2014 Website Security Statistics Report
An end to the war of languages… maybe.

Whenever beginning a new software project, you have to make a choice: what programming languages or development frameworks to use? While it would be nice to select the most secure software stack at the start of a project, more often this decision is made for different reasons and security is commonly an afterthought.

The software stack decision is commonly based upon parameters such as:

- What the development teams are most familiar with.
- The current market momentum around the latest and greatest technology.
- What will generate code the fastest and can be maintained at a low cost.
- The available talent pool as the project grows.
- And of course, whatever gets the job done.

Familiarity with a programming language or development framework can drastically impact the ‘security’ outcome – whether it is designed to be secure by default or must be configured properly, and whether various libraries are available. Still, conventional wisdom suggests that most popular modern languages and frameworks (commercial and open source) perform similarly when it comes to an overall security posture. At least in theory, none is noticeably more secure than another. That being said, suggesting that PHP, Java, C# and others are any more secure than other languages is sure to spark heated debate. This is curious since the security of various programming languages rarely comes up when choosing what to use for a project.

In the security industry, you often hear that any programming language can be coded securely – or conversely, insecurely – therefore, it really comes down to the development process. We know information security, and in particular application security, is littered with conventional wisdom and lacks foundational data. So we must ask, is this really true? Do programming languages perform relatively the same security-wise in the real-world, the only place where it matters? Or is their thought-to-be-similar-performance only true when restricted to technical documentation and anecdotes of their advocates? This area warrants deeper investigation.

In this report, we put this area of application security understanding to the test by measuring how various web programming languages and development frameworks actually perform in the field. To which classes of attack are they most prone, how often and for how long; and, how do they fare against popular alternatives? Is it really true that the most popular modern languages and frameworks yield similar results in production websites?

By analyzing the vulnerability assessment results of more than 30,000 websites under management with WhiteHat Sentinel, we begin to answer these questions. These answers may enable the application security community to ask better and deeper questions, which will eventually lead to more secure websites. Organizations deploying these technologies can have a closer look at particularly risk-prone areas. Software vendors may focus on areas that are found to be lacking. Developers can increase their familiarity with the strengths and weaknesses of their technology stack. All of this is vitally important because security must be baked into development frameworks and must be virtually transparent. Only then will application security progress be made.
Déjà vu

Literally translated in English as “already seen,” déjà vu would best describe the current state of website security. The Verizon Data Breach Incidents Report, the FireHost “Superfecta” Attack Report, and many other industry reports are in firm alignment that websites and web applications remain one of the leading targets of cyber-attacks. Further, the conclusions drawn from these varying industry reports all point to the need for more secure software.

As a society, our reliance on the web continues to grow and the financial implications of these attacks are multiplying. Those who have sought to transfer some of these risks to more traditional instruments such as the burgeoning cyber insurance market have filed claims reaching as high as $20 million, with an average payout of just above $900,000*. Unfortunately, these policies are expensive and complicated, and do not always cover damages incurred from attacks.

Tide goes in, tide goes out...

The Software Development Lifecycle (SDLC) is well-defined and can be readily explained. If asked, IT teams are likely to point to a number of reasons for their choice of software stack which was likely made during the requirements or planning phase of development. The missing piece to achieving more secure software is the absence of relevant security data assisting in the technology selection process.

The requirements phase revisited

Application security professionals understand that websites with multiple users must have a way to securely handle each user, to ensure their sessions are initiated, managed, and terminated in a secure manner, and also to determine that if these sessions are authenticated, they must begin as an encrypted authentication event to avoid session fixation. What application security professionals are not currently able to add to the decision-making process is the determination, for example, of which programming language most securely handles sessions. If the choice of language is not one that handles sessions as securely as others, other determinations of how the chosen language is affected by the ability to correct session fixation issues, comes into play. Is it difficult to address? How long is the average time to fix those issues?

The above is just one example of how our choices could be positively affected by understanding the security of the software stacks we choose. The same is reflected across the many security-focused scenarios that should be made at the requirements gathering and design phases. Another example of such a scenario:

What happens if user X posts a question on the system? What happens if user Y then reads this question? We know there is a risk of Cross Site Scripting (XSS) but what we do not know is how well-equipped the chosen technology will be to protect against XSS. Is it difficult to address XSS?

As security professionals, we need to meet the business requirements while controlling cost, and ensuring the security and integrity of our systems and data. It would be nice to get to a place where application security professionals can say “we can keep development costs down and be secure by choosing language X, given our requirements and the relative ease-to-fix issues that are known to affect our design.” This report aims to examine the prevalence of threats per programming language and the ensuing analysis reveals the relative security of those languages.

Data Set

The data

All URLs for active slots with a PE* service level were selected. Clients that are used in QA Demo and other “non-real” clients were removed from the sample.

An unknown number of scan responses were missing and therefore, no language data was collected on these URLs. These only affected URLs that did not have a vulnerability detected.

Problems with the data

Scan responses that are more than 90-days old and not associated with a vulnerability are deleted from the server. This policy resulted in an unknown number of slots missing scan responses. While there is a lack of empirical evidence, it was determined that this was occurring on large slots and we probably had a sufficient number of URLs with vulnerabilities to accurately determine the language profile on the slot.

Slots were partially validated by comparing the slot information from the Salesforce database as it is the canonical authority on the client list. However, this data is dirty in that it is missing about 118 slot IDs that were used in the analysis.

We were using the most recent completed instance or, the most recent failed instance or, the active instance. There is a notable number of instances that have missing scan responses.

It is worth noting that we were able to detect SSI, Ruby, HTC, and GO, however, the sampling of these languages were statistically to small to include. We also detected Flash applications, but as Flash is not a server-side language it was generally excluded unless specifically stated.

The distribution of URLs per slot has a strong positive skew (8.1371). That is, the majority of slots had a small number of URLs and a few had a large number. This may be the result of differences in the size of the sites, the number of interactive elements and the language that was used. Therefore the average values reported will be based on the median and not the mean.

There were a large number of slots (811) that had fewer than 10 URLs. Of the 122,800 URLs used, there were 2,652 URLs associated with those slots that had fewer than 10 URLs.

The language classification system was able to find 179,146 markers for language.

*WhiteHat Security’s PE level service combines automated scanning by the Sentinel platform and combines manual custom testing to identify business logic flaws by the WhiteHat Threat Research Center.
Methodology

Terminology
Modern websites are composed of multiple technologies. WhiteHat Security defines the boundaries of a web application as a "slot.” The research data was derived from slots that had at least three completed assessments.

Language classification
Language classification was based on file extensions and HTTP header information. The URL file extension and the HTTP response body headers were used for language classification. A list of file extensions was mapped to selected languages. Many languages had several file extensions associated with them. The headers were examined and classification criteria were used to determine which language was used. Some languages and frameworks emit characteristic headers that were mapped to a given language. The same is true for cookies. Headers such as "content-disposition" had the filename section parsed and the file extension was used for classification. The "x-powered-by" header often listed the language used. The “content-type” header often listed the type of data, such as json or flash. In addition, there were many header names, such as “x-aspnet-version” that were used to indicate which language was used. The headers were collected and classified. Any unclassified headers went into a pool to be reexamined when sufficient numbers of the headers could be classified. 56.3% of URLs were able to be classified. We were able to determine at least one language on every slot. 31.7% of the slots had multiple languages.

Methodology
Scripts were written to extract supplemental data about the slots, such as industry, class, vulnerability status (open or closed), and timestamps. These data points are stored in our database and were extracted into a form that was imported into our statistical software. This extraction was done using SQL commands or Perl calls to our custom data framework. No cleaning of the data occurred at this phase, simply extraction.

We used the R programming language to validate, clean, and link data. This is well-documented and results are reproducible. Raw data from the extraction, transformation, and loading phases were loaded into R. All cleaning and validation was done in R so that the results could be reproduced. This used a series of custom written routines on top of the R libraries. The statistical analysis was also scripted so that it could be repeated. A markdown language was used to generate a report with the results from the analysis and comments about their meaning.
Key findings

In analyzing the data, our goal was to learn about the consequences of different technology decisions and to empower security decision makers that aim for a modern approach to addressing the real business challenges they face.

Fundamental data points
- We observed .Net to be the most widely represented language choice with 28.1% of web applications using the framework, followed by Java at 24.9% and ASP at 15.9%.
- There was no significant difference between languages in examining the highest averages of vulnerabilities per slot. Java had an average of 11.36 vulnerabilities per slot. .Net was found to have an average of 11.32 and PHP came in at 10.98.
- The bottom of the spectrum, or the most “secure”, also showed no significant difference between languages with the lowest averages of vulnerabilities per slot. ColdFusion was observed as having 7 vulnerabilities per slot. Perl was found to have the fewest with an average of 6.

Risk exposure does not vary widely between languages, as language choice does not affect the number of vulnerabilities. A one-way analysis of the variance showed that there was no statistically significant difference between these languages. In fact, there was no statistical difference, in terms of the average number of vulnerabilities per slot, between any of the languages in this study.

Percent of URLs by language

The most widely used languages are .NET and Java. Many organizations use ASP, a legacy language.
Risk exposure does not vary widely between languages, as language choice does not affect number of vulnerabilities.

Vulnerability classes
- 10.59% of ColdFusion sites had at least one SQL Injection vulnerability, the highest observed, followed by ASP with 7.67% and .NET 5.78%.
- Perl sites had a 67% chance of having at least one Cross Site Scripting (XSS) vulnerability, over 11% more than any other language.
- There was less than a 2% difference among the languages with Cross Site Request Forgery (CSRF).
- Many vulnerability classes were not affected by language choice.

Language observations
- ColdFusion has the best overall remediation rates at 74.3%.
- ColdFusion is significantly affected by Abuse of Functionality vulnerabilities with 5.93% of all sites having at least one occurrence, five times that of other languages.
- PHP had the lowest observed remediation rates.
- PHP is significantly affected by Insufficient Transport Layer Protection vulnerabilities, at 4.13% versus an average of 1% of all the combined major languages (Perl, Java, ASP, .Net & ColdFusion).
- Perl has the greatest remediation time of XSS, at 265 days to resolve.
- Perl's median remediation rate of CSRF is 23.8 days, three times faster than the next closest language, PHP at 68.97 days. All other languages were over 100 days.
- ColdFusion and PHP have fast remediation times when vulnerabilities are addressed: 50.5 and 47.5 days respectively.
- ASP takes the longest to fix all vulnerabilities averaging 139.97 days. .Net averages 111.86 days and PHP rounded out the bottom with an average of 47.49 days.
- ASP is remediating vulnerabilities at the same rate as other languages, but efforts are focused on critical issues.*

Industry observations
- Financial Services, HealthCare and Insurance organizations had the highest number of ASP sites by a 3:1 ratio.
- 86% of Gaming industry sites are written in PHP.
- 36% of Banking industry sites were written in Java and 55% in .Net.
- 43% of Manufacturing sites leveraged Perl as their language of choice.
- 35% of the Technology sector wrote their sites in PHP.

*Observation based on the types of vulnerability classes being remediated.
Average vulnerabilities

The big picture

In a moment, we will return to exploring the security of software stacks. But first, knowing that there was no significant difference between languages with the highest averages of vulnerabilities per slot we took a step back to look at how each website performed as they are most often comprised of multiple technologies. The importance of knowing the security posture of your websites is, from a security perspective, more important than language choice.

Vulnerabilities per language

Now that we have examined the average number of vulnerabilities observed in websites we will drill down and explore the vulnerabilities found in each language and the possible significance of these findings.

We found that 31% of all vulnerabilities found were in .Net applications. It must be noted that there are more websites written in .Net than all of the other languages in the study and that there was no evidence to suggest that .Net is any less secure based on this data point. In fact, .Net sites have a tendency to be larger and more complicated, which is directly correlated with having a larger attack surface and consequently more vulnerabilities.

Java – by virtue of its popularity with its extensive standard library class and its familiarity among programmers – accounted for 28% of all the vulnerabilities found. Again, the number of applications written in the language along with the complexities of the websites has to be considered as a contributing factor.

Originally released almost two decades ago ASP totaled 15% of all the discovered vulnerabilities. Although lacking the complexities of modern frameworks, many of the same critical vulnerability classifications were found to be present, as we will explore.

Another popular server-side scripting language, PHP also accounted for 15% of vulnerabilities discovered. A language still popular with Retail, Technology, and Financial Services organizations, PHP sites do not have a tendency to be as large or complex as .NET or Java sites but still suffer from many of the same issues.

The percentage of vulnerabilities found among the remaining languages experiences a sharp drop off from this point. ColdFusion, another platform almost 20 years in existence, only accounted for 4% of the vulnerabilities, while Perl registered at 2%.
Median days open

By language
The next data set we examined was the average number of days vulnerabilities remained open. Vulnerabilities go unfixed for many reasons and it begged the question as to whether there was anything to be learned from studying the length of time vulnerabilities were open in each of the languages.

ASP vulnerabilities were open for an median of 139 days, more so than any of the other languages.

PHP rounds out the top of the list with an average days open of 129.5 days. The numbers begin to significantly decline beyond Java, which had an average of 90.9 days open. The other languages were all under 45 days once we hit this drop off.

By class
Cross-Site Scripting in the PHP environment was open the least median number of days at 49. Perl and ASP respectively had XSS issues open an average of 184 and 135 days. .Net did not fare much better with XSS having an average number of days open of 126. Overall XSS appears to take a relative amount of effort regardless of the language class.

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, closely followed by Perl, which had the issue open an average of 19.4 days. ColdFusion topped the list averaging 107.4 days of exposed SQL Injection vulnerabilities. ASP’s average number of days open for SQL Injection was not far off of the ColdFusion average at 97.5 days. .Net and Java fell roughly in the middle at 51.4 and 64.8 days respectively.

The vulnerability with the highest average number of days open was Weak Password Recovery Validation in ASP websites, while not the most damaging vulnerability by itself, this could speak to a number of things such as, complexities of the language itself, programming experience necessary, or simply that this vulnerability class is not a priority in that environment.
Days open of top 5 vulnerability classes

Key findings:
- ASP vulnerabilities remain open the longest at 139 days.
- Cross-Site Scripting takes the longest to close in Perl taking 184 mean days.
- ColdFusion has the largest average days open for SQL Injection vulnerabilities at 107 mean days.
- Languages with the most security controls are taking the longest to remediate.
Cross-Site Scripting (XSS)

XSS regained the number one spot for being the most common vulnerability, after being overtaken by Information Leakage last year in all but one language: .Net, which still has Information Leakage as the number one vulnerability, followed by XSS.

Stand-outs

ColdFusion has a significantly higher percentage of abuse of functionality vulnerabilities at 6% compared to Java, the language with the second highest percentage at 1%. ColdFusion does, however, boast a 100% remediation rate versus Java’s 78% remediation rate or abuse of functionality vulnerabilities.

ColdFusion also suffers from the greatest percentage of SQL Injection at 11%. ASP takes second place with 8%. Java had the lowest percentage of SQL Injection at 1%. Again we see ColdFusion with a higher remediation rate of 96% versus Java’s 89%.

PHP’s rate of Insufficient Transport Layer at 4.13% is the highest, exceeding Java by almost 4-times which came in second at 1.34%. Coupled with a low remediation rate of 52%, PHP applications are at a much higher risk of exposing sensitive information.
Vulnerability class by language (percentage)

<table>
<thead>
<tr>
<th>Vulnerability Class</th>
<th>ASP</th>
<th>ColdFusion</th>
<th>.NET</th>
<th>Java</th>
<th>Perl</th>
<th>PHP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross-Site Scripting</td>
<td>49</td>
<td>46</td>
<td>35</td>
<td>57</td>
<td>67</td>
<td>56</td>
</tr>
<tr>
<td>Information Leakage</td>
<td>29</td>
<td>24</td>
<td>44</td>
<td>15</td>
<td>11</td>
<td>17</td>
</tr>
<tr>
<td>Content Spoofing</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>SQL Injection</td>
<td>8</td>
<td>11</td>
<td>6</td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Cross-Site Request Forgery</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Insufficient Transport Layer Protection</td>
<td>0.8</td>
<td>1</td>
<td>0.9</td>
<td>1</td>
<td>0.3</td>
<td>4</td>
</tr>
<tr>
<td>Abuse of Functionality</td>
<td>0.3</td>
<td>6</td>
<td>0.3</td>
<td>0.9</td>
<td>0.5</td>
<td>0.2</td>
</tr>
<tr>
<td>HTTP Response Splitting</td>
<td>0.9</td>
<td>3</td>
<td>0.8</td>
<td>2</td>
<td>0.8</td>
<td>0.3</td>
</tr>
<tr>
<td>Predictable Resource Location</td>
<td>0.1</td>
<td>0.1</td>
<td>0.0</td>
<td>0.2</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>Brute Force</td>
<td>0.7</td>
<td>0.3</td>
<td>1</td>
<td>2</td>
<td>0.8</td>
<td>1</td>
</tr>
<tr>
<td>URL Redirector Abuse</td>
<td>0.7</td>
<td>0.4</td>
<td>0.5</td>
<td>1</td>
<td>1</td>
<td>0.9</td>
</tr>
<tr>
<td>Insufficient Authorization</td>
<td>0.2</td>
<td>0.3</td>
<td>0.5</td>
<td>0.9</td>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>Fingerprinting</td>
<td>0.3</td>
<td>0.1</td>
<td>0.5</td>
<td>0.6</td>
<td>0.3</td>
<td>0.1</td>
</tr>
<tr>
<td>Session Fixation</td>
<td>0.2</td>
<td>0.3</td>
<td>0.2</td>
<td>0.6</td>
<td>0.1</td>
<td>0.3</td>
</tr>
<tr>
<td>Directory Indexing</td>
<td>-</td>
<td>-</td>
<td>0.0</td>
<td>0.0</td>
<td>-</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Key findings:
- Cross-Site Scripting regains the number one spot after being overtaken by Information Leakage last year in all but one language. .Net has Information Leakage as the number one vulnerability, followed by Cross-Site Scripting.
- ColdFusion has a rate of 11% SQL Injection vulnerabilities, the highest observed, followed by ASP with 8% and .NET 6%.
- Perl has an observed rate of 67% Cross-Site Scripting vulnerabilities, over 17% more than any other language.
- There was less than a 2% difference among the languages with Cross-Site Request Forgery.
- Many vulnerabilities classes were not affected by language choice.

Cross-Site Scripting is a significant issue across all languages.
Remediation rates

Progress measured

Remediation rate is the key accountability metric in any web application security program. Affected by many factors – including business functionality, complexity, and priority – the rate at which vulnerabilities are addressed is a key indicator of application security maturity. Likewise, languages are affected by unique factors as they pertain to remediation rates. Some languages have frameworks that make addressing different vulnerability classes less complex than others. The skill levels of the available programmer resources and the libraries used directly affect the security of individual languages.

Perl's remediation rate of XSS vulnerabilities bested the pack boasting an 85% rate. When it came to addressing SQL Injection vulnerabilities on the other hand, Perl had the lowest remediation rate of 18%. Perl did have the lowest number of days open for SQL Injection, so it is apparent that when these vulnerabilities were addressed, they closed faster than any other language.

SQL Injection had a 96% remediation rate in ColdFusion applications and every single abuse of functionality vulnerability found in ColdFusion sites was remediated.

Legacy ASP sites exhibited remediation rates on par with other languages while suffering from large windows of exposure suggesting that there were strategic decisions being made regarding the maintenance and security of these sites. Classic ASP struggles from being developed at a time in the history of the web when the breadth of attacks were not what they are today, yet what we see here is a great amount of effort to keep pace with the remediation rates of modern frameworks.

There was no immediate data to suggest that language choice greatly affected remediation rates given that a language such as ASP is able to keep track while PHP lagged much further behind. The efforts to keep pace may be enough reason to retire these legacy sites, however more often this choice is based on the need for update functionality.
Key findings

- Perl remediates 85% of all Cross-Site Scripting vulnerabilities, the highest rate among all languages but only 18% of SQL Injection.
- Net and Java have the same remediation rate of SQL Injection at 89%.
- ColdFusion remediates 100% of its Abuse of Functionality vulnerabilities, 96% of its SQL Injection, and 87% of Insufficient Transport Layer Protection vulnerabilities.

ASP is remediating at the same rate as the other languages, focusing on mission critical vulnerabilities.
Industry analysis

Across the board

An oft-repeated response to inquiries about the languages their websites use is “a little of everything,” however, the data showed that organizations tend to have a significant amount of one or two languages with a very minimal investment in the others. The breakdown is by far not equally spread, yet what we see is an attempt to approach the security of their websites as though there were truly a distribution that favored no particular language. This tends to lead to choices in tools, services and skill sets that may not have enough focus or expertise on the areas with the greatest financial investment or impact.

There were some definite “favorites” among the industries. No doubt that there exists correlations between financial drivers and functionality. The Gaming industry favored PHP for their applications more so than other industries by a staggering amount, 83% of all the Gaming industries websites were written in PHP, likely due to it’s speed and low resource consumption. The reason for this was not immediately clear to us, however, the remediation rates among ASP, ColdFusion, .Net, and Java were considerably higher than those PHP and Perl sites. PHP was favored by Gaming and Food and Beverage, while Perl was favored by Manufacturing sectors.

Java enjoyed a relatively even distribution among all of the industries while .Net was a more favorable language choice to others, namely the Insurance, Pharmaceutical and Transportation sectors. These two languages remain the number one and two choice for those industries.

A legacy tale

Financial services favor .Net for their applications, and also maintain the largest number of legacy ASP applications by a 3:1 ratio. It is unlikely that many, if any, new applications are being built with classic ASP so it stands to reason that these applications are very important to revenue-generating and mission-critical applications. The Retail sector had the second largest collection of classic ASP applications which appear to have a similar challenge, revenue-generating sites that cannot for one business reason or another be retired.

This data point became particularly interesting to us when we considered that ASP has the greatest time-to-fix but has a remediation rate on par with all other languages. These sites, while unable to be sun-stetted because they are mission-critical and/or revenue-generating are taking a cost to risk approach of fixing vulnerabilities.

It was not lost on us that both the financial services sector and the retail industry face heavy regulation and compliance drivers. Our 2013 Website Statistics research showed that compliance was also the number one reason vulnerabilities went unresolved* so that did not explain why both of these industry’s legacy sites performed as well they did. In fact, the driver here was revenue not compliance or regulation.

No industry has an even breakdown. Everyone skews in a different direction.

Key findings
- Financial Services has the highest number of ASP sites by count, by almost 3 to 1.
- 83% of Gaming Industry sites written in PHP.
- 49% of the Banking Industry applications were written in Java & 42% in .Net.
- 32% of Manufacturing sites leveraged Perl as their language of choice.
- The Technology sector wrote 35% of their sites in PHP.
Recommendations

Language choice
Cross-Site Scripting is the one vulnerability that appears to be affected by language choice, however, regular assessments and focused remediation efforts can manage that risk. Language choice begins at the architecture and design stage of application development; security must begin here as well. Understanding the impact of those decisions early will help address the management of the risk later on. The practice of choosing languages based on business needs and functionality is not in question here; those factors are how our business derives revenue. However, we should not overly rely on frameworks to provide protection. More security controls in a framework tend to make it harder to remediate because developers don’t know how to fix it. Instead, solid coding practices and code review are your best tools. Ensuring that software is tested in all phases of development and including code reviews of web services as they are critical components to modern complex web applications.

Governance
Corporate Governance has long since spawned excellent IT governance frameworks. The inclusion of application security into your existing governance frameworks is vital for the reduction in the risk that is inherent in web applications. The application investment decisions should align with the strategic priorities of the information security group. If budget were no object this would not be necessary, however, budget is limited and security spending must be allocated to efforts that reduce as much risk as possible while balancing budgets.

Current governance frameworks can be over-arching and require a herculean effort to apply to your SDLC process. It is very important to not allow application security governance to exceed the amount of effort required to deliver a project. If a developer has to spend more time filling out request for change forms or your security department becomes inundated by the processes of application assessment, the teams will lose trust in the system and find ways around it.

When it comes to governance, one-size does not fit-all. This is why the frameworks are as large as they are. They are intended to serve as a guideline for all. The governance that you apply to your application security program must match your needs and requirements.
A risk-based approach

A risk-based approach is a management method for application security that relies on quantifying risk in dollars and cents as the main driver for security decisions. This approach cannot be applied during the language selection process, however, that choice is best made based on business needs and functionality.

The determined risk is then the probability and potential loss magnitude in dollars, represented by application vulnerabilities.

Once risk is quantified, meaningful comparisons can be made to drive decisions.

These decisions include:

- Remediation decisions (to remediate an application vulnerability or not, and when to remediate)
- Prioritization decisions (stack ranking of which vulnerabilities should be remediated first)
- Resource decisions (opportunity cost to apply limited development resources to fix vulnerabilities)
- Funding decisions (establishing business case justifications for web application security projects, how much budget to allocate towards web application security versus alternative security budgets)
- Policy decisions (establishing policies for handling web application risk – remediate, mitigate, postpone, transfer, accept)
- Exception decisions (e.g. when to allow exceptions to remediation SLAs, and for how long)
Appendix

Remediation percent by vulnerability class

<table>
<thead>
<tr>
<th>Vulnerability Class</th>
<th>ASP</th>
<th>ColdFusion</th>
<th>.NET</th>
<th>Java</th>
<th>Perl</th>
<th>PHP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross-Site Scripting</td>
<td>79</td>
<td>75</td>
<td>76*</td>
<td>71</td>
<td>85</td>
<td>65</td>
</tr>
<tr>
<td>Information Leakage</td>
<td>67</td>
<td>60</td>
<td>72</td>
<td>51</td>
<td>24</td>
<td>36</td>
</tr>
<tr>
<td>Content Spoofing</td>
<td>74</td>
<td>77</td>
<td>74</td>
<td>74</td>
<td>84</td>
<td>55</td>
</tr>
<tr>
<td>SQL Injection</td>
<td>87</td>
<td>96</td>
<td>89</td>
<td>89</td>
<td>18</td>
<td>25</td>
</tr>
<tr>
<td>Cross-Site Request Forgery</td>
<td>60</td>
<td>46</td>
<td>54</td>
<td>51</td>
<td>69</td>
<td>54</td>
</tr>
<tr>
<td>Insufficient Transport Layer Protection</td>
<td>50</td>
<td>87</td>
<td>46</td>
<td>51</td>
<td>100</td>
<td>52</td>
</tr>
<tr>
<td>Abuse of Functionality</td>
<td>62</td>
<td>100</td>
<td>62</td>
<td>78</td>
<td>70</td>
<td>65</td>
</tr>
<tr>
<td>HTTP Response Splitting</td>
<td>80</td>
<td>80</td>
<td>51</td>
<td>40</td>
<td>40</td>
<td>75</td>
</tr>
<tr>
<td>Predictable Resource Location</td>
<td>71</td>
<td>50*</td>
<td>38</td>
<td>22</td>
<td>100*</td>
<td>67</td>
</tr>
<tr>
<td>Brute Force</td>
<td>34</td>
<td>62</td>
<td>41</td>
<td>25</td>
<td>19</td>
<td>37</td>
</tr>
<tr>
<td>URL Redirector Abuse</td>
<td>44</td>
<td>53</td>
<td>49</td>
<td>66</td>
<td>26</td>
<td>32</td>
</tr>
<tr>
<td>Insufficient Authorization</td>
<td>80</td>
<td>80</td>
<td>83</td>
<td>51</td>
<td>32</td>
<td>50</td>
</tr>
<tr>
<td>Fingerprinting</td>
<td>33</td>
<td>67*</td>
<td>41</td>
<td>56</td>
<td>20*</td>
<td>33</td>
</tr>
<tr>
<td>Session Fixation</td>
<td>45</td>
<td>42</td>
<td>49</td>
<td>34</td>
<td>0*</td>
<td>48</td>
</tr>
<tr>
<td>Directory Indexing</td>
<td>—</td>
<td>—</td>
<td>100*</td>
<td>100*</td>
<td>—</td>
<td>100</td>
</tr>
</tbody>
</table>

*WhiteHat Security defines the boundaries of a web application as a “slot”.
*Limited amount of data available
More security controls in the framework correlates with longer remediation time.
### Percent of languages in use by industry

<table>
<thead>
<tr>
<th>Industry</th>
<th>ASP</th>
<th>Coldfusion</th>
<th>.NET</th>
<th>Java</th>
<th>Perl</th>
<th>PHP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automotive</td>
<td>8</td>
<td>17</td>
<td>25</td>
<td>50</td>
<td></td>
<td>33</td>
</tr>
<tr>
<td>Banking</td>
<td>19</td>
<td>12</td>
<td>42</td>
<td>49</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Education</td>
<td>25</td>
<td>17</td>
<td>33</td>
<td>33</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>Energy</td>
<td>11</td>
<td>5</td>
<td>63</td>
<td>32</td>
<td>5</td>
<td>21</td>
</tr>
<tr>
<td>Entertainment &amp; Media</td>
<td>7</td>
<td>10</td>
<td>28</td>
<td>37</td>
<td>1</td>
<td>31</td>
</tr>
<tr>
<td>Financial Services</td>
<td>32</td>
<td>7</td>
<td>55</td>
<td>36</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Food &amp; Beverage</td>
<td>6</td>
<td>28</td>
<td>28</td>
<td>33</td>
<td>6</td>
<td>50</td>
</tr>
<tr>
<td>Gaming</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>13</td>
<td>9</td>
<td>83</td>
</tr>
<tr>
<td>Government</td>
<td>50</td>
<td></td>
<td>50</td>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>Healthcare</td>
<td>31</td>
<td>10</td>
<td>47</td>
<td>40</td>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td>Insurance</td>
<td>38</td>
<td>13</td>
<td>71</td>
<td>42</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>IT</td>
<td>21</td>
<td>6</td>
<td>36</td>
<td>39</td>
<td>8</td>
<td>30</td>
</tr>
<tr>
<td>Manufacturing</td>
<td>15</td>
<td>4</td>
<td>28</td>
<td>32</td>
<td>43</td>
<td>9</td>
</tr>
<tr>
<td>Non Profit</td>
<td>10</td>
<td>60</td>
<td>50</td>
<td>30</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>Pharmaceutical</td>
<td>67</td>
<td></td>
<td>67</td>
<td>33</td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>Retail</td>
<td>28</td>
<td>16</td>
<td>37</td>
<td>42</td>
<td>5</td>
<td>23</td>
</tr>
<tr>
<td>Social Networking</td>
<td>19</td>
<td>10</td>
<td>29</td>
<td>24</td>
<td>5</td>
<td>57</td>
</tr>
<tr>
<td>Technology</td>
<td>14</td>
<td>21</td>
<td>28</td>
<td>30</td>
<td>3</td>
<td>34.6</td>
</tr>
<tr>
<td>Telecommunications</td>
<td>22</td>
<td>3</td>
<td>34</td>
<td>44</td>
<td>3</td>
<td>50</td>
</tr>
<tr>
<td>Transport &amp; Distribution</td>
<td>15</td>
<td>8</td>
<td>69</td>
<td>46</td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>
About WhiteHat Security

Founded in 2001 and headquartered in Santa Clara, California, WhiteHat Security provides end-to-end solutions for application security. The company’s cloud website vulnerability management platform and leading security engineers turn verified security intelligence into actionable insights for customers. Through a combination of core products and strategic partnerships, WhiteHat Security provides complete application security at a scale and accuracy unmatched in the industry. WhiteHat Sentinel, the company’s flagship product line, currently manages thousands of websites – including sites in highly regulated industries, such as e-commerce, financial services and healthcare companies. For more information, visit www.whitehatsec.com.