Protecting a Web Application Against Attacks Through HTML Shared Files

Francisco Corella

Revised November 10, 2008

Patent Pending

Abstract

Many Web applications have a file-sharing feature that allows Web users to share files by uploading them to, and downloading them from, a Web-accessible file repository. Shared files may include HTML files and other files containing scripts that are executed by the browser in the security context of the application user who downloads the file. This opens the door to a range of cross-user attacks, including attacks by former users and even attacks by a user of a virtual application instance against a different virtual instance of the same application. Such attacks are in essence XSS attacks, but the usual defenses against XSS are typically not available, because shared files cannot be sanitized.

This paper proposes a countermeasure that Web applications can use against attacks through HTML shared files, without sanitizing those files. The countermeasure leverages the same-origin policy by the use of carefully tailored hostnames for serving user files and application pages, including two different hostnames for downloading a shared file, linked by a redirection step. Authentication is achieved by the use of different cookies for shared files and application pages, and ephemeral file-retrieval sessions.

Acknowledgement

I would like to thank Bil Corry, Amit Klein, Adrian Pastor and Peter Watkins for comments on the original version of this paper, which can be found in an archived email thread [1]. I have made substantial changes based on those comments. There are several new references, discussion of those references, consideration of content sniffing, and mention of relevant new features in IE8. There is also an improvement, described in Section 5.1, to address the possibility that a browser may use multiple Range requests to accomplish a single file download. The issue was raised by Peter Watkins [2], and the improvement is based on a comment by Bil Corry [3].

1. Introduction

Many Web applications have a file-sharing feature that allows Web users to share files by uploading them to, and downloading them from, a Web-accessible file repository. (I will refer to files stored in the repository as *shared files* or *user files*.) Some of these applications provide file-sharing as their only purpose, while others provide file-sharing as one of several collaboration tools or social-networking features.

These applications often have multiple *virtual application instances*, each having its own file repository and its own group of instance users who are allowed to access the file repository. (Applications with multiple virtual instances are sometimes referred to as *multitenant applications* [4], where the word *tenant* refers to the group of users of a virtual instance. In this paper I will simply say *application instance* to refer to a virtual application instance.)

Sharing files has the obvious risk that a virus or other malware may be present in a file being obtained

through file sharing. This risk however is not specific to the fact of sharing; it exists whenever a file is obtained by any means. This risk is mitigated to a certain extent by warnings issued by browsers when downloading executable files, and may be further mitigated through the use of virus scanners. This paper is concerned, however, with a somewhat different and more insidious risk, one that is not mitigated by browser warnings or scanners.

Some of the files in an instance repository may be HTML files. When a user downloads an HTML file from the repository, the file may be displayed by the browser. This is a useful feature, as it allows one or more HTML shared files to serve as a private Web site accessible only to instance users, perhaps containing links to non-HTML files stored in the repository. But an HTML file displayed by the browser may contain client-side code, such as scripts written in JavaScript or ActionScript, Java applets, or ActiveX controls. A script planted by a malicious user in a shared file, when downloaded by a victim application user, runs in the security context of the victim application user, and can take advantage of any authentication cookies set by the application in the victim's browser to impersonate the victim and take advantage of any access privileges granted to the victim by the application.

Similar considerations apply to other types of files that may contain scripts, such as SWF files or PDF files. Also, a file with HTML content may be disguised as a file of a script-free type, such as GIF. The file may have a *.gif* extension and may be downloaded with an HTTP *Content-Type* header with value *image/gif*, and yet a browser that does *content-sniffing*, such as Internet Explorer (IE), may detect the HTML content, treat the file as an HTML file, and execute an attack script planted in the file. References to HTML files in the rest of the paper should be understood to apply more broadly to any type of file that contains client-side code, if the browser executes such code without warning and the effects of the code are subject to the same-origin policy [5] as discussed below.

This problem is a variation on the cross-site scripting (XSS) [6] problem faced by sites, such as social networking sites, that allow users to upload HTML content that is displayed to other users. Those sites typically defend against such attacks by sanitizing uploaded HTML content to remove or disable any scripts. Most Web applications with a file sharing feature, on the other hand, do not take precautions against attacks through shared files. Why is this?

One can think of two reasons. First, it is not a good idea to sanitize uploaded files. Users do not expect their files to be tampered with, and they may have legitimate reasons to share files that contain scripts. Second, an argument can be made that the Web application is entitled to assume that users of an application instance trust each other, and thus that it is not the responsibility of the application to prevent attacks of a user against another user (*cross-user attacks*).

The second reason, however, is not a valid one. The application has a responsibility to defend users against attacks through HTML shared files because those attacks can be used to defeat security mechanisms implemented by the application and negate security assumptions that users take for granted. For example, they can defeat user authentication and allow a user to impersonate another user, or a former user of an application instance to retain access to the instance without authorization. They can defeat the system of file permissions, if one is in place, giving a user access to files he or she is not authorized to download. They can even allow a user of one instance to access a different instance for which he or she does not have credentials (*cross-instance attack*).

This paper proposes a countermeasure that Web applications can use to protect users against attacks through HTML shared files, without sanitizing those files. The countermeasure leverages the same-origin policy by the use of carefully tailored hostnames for serving user files and pages constructed by the application (*application pages*); two different hostnames linked by a redirection step are used to download a user file. Authentication is achieved by the use of different cookies for shared files and application pages, and ephemeral file-retrieval sessions.

The rest of the paper is organized as follows. Section 2 surveys previous work. Section 3 discusses attacks through HTML shared files in more detail. Section 4 describes the proposed countermeasure. Section 5 discusses possible variations on the countermeasure and secondary benefits. References are listed in Section 6.

2. Prior Work

Little work has been done concerning the risk of cross-user attacks through HTML shared files. US-CERT discusses the risks of file sharing in a Cyber Security Tip [7] but only considers peer-to-peer file sharing. Carnegie Mellon's MySecureCyberspace Web site discusses File Sharing security [8], but only considers peer-to-peer file sharing and centralized file sharing via FTP. A recent thread in the Web Security mailing list [9] touched on Web-based file sharing security, but was mostly concerned with the general problem of malware being present in uploaded files.

Most application service providers (ASPs) do not discuss the risk, and do not take any steps to mitigate the risk; some do not even authenticate access to user files. An exception is Google: a Google article [10] explains that Google applications download untrusted files with a *Content-Disposition* HTTP header with value *attachment*. This is an unofficial HTTP header, borrowed from email, which browsers interpret by giving the user an opportunity to save the file instead of displaying it. According to the article, if the user chooses to display the file, Firefox will consider the file as originating from the user's local host, thus preventing any scripts in the file from accessing the application; IE, on the other hand will consider the file as originating from the application, thus rendering the *Content-Disposition* countermeasure ineffective; the article suggests wrapping the file in a zip archive to cope with this.

The essence of the Content-Disposition countermeasure is to prevent a file that may contain malicious scripts from being displayed. Internet Explorer 8 (IE8) will have three new features that will help with this goal, referred to a *Restrict Upsniff*, *Sniffing Opt-Out*, and *Force Save* [11][12]. Often, however, a shared HTML file is intended to be displayed rather than downloaded, hence the need for a different kind of countermeasure.

Two proposals [13][14] aimed at mitigating XSS are applicable to HTML file sharing. Both would allow the Web application to forbid the execution of scripts in files being downloaded, using HTTP headers. However they both require buy-in from the browsers, and thus are not immediately usable. Also, users may want scripts to be executed, which the countermeasure of Section 4 below allows.

The idea of using multiple hostnames as a defense against attacks through shared files or untrusted HTML content has been suggested before. Jesse Ruderman, in a Web page that gives tips to developers [15], says:

> If you must allow unsanitized, untrusted HTML to be part of your site, ensure that those pages are not on the same hostname as where other users log in. (webmail, web hosts, attachments in a bug-tracking system, Google cache) (see Gerv's proposal)

(In the quoted sentence, "Gerv's proposal" is presumably a reference to [13].) Brian Eaton, in a post to the Web Security mailing list [16] on how to avoid XSS attacks through PDF files, says:

> You could host the file using a different virtual hostname, to limit the damage that XSS could do to your site. If you use domain cookies for anything you'd need a different DNS domain as well.

And the same Google article quoted above [10] also suggests, as an additional or alternative countermeasure to the Content-Disposition header:

> ...you can set up a separate domain name to serve files for download, e.g. DOMAINNAMEmail.com serving downloads for mail.DOMAINNAME.com. The file server domain shouldn't use any cookies, taking only a file identifier and download authorization key as URL parameters.

None of these suggestions, however, amounts to a full-fledged solution.

Instead of using different hostnames, it has also been suggested to use different ports. Peter Watkins has reported [2] that the Acmemail Webmail project offered the option of using multiple port numbers, and recommended that the Web server be configured to listen on two ports, a *control port* and a *message port*.

3. Cross-user and cross-instance attacks through HTML shared files

As briefly mentioned in the introduction, scripts embedded in shared files can be used to carry out a variety of cross-user and cross-instance attacks that cannot be mitigated by browser warnings or virus scanners.

The basic attack mechanism is as follows. The attacker, a malicious user of an application instance, uploads an HTML attack file, containing an attack script, to the repository of shared files of an application instance. The victim, another user of the same instance, downloads the file, causing the victim's browser to execute the script. The script may then be able to download other user files using an authentication cookie set in the victim's browser and send their contents to the attacker, who may not be authorized to see those other files. Furthermore, if the attack file is served with the same hostname as Web pages pertaining to the application instance (*application instance pages*) the same-origin policy may allow the attack script to read the contents of application instance pages that the victim but not the attacker is authorized to access. Worse, the script may thus be able to obtain a token embedded in a hidden field of an application instance form as a countermeasure against cross-site request forgery (CSRF) [17], and use it to defeat the countermeasure. The script can then submit the form, thus allowing the attacker to act upon the application instance while impersonating the victim, and with the privileges of the victim.

As a slight variation on the basic attack, the attacker may plant the attack file in advance of an anticipated revocation of the attacker's rights to access the application instance. The attacker may thus continue to read user files, read application instance pages, and interact with the application instance after revocation. If the victim's privileges include the right to create application instance accounts for new users, the attacker may be able to restore his or her lost access rights. This variation is of concern even for applications where all users have the same privileges and are authorized to access all files, a case in which cross-user attacks would seem at first glance to be inconsequential.

With a more elaborate variation, the attacker may be able to gain access to an application instance to which the attacker has not had access before. This requires some social engineering. The attacker wants access to a target application instance to which the victim has access. The attacker uses a second application instance, to which the attacker has access, as a decoy; the decoy instance may already exist, or the attacker may register with the ASP and arrange for its creation. The attacker lures the victim into obtaining a user account for the decoy instance and logging in to the decoy instance while being logged in to the target instance. The attacker uploads an attack file with an attack script to the repository of the decoy instance and lures the victim into downloading the attack file. If the same hostname is used to serve user files and application pages pertaining to all instances, the same origin policy allows the attack script to access the target instance, read user files and application pages of the target instance and forward their contents to the attacker, interact with the target instance, and even, if the victim has sufficient privileges, create a user account at the target instance for him or herself.

4. Countermeasure

The same-origin policy prevents a script contained in an HTML document from accessing the contents

of a Web page or file downloaded from a *different origin*. Assuming that all relevant pages or files are downloaded using the same protocol and port (typically *https* and 443 respectively), the *origin* of a page or file is, in effect, the DNS domain from which the page is served, or any domain that contains it as a subdomain. In other words, the origin is the hostname portion of the URL used to retrieve the page or file (henceforth, the *hostname of the page or file*), or a broader domain to which that hostname belongs.

The same-origin policy can be used to prevent attacks through HTML shared files if the hostnames of user files and application pages are chosen so that they satisfy the following two conditions:

(1) The hostname of a user file of an application instance is different from, and is not a subdomain of, the hostname of any application page of the same or any other application instance. This ensures that an attack script in a user file will not be able to read the contents of any application page. It prevents an attacker, whether a current user or a former user, from using the script to obtain application data or to defeat CSRF countermeasures.

(2) The hostname of a user file of an application instance is different from, and is not a subdomain of, the hostname of any user file of the same or any other application instance. This ensures that an attack script in a user file will not able to read the contents of any other user file. It prevents the attacker from using the script to circumvent a system of file permissions or to continue reading user files after the attacker's credentials for accessing the application instance have been revoked.

These conditions could be satisfied, for example, by using user file hostnames of the form

userfiles-*AppInstID-FileID.RegisteredDomain*

and application page hostnames of the form

application-*AppInstID.RegisteredDomain*

where *AppInstID* is a unique application instance identifier, *FileID* is a unique file identifier, and *RegisteredDomain* is a domain name obtained by the ASP from an Internet domain registrar.

This hostname scheme, however, raises serious difficulties.

First, users may want to place links to an HTML user file in other HTML files (located anywhere in the World Wide Web). It is unreasonable to ask users to embed a file identifier in the hostname portion of the URL when constructing such a link. The file should instead be identified by an ordinary file path appended to a hostname that is the same for all user files pertaining to the same application instance.

Second, HTML user files belonging to the same application instance should be linkable to each other using relative URLs. This requires, again, that all user files pertaining to the same application instance have the same hostname.

Last but not least, an authentication cookie set upon login in a user's browser, if associated with the domain

application-*AppInstID.RegisteredDomain*,

cannot be used to authenticate user files that use different hostnames.

The solution to these difficulties lies in the combination of two ideas:

(1) Use two different URLs to retrieve a user file, with a redirection step mapping the first one to the second.

(2) Set two different authentication cookies upon login, one for user files and the other for

application pages.

Of the two different user file URLs, the first one is a *standard URL* consisting of a *standard hostname* of the form

userfiles-*AppInstID.RegisteredDomain*

and an ordinary file path that identifies the file. Here is an example of a standard URL:

https://userfiles-123456.pomcor.com/folder1/folder2/file1.html

Standard URLs are reasonable ones to use in links to user files, and since all standard URLs for a given application instance have the same hostname, it is possible to use relative URLs in links to user files from other user files.

The second user file URL is an *extended URL* consisting of an *extended hostname* of the form

userfiles-*AppInstID-FileRetrSessID.RegisteredDomain*

and the same file path, e.g.:

https://userfiles-123456-c4482d6f9e85d1a2c6441ea447c4167211ddc321.pomcor.com/folder1/folder2/file1.html

where *FileRetrSessID* is a random, high-entropy, *file-retrieval session ID*. When the application receives a standard URL request, accompanied by a user file authentication cookie, it generates this ID and uses it as the primary key of an ephemeral file-retrieval session record that contains a reference to the login session identified by the cookie; then it redirects the browser to the extended URL. When the extended URL is received, the application uses the ID embedded in the hostname to authenticate the request by locating the file-retrieval session record and the login session record that it refers to; then it deletes the file-retrieval session record.

Besides being used for authentication, the file-retrieval session ID present in the hostname also serves to prevent an attack script contained in the user file from accessing any other user file, just like the above *FileID* would.

As for the two authentication cookies, their associated domains are the standard user file hostname and the application page hostname respectively. The application can set the user file cookie upon login by redirecting the browser to a special URL whose hostname component is the standard user file hostname, and then redirecting it again to the URL of an initial application page. The special URL must have a path component that cannot be interpreted as the path of a user file; for example, user file and folder names beginning with a dot could be disallowed, and the path of the special URL could begin with a dot. The special URL must also contain an ephemeral secret, analogous to the ephemeral file-retrieval session ID, to authenticate the first redirection.

5. Remarks

5.1 Peter Watkins [2] has raised the issue that a browser may issue multiple HTTP requests in the course of a single file download. Browsers may issue multiple Range requests if the file is very large and network problems cause the connection to be lost. Some versions of IE have also been observed to issue two requests when sniffing the contents of Office files. Since the file retrieval session record is deleted when the first request is request is received, any subsequent requests will fail, and that will cause the download to be restarted or fail altogether.

If this is actually a problem, it can be solved by the following improvement, based on a comment by Bil Corry [3].

Instead of deleting the file retrieval session record, preserve the record, but generate a second session

ID, store it in a second-session-ID field of the file retrieval, and set a file-retrieval-session cookie with this second ID. This is done when the first request is received; more specifically, when the application receives the redirection of the first request to the extended user file URL. The cookie is associated with the hostname of the extended user file URL, which includes the first file-retrieval session ID, and is used to authenticate subsequent extended-user-file-URL requests made in the course of the same file download.

Thus, the following algorithm is used to authenticate an extended-user-file-URL request:

- Use a the file retrieval session ID in the hostname of the request to locate a file-retrieval session record whose primary key is that session ID. Fail if there is no such record.
- Check if the second-file-retrieval-session-ID field in the record has a value. If it does, check if the request is accompanied by a cookie containing that value. Accept the request if and only if this is the case.
- If the second-file-retrieval-session-ID field does not have a value, generate the second file retrieval session ID, add it to the record, and set the cookie, as described above.

Notice that the addition of the second file retrieval session ID to the record has the implicit effect of invalidating the first file retrieval session ID in its role as an authentication token, since the presence of the second ID in the record causes the application to look for the cookie.

To recapitulate, the following cookies are used in the course of a login session: a cookie for authenticating application page requests, associated with the hostname of the application page URL for the application instance; a cookie for authenticating standard-user-file-URL requests, associated with the standard-user-file hostname for the application instance; and, for every user-file download that takes place the login session, a file-retrieval-session cookie for authenticating extended-user-file-URL requests, associated with the extended user file hostname generated for the download. Notice that different downloads of the same file cause different file-retrieval-session records to be created, and hence different file-retrieval cookies to be set for the same file, associated with different hosntames.

(Note: The file retrieval session ID should have a short validity period, implemented by a creation timestamp stored in the record. I neglected to point this out in the original version of the paper, but it was mentioned by Bil Corry in connection with his proposed improvement [3].)

(Note: The security role of the file retrieval session ID in the extended URL, and that of the file-retrieval cookie, is only to warrant that the user has logged in. The application must still verify that the user is a valid user with permission to download the particular file at the time when an extended-file-retrieval-URL request for the file is received. It follows from this that it is not strictly necessary to record the file path in the file retrieval session record. It may be a good idea to do so, however, as an additional precaution (e.g. to avoid future bugs as the code evolves). Associating the file retrieval session ID with the particular file being downloaded was suggested by Peter Watkins [2].)

5.2 It is important that the extended user file hostname begin with the string "userfiles" or some other string that makes it clear that the hostname is used to serve user files; and the difference between user file hostnames and application page hostnames should be brought to the attention of the users. There are two reasons for this.

First, the extended user file hostname appears in the address box of the browser when an HTML user file is displayed. Making the hostname recognizable as a user file hostname will mitigate the risk of an HTML user file masquerading as the login page or some other application page.

Second, the warning issued by the browser before running an executable user file will show the hostname of the file. Making the hostname recognizable as a user file hostname will mitigate the risk

of an executable user file masquerading as a helper client-side program that the application is asking the user to install.

5.3 The countermeasure prevents a script contained in an HTML user file from reading any other user file. This imposes a constraint on the functionality of the private Web site comprised by the collection of HTML files of an application instance. It is possible to remove or relax this constraint at the cost of some reduction in security.

If there is no system of file permissions and all instance users are allowed to download all files, then the constraint could be removed by eliminating the extended user file hostname and the user file redirection. The security cost would be the risk that a former user could retain access to user files.

If there is a system of file permissions based on a partition of the file space into several *security areas*, then the constraint could be relaxed by replacing the file-retrieval session ID with a security area ID. A script contained in an HTML user file could then access other files in the same security area. The security cost would be the risk that a former user could retain access to user files in the same security areas that user could access before his or her credentials were revoked.

5.4 The countermeasure remains useful in the case where there is only one application instance. (Of course, there are then no application instance IDs in hostnames.) There is no risk of cross-instance attacks in this case, but the countermeasure still protects against cross-user attacks and former user attacks.

6. References

[1] Web Security Mailing List. Thread: countermeasure against attacks through HTML shared files. http://www.webappsec.org/lists/websecurity/archive/2008-11/msg00023.html

[2] Peter Watkins, post to the Web Security mailing list (included in [1]). http://www.webappsec.org/lists/websecurity/archive/2008-11/msg00032.html

[3] Bil Corry, post the Web Security mailing list (included in [1]). http://www.webappsec.org/lists/websecurity/archive/2008-11/msg00039.html

[4] Multitenancy. Wikipedia article, http://en.wikipedia.org/wiki/Multitenancy

[5] Same Origin Policy. Wikipedia article. http://en.wikipedia.org/wiki/Same_origin_policy

[6] Cross-Site Scripting. Wikipedia article. http://en.wikipedia.org/wiki/Cross-site_scripting

[7] Risks of File-Sharing Technology. Cyber Security Tip ST05-007. http://www.us-cert.gov/cas/tips/ST05-007.html

[8] MySecureCyberspace: File Sharing. http://www.mysecurecyberspace.com/secure/file-sharing.html

[9] Web Security Mailing List. Thread: File Uploading Vulnerabilities. http://www.webappsec.org/lists/websecurity/archive/2008-09/msg00022.html

[10] Google. ArticleUntrustedDownloads. HOWTO serve untrusted files as downloads. http://code.google.com/p/doctype/wiki/ArticleUntrustedDownloads

[11] Eric Lawrence. IE8 Security Part V: Comprehensive Protection. http://blogs.msdn.com/ie/archive/2008/07/02/ie8-security-part-v-comprehensive-protection.aspx

[12] Eric Lawrence. IE8 Security Part VI: Beta 2 Update. http://blogs.msdn.com/ie/archive/2008/09/02/ie8-security-part-vi-beta-2-update.aspx

[13] Gervase Markham, Content Restrictions. http://www.gerv.net/security/content-restrictions/

[14] Brandon Sterne, Content Security Policy. http://people.mozilla.org/~bsterne/content-security-policy/

[15] Jesse Ruderman, Security Tips for Web Developers. http://www.squarefree.com/securitytips/web-developers.html

[16] Brian Eaton, post the the Web Security mailing list. http://www.webappsec.org/lists/websecurity/archive/2007-05/msg00087.html

[17] Cross-Site Request Forgery. Wikipedia article. http://en.wikipedia.org/wiki/Cross-site_request_forgery