

Securing Web 2.0: are your web applications vulnerable?

White paper

- Introduction 2
- Web 2.0 components 2
- Vulnerabilities in Web 2.0 3
 - Cross-site scripting 3
 - Web application worms 3
 - Feed injection 4
- Securing Web 2.0 applications 4
- Conclusion 4



Introduction

The definition of Web 2.0 is still being debated despite extensive discussion. Its staunchest advocates proclaim it a complete philosophical and technological reworking of how the web functions. Others declare that it is meaningless. However, most agree on common characteristics of a Web 2.0 application, such as increased interactivity, the acceptance of user input for building community and a reliance on client-side functionality. Additionally, Web 2.0 applications can be more vulnerable to exploitation by hackers than their predecessors. Hackers spend most of their time gathering information. When Web 2.0 applications push functionality and code to users, they provide hackers with information that can be used for formulating attacks. Often, old attacks such as cross-site scripting become more dangerous when used against Web 2.0 applications. This white paper defines some of the common technological components of Web 2.0 applications and discusses ways of securing them against exploitation.

Web 2.0 components

Web 2.0 uses the web for delivering information that is often created through community contribution. Wikis and blogs are good examples of these types of applications. The main attribute of a Web 2.0 application is interactivity. More functionality is on the client, and less is on the server. As a result, requests are updated in the browser without refreshing the entire page. For example, consider Google Maps. Instead of a static page, you can drill down or zoom in and out of a map without making requests for a new page.

You can use several key technologies—or more appropriately, groupings of different technologies—to create Web 2.0 applications. The following are some of the most frequently used:

Ajax, Asynchronous JavaScript™ combined with XML, increases a web application's interactivity, responsiveness and usability by exchanging small bits of data with the server so that the entire page does not need to be refreshed every time a user makes a new request.

RSS, Really Simple Syndication or Rich Site Summary, is a collection of “feed” formats for publishing frequently updated content, such as news or blogs.

JSON, JavaScript Object Notation, is used with JavaScript in the same way that XML is used with Ajax.

Flash is a popular way to add video and interactivity to websites. Most browsers support Flash and contain a client application to run Flash files.

SOAP, Simple Object Access Protocol, is used by most web services to send XML data between a web service and a client web application making a request.

REST, Representational State Transfer, increases a web application's response time and server-loading through caching. Most blog sites are based on REST rather than RPC (Remote Procedure Call). They download an XML RSS feed file that contains links to other resources.

Used together or separately, these technologies have increased the flexibility of web applications. However, when implemented without security considerations, application inputs can be vulnerable, and old attacks can gain new traction.

Vulnerabilities in Web 2.0

The same vulnerabilities that affect standard web applications can affect Web 2.0 applications. However, other situations can create security issues specifically for Web 2.0. A Web 2.0 application can aggregate resources, such as blogs or RSS feeds, from a number of locations and then build a large repository of information for presentation on its own site. Vulnerabilities in your application can be exploited by unverified third-party content.

In Web 2.0 applications, data can be exchanged between applications in a variety of methods, including XML, JSON and proprietary structures. Often, this data is transmitted in clear text, making it easy for attackers to collect.

Although Ajax can improve the usability of a web application, it can also create attack opportunities if the application has not been designed with security in mind. Ajax creates a larger attack surface and contains more inputs to secure. Without security considerations, Ajax can expose the internal functions of the web application server and can let client-side scripts access third-party resources, increasing the opportunity for damaging attacks.

Because some Web 2.0 applications let users upload content, they can be extremely susceptible to hackers who can upload malicious content. Subsequent visitors can be infected by visiting a page.

This section discusses some common vulnerabilities for Web 2.0 applications. In many cases, these vulnerabilities are not limited to Web 2.0 applications. However, their potential for risk increases because of the increased level of interactivity of these applications.

Cross-site scripting (XSS)

Cross-site scripting has been a vulnerability since the early days of the web. Ajax and JSON increase the opportunity for cross-site scripting vulnerabilities. Such implementations require a trust relationship between the client and server—a relationship that attackers can exploit. With Ajax, cross-site scripting can make malicious requests with a user's credentials without refreshing the web page. This increases the potential for damage and theft of information. With traditional web applications, most information theft occurs with passive screen scraping. The malicious script can secretly read the contents of the HTML on a page that a user sees and send it back to the attacker. Using Ajax, a cross-site scripting attack sends requests for pages other than a page that a user sees. The attacker can actively search for specific content, potentially accessing resources and data not available from passive screen scraping.

You can find more information about cross-site scripting in the HP white paper called *Cross-site scripting: are your web applications vulnerable?*

Web application worms

With Ajax applications, cross-site scripting can propagate itself like a virus. Cross-site scripting can use Ajax requests to autonomously inject itself into pages. It can easily reinject the same host with more cross-site scripting without refreshing the page display. Thus, cross-site scripting can send multiple requests using complex HTTP methods to propagate itself invisibly to a user. Several examples have already surfaced, and one example is the Samy worm on MySpace.

Feed injection

Web feeds, such as RSS and Atom, provide users with the content and content summaries of a website without requiring users to visit the site. Unfortunately, many web applications that receive web feed data do not consider the security implications of using content from third-party locations. Vulnerabilities associated with feeds include cross-site scripting, keystroke logging and cross-site request forgery. During presentation, readers treat feed data as a literal and execute scripts within the feeds. Attackers can install malicious software on client systems, steal cookies or conduct other malicious activities. Many readers convert feed content and save it to a file on the hard disk before loading it into the viewer, which opens the local zone for attack. In the local zone, there is no limit to what a hacker can request. This lets the attacker include code in a feed to scan the ports of a back-end network, identifying open ports and potentially launching attacks automatically while behind the firewall without the user's knowledge. The potential impact of a feed-based attack increases significantly when the feed is syndicated to other websites.

More information can be found in the HP white paper called *Feed injection in Web 2.0*.

Securing Web 2.0 applications

To secure Web 2.0 applications, you should use the same measures for securing standard web applications. However, you need additional measures. Web 2.0 developers must protect users and not trust them at the same time. Because Web 2.0 application logic and functionality are on the client, you can prevent exploitation by adding data validation to the JavaScript. This does not replace server validation but complements it.

When developing web and Web 2.0 applications, use the following recommendations as security best practices:

- Make sure that the application validates all input before processing it.

- Use white listing rather than black listing for validation. White listing refers to the practice of accepting input that is good, as opposed to blocking input that is bad. For example, a ZIP code must always have five numbers. White listing ZIP code input means only accepting five numbers.
- Encode all user-supplied data to prevent sending inserted HTML to users in a format that their browsers can interpret.
- Minimize the exposed program logic.
- Document all inputs that your application allows.

End users can help protect themselves by disabling script, applet and plug-in execution, although they may limit functionality as a result.

When testing Web 2.0 applications, consider both servers and clients. Pay special attention to client-side attack vectors. Identify the libraries that are loaded in a browser—these libraries often contain known vulnerabilities. Web 2.0 applications often trust third-party information without properly verifying it. Identify sources so that you can test all inputs to the application. Because a browser runs everything in context of its Document Object Model (DOM), identify the DOM access points.

Conclusion

Web 2.0 applications have moved the Internet forward and help fulfill the promise of more interactive functionality and community building. However, security is not usually considered. The increase in functionality and interactivity has increased the ways in which an application can be attacked successfully. Even old attack methods have gained new strength when attacking Web 2.0 applications. Web 2.0 applications pose new challenges to developers, application testers and end users. Until security is part of the complete software development lifecycle, Web 2.0 applications will remain insecure and can increase the potential for harm.

To learn more, visit www.hp.com/go/software

© Copyright 2007 Hewlett-Packard Development Company, L.P. The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein. JavaScript is a U.S. trademark of Sun Microsystems, Inc.

4AA1-5390ENW, October 2007

