Toward Secure Web Application Design: Comparative Analysis of Major Languages and Framework Choices

Stephen J. Tipton
College of Arts & Sciences
Regent University
Virginia Beach, Virginia, U.S.A.

Young B. Choi
College of Arts & Sciences
Regent University
Virginia Beach, Virginia, U.S.A.

***Abstract*—We will examine the benefits and drawbacks in the selection of various software development languages and web application frameworks. In particular, we will consider five of the ten threats outlined in the Open Web Application Security Project (OWASP) Top 10 list of the most critical Web application security flaws [12], and examine the role of three popular Web application frameworks (Ruby on Rails (Ruby), Play Framework (Scala), and Zend Framework 2 (PHP)) in addressing a selection of these major threats. In addition, we will compare the strengths and weaknesses of each Web application framework as it pertains to the implementation of strong security measures. Furthermore, for each framework examined, assess how an organization should address these security threats in their software design utilizing their framework of choice. We will suggest the direction in which an organization facing such a decision ought to head; moreover, facilitate such a decision by assessing the benefits and drawbacks of each, based on the findings; and encourage one to decide what works best for the organization's technical direction.**

***Keywords*—Web; security; framework; application; authentication; ruby; ruby on rails; play framework; Scala; PHP; Zend Framework 2; SQL injection; threats**

I. Introduction

In October 2014, Drupal, the popular PHP-based open source content management platform, reported experiencing multiple exploits of vulnerability within its database abstraction API involving carefully crafted requests that resulted in the execution of arbitrary SQL statements [18]. Despite the overarching purpose of the database abstraction API in preventing such exploits, the Drupal Security Team advised site administrators utilizing Drupal 7.x to upgrade to Drupal core 7.32. Administrators who were unable to upgrade were advised to apply a patch to the database.inc file. In a subsequent announcement from the Drupal Security Team, the importance of upgrading to Drupal 7.32 was further outlined and promulgated that simply upgrading would not remove the potential for backdoors in the database, code, or various other locations [5].

A SQL injection attack is such that takes advantage of holes in web services and other web applications by inserting, or "injecting" arbitrary SQL statements "via the input data from the client to the application" [12, 20]. Such vulnerability raises the extreme potential for reading sensitive data from the database; the modification of data via Insert, Update, and/or Delete statements; and the execution of administration operations on the database [20].

WhiteHat Security's 2014 Website Security Statistics Report [25] notes that as a language, "PHP stood out from the pack when looking at SQL Injection, with the languages instances of the vulnerability exhibiting the lowest average number of days at 6.8." Java fell with a much larger gap from PHP at an average of 64.8 days [25]. It is further noted that from the perspective of the Ruby language, statistics were much too minute to include in WhiteHat's report.

While SQL injection, or injection in general, leads the OWASP Top 10 list of web security threats, Web security considerations are not limited to this vulnerability. The scope of this assessment addresses five of the leading threats listed in the Top 10, with SQL injection rounding out this list. Additionally discussed are the threats involving broken authentication and session management; cross-site scripting; insecure direct object references; and security misconfiguration. In particular, a selection of these threats are addressed in relation to three Web application frameworks: Ruby on Rails (Ruby) addressing SQL injection; Play Framework (Scala) addressing Security Misconfiguration; and Zend Framework 2 (PHP) addressing Broken Authentication and Session Management. Addressing these top threats in relation to these three frameworks, and assessing their strengths and weaknesses may facilitate an organization facing the technical decision of choosing an appropriate software stack.

II. Web Security Threat Considerations

The OWASP Top 10 list of Web security threats is rounded out by five of the most critical threats noted within the previous year. Leading the list, as previously cited, are injection attacks (e.g., SQL injection), which were outlined in the case of Drupal's vulnerability in their database abstraction API.

This section considers the leading five threats from the Top 10 list: SQL injection; broken authentication and session management; cross-site scripting; insecure direct object references; and security misconfiguration. Each threat is detailed in its nature, with the primary objective to outline the threats in relation to the scope of this research.

The secure design and implementation of software applications are critically bound to the firm understanding of the threats in which software is designed against. It is imperative that these five threats are considered in detail to provide the understanding necessary for selecting the appropriate software stack to be leveraged in the implementation of the organization's web applications. The following considerations will describe each of the five threats, and the nature imposed upon software applications. The Network Defense Security and Vulnerability Assessment, Volume 5 of the Network Security Administrator Certification [19], echoes this critical aspect due to the increasing importance of Web sites to commercial businesses.

A. SQL Injection

SQL injection attacks exploit vulnerabilities in APIs, as well as other Web applications through the insertion, or injection of arbitrary SQL commands by way of inputting data through the gateway that links the client to the application [12, 20]. Patil and Bamnote [13] cite repercussions from injection attacks including the "unauthorized access to private or confidential information stored ... [via] authentication bypassing, [and] leaking of private information." The Network Defense Security and Vulnerability Assessment [19] parallels this illustration by noting that Web applications are extremely vulnerable due to the ability to receive input data in numerous ways. In general, input data should be analyzed and effectively wrapped by a server-side validation mechanism.

B. Broken Authentication and Session Management

According to the Top 10, authentication and session management are often incorrectly implemented, leaving vulnerable web applications in a broken state in which attackers may potentially compromise user-created passwords, API keys, or session tokens; vulnerabilities left unaccounted for may also "exploit other implementation flaws to assume other users' identities." Web service authentication is not a feature that comes built-in to various Web application frameworks [6]; rather, it is the expectation of developers to implement authentication. Furthermore, this is primarily the case due to the many flavors of adding authentication to HTTP-based web services, including basic authentication, token-based authentication, and session-based authentication.

C. Cross-Site Scripting

Cross-site scripting, or XSS, is the result of "insufficient data validation, sanitization, or escaping" [9] within web applications that present an opportunity for an attacker to execute malicious browser-side code, such as JavaScript. The exploitation of this vulnerability may consummate in the "complete ... compromise of the victim's session," cites Kern. Similar to SQL injection, the Network Defense Security and Vulnerability Assessment [19] asserts that all input data should be thoroughly validated. In XSS vulnerabilities, this threat relates to the browser-side; therefore, XSS can occur when proper validation or escaping on the browser-side is non-existent. According to the Top 10, the malicious execution of scripts can result in hijacked user sessions, defaced web sites, or redirection to phishing sites.

D. Insecure Direct Object References

The Top 10 defines insecure direct object references as "a reference to an internal implementation object, such as a file, directory, or database key" that lacks necessary access controls or other protective measures. For example, web applications are frequently known to use the actual name or key of an object when generating Web pages, without verifying the authorization to access that particular object [21]. The technical impact of such flaws includes the potential for compromising the data associated with the key. To expand upon this example, one may consider a RESTful Web service's URI structure as "intuitive and guessable" [7]. To counteract this, the MVC-pattern featured in many Web frameworks establishes the role of a controller intermediary between the route (the URI structure) and the model layer.

E. Security Misconfiguration

Efficient security requires the existence of secure configuration that is both defined and deployed for the Web application, its framework(s), its server and other related servers (e.g., web, database, etc.), and its platform, according to the Top 10. Furthermore, settings should constantly be maintained. The utilization of what is referred to as "patch management," which is "the administration and supervision of the processes and technology for keeping systems updated with the latest security software defenses," goes hand-in-hand with maintaining good security configurations, and is considered a "basic security must-have" [4]. Configuration defaults are also known to be insecure. For example, the Play framework default configuration includes a generated value for the application's secret key [6]. This is also the case for the Ruby on Rails framework [3]. Furthermore, it is also common to require configuration values to be stored within environment variables, and then referenced in configuration files [6].

III. Comparative Analysis of Popular Web Appliaction Frameworks

At some point, an organization will be facing a technical decision involving the selection of a software development stack to accomplish a project that will ultimately enhance or increase business value. The importance of selecting the appropriate tool for the job is drastically increased when weighing the threats outlined in the OWASP Top 10 list. Having previously addressed in detail the five threats that round out the Top 10 list, the next measure to consider is analyzing the comparisons between three web application frameworks across three different software development languages: Ruby on Rails (Ruby); Play Framework (Scala); and Zend Framework 2 (PHP).

Each framework addressed will offer a high-level overview of the framework's features and typical use cases. In a comparative analysis, strengths and weaknesses of each framework will be weighed; the objective is to understand what each framework may or may not offer "out-of-the box," and how each framework will assist developers in designing and implementing secure web services, modular components, or full-blown web applications. From a business angle, such an understanding will facilitate a technical decision.

It ought to be understood, however, that neither of these frameworks are not in itself "more secure than another" [17]; rather, it is the functional features that reside within each framework that assist developers with the tools necessary to secure web applications.

To round out the comparative analysis of these three frameworks, each will include a real-world example within a summarized case study, demonstrating how organizations have utilized that framework of choice to deliver a secure software application. In these short studies, the scope will be limited to a single selection from the five threats that round out the OWASP Top 10 list. It is the objective of this discussion to encourage technical leadership in an organization to make a sound decision when selecting a software development stack.

A. Ruby on Rails (Ruby)

The 10,000-foot level. The overall purpose of a Web application framework is to provide a toolset to developers that facilitate the implementation of Web-based software applications. From a security standpoint, no one framework is going to outweigh another in its own security [17]. The challenge in securing web-based software is raised when developers are faced with implementing secure code. The good news is, Web frameworks provide a set of tools that make this simple for developers to achieve. Ruby on Rails is an example of a Web application framework that achieves this function. In short, the Rails framework "makes it easier to develop, deploy, and maintain web applications" [16].

The leading threat according to the OWASP Top 10 is the exploitation of API vulnerabilities using SQL injection. By virtue of "clever methods," [17] most Rails applications are nearly immune to this threat. However, this is not to assert SQL injection is impossible in Rails applications. If not utilized properly, these "clever methods" will serve no other purpose than to sit unused, leaving a Rails application open to this vulnerability. Ruby on Rails utilizes an Object Relational Mapper (ORM) called Active Record which exposes methods facilitating safe database transactions by properly escaping SQL, which in itself "is immune from SQL injection attacks" [16].

Strengths and weaknesses of the framework. In addressing SQL injection vulnerabilities within Rails applications, it is the responsibility of developers to take advantage of the toolset provided by the Rails framework. As noted previously, Rails exposes "clever methods" that facilitate a near-immunity against SQL injection. While these methods do exist, holes are occasionally uncovered that expose vulnerabilities within the internal method. For example, in January 2013, such a vulnerability was found in dynamic finder methods (e.g., `find_by_foo(params[:foo])`). The scenario was verified when applications were using the third-party authentication library Authlogic, and the secret session token was known [10].

[16] describe the functionality of Active Record and how it handles the prevention of SQL injection as follows: When multiple parameters are passed into the where method call— a method call that corresponds to the SQL where clause— the first parameter is effectively utilized as a template for generated SQL. Strengthening this feature is the utilization of placeholders, which are replaced with the values from the remaining parts of the array at runtime. Additionally, named placeholders may have their values passed in as a hash of key-value pairs (e.g., `{pay_type: pay_type, ...}`). Furthermore, these key-value pairs can be passed in as a direct hash reference (e.g. `params[:order]`) as a single argument to the where method (e.g., `Order.where(params[:order])`). This latter form is cautioned, however, as it takes in every key-value pair residing within the hash. An even more secure method would essentially white list the key-value pairs that are needed for the Active Record query (e.g., `Order.where(name: params[:name], ...)`).

Case study: Object Injection and Rails' Dependency on YAML. William (B.J.) Snow Orvis is a software programmer with Artemis Internet and iSec Partners, and has frequented the Ruby community presenting talks on addressing security issues in Ruby on Rails development. In Orvis' *Secure Development on Rails* presentation [11], he covered an object injection vulnerability (similar to SQL injection) that was discovered by Rails contributor Aaron Patterson [14]. This vulnerability affected all versions of the Rails framework, and entailed "multiple weaknesses in the parameter parsing code ... which allow[ed] attackers to bypass authentication systems, inject arbitrary SQL, inject and execute arbitrary code, or perform a DoS attack on a Rails application." It is noted that the parameter parsing code provides applications the ability to automatically typecast strings to certain data types. The caveat uncovered revealed that certain conversions, in particular the creation of symbols and parsing YAML— a highly utilized dependency in Rails— were supported in the parsing code. "These unsuitable conversions can be used by an attacker to compromise a Rails application," warned Patterson.

The previous scenario outlined by Patterson [14] varied depending on which version of Rails was being used, and whether or not the Web application depended upon support for XML parameters. Mitigating the issue followed a two-fold approach. Primarily, users who did not rely upon XML parameter support were advised to disable XML parsing entirely by deleting `Mime::XML` from `ActionDispatch::ParamsParser::DEFAULT_PARSERS` (e.g., `ActionDispatch::ParamsParser::DEFAULT_PARSERS.delete(Mime::XML)` in Rails 3.x). Alternatively, developers of applications that relied heavily upon XML parsing were advised to disable the YAML and symbol type conversion from the XML parser by deleting `Mime::YAML` from `ActionDispatch::ParamsParser::DEFAULT_PARSERS` (e.g., `ActionDispatch::ParamsParser::DEFAULT_PARSERS.delete(Mime::YAML)` in Rails 3.x). Additionally, this latter approach was further advised to be in parallel with reducing the value of `REXML::Document.entity_expansion_limit` to limit the risk of entity explosion attacks. Orvis' talk on Secure Development on Rails covered many aspects of Web security, and is recommended as a supplement to this composition.

B. Play Framework (Scala)

The 10,000-foot level. As previously discussed, Ruby on Rails experienced a vulnerability involving parameter parsing, which automatically typecasts strings to certain data types. In Play for Scala, data types are cast statically at compile time,

rather than dynamically at runtime. Furthermore, it is this "increased type safety" that garners an immediate benefit throughout the development lifecycle [6]. Play is not constrained to type safety benefits, either. It offers a declarative application URL scheme configuration; it features an HTML5-embraced architecture; it silently reloads on code changes; and more importantly, it is a full-stack framework providing persistence, security, and internationalization [6].

The OWASP Top 10 listed security misconfiguration as the fifth-most critical Web security threat in 2013. Adequate security relies on the definition and deployment of secure configuration for the web application and its numerous components. In addition, the maintenance of these configurations are of equal importance. Cyber Security [4] stresses patch management, along with good security configuration maintenance as a "basic security must-have." Expanding upon this, security misconfiguration is classified by OWASP as easily exploitable. An attacker may access default accounts, unused pages, unpatched flaws, unprotected files and directories, etc. for the primary purpose of obtaining unauthorized access to or knowledge of the system.

Strengths and weaknesses of the framework. Hilton, Bakker, and Canedo [6] confidently assert that simply creating a Play application requires no configuration. This is true as well with Ruby on Rails, which boasts of its convention over configuration. Play initializes a configuration file automatically, with almost all of the parameters being optional. However, with optional parameters, values must sensibly be defaulted. Configuration defaults, in production, are susceptible to insecurity. For example, Play adds a default configuration value for the application's secret key. As expected, these values are able to be overridden, or referenced with environment variables. Moreover, it is required to utilize environment variable references for OS-independent, machine-specific configuration; likewise, it is encouraged to use environment variable references— primarily in production environments— for sensitive configurations, such as database credentials and secret keys.

During development, there is only the need for a single configuration file (e.g., conf/application.conf). However, when deploying to production, different configuration settings will be necessary. Hilton, Bakker, and Canedo [6] note that due to the application being packaged within a JAR file, simply deploying the application, and then manually editing the configuration is inefficient. Consequently, this practice is known to be error-prone and automation-unfriendly. It is highly advised to not make the mistake of sharing identical settings for all environments (e.g., development, test, and production), to shortcut the need for separate configurations. It is likely that at some point, a developer who has shortcut this necessary step could potentially wipe out an entire production asset, such as a database, simply by mistaking which environment was currently being utilized.

It is encouraged to have a "safe" default configuration that is easily overridable by other environments, such as the test environment [6]. Play allows configuration overriding by specifying the override function on a given configuration (e.g., mail.override.address = "info@example.org"). Following any overrides, the developer would then specify the inclusion of a separate configuration file (e.g., development.conf), which would override the default configuration.

Case study: Secure Network Configuration using the Typesafe Reactive Platform and the Play Framework. Auvik Networks "is a hybrid cloud, software-as-a-service (SaaS) application that provides IT professionals with a better way to monitor, configure and automate their network" [24]. The company created a cloud-managed network automation platform to simplify enterprise networking, which has the potential of being highly complex. To deliver this business value, Auvik utilized the Typesafe Reactive Platform to provide a reliable and scalable solution, allowing a continual value add to the business [2].

Utilizing Akka, which facilitates the building of "highly concurrent, distributed, and resilient message-driven applications on the JVM" [1], Auvik leveraged the scalability, clustering, and load balancing to build and deploy their hybrid cloud configuration. Auvik delivered a cloud-based UI that allows a customer to sign up, manage, monitor, and configure their network environment— all via a Web application built on the Play framework. By using Play, and deploying onto the Typesafe Reactive Platform, Auvik was able to take advantage of developer productivity, a modern web application experience, minimal resource consumption, and a high-performing, highly scalable application.

To read more about Auvik Networks use of the Typesafe Reactive Platform and Play framework, the *Auvik Networks simplifies enterprise networking* [2] case study is recommended.

C. Zend Framework 2 (PHP)

The 10,000-foot level. Broken authentication and session management appear in the OWASP Top 10 list second to SQL injection. The vulnerability does not reside within the framework itself; rather, it is in the incorrect implementation that leaves web applications in a vulnerable state potentially allowing attackers to compromise passwords, API keys, or session tokens. Hilton, Bakker, and Canedo [6] echo this fact by disclosing against the misconception that frameworks ship with built-in authentication handling. Because of the many attributes of HTTP-based web services (e.g., basic authentication, token-based authentication, and session-based authentication), the responsibility of handling an authentication mechanism is left to developers; and since developers are the sole proprietors of enabling a secure authentication implementation, it is imperative that entry-points into a web application are efficiently secure.

In some cases, developers are encouraged to utilize open-source libraries to leverage authentication functionality. Rather than reinvent the wheel, frameworks such as Ruby on Rails, in collaboration with the rich Ruby community, foster the utilization of libraries such as Devise or OmniAuth; of course, developers may roll their own authentication implementation as well [15]. The Play framework likewise does not ship with authentication functionality built-in. In fact, rolling one's own authentication implementation in Play is a straightforward process. Hilton, Bakker, and Canedo [6] state that

authentication may be performed alongside every HTTP request, prior to an appropriate HTTP response. This allows the existence of a stateless application that requires valid credentials on every HTTP request.

When addressing user-created passwords, the obligation of encryption is introduced. Where libraries such as Devise facilitate Rails developers in integrating a robust, encrypted authentication solution, frameworks such as Zend Framework 2 (ZF2) for PHP ship with encryption components ready to deal with symmetric or asymmetric algorithms; additionally, cryptographic fingerprints [8] further protect authenticated sensitive data. When considering security benefits in ZF2, it is valuable to note that "all the cryptographic and secure coding tools you need to do things right" are readily available out-of-the-box [8].

Strengths and weaknesses of the framework. Karadzhov (2013) outlines the steps and code involved in securing a valuable authentication mechanism in ZF2 applications. One of the many components available to achieve this is Zend\Authentication\Adapter. This component receives user credentials such as username and password; however, it may also be an International Mobile Equipment Identity (IMEI) key unique to mobile devices. If authentication is verified, the identity information is stored to alleviate the need for the user to provide credentials repeatedly. Subsequent requests utilize the stored identity to check accessibility to a given controller and action in the MVC pattern. Coupled with an authentication adapter, a connection the system involved in credential verification is established. For example, when using MD5 for password hashing, an instance of a database adapter such as Zend\Db\Adapter\Adapter would be utilized along with database table information relating to storing the username and password. However, this approach is no longer considered secure [8].

As previously discussed, ZF2 ships with encryption components that ease the challenges of implementing properly secured authentication. As storing passwords hashed with the MD5 algorithm are no longer considered secure, according to Karadzhov [8], ZF2 features the Zend\Crypt\Password component that more efficiently and securely stores passwords. Furthermore, it is advised to use the *Bcrypt* algorithm in replacement of any use of MD5. Enrico Zimuel, creator of Zend\Crypt, states that Bcrypt is considered secure due to the slow computational time of a single hash; therefore, a brute force or dictionary attack would require a much larger amount of time to complete [8]. The Bcrypt algorithm is implemented via the Zend\Crypt\Password\Bcrypt class, in which an instance of this class would create a 60-character hashed string given a plain-text string:

```
use Zend\Crypt\Password\Bcrypt;

$bcrypt = new Bcrypt();

$password = $bcrypt->create('password');

#=>$2a$14$yuD/3v/ldbdOZ0pfljUyJ.a0Q4Ue0UTAoES2B
lgK0Op1Z6IF9.aTS
```

Case study: Brute-force Password Cracking. Compounding the threat of compromising passwords is a brute-force method in which bots are used to submit multiple string combinations to authentication forms. While brute-force attacks are more difficult to be successful when employing encryption algorithms such as Bcrypt in ZF2 web applications, it is still a considerable vulnerability to address. Vikram Vaswani, founder of Bombay-based web design company Melonfire, has outlined in a very robust how-to article [23] the mitigation of various security scenarios when developing web applications in the ZF2 architecture. As previously discussed, web applications are vulnerable to attacks including, but not limited to, SQL injection, XSS, CSRF, spam, and brute-force password hacking. Also outlined is the ease in protecting against such vulnerabilities when developing a PHP web application in ZF2. In Vaswani's article, he addresses countermeasures developers can take in mitigating form-based brute-force attacks.

The simplest measure to take to counteract bot interaction via web application forms is to implement a CAPTCHA [23]. ZF2 includes a component that implement this functionality: Zend\Captcha. This component can add FIGlet— ASCII-generated text banners made up of many typefaces— or an image CAPTCHA to the Web form. It also supports the third party web service reCAPTCHA, which integrates remote-generated CAPTCHAs. A caveat to the integration of reCAPTCHA lies in the requirement that the dependency would need to be specified in the Composer configuration. Aside from this, ZF2 essentially ships with many components necessary to secure Web applications.

Vaswani [23] illustrates the setup of a simple contact form, with inputs for name, email address, and CAPTCHA verification. ZF2 provides the Zend\Captcha\Image component, which accepts a number of configuration options (e.g., length of CAPTCHA word, font, directory to store the CAPTCHA, etc.) to generate the CAPTCHA. It is further noted that this component utilizes PHP's GD extension to generate the CAPTCHA image. Once the CAPTCHA is in place, validators are automatically set up and available to the controller and action via the Zend\Captcha component.

To understand more of how ZF2 can assist in securing web applications, Vaswani's thorough article, *Improve web application security with Zend Framework 2* [23], is recommended.

IV. Conclusion

As outlined in the OWASP Top 10, there is much more to securing Web applications than addressing three of the more common threats in relation to three corresponding web application frameworks. It must be restated as well that no single web application framework is going to be more secure than the other. However, there are features that prove beneficial to developers; while there are features that may not be of much assistance aside from providing necessary tools for developers. It is important to recall that most frameworks do not ship with authentication functionality, or any other fully implemented security threat mitigation. Therefore, the onus is on developers to understand the threats facing web applications. Because these threats are constantly evolving, it is important to remain engaged in current threat assessments in the industry.

We examined the benefits and drawbacks in selecting software stacks comprised of Ruby and the Ruby on Rails framework; Scala and the Play framework; and PHP and Zend Framework 2. It has further considered the leading five threats from the OWASP Top 10, and compared the three frameworks in mitigating a subset of the five threats. In exemplifying such mitigation, we covered three scenarios in which a given framework was utilized in counteracting an exploited vulnerability.

The determination of which software stack works best for a given organization's technical needs must now rely upon the technical focus of the organization. If an organization is seeking to build a robust, scalable, and easily configurable web service, along with a modern user interface, then perhaps the choice for the organization may lead to developing on the JVM using Scala and the Play framework. Companies such as Twitter, LinkedIn, DirecTV, WhitePages, and The Huffington Post have all made this decision to migrate away from their original architectures to the Reactive Platform offered by Typesafe [22].

It may be in the business' interest to quickly deliver a robust application with security-minded authentication functionality, and common threat mitigation approaches— all while not being in possession of a large, knowledgeable team of developers that would be able to roll their own approach. If this is the scenario, perhaps utilizing the Ruby on Rails framework would be the choice, with its rich community of developers and open source libraries that are able to be seamlessly integrated into a complete application.

However, it is noted that one framework is not more secure than the other; likewise, it is noted that most frameworks leave it to developers to implement security measures in Web applications, while being provided the tools necessary for it to be achieved. In retrospect, the single framework considered in this research that demonstrates the most robust set of tools is arguably Zend Framework 2. With components available to achieve more secure encrypted password functionality, ZF2 may be the choice for an organization warranting such a complete toolset.

The decision, however, is up to the organization's technical leadership. It is also highly encouraged to not only understand the threats facing today's Web technologies, but to understand what those threats mean to one's organization. By understanding these threats, and how these threats may affect one's organization, the determination of an appropriate software stack may be decided upon. We only provided a handful of tools; like many Web application frameworks, the responsibility is now up to developers. Likewise, the responsibility is now in the hands of technical leadership.

REFERENCES

[1] Akka. (2014). Retrieved from http://akka.io/.

[2] Auvik Networks simplifies enterprise networking. (2013). *Typesafe Case Studies & Stories*. Retrieved from http://downloads.typesafe.com/website/casestudies/Auvik-Case-Study.pdf?_ga=1.61301934.324464605.1417574558.

[3] Configuring Rails Applications. (n.d.). *Rails Guides*. Retrieved from http://guides.rubyonrails.org/configuring.html#initializers.

[4] Cyber Security: Doing the Right Things. (2013). Securing our connected world. *TMForum Security and Defense Publication*. Retrieved from http://www.tmforum.org/ResearchPublications/7097/home.html#TRCPublications/Link51039.

[5] Drupal Core - Highly Critical - Public Service announcement - PSA-2014-003. (2014, Oct. 29). *Drupal Security Advisories*. Retrieved from https://www.drupal.org/PSA-2014-003.

[6] Hilton, P., Bakker, E., and Canedo, F. (2014). Play for Scala. Covers Play 2. Shelter Island, NY: Manning Publications Co.

[7] Insecure Direct Object Reference or Forceful Browsing. (2014). *OWASP*. Retrieved from https://www.owasp.org/index.php/Ruby_on_Rails_Cheatsheet#Insecure_Direct_Object_Reference_or_Forceful_Browsing.

[8] Karadzhov, S. (2013). Learn ZF2 Zend Framework 2: Learning by Example. Slavey Karadzhov.

[9] Kern, C. (2014). Securing the Tangled Web: Preventing script injection vulnerabilities through software design. *Communications Of The ACM, 57*(9), 38-47. doi:10.1145/2643134.

[10] Lai, H. (2013). Rails SQL injection vulnerability: hold your horses, here are the facts. *Phusion Corporate Blog*. Retrieved from http://blog.phusion.nl/2013/01/03/rails-sql-injection-vulnerability-hold-your-horses-here-are-the-facts/.

[11] Orvis, W. S. (2013). Secure Development on Rails. *Pivotal Labs*. Retrieved from http://pivotallabs.com/bj-orvis-rails-security/.

[12] OWASP. (2013). OWASP Top 10 - 2013. The Ten Most Critical Web Application Security Risks. *The Open Web Application Security Project*. Retrieved from http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf.

[13] Patil, V. S., and Bamnote, Dr. G. R. (2014). An Overview to SQL Injection Attacks and its Countermeasures. *International Journal of Innovative Research & Development, Vol. 3, Issue 1*. Retrieved from http://ojms.cloudapp.net/index.php/ijird/article/view/45590/36927.

[14] Patterson, A. (2013). Multiple vulnerabilities in parameter parsing in Action Pack (CVE-2013-0156). Retrieved from https://groups.google.com/forum/?fromgroups=#!topic/rubyonrails-security/61bkgvnSGTQ.

[15] Rolling Your Own Auth. (n.d.). *Sessions, Cookies, and Authentication. The Odin Project*. Retrieved from http://www.theodinproject.com/ruby-on-rails/sessions-cookies-and-authentication#sts=Rolling Your Own Auth.

[16] Ruby, S., Thomas, D., and Hansson, D. H. (2011). Agile Web Development with Rails. 4th ed. Raleigh, NC; Dallas, TX: The Pragmatic Bookshelf.

[17] Ruby on Rails Security Guide. (n.d.). *Rails Guides*. Retrieved from http://guides.rubyonrails.org/security.html.

[18] SA-CORE-2014-005 - Drupal core - SQL injection. (2014, Oct. 15). *Drupal Security Advisories*. Retrieved from https://www.drupal.org/SA-CORE-2014-005.

[19] Security and Vulnerability Assessment. (2011). *Network Security Administrator Certification, Vol. 5*. EC-Council.

[20] SQL Injection. (2014, Aug. 14). *OWASP*. Retrieved from https://www.owasp.org/index.php/SQL_Injection.

[21] Top 10 2013-A4-Insecure Direct Object References. (2013). *OWASP*. Retrieved from https://www.owasp.org/index.php/Top_10_2013-A4-Insecure_Direct_Object_References.

[22] Typesafe Clients. Retrieved from https://typesafe.com/.

[23] Vaswani, V. (2014). Improve web application security with Zend Framework 2. Retrieved from http://www.ibm.com/developerworks/library/se-zend-security/index.html.

[24] What is Auvik. (2014). Auvik Networks. Retrieved from https://www.auvik.com/about/.

[25] WhiteHat Security. (2014). 2014 Website Security Statistics Report. Retrieved from http://info.whitehatsec.com/rs/whitehatsecurity/images/statsreport2014-20140410.pdf.