HEXAGON SECURITY

Dynamic Cross-Site Request Forgery

A Per-Request Approach to Session Riding

By
Nathan Hamiel and Shawn Moyer

July 24, 2009

Revision 1

Abstract

Cross-Site Request Forgery ("CSRF") is typically described as a "replay" or static type of attack, where a bad actor uses markup, Javascript, or another method to force a client browser to perform a known, repeatable in-session transaction on the affected site without the user's knowledge.

Typical defensive measures against CSRF address this by creating unique, per-session or per-request tokens that aren't typically available to an attacker, and by checking for other browser behaviors such as referring URL.

In this paper we describe a number of approaches to construct "dynamic" CSRF attacks, forging unique requests on an automated, per-target basis, even in scenarios where Cross-Site Scripting or Cross-Domain issues don't exist.

"Classical" Cross-Site Request Forgery

Pete Watkins coined the term Cross-Site Request Forgery in a post to Bugtraq in 2001[1], for what had been described in 1988 by Norm Hardy as the "Confused Deputy"[2], in reference to similar problems with compilers and timesharing systems.

The fundamental premise of CSRF is this: In the course of web browsing, a target user encounters a request from a malicious site or location that makes a request on behalf of the user to a site the user is already authenticated to.

A forged request could be in the form of an HTML tag such as an IMG, IFRAME, or META refresh, embedded in Javascript or Flash, or sent as a 300-series HTTP redirect from a location under the attacker's control. If the target user is authenticated, relevant HTTP headers and any authentication data are sent to the site along with the malicious request.

From Watkins' 2001 Bugtraq post:

1 http://www.tux.org/~peterw/csrf.txt
2 http://www.cis.upenn.edu/~KeyKOS/ConfusedDeputy.html

CSRF does not in any way rely on client-side active scripting, and its aim is to take unwanted, unapproved actions on a site where the victim has some prior relationship and authority.

Where XSS sought to steal your online trading cookies so an attacker could manipulate your portfolio, CSRF seeks to use your cookies to force you to execute a trade without your knowledge or consent (or, in many cases, the attacker's knowledge, for that matter).

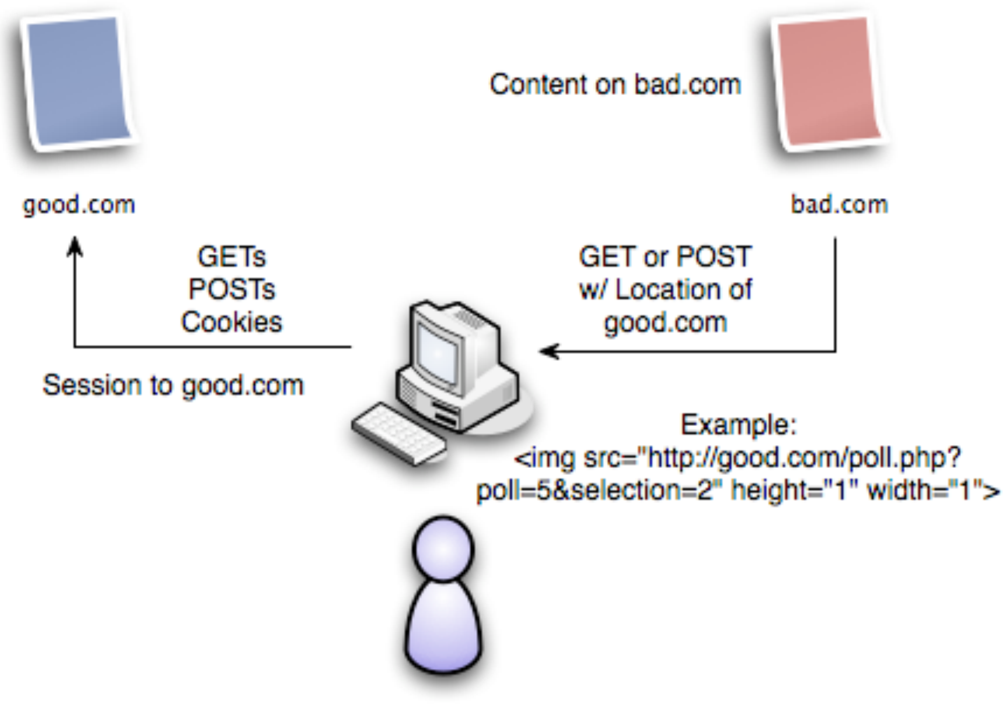

Figure 1*: An example of "Classic" CSRF. Hidden markup on bad.com forges a client request to an authenticated session on good.com.*

In 2004, Thomas Schreiber's paper "Session Riding" [3] further expanded on Watkins post, and went on to describe a number of attack scenarios where CSRF was possible and valuable to an attacker, providing the most detailed accounting of the problem at the time.

Schreiber went on to recommend tokenization as a solution, in addition to URL rewriting:

3 http://www.securenet.de/papers/Session_Riding.pdf

> *To make a Web application safe against Session Riding, introduce a secret (aka hash, token) and put it into every link and every form, at least into every form that is sensitive. The secret is generated after the user has logged in. It is stored in the server-side session. When receiving an http request that has been tagged with this information, compare the sent secret with the one stored in the session. If it is missing in the request or the two are not identical, stop processing the request and invalidate the session.*

Schreiber's is the most common approach to CSRF protection, and is the typical mitigation approach in place today. While in the case of some newer frameworks, a CSRF token may be unique to a given generated request and recreated per-pageview, in practice CSRF token values are often valid for a given lifetime, such as a user's session, and not always mapped to a specific session at all.

Interestingly, Watkins' original 2001 post saw the flaw in tokenization alone as a defensive approach:

> *A combination of cookie + URL mangling might not be bad, though in the message board case, a CSRF attacker could use an intermediate redirect (as described earlier) to get the URL mangling (from the Referer), and redirect back to the messageboard with the proper mangling as well as all cookies that might be expected/needed. So in your example case, URL mangling would buy nothing.*

Watkins' point, that tokens or "URL mangling" buy nothing for sites where offsite linking and referrer leakage is possible, remains valid today, and with the web's increasing emphasis on offsite content, shared APIs, and user-generated content, a number of opportunities exist to prove Watkins' point.

Additionally, while the tokenization approach is sound, it must be implemented properly, with tokens being mapped to a given user session, not reusable, and sufficiently large and random enough to prevent brute-force attacks.

A game-changing moment for CSRF was the Samy[4] worm, which combined Cross-Site Scripting with CSRF to propagate. In cases where this is approach feasible, CSRF is often a payload of choice for Cross-Site-Scripting, using the XSS to forge requests that include user-specific request tokens and other relevant session information.

"Dynamic" Cross-Site Request Forgery

4 http://namb.la/popular/tech.html

In a “dynamic” CSRF scenario, expanding on the approach described by Watkins, an attacker creates a customized, per-request forgery, based on each user’s session-specific information, including valid CSRF tokens and other parameters specific to the user’s session.

In order for this approach to succeed *without* Cross-Site Scripting, a user would typically need to make a request to a site under attacker control, and the useful information to construct the request must be accessible to the attacker.

This scenario is more likely than it would first appear. Clicking on a link in the context of a webmail application or message forum may leak the necessary information through referrer, or if the target site permits the inclusion of offsite content, such as offsite images (typical on many web forums and social networks), an attacker could silently obtain the necessary information to construct a custom forged request, and send a customized response to the target browser to call the “dynamic” CSRF.

Additionally, other scenarios such as session fixation or brute-force against tokens from browser history[5] might be feasible as methods to construct a “dynamic” CSRF, again constructing the forgery “on the fly” for each target request.

5 http://securethoughts.com/2009/07/hacking-csrf-tokens-using-css-history-hack/

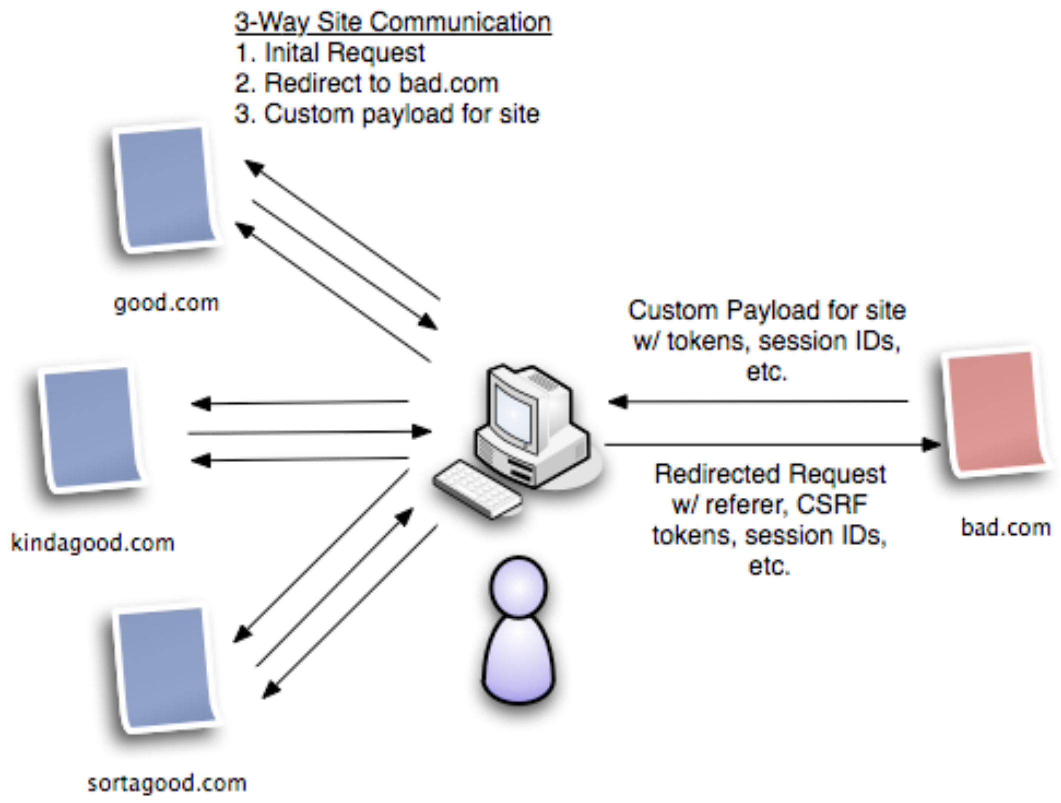
3-Way Site Communication
1. Inital Request
2. Redirect to bad.com
3. Custom payload for site
good.com
kindagood.com
sortagood.com
Custom Payload for site w/ tokens, session IDs, etc.
Redirected Request w/ referer, CSRF tokens, session IDs, etc.
bad.com

Figure 2: *"Dynamic" CSRF scenario: User request is redirected to a CSRF with per-user session information included, such as CSRF token and session ID.*

In short, if it's possible to obtain session specific information from a target browser in some fashion, "dynamic" CSRF is possible. To take advantage of this problem on an automated basis, an application controlled by the attacker could package different payloads based on domain or URL from which a given request came, and include any relevant "secret" URL-based information in the forged request, such as valid Session IDs or CSRF tokens, obtained either via cross-domain referrer leakage or from other means such as Session Fixation and other exposures.

As a trivial example, if the following value in the referrer were obtained from a request to an attacker-controlled server, originating from a target browser:

http://good.com/function.php?val1=foo&val2=bar&token=765234

In "dynamic" CSRF, a simple application under attacker control could repackage the request, sending a new request to the browser with a custom payload including the a valid token, such as:

http://good.com/anotherfunction.php?val4=morefoo&val5=morebar&token=765234

If the user's browser encounters the custom request as a redirect, a hidden form POST, or some other method, the request is sent along with any relevant HTTP authentication headers for good.com.

"Dynamic" CSRF Payloads

An application that constructs a dynamic payload for CSRF could take a number of approaches. Each scenario will vary depending on what type of data the attacker has the ability to put in the victims path, and the specific function being targeted on the site in question.

HTTP Redirect Payload

A trivial payload, where possible from cross-domain referrer leakage, is the use of a simple HTTP 302 redirect. In this case the attacker would use leaked session data and construct a redirect with a location value populated with the relevant session data obtained from the target browser.

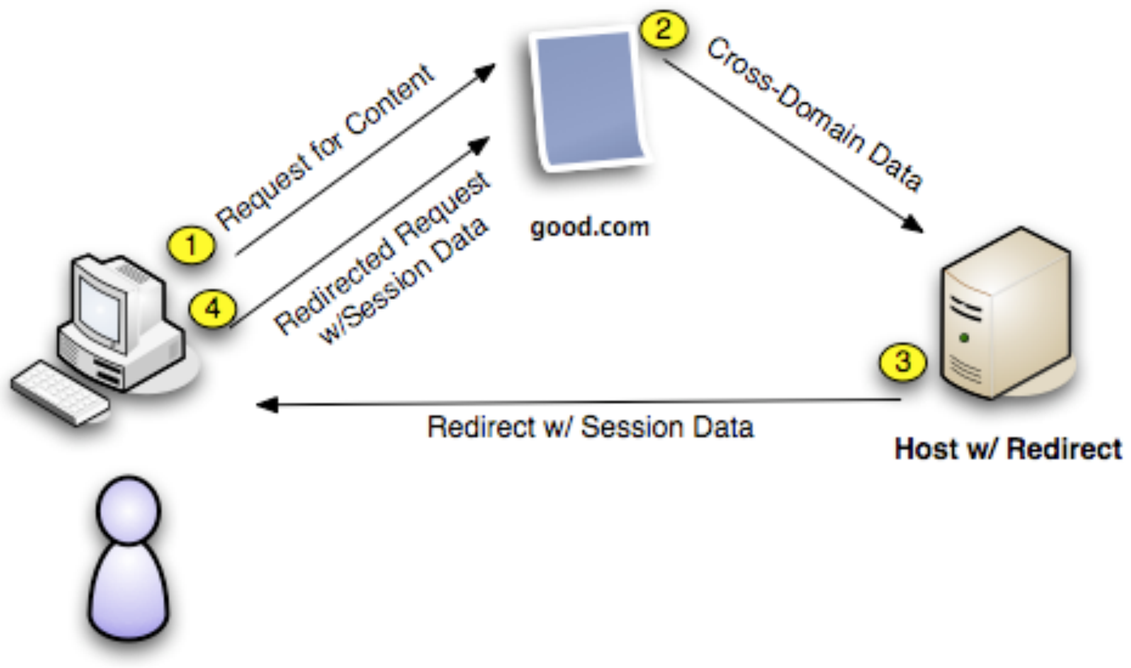

Figure 3: *Redirect payload, sending custom redirect to client browser via HTTP 302.*

A dynamic payload is created with the associated token value and other session data is sent. This would allow an attacker to potentially bypass CSRF protections in place and execute a request of the attacker's choice, and if the redirect is sent from a link in the context of a target site, referrer checks wouldn't apply - the referring site would remain the source of the request, since browsers typically leave the original referrer intact.

HTML-based Redirect

An application that creates "Dynamic" CSRF payloads could also create a custom page based on referrer content and redirect the user to another site, performing the CSRF in the background.

An interesting scenario here would be the use of a URL shortening service such as TinyURL, Is.gd, or others, sending a redirect to an ultimate "legitimate" destination, but embedding a custom CSRF payload in as part of an interstitial HTML page that includes a meta refresh to another location that appears legitimate. The generated HTML could construct a GET request with session data taken from cross-domain data, or even a self-submitting form that constructs a POST request.

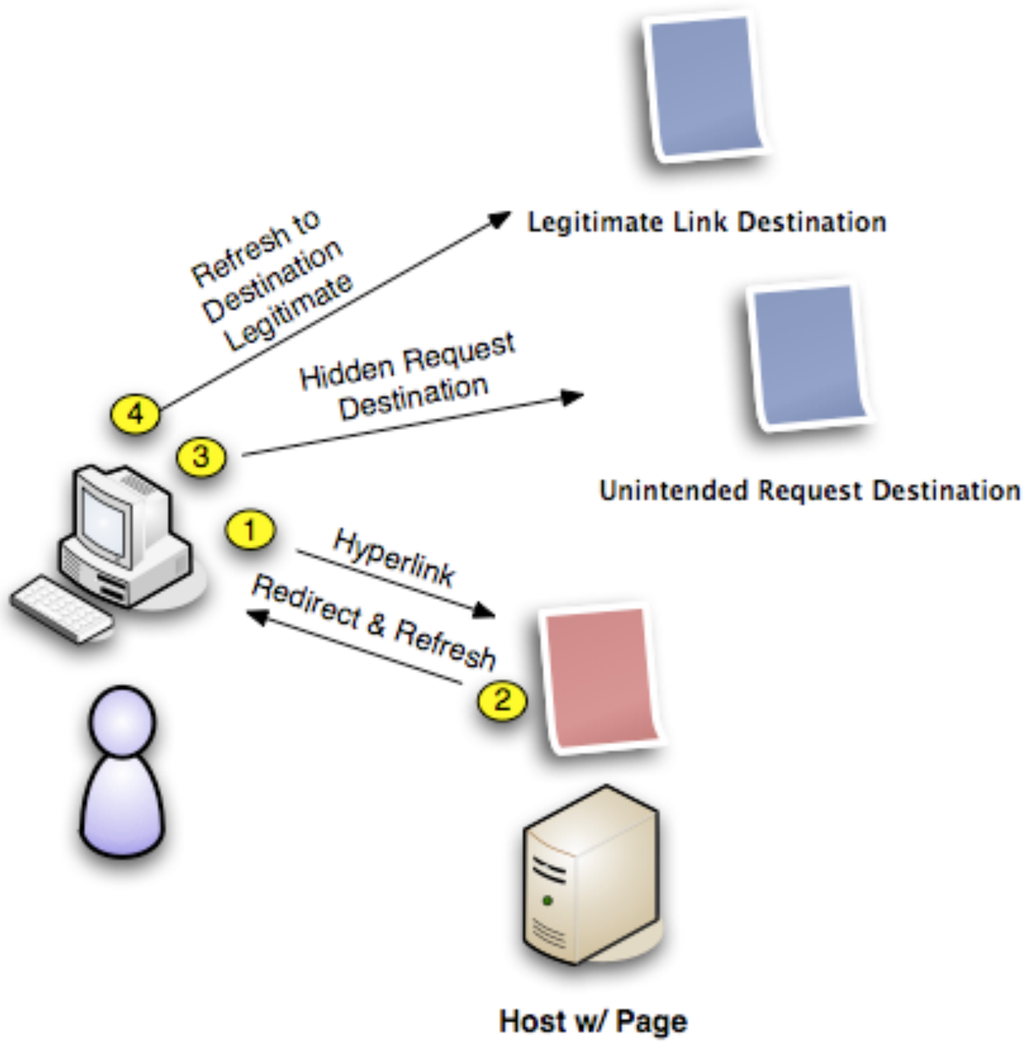
Legitimate Link Destination
Refresh to Legitimate Destination
4
Hidden Request Destination
3
Unintended Request Destination
1
Hyperlink
Redirect & Refresh
2
Host w/ Page

***Figure 4**: META refresh which includes hidden "dynamic" CSRF constructed from referrer leakage.*

Token Fixation “Dynamic” CSRF

In a scenario where CSRF tokens are reusable and not linked to a given user session, a “dynamic” CSRF could construct a payload that issues a valid token to the user obtained by making a request to the site directly and obtaining a valid token. This approach is similar to session fixation attacks where SessionID is not paired to a given IP address. A token is obtained, and if a user can be enticed to visit a site under attacker control, a request is made on behalf of the user in the context of the target site.

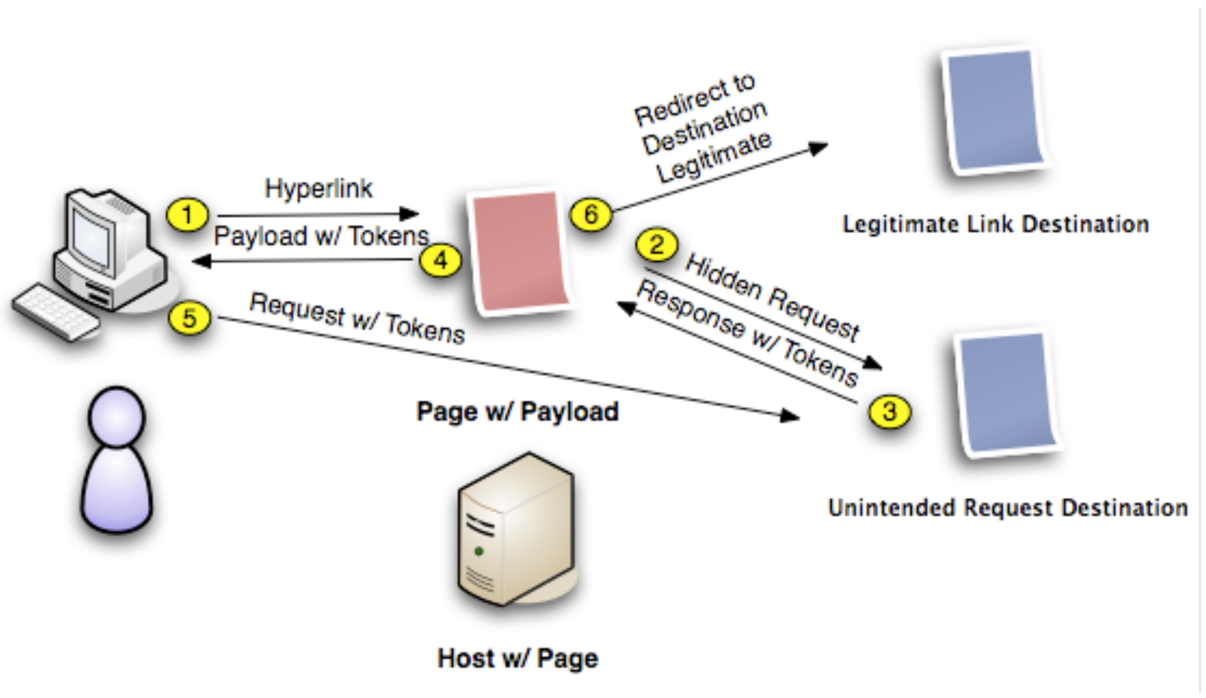

***Figure 5**:* "Dynamic" CSRF with token/session fixation. Application under attacker control makes a request for a valid token that isn't mapped to a specific session, and issues it to the user.

POST-based "Dynamic" CSRF

It is also possible to construct a POST-based "dynamic" CSRF request that requires no interaction from the user's browser. This could be possible in a multi-stage scenario, where for example CSRF tokens and other relevant information that are valid for the duration of a session are leaked via an offsite image or offsite content, and could then be combined into a customized forged POST request.

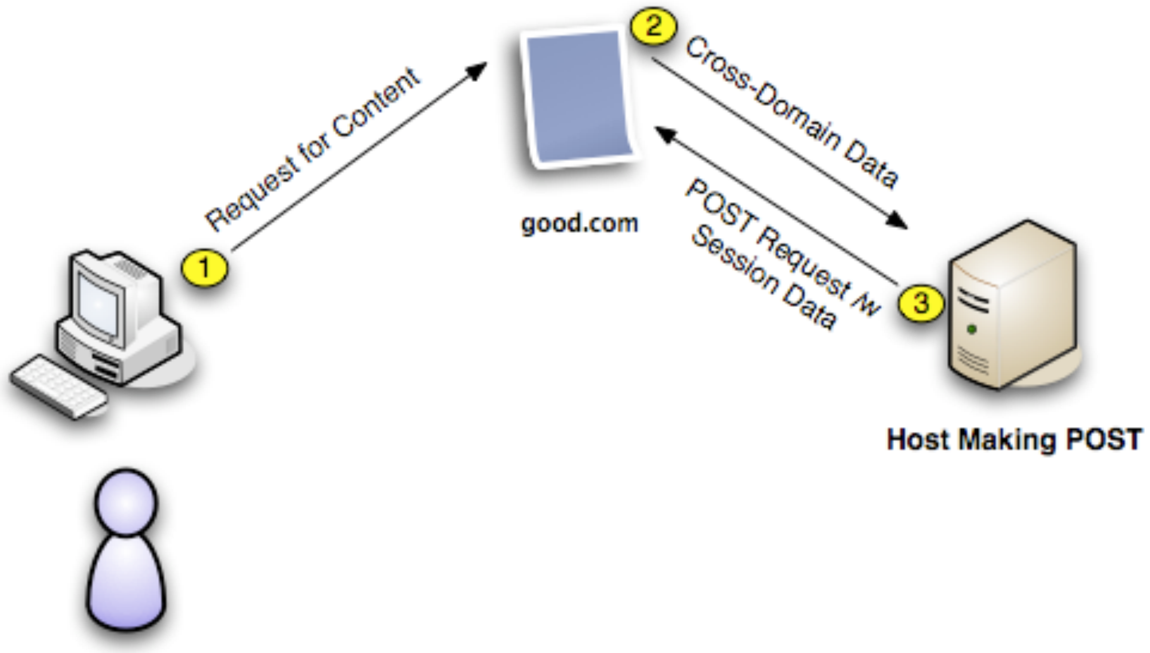

***Figure 5**: POST-based "dynamic" CSRF scenario.*

Further Thoughts

Theft of CSRF tokens isn't the only use case for this approach. Even for CSRF where tokens or authentication mechanisms aren't in use or can be bypassed , dynamically constructing the forgery on a per-request basis may be useful, such as targeting parameters in the URL that are specific to a given authenticated user.

The larger point here is simply to remember the statement Watkins made eight years ago, which seems to have been missed: CSRF is not purely a static attack based only on known, repeatable parameters, it can be fluid, dynamic, and tailored to each target.

References

1. Norm Hardy, “The Confused Deputy”, 1988, Available: http://www.cis.upenn.edu/~KeyKOS/ConfusedDeputy.html
2. Pete Watkins, “Cross-Site Request Forgery”, June 13, 2001, Available: http://www.tux.org/~peterw/csrf.txt
3. Thomas Schreiber, “Session Riding”, December, 2004, Available: http://www.securenet.de/papers/Session_Riding.pdf
4. Samy Kamkar, “MySpace Worm Explanation”, June, 2006, Available: http://namb.la/popular/tech.html
5. SecureThoughts.com, “Hacking CSRF Tokens using CSS History Hack”, July, 2009, Available: http://securethoughts.com/2009/07/hacking-csrf-tokens-using-css-history-hack/