WhiteHat SECURITY

WhiteHat Website Security Statistic Report

Spring 2010, 9th Edition

9th edition

Introduction

Security-conscious organizations make implementing a software security development lifecycle a priority. As part of the process, they evaluate a large number of development technologies for building websites. The assumption by many is that not all development environments are created equal. So the question often asked is, "What is the most secure programming language or development framework available?"

Clearly, familiarity with a specific product, whether it is designed to be secure-by-default or must be configured properly, and whether various libraries are available, can drastically impact the outcome. Still, conventional wisdom suggests that most popular modern languages / frameworks (commercial & open source) perform relatively similarly when it comes to an overall security posture. At least in theory, none is markedly or noticeably more secure than another. Suggesting PHP, Java, C# and others are any more secure than other frameworks is sure to spark heated debate.

As has been said in the past, "In theory, there is no difference between theory and practice. But, in practice, there is." Until now, no website security study has provided empirical research measuring how various Web programming languages / frameworks actively 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 popular modern languages / frameworks yield similar results in production websites?

By analyzing the vulnerability assessment results of nearly 1,700 websites under WhiteHat Sentinel management, we may begin to answer some of these questions. These answers may enable the website 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 found lacking; and, developers will increase their familiarity with the strength and weaknesses of their technology stack. All of this is vitally important because security must be baked into development frameworks and be virtually transparent. Only then will application security progress be made.

Which Web programming languages are most secure?

Cyber-criminals are evolving. Many are in it for the money, others the data, some prefer silent command & control, and more still seek to embarrass or harass their victims. While attackers' motivations are consistent, their methods and techniques are anything but predictable. This has made Web security a moving target. To protect themselves, organizations need timely information about the latest attack trends and defense measures, as well as visibility into their website vulnerability lifecycle.

Through its Software-as-a-Service (SaaS) offering, WhiteHat Sentinel[1], WhiteHat Security is able to deliver the knowledge and solutions that organizations need to protect their brands, attain PCI-DSS[2] compliance and avert potentially devastating and costly breaches.

The WhiteHat Security Website Security Statistics Report provides a one-of-a-kind perspective on the state of website security and the issues that organizations must address to safely conduct business online. The WhiteHat Security report presents a statistical picture of current website vulnerabilities, accompanied by WhiteHat expert analysis and recommendations.

WhiteHat's report is the only one in the industry to focus solely on unknown vulnerabilities in custom Web applications, code unique to an organization, within real-world websites.

Key Findings

- Empirically, programming languages / frameworks do not have similar security postures when deployed in the field. They are shown to have moderately different vulnerabilities, with different frequency of occurrence, which are fixed in different amounts of time.
- The size of a Web application's attack surface alone does not necessarily correlate to the volume and type of issues identified. For example Microsoft's .NET (ASPX) and Struts (DO), with near-average attack surfaces, turned in the two lowest historical vulnerability averages. ASPX and DO websites have had an average of 18.7 and 19.9 serious* vulnerabilities respectively.
- Perl (PL) had the highest average number of vulnerabilities found historically by a wide margin, at 44.8 per website and also the largest number currently at 11.8.
- Struts (DO) edged out Microsoft's .NET (ASPX) for the lowest average number of currently open vulnerabilities per website at 5.5 versus 6.2.
- Cold Fusion (CFM) had the second highest average number of vulnerabilities per website historically at 34.4, but has the lowest likelihood of having a single serious* unresolved vulnerability if currently managed under WhiteHat Sentinel (54%). Closely following was Microsoft ASP Classic, which at 57% beat its successor Microsoft .NET by a single point.
- Perl (PL), Cold Fusion (CFM), JSP, and PHP websites were the most likely to have at least one serious* vulnerability, at roughly 80% of the time. The other languages / frameworks were only within ten percentage points.
- Among websites containing URLs with Microsoft's .NET (ASPX) extensions, 36% of their vulnerabilities had Microsoft ASP Classic extensions. Conversely, 11% of the vulnerabilities on ASP websites had ASPX extensions.
- 37% of Cold Fusion (CFM) websites had SQL Injection vulnerabilities, the highest of all measured, while Struts (DO) and JSP had the lowest with 14% and 15%.
- At an average of 44 days, SQL Injection vulnerabilities were fixed the fastest on Microsoft ASP Classic websites, just ahead of Perl (PL) at 45 days.
- 79% of "Urgent" Severity SQL Injection vulnerabilities were fixed on Struts (DO) websites, the most of the field. This is followed by Microsoft's .NET (ASPX) at 71%, Perl (PL) at 71,% and remainder between 58% and 70% percent.
- More than 8 in 10 Perl (PL) websites have Cross-Site Scripting issues (the highest), as compared to just over half of ASPX (the lowest).
- PHP and Perl websites were among the worst in average vulnerability counts, but had fastest average Cross-Site Scripting remediation times -- 52 and 53 days respectively. At the same time, Microsoft's .NET (ASPX) performed among the best in vulnerability count averages, but placed dead last for remediation times at 87 days.
- The vulnerability resolution rate for "Critical" severity Cross-Site Scripting vulnerabilities (non-persistent) in all measured languages / frameworks hovered in the 50% - 60% range, with PHP the only standout at 66%.
- Perl and JSP performed impressively on Insufficient Authorization fix times with 20 and 29 days respectively. Historically, Insufficient Authorization vulnerabilities have taken over 50 days to fix.

Data Overview

- Data collected from January 1, 2006 to March 25, 2010
- 1,659 total websites. (Over 300 organizations, generally considered security early adopters, and serious about website security.)
- 24,286 verified custom Web application vulnerabilities
- Vast majority of websites assessed for vulnerabilities multiple times per month.
- Vulnerabilities classified according to WASC Threat Classification, the most comprehensive listing of Web application vulnerabilities (Figure 1)
- Vulnerability severity naming convention aligns with PCI-DSS

Vulnerability Assessment & Data Collection Process

Built on a Software-as-a-Service (SaaS) technology platform, WhiteHat Sentinel combines proprietary scanning technology with expert analysis, to enable customers to identify, prioritize, manage and remediate website vulnerabilities. WhiteHat Sentinel focuses solely on previously unknown vulnerabilities in custom Web applications -- code unique to an organization (Figure 2). Every vulnerability discovered by any WhiteHat Sentinel Service is verified for accuracy and prioritized by severity and threat.

In order for organizations to take appropriate action, each website vulnerability must be independently evaluated for business criticality. For example, not all Cross-Site Scripting or SQL Injection vulnerabilities are equal, making it necessary to consider its true "severity" for an individual organization. Using the Payment Card Industry Data Security Standard[4] (PCI-DSS) severity system (Urgent, Critical, High, Medium, Low) as a baseline, WhiteHat Security rates vulnerability severity by the potential business impact if the issue were to be exploited and does not rely solely on default scanner settings.

Technical Vulnerabilities	Business Logic Flaws
Command Execution – Buffer Overflow – Format String Attack – LDAP Injection – OS Commanding – SQL Injection – SSI Injection – XPath Injection	**Authentication** – Brute Force – Insufficient Authentication – Weak Password Recovery – Validation – Cross-Site Request Forgery
Information Disclosure – Directory Indexing – Information Leakage – Path Traversal – Predictable Resource Location	**Authorization** – Credential/Session Prediction – Insufficient Authorization – Insufficient Session Expiration – Session Fixation
Client-Side – Content Spoofing – Cross-site Scripting (XSS) – HTTP Response Splitting	**Logical Attacks** – Abuse of Functionality – Denial of Service – Insufficient Anti-automation – Insufficient Process Validation

Figure 1. WASC Threat Classification Version 1

WhiteHat Security is currently in the process of migrating to the newly released WASC Threat Classification V2[3]

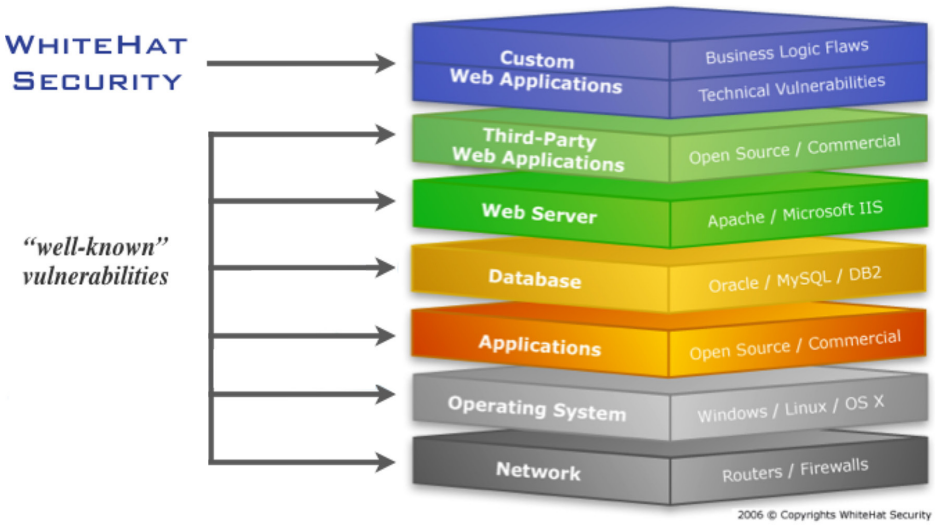

Figure 2. Software / Vulnerability Stack

The websites covered by WhiteHat Sentinel likely represent the most "important" and "secure" sites on the Web, owned by enterprises that are serious about their security.

WhiteHat Sentinel offers three different levels of service (Premium, Standard, and Baseline) to match the level of security assurance required by the organization[5]. And, WhiteHat Sentinel exceeds PCI 6.6 and 11.3.2 requirements for Web application scanning[6].

Proprietary Scanning Technology

- "Production safe" scanning – Non-invasive testing with less performance impact than a single user.
- Battlefield testing – Track record of identifying more vulnerabilities than commercial scanners.
- Highly Accurate – False-positives are virtually eliminated by the WhiteHat Security Operations Team.
- Seamless support for Web 2.0 technology – modern websites using JavaScript, Macromedia Flash, AJAX, Java Applets, or ActiveX
- Authenticated scans – Patented automated login technology for complete website mapping.
- Business Logic Coverage – customized tests analyze every Web form, business process, and authentication / authorization component.

Factors influencing the results:

Websites range from highly complex and interactive with large attack surfaces to static brochureware. Static brochureware websites, because of a generally limited attack surface, will have a limited number of "custom" Web application vulnerabilities.

Vulnerabilities are counted by unique Web application and class of attack. If there are five parameters in a single Web application (/foo/webapp.cgi), three of which are vulnerable to SQL Injection, it is counted as one vulnerability (not three).

"Best practice" findings are not included in the report. For example, if a website mixes SSL content with non-SSL on the same Web page, while this may be considered a business policy violation, it must be taken on a case-by-case basis. Only issues that can be directly and remotely exploitable are included.

Vulnerability assessment processes are incremental and ongoing, the frequency of which is customer-driven and as such should not automatically be considered "complete." The vast majority of WhiteHat Sentinel customers have their sites assessed multiple times per month.

New attack techniques are constantly being researched to uncover previously unknown vulnerabilities. This makes it best to view the data as a best-case scenario. Likewise, assessments may be conducted in different forms of authenticated states (i.e. user, admin, etc.).

Websites may be covered by different WhiteHat Sentinel service levels (Premium (PE), Standard (SE), Baseline (BE)) offering varying degrees of testing criteria, but all include verification. PE covers all technical vulnerabilities and business logic flaws identified by the WASC Threat Classification v1 (and some beyond). SE focuses primarily on the technical vulnerabilities. BE bundles critical technical security checks into a production-safe, fully-automated service.

Data Analysis

For our report's first attempt at contrasting and comparing various server-side Web programming languages / frameworks, we sliced our current statistics report template by well-recognized file extensions within URLs. Specifically we focused on ASP, ASPX, CFM, DO, JSP, PHP, and PL extensions. Microsoft ASP Classic, Microsoft .NET (normally developed in Visual Basic or C#), Adobe's Cold Fusion, Struts (typically Apache Struts Java), Java Server Pages, PHP, and Perl. We did not have a statistically representative sample (40+ websites) to evaluate Web applications written in Ruby on Rails, Python, C++, etc.

What is a "website"?

Websites, which may be a collection of multiple Web servers and hostnames, often utilize more than one programming language or framework. As such, a single website may contain vulnerabilities with multiple different extensions. For example (Figure 3), among websites with ASPX URLs, 36 percent of their vulnerabilities have ASP extensions. Conversely, 11 percent of the vulnerabilities on ASP websites have ASPX extensions. In essentially all cases we see a significant overlap in vulnerable extensions. This should be taken into consideration when viewing all charts contained within this report.

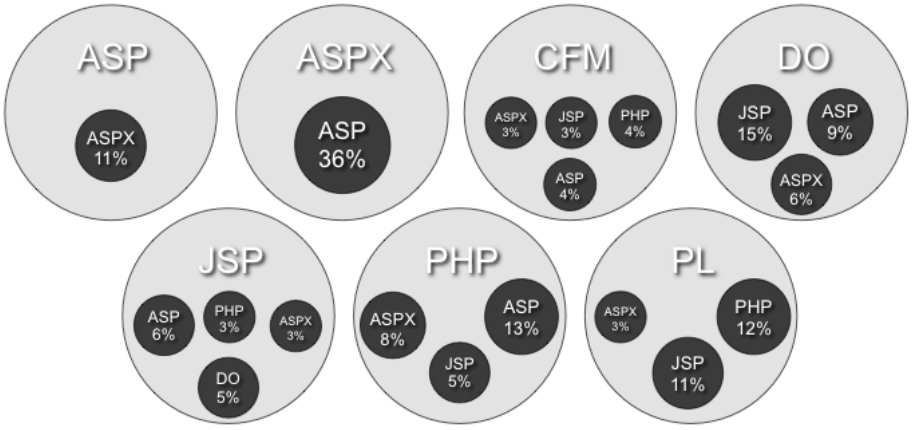

Figure 3. Vulnerability Overlap.

Percentage of a websites vulnerabilities with extensions different than the programming language / development framework being measured.

	ASP	ASPX	CFM	DO	JSP	PHP	PL
Average # of inputs (attack surface) per website	470	484	457	569	919	352	588
Average ratio of vulnerability count / number of inputs	8.7%	6.2%	8.4%	6.3%	9.8%	8.1%	11.6%

Figure 4. Attack Surface and Vulnerable Input Ratio (Compared by Extension)

Attack Surface and Number of Vulnerabilities

The size of an application's attack surface is an extremely important security metric. Application inputs are areas where arbitrary data is received, potentially leaving the software open to attack (attack surface). Application inputs include, but are not limited to, query and POST data parameter names/values, cookies, and files paths/names. These numbers (Figure 4) come from crawling all of a website's Web pages while maintaining a logged-in state.

The overall average number of input points for all websites is 548, with PHP on the low-end with 352, JSP on the high end at 919, and the remainder falling in-between. Generally, the larger the attack surface, the more vulnerabilities one would expect to find, or at least the odds of this being true increases. With this assumption, we would expect similarly ordered results within the "Avg. # of serious* vulnerabilities per website during the WhiteHat Sentinel assessment lifetime" in (Figure 5).

However, in this case we see two of the three development environments with the lowest attack surface, PHP and CFM, actually have two of the three highest average number of serious* vulnerabilities historically -- 26.6 and 34.3 respectively. ASPX and DO, with near-average attack surfaces, turned in the two lowest average serious** vulnerability numbers with 18.7 and 19.9. Perl, also with a near-average attack surface, claimed the highest number by a wide margin at 44.8 serious* vulnerabilities per website.

Comparing historical vulnerability numbers against those currently unresolved is also particularly insightful because progress over time can be measured. All languages / frameworks across the range have made huge improvements with overall vulnerability resolution, between an average of 5.5 and 11.8, but their relative positioning has changed only slightly. All are within about three to five average vulnerabilities from each other. Interestingly, Struts (DO) closely edged out Microsoft's .NET (ASPX) for the lowest average number of vulnerabilities currently open.

As anyone with operational security experience will attest, attackers only need to exploit a single vulnerability to compromise a system, the data contained within, or its users. A large volume of vulnerabilities only makes the odds of their discovery and malicious use greater. With free and readily available tools, often this hurdle becomes moot. Measuring the percentage likelihood of "websites having had at least one serious*** vulnerability," both historically and currently, provides much needed context against total counts.

	ASP	ASPX	CFM	DO	JSP	PHP	PL
Websites having had at least one serious* vulnerability	74%	73%	86%	77%	80%	80%	88%
Websites currently with at least one serious* vulnerability	57%	58%	54%	56%	59%	63%	75%
Avg. # of serious* vulnerabilities per website during the WhiteHat Sentinel assessment lifetime	25	18.7	34.3	19.9	25.8	26.6	44.8
Avg. # of serious* severity unresolved vulnerabilities per website	8.9	6.2	8.6	5.5	9.6	8.3	11.8

Figure 5. Vulnerability Overview (Compared by Extension)

* *WhiteHat Sentinel seeks to identify all of a website's externally available attack surface, which may or may not require crawling all of its available links.*

** *The framework's current version or configuration was not taken into account, which may often impact its security posture. In future reports, we'd like to improve our identification methodology to provide more fine-grained results.*

*** *Serious vulnerabilities are those of HIGH, CRITICAL, or URGENT severity as defined by PCI-DSS naming conventions. Exploitation could lead to significant and direct business impact.*

Historically PL, CFM, JSP, and PHP websites have been the most likely to have at least one serious* issue, occurring in 80% or more (Figure 5). Notably, the remainder of the field was not far behind, within only ten percentage points. Compared against the "Avg. # of serious* severity unresolved vulnerabilities per website," we again see large improvement across the group with all dropping into the 50 percent range, with Perl and PHP being the exceptions. There are also two significant stand outs: Cold Fusion (CFM), which has the second to highest average number of vulnerabilities historically. Now, the average Cold Fusion (CFM) website actively managed under WhiteHat Sentinel has the lowest likelihood (54%) of having a serious* unresolved vulnerability. This is followed closely by Microsoft ASP Classic, which at 57 percent beats its successor Microsoft .NET by a single point.

Vulnerability Pervasiveness

The most prevalent issues are calculated by the percentage likelihood of a particular vulnerability class occurring within websites (Figure 6). This approach minimizes data skewing in website edge cases that are either highly secure or extremely risk-prone.

When it comes to the likelihood of a single vulnerability appearing in a website, the ordering of the classes across the languages / frameworks is very similar. To what degree they are affected (or more / less likely to be affected) is not similiar. For example, note that 37 percent of CFM websites have SQL Injection, while DO and JSP are close at the lower rates of 14 and 15 percent respectively. More than 8 in 10 Perl websites have Cross-Site Scripting issues, as compared to just over half of ASPX websites. Curiously, CSRF did not make the Top Ten for either Perl or PHP, but Directory Indexing did. In all cases, the diversity of issues present across all languages / frameworks continues to be striking.

To supplement vulnerability likelihood statistics, the following graph (Figure 7) illustrates prevalence by class in the overall vulnerability population. Notice how it differs in type and degree from the Top Ten graph. The reason is that one website may possess hundreds of unique issues of a specific class, such as Cross-Site Scripting, Information Leakage, or Content Spoofing, while another website may not contain any.

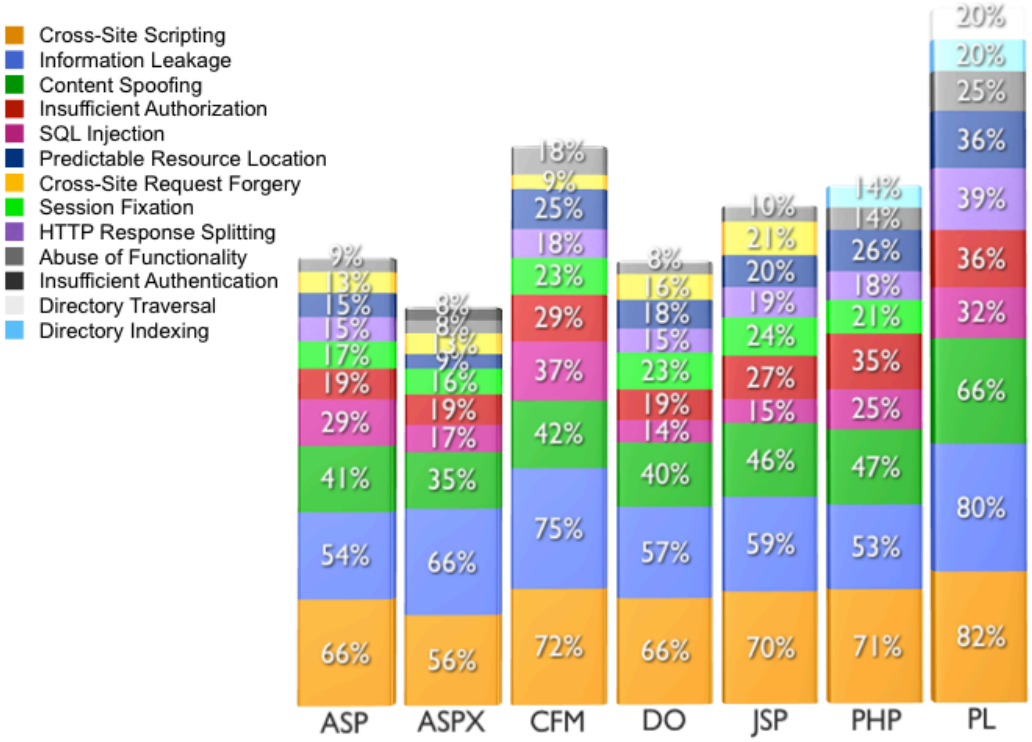

Figure 6. Top Ten Vulnerability Classes (Compared by Extension)

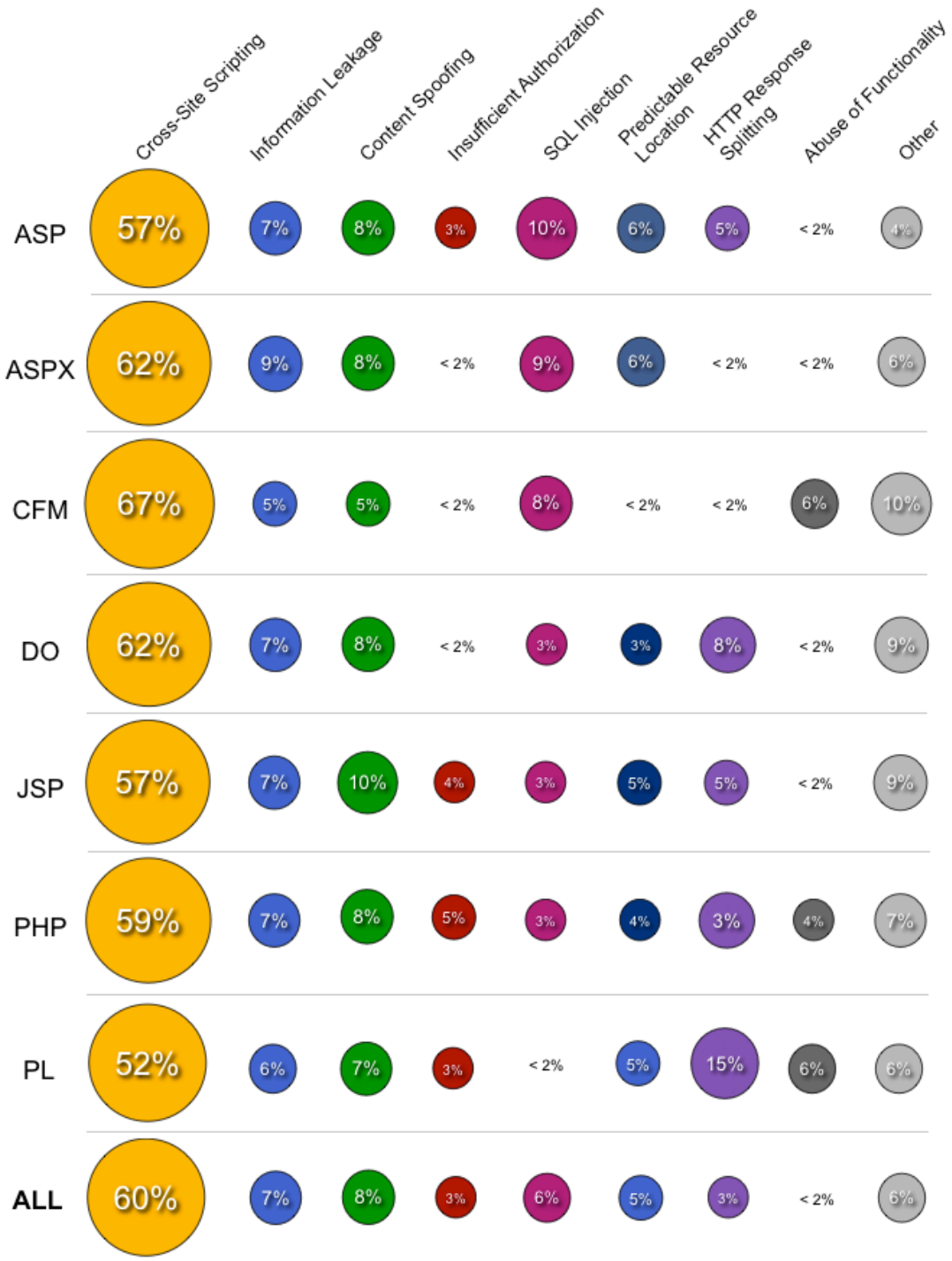

Figure 7. Vulnerability Population (Compared by Extension)

SQL Injection and Cross-Site Request Forgery are likely under-represented in both Figure 6 and Figure 7. To protect against SQL Injection attacks, industry best-practices suggest verbose error messages should be disabled to increase the difficulty of its exploitation. This act also has the side effect of increasing the difficulty for scanning technology to identify open issues. Cross-Site Request Forgery is under-represented because scanning technology industrywide is still extremely limited in its detection capability. Most serious* issues are still found by hand as were the majority of CSRF vulnerabilities identified in this report.

Time to Fix

When website vulnerabilities are identified, there is a certain amount of time required for the issue to be resolved. Resolution could take the form of a software update, configuration change, Web application firewall rule, etc. Ideally, the time to fix should be as short as possible because an open vulnerability represents an opportunity for hackers to exploit the website, but no remedy is instantaneous. To perform this analysis, we focused on the most common vulnerability classes identified (and resolved) within the last twelve months between March 25, 2009 and March 25, 2010 (Figure 8).

Factors influencing the results:

Should a vulnerability be resolved, it could take up to seven days before it is retested and confirmed closed by WhiteHat Sentinel, depending upon the customer's scan schedule. A customer can proactively use the auto-retest function to get real-time confirmation of a fix.

Not all vulnerabilities identified within this period have been resolved, which means the time to fix measurements are likely to grow (Figure 8).

Cross-Site Scripting (XSS), is well-known for being the most commonly found vulnerability, and every day it seems this issue is becoming more widely exploited. So while PHP and Perl placed among the worst in various vulnerability counts, it is curious to see them having the fastest average remediation times – 52 and 53 days respectively. At same time, it is surprising that Microsoft's .NET (ASPX), which performed amongst the best in vulnerability count averages, placed dead last at 87 days for remediation. Second to last was Microsoft ASP Classic (ASP) at 84 days.

At this point we can only theorize about why ASPX websites have such long vulnerability remediation times. One reason could be that Microsoft .NET developers tend to over-rely upon native XSS protections enabled by default in the framework and do not independently secure their code with output filters. This becomes problematic in two different scenarios:

1. *When a XSS filter-bypass vulnerability is found in the .NET framework itself, the organization may select to patch their production systems, as opposed to fixing all their application code (not an unreasonable cost-benefit choice). Depending on the circumstances of the issue and patch deployment cycle, this process may not occur quickly in either case.*
2. *A .NET security feature may be purposely disabled because it breaks expected website functionality, which exposes all the vulnerable application code.*

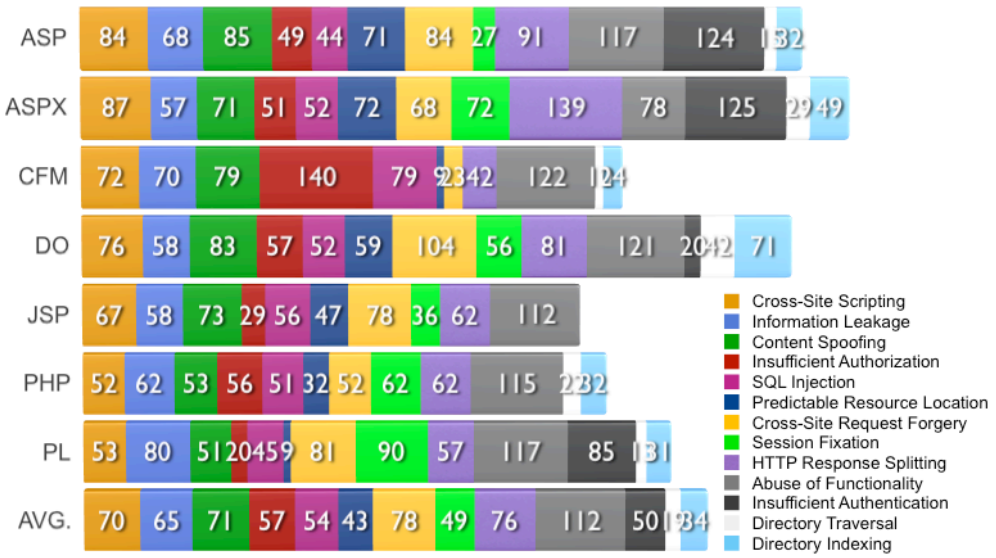

Figure 8. Average number of days for vulnerability resolution

In both circumstances, lengthy fix times may be compounded by the fact that one-third of .NET websites have ASP extensions, which are not naturally resilient to XSS. Secondly, PHP and Perl developers do not typically have the luxury of a native framework providing security against this attack transparently and by default. They become more familiar with how to defend their code directly. On the other hand, this may not entirely explain the lower marks of the others.

In another odd twist, SQL Injection, another common and devastating issue, was fixed the fastest on ASP websites at 44 days, just ahead of PL at 45 days. Again as described above, perhaps developers in these languages are more familiar with SQL Injection and how to remediate because default framework protection is unusual. CFM websites were the slowest to remediate SQL Injection at 79 days, while all the other languages placed in the 50-day range. Lastly, PL and JSP performed impressively on Insufficient Authorization fix times with 20 and 29 days respectively, which historically have taken over 50 days to fix. 140 days for CFM in this category is a negative standout.

Across essentially all classes of attack, remediation requires weeks to months. In today's threat landscape, where most incidents are Web-based, this average response time is not nearly fast enough. This signifies a need for organizations to improve their capability to monitor incoming attacks and respond to verified vulnerabilities. Unless a software security development lifecycle effort can guarantee perfection, which it can't, organizations must be prepared to react appropriately without being operationally disruptive. We have seen several cases where issues have been fixed within a 24 to 48 hour window, through prioritization and use of tools like Web Application Firewalls, demonstrating that it is possible.

Resolution Rates

Even if vulnerabilities are identified, it does not necessarily mean they are fixed quickly, or ever. As a result, it is important to analyze the types and severity of the vulnerabilities that do get fixed (or not) and in what volumes (Figure 9). Some organizations target the easier issues first to demonstrate their progress by vulnerability reduction. Others prioritize the high severity issues to reduce overall risk. Also, languages / framework may make some issues easier to fix issues than others.

Class of Attack	Severity	ASP	ASPX	CFM	DO	JSP	PHP	PL
SQL Injection	Urgent	70%	72%	66%	79%	58%	70%	71%
Insufficient Authorization	Urgent	21%	45%	46%	20%	25%	18%	10%
Directory Traversal	Urgent	43%	20%	67%	0%	33%	32%	16%
Cross Site Scripting	Urgent	100%	0%	100%	0%	0%	50%	0%
Cross-Site Scripting	Critical	51%	57%	50%	51%	52%	66%	54%
Cross-Site Request Forgery	Critical	18%	34%	17%	27%	39%	57%	27%
Session Fixation	Critical	19%	18%	0%	36%	50%	50%	100%
Abuse of Functionality	Critical	76%	23%	82%	38%	57%	59%	97%
Insufficient Authentication	Critical	55%	37%	0%	33%	71%	0%	100%
Information Leakage	High	32%	34%	57%	49%	45%	39%	29%
Content Spoofing	High	31%	30%	43%	37%	44%	46%	69%
Predictable Resource Loc.	High	29%	64%	85%	64%	53%	56%	29%
HTTP Response Splitting	High	28%	24%	33%	10%	36%	42%	35%
Directory Indexing	High	33%	56%	40%	25%	27%	33%	18%
TOTAL		65%	67%	75%	72%	63%	69%	74%

Figure 9. Percentage of vulnerabilities resolved (Compared by Extension)

Factors inhibiting organizations from remediating vulnerabilities:

No one at the organization understands or is responsible for maintaining the code.

No one at the organization knows about, understands, or respects the vulnerability.

Feature enhancements are prioritized ahead of security fixes.

Lack of budget to fix the issues.

Affected code is owned by an unresponsive third-party vendor.

Website will be decommissioned or replaced "soon." To note, we have experienced deprecated websites under the Sentinel Service still in active use for over two years.

Risk of exploitation is accepted.

Solution conflicts with business use case.

Compliance does not require it.

"Urgent" severity SQL Injection issues are widely exploited, so it is important to see very high remediation rates. Struts (DO) is noticeably ahead at 79%, with ASPX at 72% and followed by PL at 71%. The remainder of the field lay between 58 and 70 percent. While roughly 7 in 10 SQL Injection issues are fixed, ideally this should be much higher on average.

As demonstrated by the targeted attack against the Apache Foundation[7, 8], "Critical" severity Cross-Site Scripting vulnerabilities (non-persistent) are taking their place among the most exploited -- and not just the most pervasive. So it is concerning to see that nearly all languages / frameworks hover only in the 50 percent range for remediation, with PHP the only standout at 66%. Given the recent media exposure, it should be safe to expect these numbers to elevate throughout 2010.

The total remediation rates for CFM, PL, DO top the stack scoring in the mid-70 percent range with the remaining players not too far behind in the mid to upper 60 percent range. While we do not have an outside point of reference to determine if these numbers are good or bad, we are seeing them steadily improve over time. This is very encouraging. This indicates website security is maturing and organizations are bettering their capabilities, even if the remediation times remain slower than would be preferred.

Comparing Industry Verticals

While it is difficult to draw conclusions, we decided to include more granular data about how each language / framework performed by industry vertical (Figures 10 & 11). Perhaps how each industry deployed these technologies plays a role in the issues to which they are prone.

Roughly half of the Insurance and Pharmaceutical websites use .NET (ASX), while another one-third use ASP Classic. Totaling more than 80 percent representation collectively, clearly these two segments are heavily invested in Microsoft Technology. The same can be said for Financial Services, Retail, and Social Networking, in which more than half the websites using .NET or ASP Classic. The other half of these websites is largely dominated by Java, either with Struts (DO) or JSP.

Also notable is that we see ASP Classic, while a dated framework, existing across all industry verticals from 11 percent in education websites to as high as 35 percent in insurance websites.

Perl to a lesser extent also receives consistent usage between one and four percent. Social Network websites seem to make use of PHP the most, followed by IT, and Telecommunications. Cold Fusion is nearly non-existent in everything but Healthcare, Education, and Retail, and then only at marginal levels.

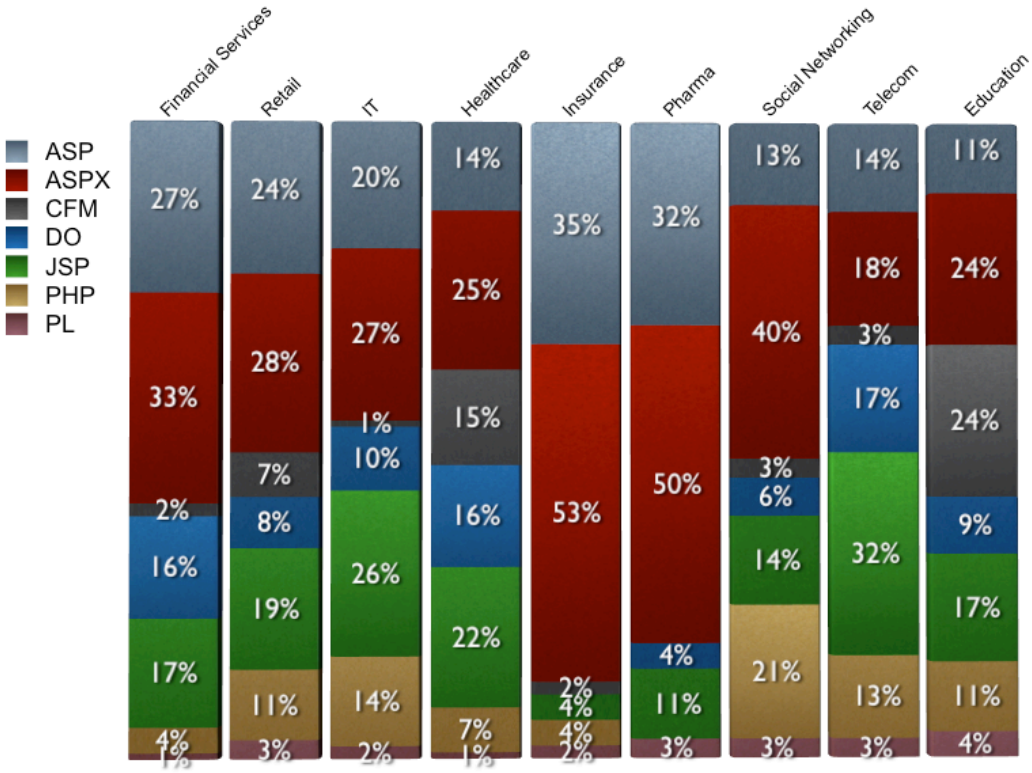

Figure 10. Technology in Use

Figure 11. Top Five Vulnerability Classes by Top Four Industry Verticals
(Must have at least 20 websites for inclusion)

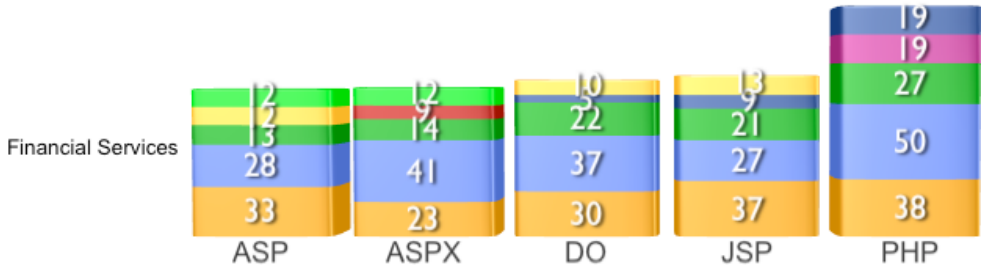

Figure 11a. Financial Services

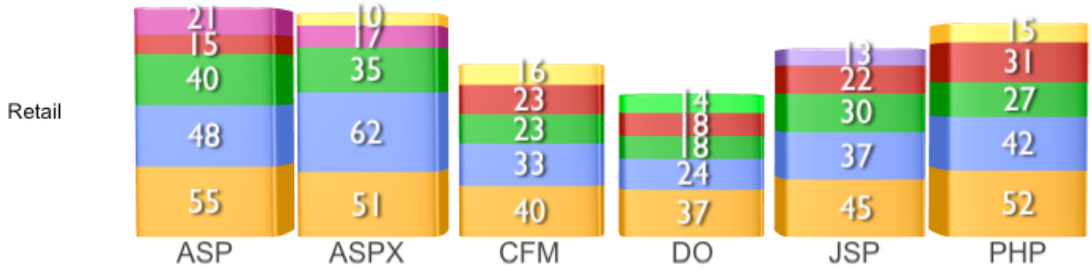

Figure 11b. Retail

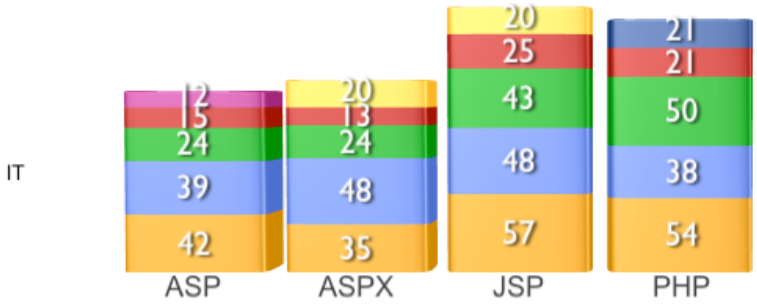

Figure 11c. Information Technology

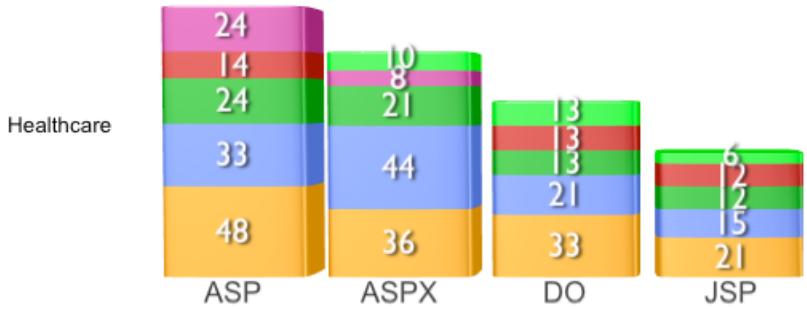

Figure 11d. Healthcare

Conclusions and Recommendations

Technically speaking, one Web application programming language / development framework can be made basically just as secure (or not) as any other. Empirically, languages / frameworks only show moderately varying security characteristics, such as the classes of attack to which they are most prone, how long they tend to be vulnerable, and so on. From a security posture perspective however, the languages / frameworks appear more the same than different. As such, we concluded that an organization's chosen website development technology alone does not guarantee application security success (or failure) – many other factors must be considered.

In our experience, an executive-level application security mandate is essential. Of course, people, process, and technology are necessary as well, but, if the business doesn't directly value application security, then such investments will be resisted or marginalized. Today, the desire for application security for most organizations originates from hacking incidents, compliance mandates, and customer demand. Whatever the circumstance, here are our recommendations:

- *You can't secure what you don't know you own – Inventory your Web applications to gain visibility into what data is at risk and where attackers can exploit the money or data transacted.*
- *Assign a champion – Designate someone who can own and drive data security and is strongly empowered to direct numerous teams for support. Without accountability, security and compliance will suffer.*
- *Don't wait for developers to take charge of security – Deploy shielding technologies to mitigate the risk of vulnerable Web applications.*
- *Shift budget from infrastructure to Web application security – With the proper resource allocation, corporate risk can be dramatically reduced.*

Glossary – Web Security Threat Classification v2.0 (Classes of Attack)[7]

Cross-Site Scripting[8] (XSS) is an attack technique that involves echoing attacker-supplied code into a user's browser instance. A browser instance can be a standard web browser client, or a browser object embedded in a software product such as the browser within WinAmp, an RSS reader, or an email client. The code itself is usually written in HTML/JavaScript, but may also extend to VBScript, ActiveX, Java, Flash, or any other browser-supported technology.

When an attacker gets a user's browser to execute his/her code, the code will run within the security context (or zone) of the hosting web site. With this level of privilege, the code has the ability to read, modify and transmit any sensitive data accessible by the browser. A Cross-site Scripted user could have his/her account hijacked (cookie theft), their browser redirected to another location, or possibly shown fraudulent content delivered by the web site they are visiting. Cross-site Scripting attacks essentially compromise the trust relationship between a user and the web site. Applications utilizing browser object instances which load content from the file system may execute code under the local machine zone allowing for system compromise.

Information Leakage[9] is an application weakness where an application reveals sensitive data, such as technical details of the web application, environment, or user-specific data. Sensitive data may be used by an attacker to exploit the target web application, its hosting network, or its users. Therefore, leakage of sensitive data should be limited or prevented whenever possible. Information Leakage, in its most common form, is the result of one or more of the following conditions: A failure to scrub out HTML/Script comments containing sensitive information, improper application or server configurations, or differences in page responses for valid versus invalid data.

Content Spoofing[10] is an attack technique that allows an attacker to inject a malicious payload that is later misrepresented as legitimate content of a web application.

Insufficient Authorization[11] results when an application does not perform adequate authorization checks to ensure that the user is performing a function or accessing data in a manner consistent with the security policy. Authorization procedures should enforce what a user, service or application is permitted to do. When a user is authenticated to a web site, it does not necessarily mean that the user should have full access to all content and functionality.

SQL Injection[12] is an attack technique used to exploit applications that construct SQL statements from user-supplied input. When successful, the attacker is able to change the logic of SQL statements executed against the database.

Structured Query Language (SQL) is a specialized programming language for sending queries to databases. The SQL programming language is both an ANSI and an ISO standard, though many database products supporting SQL do so with proprietary extensions to the standard language. Applications often use user-supplied data to create SQL statements. If an application fails to properly construct SQL statements it is possible for an attacker to alter the statement structure and execute unplanned and potentially hostile commands. When such commands are executed, they do so under the context of the user specified by the application executing the statement. This capability allows attackers to gain control of all database resources accessible by that user, up to and including the ability to execute commands on the hosting system.

Predictable Resource Location (PRL)[13] is an attack technique used to uncover hidden web site content and functionality. By making educated guesses via brute forcing an attacker can guess file and directory names not intended for public viewing. Brute forcing filenames is easy because files/paths often have common naming convention and reside in standard locations. These can include temporary files, backup files, logs, administrative site sections, configuration files, demo applications, and sample files. These files may disclose sensitive information about the website, Web application internals, database information, passwords, machine names, file paths to other sensitive areas, etc...

This will not only assist with identifying site surface which may lead to additional site vulnerabilities, but also may disclose valuable information to an attacker about the environment or its users. Predictable Resource Location is also known as Forced Browsing, Forceful Browsing, File Enumeration, and Directory Enumeration.

Cross-Site Request Forgery[14] is an attack that involves forcing a victim to send an HTTP request to a target destination without their knowledge or intent in order to perform an action as the victim. The underlying cause is application functionality using predictable URL/form actions in a repeatable way. The nature of the attack is that CSRF exploits the trust that a web site has for a user. By contrast, cross-site scripting (XSS) [9] exploits the trust that a user has for a web site. Like XSS, CSRF attacks are not necessarily cross-site, but they can be. Cross-site request forgery is also known as CSRF, XSRF, one-click attack, session riding, confused deputy, and sea surf.

Session Fixation[15] is an attack technique that forces a user's session ID to an explicit value. Depending on the functionality of the target web site, a number of techniques can be utilized to "fix" the session ID value. These techniques range from Cross-site Scripting exploits to peppering the web site with previously made HTTP requests. After a user's session ID has been fixed, the attacker will wait for that user to login. Once the user does so, the attacker uses the predefined session ID value to assume the same online identity.

HTTP Response Splitting[16] – The essence of HTTP Response Splitting is the attacker's ability to send a single HTTP request that forces the web server to form an output stream, which is then interpreted by the target as two HTTP responses instead of one response, in the normal case. The first response may be partially controlled by the attacker, but this is less important. What is material is that the attacker completely controls the form of the second response from the HTTP status line to the last byte of the HTTP response body. Once this is possible, the attacker realizes the attack by sending two requests through the target. The first one invokes two responses from the web server, and the second request would typically be to some "innocent" resource on the web server. However, the second request would be matched, by the target, to the second HTTP response, which is fully controlled by the attacker. The attacker, therefore, tricks the target into believing that a particular resource on the web server (designated by the second request) is the server's HTTP response (server content), while it is in fact some data, which is forged by the attacker through the web server - this is the second response.

Abuse of Functionality[17] is an attack technique that uses a web site's own features and functionality to attack itself or others. Abuse of Functionality can be described as the abuse of an application's intended functionality to perform an undesirable outcome. These attacks have varied results such as consuming resources, circumventing access controls, or leaking information. The potential and level of abuse will vary from web site to web site and application to application. Abuse of functionality attacks are often a combination of other attack types and/or utilize other attack vectors.

Insufficient Authentication[18] occurs when a web site permits an attacker to access sensitive content or functionality without having to properly authenticate. Web-based administration tools are a good example of web sites providing access to sensitive functionality. Depending on the specific online resource, these web applications should not be directly accessible without requiring the user to properly verify their identity.

Directory / Path Traversal[19] technique allows an attacker access to files, directories, and commands that potentially reside outside the web document root directory. An attacker may manipulate a URL in such a way that the web site will execute or reveal the contents of arbitrary files anywhere on the web server. Any device that exposes an HTTP-based interface is potentially vulnerable to Path Traversal.

Directory Indexing[20] – Automatic directory listing/indexing is a web server function that lists all of the files within a requested directory if the normal base file (index.html/home.html/default.htm/default.asp/default.aspx/index.php) is not present. When a user requests the main page of a web site, they normally type in a URL such as: http://www.example.com/directory1/ - using the domain name and excluding a specific file. The web server processes this request and searches the document root directory for the default file name and sends this page to the client. If this page is not present, the web server will dynamically issue a directory listing and send the output to the client. Essentially, this is equivalent to issuing an "ls" (Unix) or "dir" (Windows) command within this directory and showing the results in HTML form. From an attack and countermeasure perspective, it is important to realize that unintended directory listings may be possible due to software vulnerabilities (discussed in the example section below) combined with a specific web request.

References

1 *WhiteHat Sentinel Service – http://www.whitehatsec.com/home/services/services.html*

2 *PCI Data Security Standard – https://www.pcisecuritystandards.org/*

3 *WASC Threat Classification v2.0 – http://projects.webappsec.org/Threat-Classification*

4 *PCI Data Security Standard – https://www.pcisecuritystandards.org/*

5 *WhiteHat Sentinel Selection Guidelines – http://www.whitehatsec.com/home/services/selection.html*

6 *Achieving PCI Compliance with WhiteHat Sentinel – http://www.whitehatsec.com/home/services/pci.html*

7 *Apache Foundation Hit by Targeted XSS Attack – http://threatpost.com/en_us/blogs/apache-foundation-hit-targeted-xss-attack-041310*

8 *Apache.org incident report for 04/09/2010 – https://blogs.apache.org/infra/entry/apache_org_04_09_2010*

9 *WASC Threat Classification v2.0 – http://webappsec.pbworks.com/Threat-Classification10*

10 *http://webappsec.pbworks.com/Cross-Site+Scripting*

11 *http://webappsec.pbworks.com/Information-Leakage*

12 *http://webappsec.pbworks.com/Content-Spoofing*

13 *http://webappsec.pbworks.com/Insufficient-Authorization*

14 *http://webappsec.pbworks.com/SQL-Injection*

15 *http://webappsec.pbworks.com/Predictable-Resource-Location*

16 *http://webappsec.pbworks.com/Cross-Site-Request-Forgery*

17 *http://webappsec.pbworks.com/Session-Fixation*

18 *http://webappsec.pbworks.com/HTTP-Response-Splitting*

19 *http://webappsec.pbworks.com/Abuse-of-Functionality*

20 *http://webappsec.pbworks.com/Insufficient-Authentication*

21 *http://webappsec.pbworks.com/Path-Traversal*

22 *http://webappsec.pbworks.com/Directory-Indexing*

The WhiteHat Sentinel Service – Website Risk Management

WhiteHat Sentinel is the most accurate, complete and cost-effective website vulnerability management solution available. It delivers the flexibility, simplicity and manageability that organizations need to take control of website security and prevent Web attacks. WhiteHat Sentinel is built on a Software-as-a-Service (SaaS) platform designed from the ground up to scale massively, support the largest enterprises and offer the most compelling business efficiencies, lowering your overall cost of ownership.

Cost-effective Website Vulnerability Management – As organizations struggle to maintain a strong security posture with shrinking resources, WhiteHat Sentinel has become the solution of choice for total website security at any budget level. The entire Sentinel product family is subscription-based. So, no matter how often you run your application assessments, whether it's once a week or once a month, your costs remain the same.

Accurate – WhiteHat Sentinel delivers the most accurate and customized website vulnerability information available– rated by both threat and severity ratings – via its unique assessment methodology. Built on the most comprehensive knowledgebase in Web application security, WhiteHat Sentinel verifies all vulnerabilities, virtually eliminating false positives. So, even with limited resources, the remediation process will be sped up by seeing only real, actionable vulnerabilities, saving both time and money, dramatically limiting exposure to attacks.

Timely – WhiteHat Sentinel was specifically designed to excel in rapidly-changing threat environments and dramatically narrow the window of risk by providing assessments on your schedule. Whether it's a quarterly compliance audit, new product roll-out, or weekly business-as-usual site updates, WhiteHat Sentinel can begin assessing your websites at the touch of a button.

Complete – WhiteHat Sentinel was built to scale to assess hundreds, even thousands of the largest and most complex websites simultaneously. This scalability of both the methodology and the technology enables WhiteHat to streamline the process of website security. WhiteHat Sentinel was built specifically to run in both QA/development and production environments to ensure maximum coverage with no performance impact. And, WhiteHat Sentinel exceeds PCI 6.6 and 11.3.2 requirements for Web application scanning.

Simplified Management – WhiteHat Sentinel is turnkey – no hardware or scanning software to install requiring time-intensive configuration and management. WhiteHat Sentinel provides a comprehensive assessment, plus prioritization recommendations based on threat and severity levels, to better arm security professionals with the knowledge needed to secure an organization's data. WhiteHat Sentinel also provides a Web services API to directly integrate Sentinel vulnerability data with industry-standard bug tracking systems, or SIMs or other systems allowing you to work within your existing framework. With WhiteHat, you focus on the most important aspect of website security – fixing vulnerabilities and limiting risk.

About WhiteHat Security, Inc.

Headquartered in Santa Clara, California, WhiteHat Security is the leading provider of website risk management solutions that protect critical data, ensure compliance and narrow the window of risk. WhiteHat Sentinel, the company's flagship product family, is the most accurate, complete and cost-effective website vulnerability management solution available. It delivers the visibility, flexibility, and control that organizations need to prevent Web attacks. Furthermore, WhiteHat Sentinel enables automated mitigation of website vulnerabilities via integration with Web application firewalls. To learn more about WhiteHat Security, please visit our website at www.whitehatsec.com.

WhiteHat SECURITY
WhiteHat Security, Inc. | 3003 Bunker Hill Lane | Santa Clara, CA 95054
408.343.8300 | www.whitehatsec.com

050610