

The Usual Suspects: Textbook Web application vulnerabilities

By Jason Reed

The same old vulnerabilities continue to plague Web applications, due to three main drivers: the pervasiveness of highly customized Web applications that interact with users, the growing complexity of the applications, and Web application developers continuing to make the same mistakes over and over again.

Web applications have come a long way in recent years in their ability to disseminate user-specific information effectively. Changes in technology have prompted the creation of frameworks to simplify application development and deployment. Application developers feel less of the burden of accepting user input, implementing security, displaying results, and so forth.

However, while the trends continue toward built-in, automated security mechanisms, the same vulnerabilities continue to plague Web applications. This is due to three main drivers: the pervasiveness of highly customized Web applications that interact with users, the growing complexity of the applications, and Web application developers continuing to make the same mistakes over and over again.

The Open Web Application Security Project (www.owasp.org) created the Top Ten Project, which maintains the list of the OWASP Top Ten – an agreement upon the most critical Web application vulnerabilities. Now translated into several languages, the list has been accepted and is used by both private corporations around the world and agencies within the US Federal Government. The list presents categories that allow testers and application developers to label problems and coordinate similar solutions across platforms and technologies.

With the introduction and subsequent acceptance of the Top Ten, application developers and their respective security departments could begin to speak about vulnerabilities with a common vocabulary. The Top Ten represents a

newfound confluence that helped push organizations to change the way their business units developed, tested and deployed Web applications.

Additionally, developers have been learning from past vulnerabilities, and frameworks are always being enhanced in order to minimize reinvention of the wheel. However, it is the highly customized and dynamic nature of today's applications that makes them difficult to secure. This is exacerbated by the trend toward overdependence on standard functions called from customized applications.

The Most Common Mistakes Developers Continue Making: With OWASP Top Ten category

1. **A2 – Broken Access Control:** Failure to consistently apply access control measures
 - a. URL/POST data parameter manipulation
 - b. Forced browsing
2. **A1 – Unvalidated Input:** Not consistently checking user input
 - a. SQL injection and cross-site scripting (XSS)
 - b. Field type validation
3. **A10 – Insecure Configuration Management:** Improper Web platform settings
4. **A3 – Broken Authentication and Session Management:** Not performing acceptable session management

Two of the most frequent questions posed when performing Web application vulnerability assessments are, “What are their developers doing better?” and “What do they still need to improve?” The problem is that the answers, even given all the trends of more inclusive security, largely haven’t changed over time.

Broken Access Control: Failure to consistently apply access control measures

Companies have moved toward dynamic applications that provide specific information for each user. For example, online banking applications provide information about a customer’s account balance and also support integrated loan applications, rather than just listing an account balance on a static page. Because of this, the same pages are used to deliver different content to different users.

Given the recent high-profile cases of identity theft and information exposure, and of subsequent fraud, the computing public has little patience and even less compassion for companies that inadvertently leak personal information.

Past: Historically, the mechanism for differentiating users was either account information passed in the URL or POST data or a session-level identifier that contained a user’s specific information (account number). Both of these methods are unacceptably insecure because the application must trust the user to pass in information for only his account.

Now: User data is set in the session state (generally) and passed to the application code by the application server instead of a user’s request. While this is better because it limits the ability of a malicious user to cross over to other accounts, the controls are not being applied consistently across the entire application. A spider of the Web application (using valid credentials) will quickly reveal fields where the application is accepting user input and overriding the session data meant to limit user manipulation of inputs. The page called will return personally identifiable information. This is a high-risk vulnerability. Given the recent high-profile cases of identity theft and information exposure, and of subsequent fraud, the computing public has little patience and even less compassion for companies that inadvertently leak personal information.

Another problem with access controls is when developers think only about the data being returned by a page and not about whether the page should be allowed to be called by the user. Applications typically have at least two user roles (some have dozens) that include classes such as anonymous users, specific users, coordinators, and administrators.

Past: Developers followed a security-by-obscurity rule that said if the users could not see the link in their navigation pane, they would not call the page. Because of this, “forced browsing” became a great tool with which attackers could call and use administrative pages if they could discover their URLs (often administrator pages were in the /admin/ folder).

Now: Developers are using designators in the page code that first check the level of the user before allowing the rest of the page to

It is puzzling that given the availability of Web platform scanning tools such as Nikto and Nessus, deployment managers do not test the configuration of their servers to verify that the most common holes have been plugged.

be executed and returned. However, like URL and POST data parameters, the controls are still not applied consistently across the entire application. It is not uncommon to see fail-open and fail-closed code for the same role in the same application. Also, some pages just do not contain the checks at all. By manipulating the role setting in the session state data, attackers can quickly find pages that fail open based on a bogus role identifier. If developers cannot secure the pages, the application should be split to a site for specific users and another for roles with higher privileges.

Unvalidated Input: Not consistently checking user input

SQL injection and cross-site scripting (XSS) continue to plague all levels of applications. While some cross-site scripting problems have been mitigated (e.g., by sanitizing input before storing it in a long-term database), reflective XSS opportunities still abound in most Web applications. Developers continue to struggle with users’ input and are applying controls inconsistently. Parts of the application, in particular users’ profile data, have seen great improvements, but the remaining fields that accept input are often either unchecked or the checks do not verify type. Storing numbers for a person’s telephone number and ZIP code may not immediately seem problematic, but the unchecked and usually unbounded fields can be made to store XSS or SQL injection characters.

Developers need to sanitize and escape all user input, and verify the type against templates – e.g., all numbers, all letters, contains a “@” sign, etc.

Insecure Configuration Management: Improper Web platform settings

It is puzzling that given the availability of Web platform scanning tools such as Nikto and Nessus, deployment managers do not test the configuration of their servers to verify that the most common holes have been plugged. In performing security assessments, we typically find at least an information leak on the server, and often more critical problems (such as the manager or admin application running on Tomcat application servers). Verifying a proper installation not only helps to secure the server, but also allows the application to run in an environment more in tune with its purpose (e.g., performance gains from settings like connection pools).

Poor server hygiene is not solely a failure of deployment managers; most application developers do not provide a punch list of options and modules that are needed on the Web and application servers. Simply creating such a list allows system administrators to disable the remaining components and allow only the services necessary to support the application.

Broken Authentication and Session Management: Not performing acceptable session management

One of the greatest improvements in session management is that developers are no longer writing their own algorithms to generate session identifiers.

Past: Developers first used user-identifiable information as the session ID, which was a terrible practice, especially when the identifiers were set as persistent cookies. Next, they obfuscated the information, but the data was still in there, which was almost as bad. Then they got a little smarter and started associating the user with the identifier in session state, but this solution was flawed because the developers concentrated on unique identifiers, not unique and hard to guess. This led to time-based identifiers, which, while unique, could easily be predicted.

Now: Developers are now using the session data to hold most of the information about a user and his session. The reference to the session is now more standard and harder to guess, derived from algorithms that come with the platforms.

However, the problem is that these (now better constructed) session identifiers are still being sent over unencrypted channels, and in a number of cases stored as persistent cookies on the users' systems. Also, the timeout periods for these sessions are still far too long for what is required for a user to transact business with the site. An on-line banking user who just needs to view an account balance and transfer money from account A to account B does not need an eight-hour session window. Application developers need to work with

their users to determine the minimum and maximum window that a user might require, and set the timeouts accordingly.

Conclusion

Application developers are learning from some of their past mistakes, but there is still more work to be done. The application frameworks are serving to bridge the gap to enable developers to concentrate on functionality and not reinvent security with each application. However, overdependence on framework functions in this era of highly customized applications can lead to continued problems with Web application security. Organizations must work to develop secure applications that provide enhanced business functionality, not functional business applications with security as an afterthought.

About the Author

Jason Reed, a Principal Consultant with SystemExperts Corporation (www.systemexperts.com), is a security professional with exceptional experience in Web application penetration testing, intrusion detection, and incident response in mission-critical production environments. Jason has worked on and led projects within multiple vertical markets including financial services, manufacturing, Web services, online retail, brick and mortar retail, and the insurance industry. Jason has successfully completed projects with governmental organizations as well, including bankruptcy trustees throughout the US.

References

The OWASP Top Ten Project:
http://www.owasp.org/index.php/OWASP_Top_Ten_Project