configurations of the environment and the applications and report potential vulnerabilities along with the severity scores of the vulnerabilities. After collecting the vulnerability reports, we interpret the individual vulnerability report pertinent to the potential attack it might lead to.

Secondly, we run popular web application vulnerability scanners on the same set of subject applications (with the same configuration used for our tool) to compare the analysis results from security configuration vulnerability standpoint. For this purpose, we used the `w3af` [26], `skipfish` [27], and `Websecurify` [28] to make the comparison. In doing so, we carefully inspected vulnerability reports of the three scanners and filtered only configuration specific vulnerabilities.

V. Evaluation Results

This section presents evaluation results comparison of our approach with generic web vulnerability scanners.

A. Default Installation Configuration Vulnerability Analysis

Table III shows the configuration vulnerability analysis results for default installations of the subject web applications. In practice, customization and deployment of these subject applications involve a great deal of modifying the defaults. Vulnerabilities encountered thereafter could be minimal or catastrophic depending on the experience of the developer doing the customization. Nonetheless, from our evaluation, analyzing just the default deployment of the applications without customization shows potential vulnerabilities.

Table III: Configuration vulnerability analysis results for default installations of subject web applications. The results refer to application level configuration vulnerabilities.

Application	PHP, Apache, MySQL	Affected Directives
Joomla	4, 0, 0	display_errors @/administrator/includes/framework.php, /cli/update_cron.php, /cli/finder_indexer.php, /includes/framework.php
Drupal	1, 1, 0	display_errors @ drupal/update.php, options @ drupal/.htaccess
WordPress	2, 0, 0	display_errors @ wp-admin/update.php, wp-admin/plugins.php
phpBB	1, 0, 0	memory_limit @ /includes/acp/acp_search.php
phpMyAdmin	2, 0, 0	display_errors @ /libraries/config/validate.lib.php, memory_limit @ /libraries/import/shp.php
SquirrelMail	0, 0, 0	-
osCommerce	2, 0, 0	session.bug_compact_42 @ catalog/admin/includes/functions/sessions.php, catalog/includes/functions/sessions.php
Gallery	0, 0, 0	-
Elgg	5, 1, 0	options @ elgg/.htaccess, display_errors @ /mod/developers/start.php, post_max_size, upload_max_size, memory_limit @ elgg/.htaccess, display_errors @ elgg/upgrade.php
Moodle	8, 0, 0	memory_limit @ /admin/cli/install.php, /lib/setuplib.php, display_errors @ admin/cli/install.php, moodle/install.php, /lib/setup.php, /lib/installlib.php, /lib/phpunit/bootstrap.php, max_execution_time @ /lib/xhprof/xhprof_html/call_graph.php
SugarCRM	10, 0, 0	max_execution_time @ /jssource/minify_utils.php, /modules/Administration/callJSRepair.php, /modules/Emails/EmailUIAjax.php, /modules/MailMerge/save.php, error_reporting @ /modules/MailMerge/save.php, memory_limit @ /modules/UpgradeWizard/silentUpgrade_dce_step2.php, /modules/UpgradeWizard/silentUpgrade_step1.php, /modules/UpgradeWizard/silentUpgrade_step2.php, /modules/UpgradeWizard/silentUpgrade_dce_step1.php
myBB	0, 0, 0	-
MediaWiki	2, 0, 0	memory_limit @ /includes/normal/UtfNormalMemStress.php, display_errors @ /maintenance/dev/includes/rounter.php
TestLink	0, 0, 0	-

In this regard, our tool reported no security configuration vulnerabilities in four applications: `SquirrelMail`, `Gallery`, `myBB`, and `TestLink`. For the remaining 10 applications, it reported security configuration vulnerabilities from as low as 1 to as high as 10 in an application. A particularly critical insight from the results is that some vulnerabilities are in parts of the application where maximum security is expected. For instance, the *display_errors* directive was found to be enabled in the *administrator/* directory of `Joomla`, the *wp-admin/* directory of WordPress, and the *admin/* directory of Moodle. These details are shown in the "Affected Directives" column of Table III. From the results, we group the unsafely set directives into three classes pertinent to the potential attacks they are likely to result in. These attacks are `Information Disclosure`, `Denial-of-Service` (buffer overflow), and `Session Hijacking`.

Information Disclosure. From the results in Table III, *display_errors* is unsafely set in 7 of the 14 applications. The *error_reporting* directive is unsafely set to *E_ALL* in SugarCRM. When combined with unsafe *display_errors* directive, *error_reporting* set to *E_ALL* (for instance after customization) could lead to a worst case scenario in which the application reports all errors to the public. Chances are, these errors may contain sensitive information about the application environment and data stored and manipulated by the application. For an attentive attacker, such detail of error messages is a low-hanging fruit to take advantage of and craft attacks. As demonstrated in Section II-B, injection attacks could be crafted to compromise these applications due to such information disclosure vulnerabilities. Moreover, malware may exploit information disclosure (e.g., a vulnerable *mod_ssl* in Apache) to launch attacks.

Denial-of-Service (Buffer Overflow). Unusually large values of the *memory_limit* (in `phpBB`, `phpMyAdmin`, `Elgg`, `Moodle`, `SugarCRM`, and `MediaWiki`) combined with abnormally large sizes for *upload_max_size* and *post_max_size* (in `Elgg`), and *max_execution_time* (in `Moodle`, `SugarCRM`) could lead to unresponsive server due to resource depletion (e.g., memory overflow). Attackers can exploit such vulnerabilities to initiate denial-of-service attacks by overwhelming the server with multiple and resource-hungry requests.

Session Hijacking. The *session.bug_compat_42* directive

is unsafely set in SugarCRM. This allows initializing a session variable in the global scope. An attacker can manipulate this vulnerability to steal session IDs and impersonate legitimate users of the web application.

B. Comparison with Generic Vulnerability Scanners

The vulnerability analysis results of generic vulnerability scanners is shown in Table IV. For all the three vulnerability scanners compared with Confeagle, they reported configuration related vulnerabilities of the environment (e.g, server fingerprinting) among a number of other vulnerabilities (e.g., XSS, SQLI, CSRF, Buffer Overflow). As can be seen from Table IV, configuration vulnerabilities at the application level are not reported by all the three scanners we evaluated on the subject applications. In what follows, we give more context to the results of comparison.

1) w3af (Version1.0-stable): We ran this tool by enabling all the 27 audit plugins including scans for vulnerabilities in file uploads, local/remote file inclusion, and htaccess methods. In all the subject applications, no application specific configuration vulnerability is reported. For all the applications, only 2 environment fingerprinting vulnerabilities due to exposed server signature (in this case Apache) are consistently reported. For most of the applications, URLs with injection points (mainly XSS and XST) are reported. From the results of w3af, we confirmed that few configuration vulnerabilities at the environment level are considered by the scanner while at the application level, we could not confirm any configuration related vulnerabilities.

2) skipfish (Version 1.85b): By disabling the dictionary based brute-force feature, we ran this tool from a BackTrack5 [29] virtual guest with the applications on a MacOSX host. While there are additional details of the scanner report, for the sake of comparison, we focused on the high risk, medium risk, and low risk entries in the report as our tool has similar classification of vulnerability scores. In SugarCRM, skipfish reported a successful Remote File Inclusion attack due to the vulnerability of the *allow_url_fopen* directive on the deployment environment. We confirmed that Confeagle also reported this vulnerability with vulnerability score of 8. Like w3af and Websecurify, this scanner also reported server signature vulnerability at the environment level.

3) Websecurify (Version 2.1.1): We used the Chrome extension of this tool and supplied each of our subject applications. The scanner has found a number of vulnerabilities for our subject applications. For instance, the scanner was able to discover 46 vulnerabilities in Elgg social network application. However, 42 stem from a Autocomplete enabled vulnerabilities via the Autocomplete HTML tag attribute that enabled 42 forms within the application to be auto completed by the browser. Most of the vulnerabilities are related to Information Disclosure (e.g., error, banner, email), where *banner disclosure* vulnerability is detected in all the subject applications. We noted that the server disclosed its type via the HTTP response headers because the *expose_php* directive was enabled in *php.ini* file. Our tool also reported this issue but at the level of the applications though. The scanner did not reveal any application-level configuration vulnerabilities.

In summary, although the generic scanners seem to focus on environment specific configuration vulnerabilities, not much of these vulnerabilities are reported either. For the environment, across all the applications, the three scanners consistently reported server-signature vulnerability related to the *expose_php* directive. skipfish reported two more environment level configuration vulnerabilities linked with the *allow_url_fopen* directive and arbitrary MIME type. For some applications (e.g., WordPress, Elgg, phpBB) Websecurify reported server configuration vulnerability that discloses administrator email. However, Confeagle reported 28 environment level configuration vulnerabilities.

VI. Related Work

In this section, we position our approach with respect to existing literature focusing on automated configuration vulnerability analysis in web applications.

In [12], an automated tool to audit misconfiguration vulnerabilities in web server environments is presented. The authors evaluated the tool on eleven widely used server environments across three popular OS platforms. The results indicate that the tool can reveal security configuration vulnerabilities in default installations of the server environments. However, this approach does not address application level configuration vulnerability analysis. Moreover, the tool assumes that all security configuration directives have equal weight. Our work widens the scope of this work and complements it by addressing the above limitations.

A language-based approach to specify and execute declarative and unambiguous security checks for detecting vulnerabilities caused by a system misconfiguration is proposed in [14]. The proposed language is based on the SCAP specification [30] and extends the OVAL configuration validation standard [31]. The language allows the definition of configuration checks, the target software components as well as the actual configurations by specifically separating the checking logic from the configuration retrieval. Unlike ours, this approach focuses on detecting system-level configuration vulnerabilities. The link between a vulnerability (if discovered) and potential attack(s) it can lead to is not obvious to infer from the approach.

In [13], a tool is presented to detect configuration vulnerabilities and measure their severities. The approach integrates the CVSS scoring metrics into Nessus [32], which in turn passes the scanning report to their configuration scanner, which uses the CCE configuration scanner [33]. By using the generated report together with user supplied inputs (such

Table IV: Configuration vulnerability analysis results with generic vulnerability scanners compared to our tool. The comma separated values refer to configuration vulnerabilities reported at the *environment* and the *application* level respectively.

Application	***Confeagle***	**w3af [26]**	**skipfish [27]**	**Websecurify [28]**
Joomla	28, 4	2, 0	3, 0	1, 0
Drupal	28, 2	2, 0	3, 0	1, 0
WordPress	28, 2	2, 0	3, 0	2, 0
phpBB	28, 1	2, 0	3, 0	2, 0
phpMyAdmin	28, 2	2, 0	3, 0	1, 0
SquirrelMail	28, 0	2, 0	3, 0	1, 0
osCommerce	28, 2	2, 0	3, 0	1, 0
Gallery	28, 0	2, 0	3, 0	1, 0
Elgg	28, 6	2, 0	3, 0	2, 0
Moodle	28, 8	2, 0	3, 0	2, 0
SugarCRM	28, 10	2, 0	3, 0	2, 0
myBB	28, 0	2, 0	3, 0	2, 0
MediaWiki	28, 2	2, 0	3, 0	1, 0
TestLink	28, 0	2, 0	3, 0	1, 0

as policies, checklists and environment information), vulnerabilities are scored with CVSS-based metrics. Although this work shared the same spirit with our work in assisting administrators to understand the severity of vulnerabilities, it only focuses on scanning and scoring aspects of configuration vulnerabilities. Moreover, it does not identify configuration vulnerabilities at the application level.

Mendes et al. [34] proposed a technique to determine the security of web server configurations by analyzing their installations against a number of security recommendations. A metrics based on weighted percentage of security recommendations is used to indicate the security level of the assessed web server. Although the approach attempted to discover vulnerabilities in server environments, it is limited to only Apache environment and the analysis is carried out manually, which can also be error-prone and may end up with inconsistencies. Our approach aims not only at finding configuration vulnerabilities at the level of the environment but also at application level.

Generic web application vulnerability scanners based on white-box [9], [10] or black-box [6]–[8], [26]–[28] techniques are effective at different scales while varying in terms of their coverage of vulnerabilities, false positives, and application coverage. However, compared to our approach, they all lack the consideration that configuration vulnerability deserves given its criticality in practice. In particular, compared to `Confeagle` which analyzes configuration vulnerabilities at environment and application level, these scanners report configuration vulnerabilities only of the environment. We confirmed this observation through our experimental evaluation on `w3af` [26], `skipfish` [27], and `Websecurify` [28].

VII. Conclusion

In web applications, configuration vulnerabilities may appear at any level of the application stack —from the underlying platform to business logic code. Unfortunately, it received little attention in web application security research despite its criticality. We presented a novel approach to reinforce automated web vulnerability analysis techniques by analyzing configuration vulnerabilities in web applications. Our approach effectively combines hierarchical application structure scanning with preliminary source code analysis to analyze, quantify, and fix configuration vulnerabilities in complex web applications.

We implemented and evaluated our tool, `Confeagle`, on 14 popular PHP applications. The evaluation results confirmed that our approach is more precise than generic web vulnerability scanners in detecting configuration vulnerabilities at the application level. We believe that our approach supplements web application vulnerability scanners by filling the missing consideration for configuration vulnerability analysis, specially at application level. More specifically, the major beneficiaries of our approach are Web Developers and Web Masters. For Web Developers, it supplements test-driven and security-aware development of web applications. It does so by conducting detailed analysis of configuration vulnerability risks before deploying web applications. For Web Masters, it assists in auditing the web application for configuration vulnerabilities before committing maintenance changes.

In the future, we plan to enhance `Confeagle` so as to incorporate different versions of Apache, PHP, and MySQL. Furthermore, we plan to apply our approach in other classes of web applications (e.g., those written in Python, Java).

References

[1] T. Holz, S. Marechal, and F. Raynal, "New Threats and Attacks on the World Wide Web," *IEEE Security and Privacy*, vol. 4, pp. 72–75, March 2006.

[2] P. Tramontana, T. Dean, and S. Tilley, "Research Directions in Web Site Evolution II: Web Application Security," in *Proceedings of the 2007 9th IEEE International Workshop on Web Site Evolution*, 2007, pp. 105–106.

[3] Symantec, "Symantec report on attack kits and malicious websites," http://symantec.com/content/en/us/enterprise/other_resources/b-symantec_report_on_attack_kits_and_malicious_websites_21169171_WP.en-us.pdf, July 2011.

[4] T. Scholte, W. K. Robertson, D. Balzarotti, and E. Kirda, "Preventing input validation vulnerabilities in web applications through automated type analysis," in *COMPSAC*, 2012, pp. 233–243.

[5] M. Fossi, "Symantec Global Internet Security Threat Report," http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xv_04-2010.en-us.pdf, Symantec, Tech. Rep. Volume XV, April 2010.

[6] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "Secubat: a web vulnerability scanner," in *WWW*, 2006, pp. 247–256.

[7] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: a state-aware black-box web vulnerability scanner," in *USENIX Security Symposium*, 2012, pp. 26–26.

[8] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in *SP '10*, 2010, pp. 332–345.

[9] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities (short paper)," in *SP '06*, 2006, pp. 258–263.

[10] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *WWW*, 2004, pp. 40–52.

[11] OWASP, "OWASP Top 10 Project," https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, January 2013.

[12] B. Eshete, A. Villafiorita, and K. Weldemariam, "Early Detection of Security Misconfiguration Vulnerabilities in Web Applications," in *ARES*. IEEE, 2011, pp. 169–174.

[13] C.-H. Lin, C.-H. Chen, and C.-S. Laih, "A study and implementation of vulnerability assessment and misconfiguration detection," in *APSCC*, 2008, pp. 1252–1257.

[14] M. M. Casalino, M. Mangili, H. Plate, and S. E. Ponta, "Detection of configuration vulnerabilities in distributed (web) environments," *CoRR*, vol. abs/1206.6757, 2012.

[15] K. Scarfone and P. Mell, "Vulnerability scoring for security configuration settings," in *Proceedings of the 4th ACM workshop on Quality of protection*, 2008, pp. 3–8.

[16] OWASP, "OWASP Configuration Guide," https://www.owasp.org/index.php/Configuration, January 2013.

[17] Symantec, "Securing Apache: Step-by-Step," http://www.symantec.com/connect/articles/securing-apache-step-step, January 2013.

[18] SANS, "Building Servers as Appliances for Improved Security," http://www.sans.org/reading_room/whitepapers/bestprac/building-servers-appliances-improved-security_33304, January 2013.

[19] Oracle, "Web Application Security Configuration Guide," http://docs.oracle.com/cd/E28595_01/Web_App_Security_Guide.pdf, January 2013.

[20] PHP, "PHP Security Manual," http://php.net/manual/en/security.php, January 2013.

[21] Apache, "Apache Security Tips," http://httpd.apache.org/docs/2.2/misc/security_tips.html, January 2013.

[22] MySQL, "MySQL Secure Installation," http://dev.mysql.com/doc/refman/5.0/en/mysql-secure-installation.html, January 2013.

[23] Cyberciti, "Linux: 25 PHP Security Best Practices For Sys Admins," http://www.cyberciti.biz/tips/php-security-best-practices-tutorial.html, January 2013.

[24] TechRepublic, "10 things you should do to secure Apache," http://www.techrepublic.com/blog/10things/10-things-you-should-do-to-secure-apache/477, January 2013.

[25] K. S. P. Mell, "The common configuration scoring system (ccss): Metrics for software security configuration vulnerabilities," http://csrc.nist.gov/publications/nistir/ir7502/nistir-7502_CCSS.pdf, December 2010.

[26] "Web Application Attack and Audit Framework," http://w3af.sourceforge.net/, January 2013.

[27] Google, "skipfish: web application Security Scanner," http://code.google.com/p/skipfish/, January 2013.

[28] Websecurify, "Cross-platform Web application Security toolkit," http://www.websecurify.com/, January 2013.

[29] BacktrackLinux, "Backtrack Linux," http://www.backtrack-linux.org/backtrack/backtrack-5-release/, January 2013.

[30] S. Quinn, D. Waltermire, C. Johnson, K. Scarfone, and J. Banghart, "The technical specification for the security content automation protocol (scap)," National Institute of Standards and Technology, Tech. Rep. Version 1.0, November 2009.

[31] S. Q. John Banghart and D. Waltermire, "Open vulnerability and assessment language (oval) validation program test requirements (draft)," Computer Security Division Information Technology Laboratory National Institute of Standards and Technology, NIST Interagency Report 7669, March 2010.

[32] Nessus: The Network Vulnerability Scanner. [Online]. Available: http://www.nessus.org/nessus/

[33] CCE: Common Configuration Enumeration. [Online]. Available: http://cce.mitre.org/

[34] N. Mendes, A. A. Neto, J. a. Durães, M. Vieira, and H. Madeira, "Assessing and Comparing Security of Web Servers," in *Proceedings of the 2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*, 2008, pp. 313–322.