THE 2011 MID-YEAR TOP CYBER SECURITY RISKS REPORT

Table of contents

Contributors

Producing the Top Cyber Security Risks Report is a collaborative effort among HP DVLabs and other HP teams, such as Fortify and the Application Security Center. We would like to sincerely thank the Open Source Vulnerability Database (OSVDB) for allowing print rights to their data in this report. For information on how you can help OSVDB:

https://osvdb.org/account/signup

http://osvdb.org/support

Contributor	Title
Mike Dausin	Advanced Security Intelligence Team Lead
Adam Hils	Application Security Center Product Manager
Dan Holden	Director, HP DVLabs
Prajakta Jagdale	Web Security Research Group Lead
Jason Jones	Advanced Security Intelligence Engineer
Rohan Kotian	Digital Vaccine Team Lead
Jennifer Lake	Product Marketing, HP DVLabs
Mark Painter	Application Security Center Content Strategist
Taylor Anderson McKinley	Director, Fortify on Demand
Alen Puzic	Advanced Security Intelligence Engineer
Bob Schiermann	Senior Technical Publications Writer

Overview

Increasingly, organizations face security risks imposed upon them by attackers intent on achieving fame, glory, or profit. Attackers, familiar with common vulnerabilities inherent in many of today's websites, know how to exploit those vulnerabilities with attacks designed specifically to take advantage of them. Examples of such destructive activities have recently hit the news in stories about "hacktivist" groups such as LulzSec and Anonymous.

The HP 2011 mid-year edition of the biannual Top Cyber Security Risks Report features in-depth analysis and attack data from HP DVLabs, Application Security Center, and Fortify security units as well as vulnerability disclosure data garnered from the OSVDB. Given the media attention paid to these recent attacks, as well as data HP obtained from its partners and customers, the bulk of this report is focused on Web applications, including the vulnerabilities that exist and the attacks that are exploiting those weaknesses.

This report is intended for IT, network, and security administrators who are responsible for securing the public-facing communication with an organization's customers, partners, and employees. The primary objective of this edition of the Top Cyber Security Risks Report is to clearly articulate the risks and weaknesses inherent in Web applications. We'll highlight the overall vulnerability landscape, including vulnerabilities in commercially available and custom-built applications that can lead to attacks, as well as how often these are being reported. The report will also highlight the rising number of attacks that are leveraging the vulnerabilities discussed throughout the paper.

Key findings from this report include:

The number of Web application vulnerabilities that are reported differs significantly from the number that actually exist.

The Open Source Vulnerability Database (OSVDB) monitors vulnerability discovery and reporting through disclosure programs. Data from the first six months of 2011 shows a distinct and significant decrease in the disclosure of new vulnerabilities. While this might seem like good news, it is actually the opposite. Data collected from scans of actual customer Web application deployments indicates that the number of vulnerabilities is not decreasing; it is only the number of reported new vulnerabilities that is decreasing. Production websites for some of the world's leading organizations are still bursting with vulnerabilities that leave the websites open to devastating attacks.

Web application attacks are on the rise, despite the lack of new vulnerabilities being disclosed.

HP DVLabs compiled attack data from its network of HP TippingPoint intrusion prevention systems (IPS) to determine the danger these vulnerabilities pose to Internet security. Information pulled from these systems shows that the number of attacks on Web applications is ten times the number of vulnerabilities being reported. This fact leads us to believe that attackers either don't need any new vulnerabilities to achieve their goals, or that there are plenty of vulnerabilities in custom applications that are unknown or untracked, increasing the attack surface to attackers. The reality is likely a mixture of both.

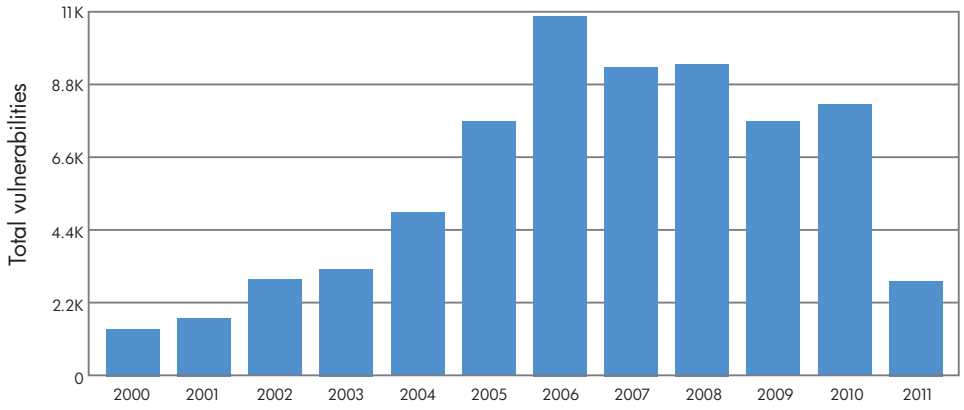
Web application vulnerabilities are easy to exploit with a variety of attack techniques and tools.

Two of the most common Web application attack types, Cross-Site Scripting (XSS) and SQL Injection (SQLi), are covered in-depth in this report. Based on data obtained from HP TippingPoint IPS devices, these are two of the most frequently used attack types—though many times for different reasons. XSS, which is often used for spam or phishing attempts, provides an easy way to distribute an attack on a wide scale. Conversely, SQLi can be used not only for overwriting a database and then redirecting visitors to a malicious site—similar in fashion to how XSS is leveraged—but also for massive database theft.

The information in this report comes from various sources, allowing HP DVLabs to obtain a broad set of data from which to correlate meaningful findings. These sources include:

- A worldwide network of HP TippingPoint Intrusion Prevention Systems
- Vulnerability information from OSVDB and the Zero Day Initiative (ZDI)
- Web application data from the ASC Web Security Research Group, the EB SW BTO Professional Services Organization, and Fortify on Demand

Figure 1



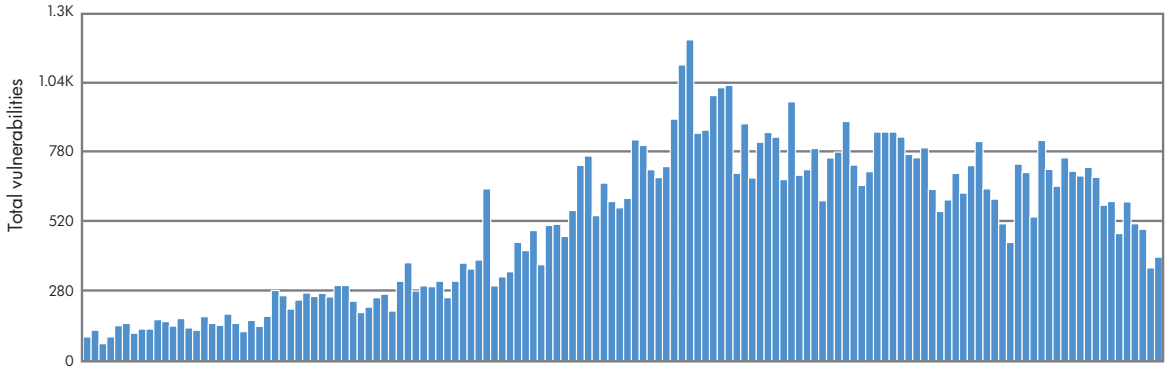
Disclosed vulnerabilities according to OSVDB, 2000–2010

Vulnerability trends

To better understand the threat landscape, it is important to start with the weaknesses present in computing infrastructures. These weaknesses typically manifest themselves as application vulnerabilities, which are the focus of this section of the report. The vulnerability landscape is discussed in the following three sections:

- **Discovery and disclosure of new vulnerabilities**: This section describes current trends in vulnerability reporting, highlighting vulnerabilities that have been disclosed in commercially available computing systems, including Web applications. Based on the trend information, one can discern the volume and category of newly discovered vulnerabilities, which provides insight into how such vulnerabilities attract attackers' attention.
- **Trends in vulnerability research**: This section highlights data from the HP DVLabs' Zero Day Initiative (ZDI) vulnerability research program. It provides a deeper look into the types of vulnerabilities that the ZDI researches and discovers in an effort to get a better sense of what drives a security attack.
- **Vulnerabilities discovered in production Web application environments**: This section highlights results from scans of live Web applications. Data in this section demonstrates the vulnerabilities that are present in real-world Web applications, including new vulnerabilities that are unreported as well as those that were previously disclosed, and as-yet unfixed vulnerabilities.

Figure 2



Vulnerability disclosure according to OSVDB, 2000–2011, broken down by month

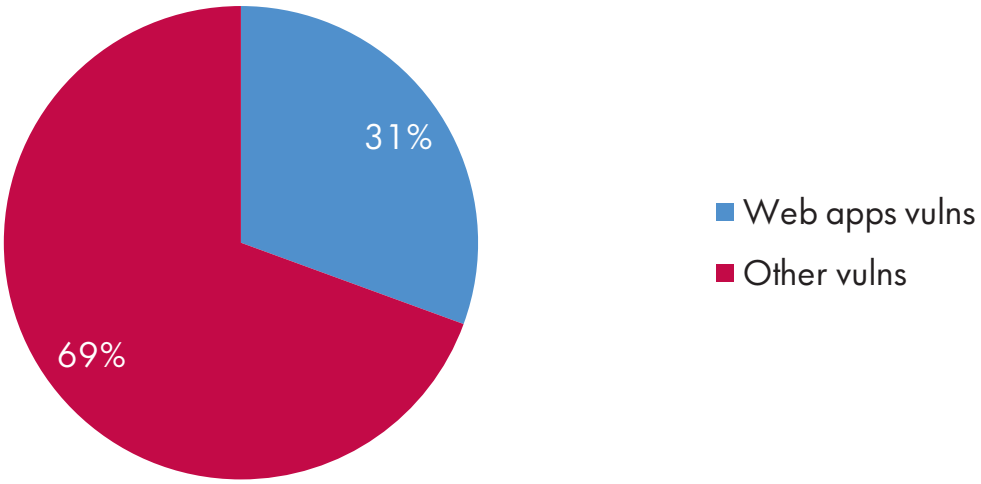
Discovery and disclosure of new vulnerabilities

Based on data pulled from OSVDB, the total number of new vulnerabilities reported for the first half of 2011 is about 25 percent lower than the number of new vulnerabilities reported at mid-year 2010 and previous years. As of June 30, 2011, OSVDB cataloged 3,087 **(Figure 1)** reported vulnerabilities in Internet-based systems, applications, and other computing tools, as compared to 4,091 cataloged in the corresponding period in 2010.

After peaking in 2006, vulnerability reporting—for commercially available products—has been in a slow decline **(Figure 2)**. The reasons for this decline are varied, but several likely reasons stand out. Software makers and system developers have increased their security awareness and have taken steps to reduce vulnerabilities prior to releasing their products. The second reason is a reduction in the disclosures of discovered vulnerabilities, motivated by a desire to instead sell the vulnerability for profit. Another is that some organizations would rather announce details regarding vulnerabilities only after they've been fixed.

Despite the overall decline in new vulnerabilities being discovered and reported, it is important to note that the ratio of vulnerabilities discovered in Web applications still makes up 31 percent **(Figure 3)** of all vulnerabilities disclosed. It is worth noting that roughly half of all vulnerability disclosures since 2006 have involved Web applications, so the downward trend for the first half of 2011 is yet another proof point for the overall drop in vulnerability disclosure thus far.

Figure 3



Comparison of Web application vulnerabilities versus non-Web application vulnerabilities, January–June 2011

The reason for the high number of Web application vulnerabilities is a matter of opportunity and profit. First, the number of Web applications in circulation grows steadily every day. A Web application in its simplest form ties together an operating system, a Web server, a database, and some top-level application that customers use to interact with the back-end systems. Many organizations have adopted this model for interacting with customers through retail sites, online banking or finance applications, or even appointment scheduling. In addition, many organizations have confidential or sensitive data stored in the database(s) connected to these applications, offering an almost limitless field of opportunity for attackers, who view it all as a very lucrative proposition.

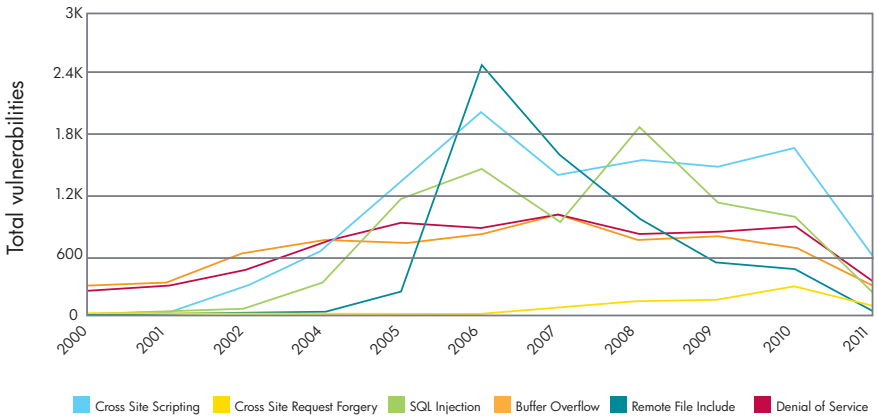
When vulnerabilities are broken down by category, some interesting trends begin to emerge. Data presented in **Figure 4** shows that certain types of vulnerabilities are more frequently discovered and disclosed. Cross-Site Scripting (XSS) still comprises the most significant amount of new Web application vulnerability disclosures. XSS is commonly used for spam, phishing, and Web browser exploits. Buffer Overflow and Denial of Service (DoS) vulnerabilities round out the top three.

Buffer Overflow
A buffer overflow attack occurs when attackers purposely overload a systems' temporary memory (called a buffer) to wreak havoc on a victim's machine. Oftentimes attackers also include instructional code in information they use to overflow the memory. That code can instruct the affected system to access or change confidential data or even send information back to the attacker.

Denial of Service (DoS)
A type of vulnerability that allows an attacker to exhaust computer resources on a vulnerable system to a point where legitimate usage of that system is impossible.

Distributed Denial of Service (DDos)
A type of DoS attack that employs a number of separate computers, which simultaneously launch a Denial of Service attack against a single application or system.

Figure 4



Disclosed vulnerabilities broken down by category, January 2000–June 2011

It is interesting to note that the breakdown of popular vulnerabilities by category remains fairly consistent over the past three years **(Figure 5)**, likely because XSS vulnerabilities are relatively easy to find and are very useful to attackers. Spammers and phishers are always looking for ways to make their trade more profitable, and XSS continues to be a useful vulnerability for these purposes.

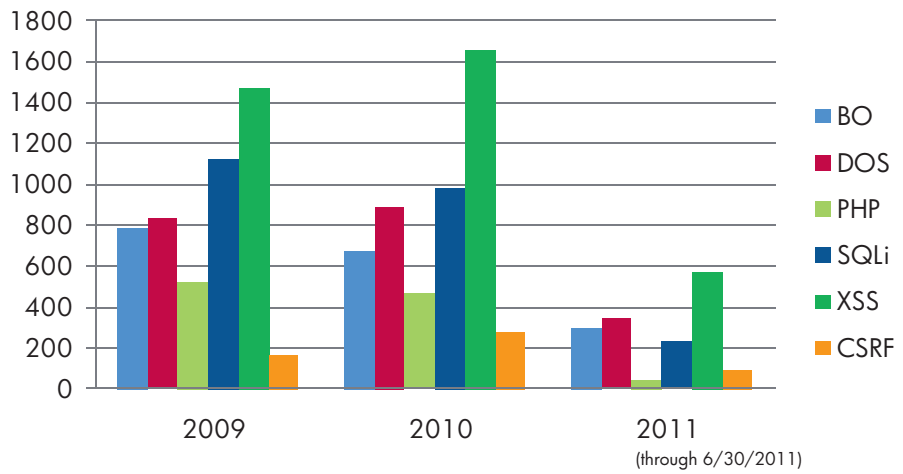
Further analysis: Zero Day Initiative

The Zero Day Initiative (ZDI), founded by HP TippingPoint in 2005, is a program for rewarding security researchers for responsibly disclosing vulnerabilities. The program is designed so that researchers provide HP TippingPoint with exclusive information about previously unpatched vulnerabilities they have discovered. HP DVLabs validates the vulnerability and then works with the affected vendor until the vulnerability is patched. At the same time, HP DVLabs develops a security solution that provides preemptive protection for HP's customers even before the application vendor distributes a fix for the vulnerability.

From 2005 through June 2011, ZDI and HP DVLabs researchers have discovered and responsibly disclosed more than 980 vulnerabilities in popular computing systems including Web browsers, media players, and document readers.

In 2010, ZDI announced changes to its disclosure policy that incent vendors to introduce more timely bug fixes into their products. Under the new policy, ZDI offers the affected vendors six months to issue patches, fixes, or workarounds for undisclosed vulnerabilities reported to them via the ZDI program. If, after six months, the vendor has not issued a fix or cleared an exception with ZDI, limited detail of the vulnerability will be disclosed so that the defensive community and consumers of these affected applications can find their own ways to mitigate the risk associated with these open bugs. Further information regarding the ZDI disclosure policy can be found here: http://www.zerodayinitiative.com/advisories/disclosure_policy/.

Figure 5

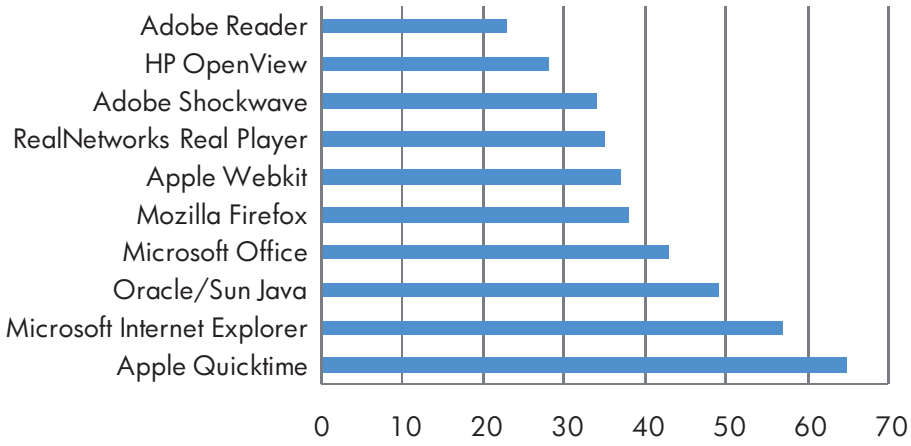


Three-year view: popular vulnerabilities by category

In the table **(Figure 6)** below you can see the top 10 applications with vulnerabilities disclosed through the ZDI since the program was started in 2005.

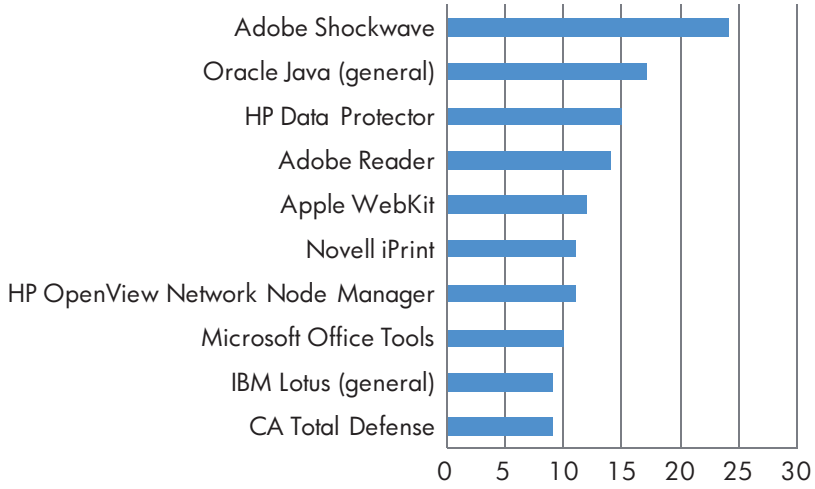
For the first half of 2011, DVLabs and the ZDI either discovered or acquired, and disclosed to affected vendors, 231 vulnerabilities in a wide range of products. On the next page **(Figure 7)** you can see the top 10 applications for which vulnerabilities were disclosed through the ZDI. While only four of the 10 applications are related to Web browsers, the total number of vulnerabilities from these applications is staggering.

Figure 6



Most frequently reported vulnerabilities disclosed through ZDI from 2005–2011

Figure 7



Most frequently reported vulnerabilities disclosed through ZDI in 2011

Seeing the big picture: where the vulnerabilites are

So far, this report has focused primarily on vulnerability disclosure, which may or may not reflect the complete picture of vulnerability trends unfolding on the Internet. In an effort to see a clearer picture of the real-world vulnerability landscape, the HP Application Security Center Web Security Research Group (WSRG) compiled results from over 2,750 security assessments performed against a variety of customer Web applications during the first six months of 2011. While it is good to see an overall reduction in the number of new vulnerabilities being reported, that has unfortunately had no impact on the dangers of exploitation. These results have been divided into three sections:

- The first correlates results from almost 250 Web applications analyzed statically (at the line-of-code level) for Web application vulnerabilities.
- The second group includes results from dynamic analysis (during the actual running of the application) conducted against over 2,500 unique Web applications.
- Finally, the third set of analyses includes a closer inspection of a much smaller group of assessments to explore the different ways in which Web application vulnerabilities can be exploited, how that impacts the overall risk standing of the application, and what mitigation measures developers are employing that are not working.

Each set of analyses will shed light on the nature—and seriousness—of Web application vulnerabilities and how prevalent they are. What security professionals have steadily witnessed in the last decade is that attacks have moved from defacement and general annoyance to one-time attacks designed to steal as much data as possible, and from there to pernicious ongoing attacks that attempt to distribute malware and steal as much data for as long as possible without being detected. It is imperative to realize that it often takes only one Web application vulnerability for an entire system to be compromised. The enormity of the danger cannot be overstated.

Static analysis

The first set of applications was statically analyzed by the WSRG in conjunction with the HP Fortify on Demand group and included 236 unique applications. The first statistic is truly staggering: a full 69% of the applications tested contained at least one SQLi flaw. Fundamentally, SQLi is an attack upon the Web application, not the Web server or the operating system itself. As the name implies, it is the act of adding unexpected SQL commands to a query, thereby manipulating the database in ways unintended by the database administrator or developer. When the attack is successful, data can be extracted, modified, inserted, or deleted from database servers that are used by vulnerable Web applications. If attackers can find one SQL Injection vulnerability in an application, there's a very good chance they can compromise it completely. A lot of high-profile attacks over the course of the first half of 2011 were the direct result of SQLi. Even Lady Gaga isn't immune to SQLi.

The second most prevalent vulnerability discovered in this series of assessments was Cross-Site Scripting (XSS), (specifically, the reflected variety). Put simply, reflected XSS attacks come from somewhere else, such as when user-supplied input from a Web client is immediately included via server-side scripts in a dynamically generated Web page. Using some social engineering, an attacker can trick a victim, perhaps through a malicious link or a "rigged" form, to submit information which will be altered to include attack code and then sent to the legitimate server. The injected code is then reflected back to the user's browser which executes it because it came from a trusted server. 64% of the assessed applications contained at least one reflected XSS flaw.

Its sister vulnerability, persistent XSS, was discovered in 42% of the applications tested in this group. Persistent attacks are just that: in some form they are stored on the target server, such as in a database, or via a submission to a bulletin board or visitor log. The victim will retrieve and execute the attack code in his browser when a request is made for the stored information. What's also interesting about this particular vulnerability is that even though it was found in 27% fewer of the applications than SQLi, there were actually more unique instances of persistent XSS discovered than any other vulnerability for which the WSRG tested. The impacts of each flavor of XSS are the same.

A more generic vulnerability, Header Manipulation, was found in 37% of the applications. Header Manipulation vulnerabilities occur when data enters a Web application through an untrusted source, most frequently a Web request. The data is included in an HTTP response header sent to a Web user without being validated. As with many Web application security vulnerabilities, Header Manipulation is a means to an end, not an end in itself. At its root, the vulnerability is straightforward: an attacker passes malicious data to a vulnerable application, and the application includes the data in an HTTP response header. Including unvalidated data in an HTTP response header can enable cache-poisoning, XSS, cross-user defacement, page hijacking, cookie manipulation, or open redirect vulnerabilities. And 18% of the applications also contained a specific cookie Header Manipulation vulnerability.

Cross-Site Scripting (XSS)
A type of Web application vulnerability that takes advantage of a lack of input validation to enable an attacker to inject malicious client-side code into a Web page which is viewed by a victim's Web browser. Various forms of XSS are currently being used to phish website users into revealing sensitive information such as usernames, passwords, and credit card details. XSS can generally be divided into stored, reflected, and DOM-based attacks. Stored XSS results in the payload being persisted on the target system in either the database or the file system. The victims will retrieve and execute the attack code in their browser when a request is made for the stored information. Execution of the reflected XSS attacks, on the other hand, occurs when user input from a Web client is immediately included via server-side scripts in a dynamically generated Web page. DOM-based XSS attacks rely on malicious modification of the DOM environment in a victim's browser. It differs from the stored and reflected XSS in the fact that the malicious data is never sent to the server. Via some social engineering, an attacker can trick a victim, such as through a malicious link or "rigged" form, to submit information that will be altered to include attack code and then sent to the legitimate server.

Command Execution
A type of vulnerability that takes advantage of a lack of input validation on a website in order to run operating system commands on the vulnerable application server. Typically, this vulnerability category allows attackers to exploit Web applications that pass user data as parameters to I/O operations by appending OS commands to user supplied input using special characters such as a pipe (|).

Another widespread vulnerability discovered during the WSRG analysis was Path Manipulation. This occurs when user-supplied input can control or otherwise influence file names or paths utilized in file system operations, which can then give an attacker the means to access or change protected system resources. 63% of the scans detected a Path Manipulation vulnerability.

While none of the vulnerabilities discussed so far can be considered innocuous, one extremely dangerous vulnerability was also detected in large numbers. Command Injection occurs when a remote user can supply a specially crafted value to execute arbitrary operating system commands on the target system. 35% of the applications contained at least one Command Injection vulnerability.

Another significant vulnerability occurs when developers leave passwords hardcoded in their code. Hardcoded passwords were discovered in 30% of the applications. An attacker who discovers a hardcoded password could obviously gain unintended access to the application. The damages would depend on the functionality of the application itself.

5% of the applications contained XPath Injection vulnerabilities. XPath Injection is very similar to SQLi. In that scenario, SQL commands are modified by an attacker to gain access to database contents and information. In XPath Injection, XPath statements are modified to gain access to the data contained within an XML document, which often serves as the "XML database." Importantly, XPath does not utilize access control restrictions as SQL does via privileges, so a successful XPath Injection attack will yield complete results in that all the data in the document will be revealed. The XPath language is also uniform, unlike SQL, so that any installed implementation is potentially vulnerable. In these aspects, XPath Injection is easier to execute than SQLi and has greater results returned on affected systems.

Another interesting set of data describes the number of vulnerabilities found per application, and per 1000 lines of code. During initial scans (before remediation efforts), 410 vulnerabilities were found on average for each of the 236 applications evaluated, equating to 4.6 vulnerabilities per 1000 lines of code. Of the three languages counted, PHP was the most vulnerable programming language, with 13.1 vulnerabilities per 1000 lines, followed by .Net at 7.7. Java was the most secure, at 4.1.

Dynamic analysis

The second set of data was collected by the WSRG in conjunction with the HP Enterprise Business Software BTO Professional Services organization and was split across three enterprise-level organizations: one from the energy sector, one from banking and finance, and one from product manufacturing and distribution to see how Web application vulnerabilities are presented in real-world applications and are encountered across all types of businesses. Each assessment was conducted using dynamic (real-time) analysis methods. The first two sets of data were analyzed against a small (less than 20) number of applications, while the last set consisted of more than 2,300 scans and was actually analyzed in far greater depth than the first two. However, all three sets of data yielded interesting results. Each was tested for a series of common, yet dangerous, Web application vulnerabilities.

Cross-Site Request Forgery (CSRF)
A type of Web application vulnerability that takes advantage of a lack of authorization on a vulnerable Web application to allow an attacker to execute application commands on behalf of another user of the application. The typical scenario of a Cross-Site Request Forgery attack involves an attacker tricking a victim into clicking on a specially crafted link that is designed to perform a malicious operation on behalf of the victim. For example, a victim may click on a malicious link that forces the victim to transfer money from the victim's bank account to an attacker's bank account.

Remote File Include
A type of Web application vulnerability that takes advantage of a lack of input validation on a website in order to execute unauthorized code (typically PHP or ASP) on a vulnerable server. Remote File Include attacks typically arise from a scripting language's inherent ability to include code from external URLs, or arbitrary local files. It is this ability that allows the attacker to include unauthorized code from an external source.

The energy sector applications contained a number of vulnerabilities that could be utilized to compromise the system. For example, 23% were vulnerable to SQLi. 15.3% were vulnerable to Remote File Include (RFI) vulnerabilities. 53.8% were vulnerable to Reflected XSS, while another 23% were vulnerable to Persistent XSS. Also, 38.4% were vulnerable to Cross-Site Request Forgery. Cross-Site Request Forgery relies on a browser to retrieve and execute an attack. It includes a link or script in a page that connects to a site that the user may have recently used. The script then conducts seemingly authorized, yet malicious, actions on the user's behalf. Other vulnerabilities could be exploited to block the access of legitimate users. 30.76% were susceptible to a Buffer Overflow, with another 15.3% vulnerable to a Denial-of-Service attack. It is important to note that an application that suffers from any one of these vulnerabilities would fail a PCI compliance audit.

While not as many specific vulnerabilities were detected, the banking and finance sector applications also contained a large number of disconcerting vulnerabilities. 58.3% of the applications were vulnerable to Reflected XSS, but only 8.3% contained a Persistent XSS vulnerability. 16.6% were vulnerable to Cross-Site Request Forgery. Exploitation of any of those vulnerabilities could result in an attacker gaining legitimate authentication credentials, in addition to other possibilities. In a bit of a good anomaly for this particular organization, only 8.3% of the applications were found to contain either a SQLi or RFI vulnerability. Yet, that positive security posture is somewhat lessened by the fact that 21.4% of the applications weren't using SSL cookies. And 50% suffered from Directory Path Disclosure vulnerabilities, which attackers can utilize to formulate more damaging attacks (think of this as a step in reconnaissance—if you know where something is, it's much easier to attack it).

Finally, the third set of applications (all 2,345 of them) is utilized by a very large product manufacturing and distribution organization. Although greater in number, these applications were actually assessed at a higher level of granularity than the preceding sets of data. Of these applications, 31% were vulnerable to XSS and 15% were vulnerable to a version of XSS that required user interaction, such as clicking a link or moving the mouse pointer over text. Another 6.5% were vulnerable to a specific form of XSS resulting from the way Apache Web servers incorrectly filtered input in the "Expect" header. Another 5.8% were vulnerable due to specific filtering vulnerabilities in Microsoft® ASP.NET.

While only 1.9% of the applications were confirmed to be vulnerable to SQLi, and 2.3% registered as vulnerable to SQLi albeit with no data able to be extracted, 18% were still vulnerable to Blind SQLi. Normal SQLi attacks depend in a large measure on an attacker reverse-engineering portions of the original SQL query using information gained from error messages. However, applications can still be susceptible to Blind SQLi even if no error message is displayed. The consequences are the same.

One interesting statistic is that only two of these applications registered as being vulnerable to Cross-Site Request Forgery. When coding applications, developers tend to make the same security mistakes in more than once place. In other words, if an application is vulnerable to SQLi, chances are it's vulnerable in many locations, not just one. However, the opposite can hold true, too. It is apparent that these developers utilized anti-CSRF tokens or other effective counter measures in their applications.

Another issue that the WSRG examined was that of information leakage. Information leakage consists of directory probing, error messages that reveal information unintended by the developer, common "guessed" directories, and other items that could reveal information beneficial to escalating attack methodology. Successful exploitation would give an attacker unauthorized access to sensitive information. The main problem with information leakage is that the information gained from these attacks can be used to conduct far more damaging attacks.

18.8% of the applications contained login information sent over unencrypted connection. Any area of a Web application that possibly contains sensitive information or access to privileged functionality such as remote site administration functionality should utilize SSL or another form of encryption to prevent login information from being sniffed or otherwise intercepted or stolen. 5.6% of the applications contained a known file or directory. One of the most important aspects of Web application security is to restrict

access to important files or directories to only those individuals who actually need to access them. 2.2% contained some form of code disclosure vulnerability. An attacker who gains access to the source code of an application obviously has an upper hand in determining the best method of attacking it.

Manual analysis

The WSRG also conducted extensive manual analysis of vulnerabilities discovered while conducting automated security tests for a different group of commercial applications. The analysis focused on discovering trends that help:

1. Determine the impact of various factors pertaining to the vulnerability source/context on its criticality and exploitability
2. Assess the mitigations put in place by developers to secure their Web applications against the most common vulnerability categories and understand their shortcomings

While securing Web applications against every possible threat is important, not all vulnerabilities are created equal, even if they belong to the same category. In the case of production systems, the discovery of critical vulnerabilities necessitates an immediate response. This, in turn, entails prioritizing the discovered issues based on the exploitability, severity, and impact on the security posture of the overall system. The manual analysis helped the WSRG identify the following numerous factors that impact a vulnerability's true risk rating.

1. Access control requirement for the resource

   Any resource requiring the user to authenticate adds an extra layer of complexity for the attacker in discovering the issue. However, once discovered, a successful exploitation can prove deadly, allowing the attacker to bypass the access control, escalate privileges, and gain control over protected and possibly sensitive sections of the system. 46% of the vulnerabilities were discovered in protected resources.

2. Function/Purpose of the resource

   Obviously, the sensitivity of the Web page content and its purpose greatly impacts the criticality of any vulnerability. An XSS vulnerability in the login page provides the attacker with more leverage to achieve a complete takeover of the system than one that exists on a search page. Ultimately, any given security issue can be turned into a deadly weapon against a Web application. However, the more effort required to exploit a vulnerability, the less attractive the application becomes to an attacker. In the sample set, 31% of vulnerabilities were detected in login scripts while 46% were detected in search pages.

3. In-house applications vs. third-party components

   During manual analysis, the WSRG discovered that vulnerabilities were detected in both custom code as well as in that of third-party applications. In 62% of the applications, the vulnerabilities were concentrated in the sections of applications that were developed in-house. In 31% of the applications, the distribution was exactly reversed. Security issues in third-party software are definitely more concerning since those allow the attackers to compromise multiple systems by possibly using the same exploit. Issues detected within custom code could take longer to fix since they will require understanding of all the vulnerable input usages and then applying a separate fix for each.

4. Complexity of the exploit that led to the discovery of the vulnerability

   Attackers always prefer targeting systems that are easier and faster to compromise. Thus, any application vulnerable to simple attack vectors will attract more attackers than one that has at least some mitigating security controls in place. 69% of vulnerabilities were discovered using "plain vanilla" attack vectors.

Despite the high-profile nature of recent Web application compromises, data breaches, and increased fines for noncompliance with governmental regulations, a large number of applications still remain vulnerable to the most rudimentary Web attacks. The WSRG has determined that a few recurring issues contribute to this problem:

1. Mitigations applied without accurately understanding the usage context of the user input

   23% of the Web applications in the sample set actually employed the HTML encoding technique to protect against XSS, 15% used a blacklisting approach, and 54% had no protection mechanisms implemented. The mitigations failed to provide any protection because in 54% of the applications, the reflections occurred within the client-side JavaScript code blocks, making the security controls ineffective.

2. Reliance on specific mitigation techniques instead of a holistic approach

   Manual inspection of the client-side application source code indicated that the few security controls employed by the developers were applied in response to individual vulnerabilities discovered more than likely during automated scans. There was no indication of development having adhered to any form of a Secure Development Lifecycle (SDLC) process to develop any of the tested applications. This was evident in the unbalanced distribution of vulnerabilities in different sections of the applications.

3. Lack of uniformity in implementation of security controls

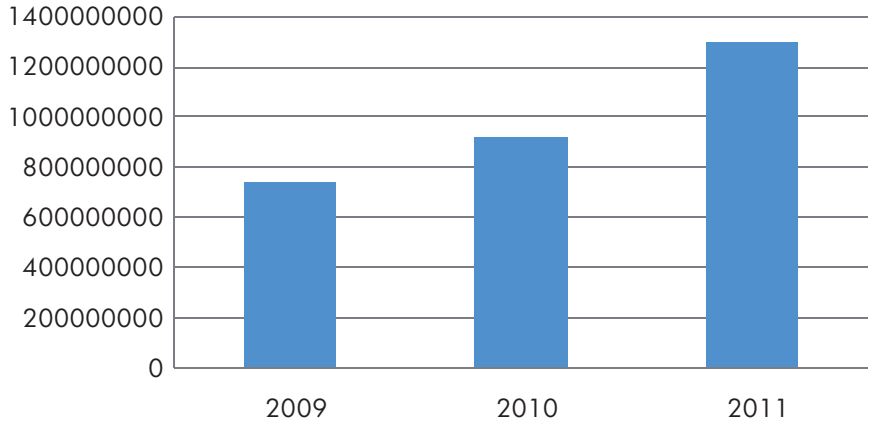
   Also, the manual analysis revealed that while certain sections of Web pages were protected against XSS attacks, others were left open to exploitation. This behavior could be attributed to various factors such as the mixture of in-house code vs. third-party application code, division of application development efforts with no uniform process put in place to govern best practices, a reactive approach to securing applications, and so on. This lack of uniformity obviously makes vulnerability patching extremely complex and challenging. The best approach is to establish a well-defined secure development process that is uniformly adhered to by all the parties involved in the creation of the application.

Attack trends

The previous section provided a view into the vulnerability landscape—specifically where and how applications can be compromised. While vulnerabilities provide a solid understanding of what exists, looking at what attacks are exploiting those vulnerabilities and how often will provide a deeper understanding of enterprise risk. Attack data from this section is broken out into three areas:

- **Frequency and number of attacks.** Data in this section is obtained from a network of HP TippingPoint Intrusion Prevention System (IPS) devices. This data is important for understanding the risk severity of particular vulnerabilities.
- **A deeper look at Cross-Site Scripting (XSS) attacks.** XSS is one of the most frequent attacks measured on HP TippingPoint IPS devices. This section delves into the different types of XSS attacks and the specific danger inherent in these variants.
- **A timeline and breakdown of SQL Injection (SQLi) attacks.** SQLi is the most frequent Web application attack that we track. This section looks at how SQLi has evolved and why it poses such a huge risk for today's enterprises.

Figure 8



Total number of attacks at mid-year, 2009–2011

New vulnerabilities are unnecessary; attacks continue to rise regardless

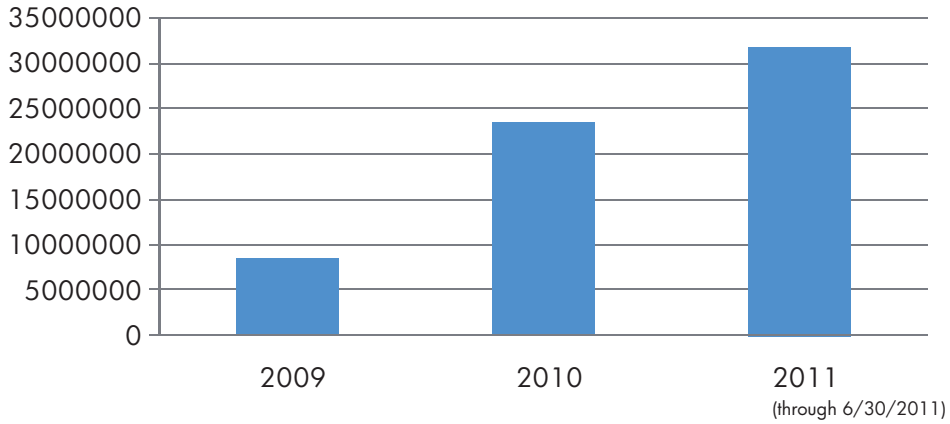
Despite the fact that the level of new vulnerability discoveries is dropping, there is no shortage of attacks. While this is true across the board—every system, every category—it is especially significant when discussing Web applications. Given the levels of vulnerabilities that are present in so many Web applications, it's a fair assessment that this rise in attacks is due to attackers leveraging existing vulnerabilities.

First, the trend of attacks for the first half of the year for the past three years **(Figure 8)** depicts a distinct upward spike.

Next, comparing Web application attacks at the mid-year for the past three years **(Figure 9)**, there is a distinct increase in attacks on Web applications—nearly a double year-on-year growth for attacks aimed at these applications.

Looking at the data another way—comparing numbers of Web application attacks to all attacks in the graph **(Figure 10)** on the next page—it is interesting to note that the ration of Web application attacks is actually a bit higher.

Figure 9

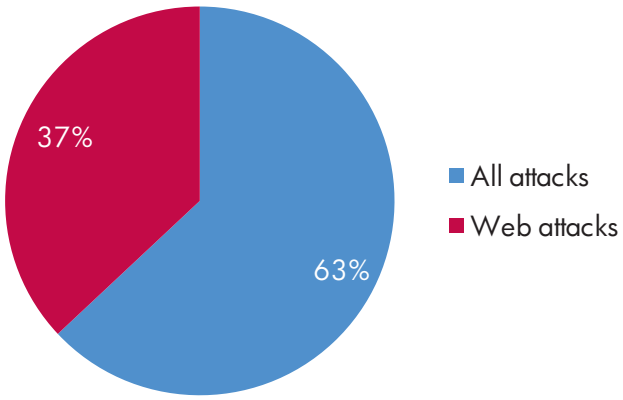


Total number of Web application attacks at mid-year, 2009–2011

When Web application attacks are broken down by category, we can see some definite trends taking shape. Let's refer back to the three types of Web application vulnerabilities discussed in an earlier section: PHP/Remote File Include (PHP/RFI), SQL Injection (SQLi), and Cross-Site Scripting (XSS). While XSS vulnerabilities are disclosed more often, it is SQLi vulnerabilities that are being attacked the most **(Figure 11)**—by a significant margin.

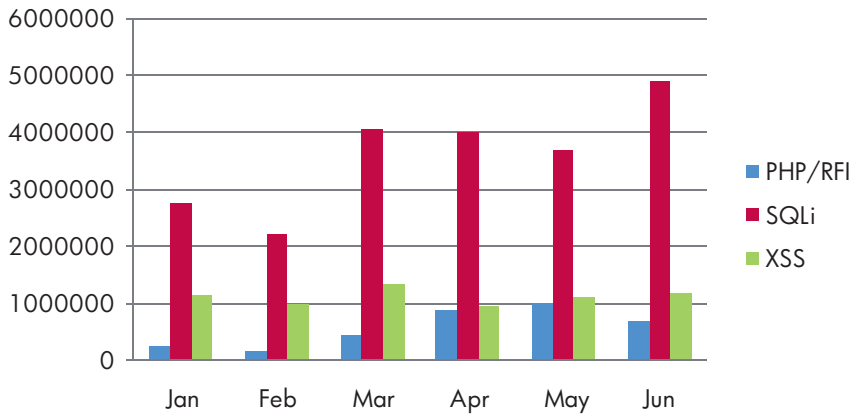
Data in **Figure 12** (on page 18) shows that Web application attacks are increasing so rapidly that the number of attacks for the first half of 2011 are nearly at the same levels as for the full years in 2009 and 2010. And in one case—SQLi—the numbers are higher than in the previous year.

Figure 10



Web application attacks versus non-Web application attacks, January–June 2011

Figure 11



Web application attacks in the first half 2011, broken down by category

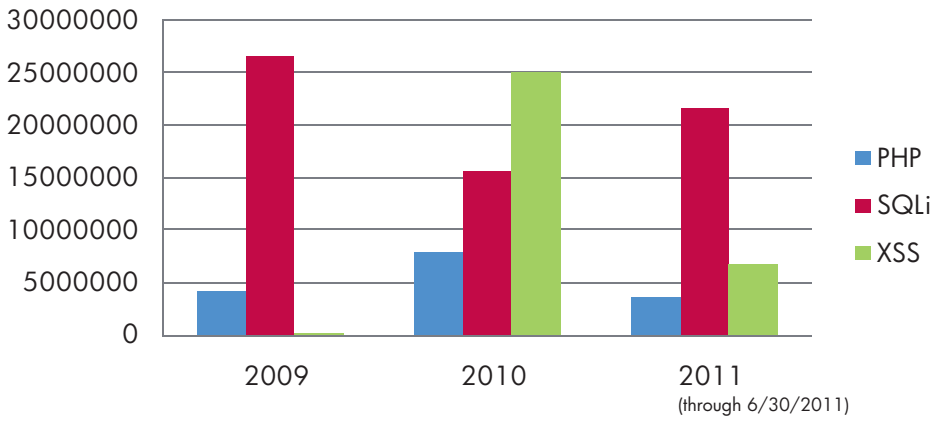
Cross-Site Scripting

Cross-Site Scripting (XSS) vulnerability has been around for a while and has been well-documented over the years. To refresh, XSS is a hacking technique that allows attackers to exploit vulnerabilities in Web applications and inject client-side script into the vulnerable Web pages that are viewed by unsuspecting users. A successful attack will allow an attacker to hijack user sessions, steal sensitive information, or deface websites. There are two primary types of XSS vulnerability: non-persistent (or reflected), and persistent (or stored).

The reflected (non-persistent) XSS is by far the most common type of XSS attack. The root cause is the improper handling (lack of sanitization) of HTTP request data by the server code, allowing malicious sites to "reflect" malicious code and attack the user. The main attack vector is usually an email message containing a malicious URL. When the user clicks on the URL, they are taken to the vulnerable site where the malicious code is executed and reflected back on the user in order to execute the attack. It is the XSS vulnerability of the website that allows this type of attack to happen. The Web browser executes the code because it believes the code originates, and is unaltered, from a trusted website.

A persistent, or stored, XSS attack is a far more devastating XSS variant. This attack does not require users to click URLs in order to pass malicious code back to the vulnerable website and attack the user. In this case, the malicious code is able to live on the vulnerable server and is served up alongside regular HTML content. Again, this type of attack is a direct result of poor input validation on the server side, which allows for non-sanitized input to end up being displayed on the site. This type of attack is particularly risky not only because it does not require direct user interaction but also because it has a much wider scope. With non-persistent attacks, the only users who get attacked are the ones who reflect the malicious code to the site by clicking the URL. With persistent XSS attacks, every visitor to the site may get compromised as the malicious code lives on the server itself. Also, this malicious code can be self-propagating, creating a type of client-side worm.

Figure 12



Web application attacks, 2009–2011

Over the last decade, XSS has been a popular part of the security threat landscape. According to vulnerabilities documented by Symantec in 2007, XSS accounted for roughly 80% of all the security vulnerabilities. That percentage has leveled off over the recent years, but XSS is still the second most popular type of Web application vulnerability. According to the Open Web Application Security Project (OWASP) 2010 Top Ten, XSS was second only to SQLi. What makes XSS even more dangerous is that it could be leveraged by an attacker to exploit other Web application vulnerabilities such as Information Disclosures, Content Spoofing, and more.

While organizations have a better understanding of the risks posed by XSS attacks, these types of vulnerabilities still make up a high percentage of bugs being disclosed every year. So while many XSS vulnerabilities have a low common vulnerability scoring system (CVSS) score, their prevalence increases the overall attack surface of a Web application, which put enterprises at a high risk for exposure and can be costly to fix.

SQL Injection plays a starring role

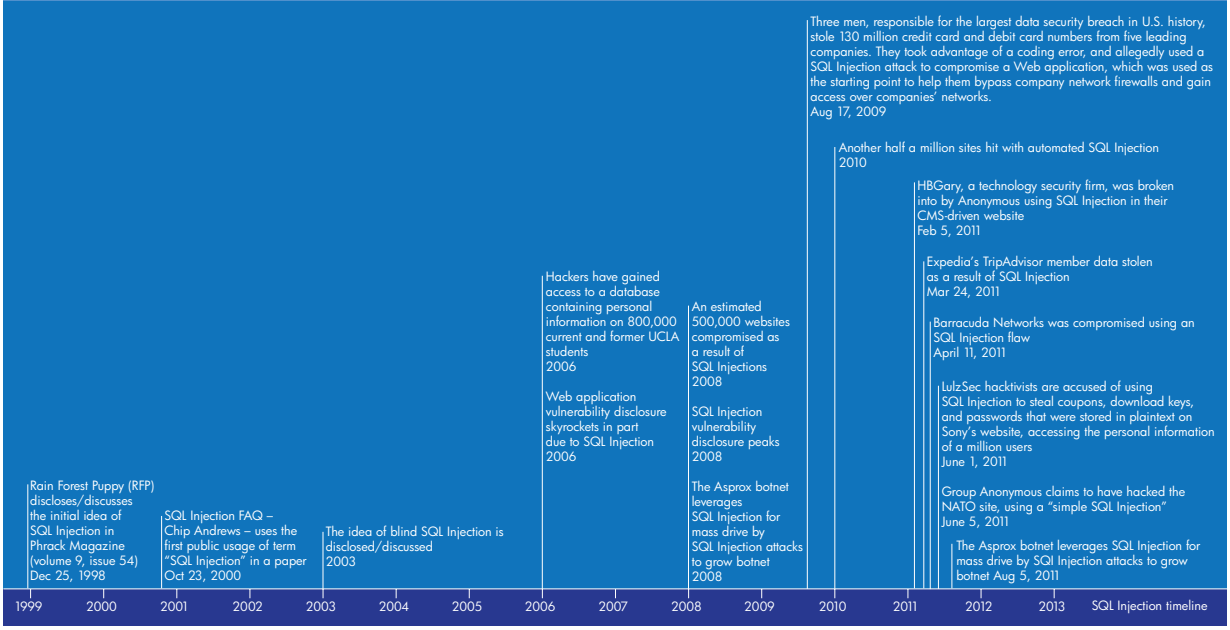
SQLi attacks gained media attention this year from the hacktivist groups LulzSec and Anonymous, who used this type of attack to compromise systems of several high-profile organizations. Data from the earlier graph **(Figure 12)** shows that this type of attack is on the rise and has been extensive for some time.

The chart and timeline on the next page **(Figure 13)** demonstrate how SQLi attacks have evolved over the years.

- 1998—Rain Forest Puppy (RFP) discloses/discusses the initial idea of SQL Injection in *Phrack Magazine* (Volume 9, Issue 54)
- 2000—SQL Injection FAQ—Chip Andrews—uses the first public usage of term "SQL Injection" in a paper
- 2003—The idea of blind SQL Injection is disclosed/discussed
- 2006—Web application vulnerability disclosure skyrockets in part due to SQL Injection
- 2008—SQL Injection vulnerability disclosure peaks

SQL Injection (SQLi)
A type of Web application vulnerability that takes advantage of a lack of input validation on a website in order to execute unauthorized database commands on a Web applications database server. When successfully exploited, data can be extracted, modified, inserted, or deleted from database servers that are used by the vulnerable Web application. In certain circumstances, SQL Injection can be utilized to take complete control of a system.

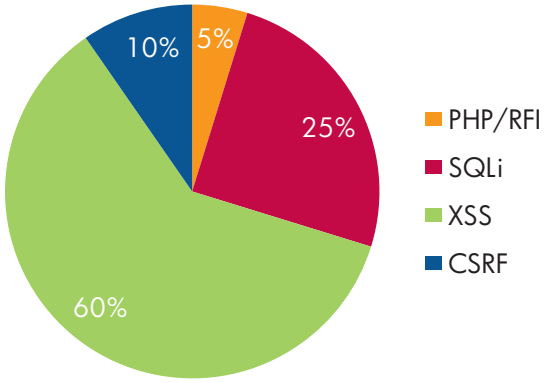
Figure 13



Timeline and evolution of SQL Injection attacks

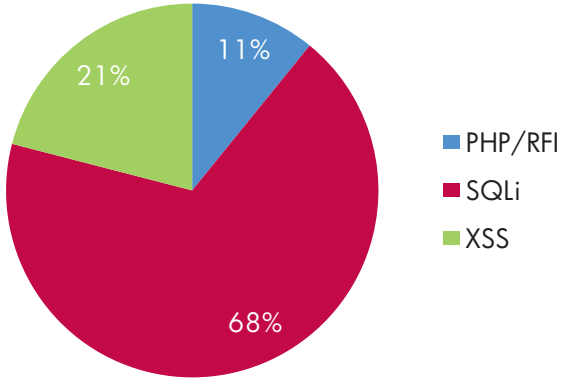
- 2008—The Asprox botnet leverages SQL Injection for mass drive by SQLi attacks to grow botnet (**http://en.wikipedia.org/wiki/Asprox**). From at least April through August, a sweep of attacks began exploiting the SQL Injection vulnerabilities of Microsoft's IIS Web server and SQL Server database server. The attack does not require guessing the name of a table or column, and it corrupts all text columns in all tables in a single request. An HTML string that references a malware JavaScript file is appended to each value. When that database value is later displayed to a website visitor, the script attempts several approaches at gaining control over a visitor's system. The number of exploited Web pages is estimated at 500,000.
- On August 17, 2009, the U.S. Justice Department charged an American citizen Albert Gonzalez and two unnamed Russians with the theft of 130 million credit card numbers using a SQL Injection attack. In reportedly "the biggest case of identity theft in American history," the man stole cards from a number of corporate victims after researching their payment processing systems. Among the companies hit were credit card processor Heartland Payment Systems, convenience store chain 7-Eleven, and supermarket chain Hannaford Brothers.
- On February 5, 2011, HBGary, a technology security firm, was broken into by Anonymous using a SQL Injection in their CMS-driven website.
- On April 11, 2011, Barracuda Networks was compromised using a SQL Injection flaw. Email addresses and usernames of employees were among the information obtained.
- On June 1, 2011, "hacktivists" of the group LulzSec were accused of using SQLi to steal coupons and to download keys and passwords that were stored in plaintext on Sony's website, accessing the personal information of a million users.
- In June 2011, Group Anonymous claims to have hacked the NATO site, using a "simple SQL Injection."

Figure 14



Web application vulnerabilities disclosed, January–June 2011

Figure 15



Total Web application attacks, January–June 2011

With SQLi attacks, it is readily apparent that attackers are content leveraging existing vulnerabilities for their exploits. The graphs above **(Figures 14 and 15)** show a side-by-side comparison of the most reported types of Web application vulnerabilities verses the most attacked Web application vulnerabilities. SQLi vulnerabilities make up a quarter of the new vulnerabilities *reported* for the first half of 2011. Yet SQLi attacks make up more than 60 percent of the Web application attacks *seen* in the HP TippingPoint IPS.

Web applications are affected by multiple types of attacks, and SQLi and XSS are just two that have received a significant amount of media attention over the last few months. The next section of this paper presents general mitigation strategies for protecting Web applications and decreasing the risk of outages, data loss, or network compromise that can result.

Mitigation

Visibility is increasingly becoming one of the most important aspects of information security, along with reducing the overall attack surface made available to attackers. To mitigate risk responsibly, organizations should test code in development, scan for vulnerabilities in QA before staging, and test applications in production on an ongoing basis. The following information is intended to help developers correct certain specific categories of critical Web application vulnerabilities.

Cross-Site Request Forgery

Resolving Cross-Site Request Forgery is not a simple task, and it actually may require recoding every form and feature of a Web application. While no method of preventing Cross-Site Request Forgery is perfect, using Cross-Site Request Forgery nonce tokens eliminates most of the risk. Although an attacker may guess a valid token, nonce tokens are nevertheless the most effective solution for preventing Cross-Site Request Forgery attacks. A user can be verified as legitimate by generating a "secret," such as a secret hash or token, after the user logs in. "The secret" should be stored in a server-side session and then included in every link and sensitive form. Each subsequent HTTP request should include this token; otherwise, the request is denied and the session invalidated. The token should not be the same as the session ID in case a Cross-Site Scripting vulnerability exists. Initialize the token as other session variables. It can be validated with a simple conditional statement, and it can be limited to a small timeframe to enhance its effectiveness. Attackers need to include a valid token with a Cross-Site Request Forgery attack in order to match the form submission. Because the user's token is stored in the session, any attacker would need to use the same token as the victim.

CAPTCHA can also prevent Cross-Site Request Forgery attacks. With CAPTCHA, a user needs to enter a word shown in distorted text, contained inside an image, before continuing. The assumption is that a computer cannot determine the word inside the graphic, although a human can. CAPTCHA requires that a user authorize specific actions before the Web application initiates them. It is difficult to create a script that automatically enters text to continue, but research is underway on how to break CAPTCHAs, so strong CAPTCHAs are a necessity. Building a secure CAPTCHA takes more effort. In addition to making sure that computers cannot read the images, you need to make sure that the CAPTCHA cannot be bypassed at the script level. Consider whether you use the same CAPTCHA multiple times, making an application vulnerable to a replay attack. Also make sure the answer to the CAPTCHA is not passed in plaintext as part of a Web form.

SQL Injection

SQL Injection arises from an attacker's manipulation of query data to modify query logic. The best method of preventing SQL Injection attacks is, therefore, to separate the logic of a query from its data; this will prevent commands inserted from user input from being executed. The downside of this approach is that it can have an impact on performance, albeit slight, and that each query on the site must be structured in this method for it to be completely effective. If one query is inadvertently bypassed, that could be enough to leave the application vulnerable to SQL Injection. The following code shows a sample SQL statement that is SQL injectable.

```
sSql = "SELECT LocationName FROM Locations";
sSql = sSql + "WHERE LocationID =" + Request["LocationID"];
oCmd.CommandText = sSql;
```

The following example utilizes parameterized queries and is safe from SQL Injection attacks.

```
sSql = "SELECT * FROM Locations";
sSql = sSql + "WHERE LocationID = @LocationID";
oCmd.CommandText = sSql;
oCmd.Parameters.Add("@LocationID", Request["LocationID"]);
```

The application will send the SQL statement to the server without including the user's input. Instead, a parameter-@LocationID- is used as a placeholder for that input. In this way, user input never becomes part of the command that SQL executes. Any input that an attacker inserts will be effectively negated. An error would still be generated, but it would be a simple data-type conversion error, and not something that a hacker could exploit.

The following code samples show a product ID being obtained from an HTTP query string and then used in a SQL query. Note how the string containing the "SELECT" statement passed to SqlCommand is simply a static string and is not concatenated from input. Also note how the input parameter is passed using a SqlParameter object, whose name ("@pid") matches the name used within the SQL query.

C# sample:

```
string connString = WebConfigurationManager.ConnectionStrings["myConn"].ConnectionString;
using (SqlConnection conn = new SqlConnection(connString))
{
  conn.Open();
  SqlCommand cmd = new SqlCommand("SELECT Count(*) FROM Products WHERE ProdID=@pid",
  conn);
  SqlParameter prm = new SqlParameter("@pid", SqlDbType.VarChar, 50);
  prm.Value = Request.QueryString["pid"];
  cmd.Parameters.Add(prm);
  int recCount = (int)cmd.ExecuteScalar();
}
```

VB.NET sample:

```
Dim connString As String = WebConfigurationManager.ConnectionStrings("myConn").
ConnectionString
Using conn As New SqlConnection(connString)
  conn.Open()
  Dim cmd As SqlCommand = New SqlCommand("SELECT Count(*) FROM Products WHERE
  ProdID=@pid", conn)
  Dim prm As SqlParameter = New SqlParameter("@pid", SqlDbType.VarChar, 50)
  prm.Value = Request.QueryString("pid")
  cmd.Parameters.Add(prm)
  Dim recCount As Integer = cmd.ExecuteScalar()
End Using
```

Cross-Site Scripting

Cross-Site Scripting attacks can be avoided by carefully validating all input and properly encoding all output. When validating user input, verify that it matches the strictest definition possible of valid input. For example, if a certain parameter is supposed to be a number, attempt to convert it to a numeric data type in your programming language.

PHP: intval("0".$_GET['q']);

ASP.NET: int.TryParse(Request.QueryString["q"], out val);

The same applies to date and time values, or anything that can be converted to a stricter type before being used. When accepting other types of text input, make sure the value matches either a list of acceptable values (white-listing), or a strict regular expression. White-listing involves creating a list of acceptable characters, as opposed to black-listing, which is a list of unacceptable characters. If at any point the value appears invalid, do not accept it. Also, do not attempt to return the value to the user in an error message.

Most server-side scripting languages provide built-in methods to convert the value of the input variable into correct, non-interpretable HTML. These should be used to sanitize all input before it is displayed to the client.

PHP: string htmlspecialchars (string string [, int quote_style])

ASP.NET: Server.HTMLEncode (strHTML String)

When reflecting values into JavaScript or another format, make sure to use a type of encoding that is appropriate. Encoding data for HTML is not sufficient when it is reflected inside of a script or style sheet. For example, when reflecting data in a JavaScript string, make sure to encode all non-alphanumeric characters using hex (\xHH) encoding.

If you have JavaScript on your page that accesses unsafe information (like location.href) and writes it to the page (either with document.write, or by modifying a DOM element), make sure the data is encoded for HTML before writing it to the page. JavaScript does not have a built-in function to do this, but many frameworks do. If you are lacking an available function, something like the following will handle most cases:

s = s.replace(/&/g,'&').replace(/"/i,'"').replace(/</i,'<').replace(/>/i,'>').replace

(/'/i,''')

Ensure that you are always using the right approach at the right time. Validating user input should be done as soon as it is received. Encoding data for display should be done immediately before displaying it.

Remote File Includes

As the saying goes, security is baked in, not brushed on. Any application under development should be designed with security in mind from the onset. The following recommendations will help you build Web applications that are not susceptible to parameter include vulnerabilities.

- Define what is allowed. Ensure that the Web application validates all input parameters (cookies, headers, query strings, forms, hidden fields, etc.) against a stringent definition of expected results. The best method of doing this is via "white-listing"; this is defined as only accepting specific account numbers or specific account types for those relevant fields, or only accepting integers or letters of the English alphabet for others. Many developers will try to validate input by "black-listing" characters, or "escaping" them. Basically, this entails rejecting known bad data by placing an "escape" character in front of it so that the item that follows will be treated as a literal value. This approach is not as effective as white-listing because it is impossible to know all forms of bad data ahead of time.
- Check the responses from POST and GET requests to ensure what is being returned is what is expected, and is valid.
- Verify the origin of scripts before you modify or utilize them.
- Do not implicitly trust any script given to you by others (whether downloaded from the Web or given to you by an acquaintance) for use in your own code.

References

http://techtimely.wordpress.com/2011/04/22/web-hacking-threats/

https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

http://en.wikipedia.org/wiki/Cross-site_scripting

http://www.phrack.org/issues.html?id=8&issue=54

http://sqlsecurity.com/FAQs/SQLInjectionFAQ/tabid/56/Default.aspx

http://www.isti.tu-berlin.de/fileadmin/fg214/Papers/ravi-asprox.pdf

http://arstechnica.com/tech-policy/news/2011/02/anonymous-speaks-the-inside-story-of-the-hbgary-hack.ars/3

http://nakedsecurity.sophos.com/2011/06/02/sony-pictures-attacked-again-4-5-million-records-exposed/

4AA3-7045ENW, Created September 2011

This is an HP Indigo digital print.