Joe Rozner Follow
Startups, infosec, snowboarding. I build shit in Los Angeles
Oct 17, 2017 · 20 min read

Wiping Out CSRF

It's 2017 and there shouldn't be much more to say about Cross-Site Request Forgery (CSRF) that hasn't already been said. It's a vulnerability that has been known for years with well-documented and proven solutions that have been implemented in popular web development frameworks. So why are we still talking about it? There are a few reasons: 1) legacy applications lack CSRF protection, 2) some frameworks use poorly built implementations, 3) applications don't take advantage of proven framework implementations, or 4) new applications don't use modern frameworks that provide CSRF protection. CSRF is still a prevalent vulnerability that continues to be commonly found in web applications.

This post will start with a deep dive to explain how CSRF works and how modern countermeasures work to protect against it. We'll then offer a solution that can be applied after an application has been written, without requiring source code modifications. Lastly, we'll examine a new extension to cookies that, if turned into a standard, could be the final nail in the coffin for most cases of CSRF. A companion code repository goes along with this post that contains a sample implementation with test cases. GitHub Repository

Understanding the Attack

At its most fundamental level, CSRF is a vulnerability where an attacker forces a victim to make an HTTP request on the attacker's behalf. It's an attack that occurs entirely on the client's side (e.g. web browser) where countermeasures trust that the victim is sending an application information that can be trusted. There are three components that enable the attack to occur: Improper usage of unsafe HTTP verbs, web browser cookie handling, and Cross-Site Scripting (XSS).

The HTTP specification separates verbs into two categories safe and unsafe. Safe verbs (GET, HEAD, and OPTIONS) are intended for use in read only operations. Requests that use them are designed to return information about the resource that is requested and should have no side effects on the server. Unsafe verbs (POST, PUT, PATCH, and DELETE) are intended for modification, creation, or deletion of a resource. Unfortunately, it's possible for an HTTP verb to be ignored or it's intended behavior to not be strictly enforced.

A major reason for incorrect verb use is due to the historically poor browser support of the HTTP specification. Up until the prevalence of XML HTTP

Request (XHR), it was simply not possible to use a verb other than GET or POST without relying on specific framework and library hacks. This limitation resulted in the practical irrelevance of distinguishing between HTTP verbs. This alone is not sufficient to create the condition of CSRF but it contributes and makes protection against it significantly more difficult. The biggest factor that contributes to the vulnerability of CSRF is the handling of cookies by the browser.

HTTP was originally designed to be a stateless protocol with a single request corresponding to a single response and no state carried between requests. To support complex web applications, cookies were created as a solution to keep state between related HTTP requests. Cookies reside at the global level of the browser and are shared across instances, windows, and tabs. Users depend on web browsers to automatically transmit cookies for every request. Since cookies are accessible/modifiable in the browser and lack anti-tamper protection, the storage of state has shifted to server managed sessions. In this model, a unique identifier is generated on the server and stored in the cookie. Each browser request sends the cookie with it and the server is able to look up the identifier to determine if it is a valid session. When the session ends, the server forgets the identifier and invalidates any further requests that send it.

The issue lies in how cookies are managed by browsers. A cookie is composed of a handful of attributes, but the most important one we care about here is the Domain attribute. The intended functionality of the Domain attribute is to scope the cookie to a specific host that match the domain attribute of the cookie. This was designed as a security mechanism to avoid sensitive information, such as session identifiers, from being stolen by malicious websites where attackers could perform session fixation attacks. The flaw here is that the domain attribute does not depend on the Same Origin Policy (SOP); it simply compares the domain value of the cookie to the host in the request. This allows requests originating from a different origin to also carry along with it any cookies for that host. This is safe behavior if and only if safe and unsafe verbs are used correctly; example: a safe request (GET) should never change state but as we've seen correct use cannot necessarily be trusted. If you don't know about the SOP, then you should read more about it: https://en.wikipedia.org/wiki/Same-origin_policy.

The last component to focus on is Cross-Site Scripting (XSS). XSS is the ability for attacker controlled JavaScript or HTML to be rendered in the DOM by a victim. If XSS exists in the application it's essentially game over when it comes to stopping CSRF attacks. The main countermeasure that we're going to

discuss in this post, and most applications depend on, can be bypassed if XSS is possible.

Performing the Attack

Now that we're aware of the contributing factors, let's dive deeper to understand just how CSRF works. If you haven't set it up yet, now would be a good time to follow the instructions in the companion repository and get the examples running. Instructions for a basic setup are available in the README as well as a short walkthrough for getting started.

There are three traditional distinct types of CSRF that we're going to cover:

1. **Resource inclusion**
2. **Form-based**
3. **XMLHttpRequest**

Resource inclusion is the type that you'll likely see in most demos or basic lessons that introduce the concept of CSRF. This type boils down to an

attacker controlling the resource that is included by an HTML tag such as image, audio, video, object, script, etc. Any tag that includes a remote resource allows this if the attacker is capable of influencing the URL that is loaded. Due to the lack of origin checking with cookies, as described above, this attack does not require XSS and can be performed by any attacker controlled site or the site itself. This type is limited exclusively to GET requests, because those are the only types of requests that browsers will make for resource URLs. The main limitation of this type is that it requires improper use of safe HTTP verbs.

The second type we'll discuss is form-based CSRF, commonly seen when safe verbs are used correctly. It is performed by the attacker creating their own form that mimics the one they want the victim to submit; it includes a snippet of JavaScript that forces the victim's browser to submit the form. The form can consist entirely of hidden elements and the form submission should happen so quickly that the victim never actually sees it. Due to the handling of cookies, this can be hosted on any site by the attacker and the attack will succeed as long as the victim is logged in with valid cookies. A successful attack will land the victim on whatever page they would normally be taken to

if the request was intentional. This method works particularly well for phishing attacks where an attacker can direct the victim to the page.

The last major type we'll discuss is XMLHttpRequest (XHR). This is probably the least likely one to see, due to the number of requirements. Since many modern web applications rely on XHR, we'll spend a lot of time building and implementing this particular countermeasure. XHR-based CSRF typically comes in the form of an XSS payload due to the SOP. Without Cross Origin Resource Sharing (CORS), XHR is limited to requests only to the origin which limits an attacker from hosting their own payloads. An attack payload for this type of CSRF is essentially a standard XHR that an attacker has found some way to inject into the victim's browser DOM.

The following solution is a great simplification of an actual implementation that can be used for a real deployment. It focuses primarily on the client-side and token management. Interception and modification of requests and responses has many strange edge cases that require large amounts of specific domain knowledge about the platform it's being performed on. There is a massive trade off in understanding platform complexity for avoiding understanding and handling the complexity of CSRF itself. Ideally the best

solution would be to use a framework that provides CSRF protections built in and utilize that. Despite the disclaimer, there remain many reasons a solution like the following makes sense.

Modern Protections

There are many cases where modifying an application to protect against CSRF isn't possible. Either the source code isn't available, the risk of application modification is too high, or it's not easily done within the application's constraints. This solution lends itself particularly well to being deployed within a RASP, WAF, reverse proxy, or load balancer, and can be used to provide protection to a single application or many all with the same configuration. This is particularly useful when the deployment platform is understood well but the applications it is being applied to aren't. Let's discuss the current solutions that are typically used for protecting against CSRF and how we can build them with the above requirements.

First and foremost, correct usage of safe and unsafe HTTP verbs is important. This alone is not a valid solution but it will make everything significantly easier and the next two methods depend upon it. Unfortunately, there isn't a

solution for this that can be applied after the fact. This is something that needs to be done at the time of building the application and requires design and architecture. Fortunately, most modern web frameworks have a concept of a router that enforces endpoints to be paired with an HTTP verb. In modern frameworks, requests for an endpoint with the wrong verb results in an error. If this is something that can't be implemented for your application, we'll discuss workarounds later.

The next protection is verifying the request's origin. This countermeasure is designed to ensure that requests coming into the application originate from a source inside the application (or an otherwise trusted origin with CORS). Correct verb usage is important here simply because if we can assume that only state changing requests are coming from unsafe requests then we only need to validate the origin for unsafe requests. Validating the origin for safe requests is problematic due to the issues we discussed above. If this is required then one solution is to create an exclusion list of known safe URLs, such as the index and/or likely landing pages users will go to on first visit. This will protect against CSRF from external sources but allow the desired behavior of users reaching the site and remaining logged in on first visit.

This protection is not strictly necessary but it adds additional layers and is likely a solution that you'll want to use if depending on CORS. CORS specifically makes a token implementation difficult due to the SOP and token distribution across applications. Origin verification also depends on the presence of HTTP headers which may not be present due to browser differences, browser extensions, or certain request conditions. If request headers are missing then the default option should always be to fail open and rely on a different layered solution for mitigating CSRF.

The protection works by comparing the Origin or Referer header with the Host header from the request. The Origin header is only used in certain circumstances, such as XHR, and may not be present in all requests. It is made up of the full host, including port. The Referer header is significantly more common and is the full URL of the location of the browser when the request was made. Lastly, the Host header is the browser informing the server of the host, including the port if not 80, that it wishes to communicate with. This is needed to support virtual hosts or multiple sites on a single application server. In this case we are using the the Host header as the source of truth for the comparison as we know that the Host header will correspond to the host that we want to enforce origin checking for. The Origin header is checked first and

then falls back to checking the referer. The order doesn't matter and can be swapped with no meaningful difference. Some basic parsing is required and it's important to make sure that only the host and port are compared.

One thing you may be wondering is can the guarantee of comparing the referer be trusted given the possibility and ease of referer spoofing. There are two parts that make this not matter. The first is that the only way to spoof the referer is directly from the victim. As mentioned earlier, this is entirely a client-side attack so the victim's browser would have to be intentionally falsifying the referer to something that would bypass this check. This is something that is unlikely to intentionally happen. The second factor is that these headers, origin and referer, cannot be set by JavaScript as they are protected and will cause an error if the attacker's XSS payload attempts to set them. This restricts any dangerous modification of these headers to the victim's browser and it's probably safe to assume a user would never perform the attack on herself/himself unless it is intentional or their browser/machine is already compromised.

The third, and most commonly used, countermeasure is tokens. Tokens come in a few different varieties but just about every implementation ends up using

synchronizer tokens. For a more complete understanding, you should read about “double submit” tokens and “encrypted” tokens. Double submit tokens should be usable with the following solution, though the solution discussed here is simpler, while encrypted tokens are typically less efficient due to the cost of AES or the otherwise chosen encryption scheme. Instead we’ll use a hybrid of synchronizer and encryption that provides the best of both solutions.

Synchronizer tokens work by synchronizing the server and the browser with a unique token. Requests for safe methods return a token that the browser will send up with each unsafe request, typically in the form body or request header, depending on the type of request. The server validates that the token is authentic and valid before allowing the request to continue; the server will also provide a new token so that tokens are not continually reused or open to replay attacks. This stops CSRF payloads on attacker controlled hosts due to the SOP. The attacker will be unable to get access to the token and insert it into the request because doing so would require that the attacker be able to force the victim to make a request to the remote site and access the response —exactly what the SOP is designed to stop. The only way an attacker would be able to do this is via XSS in the application.

Tokens are composed of four parts that must maintain integrity to be effective. The loss of any of them will significantly weaken the protection of the token. These four parts are a random nonce, user identifier, expiration, and authenticity verification:

1. **The keyspace size** of the nonce is not overly important so long that it's reasonably large enough to ensure a lack of repeats.
2. **The user identifier** can be any value that is unique to the user. In our implementation we'll choose to use the session identifier.
3. **The life span** or expiration time is the length that the token is valid for. Ideally you'll want this to be short enough that a stolen one can't be continually used for long periods of time but long enough that it won't expire while a real user is using it leaving them with a failed request. In most framework implementations this is usually managed by storing the nonce in a session and expiring the token when the session expires. This has the benefit of only one value being valid at any one time which in our case is not true so a discrete expiration is needed. In our case we'll default it to an hour.

4. **There must be some way** to enforce that the token is authentic and hasn't been tampered with or forged. Solutions implemented within frameworks typically get this for free by storing the value in server-side session storage that the user is never able to access. In our case we'll rely on HMAC-SHA256 and provide a signature with the token that we can verify later. This specifically is another reason the nonce size isn't overly important because an attacker would also need to get the key for the HMAC in order to forge a token. If the key is compromised the entire token is and the keyspace is irrelevant. The nonce in this case is just to offer some additional entropy to the token value to minimize the usefulness of a stolen or leaked token. This also is how we avoid the need for session storage that most frameworks depend on while getting significantly better performance than most of the encrypted token solutions.

There are two aspects to the token implementation, the server-side, which handles generation/validation of tokens, and the client-side, which sends the token to the server for requests that require it. We're not going to dig too deep into the server-side implementation other than provide an example for generation and validation because, as I mentioned earlier in the disclaimer, the specifics of dealing with interception of requests/responses differ from

platform to platform so much. Suffice it to say, dig into the middleware APIs for your specific platform and use it to implement something close to the following steps.

1. **Does the session currently have a token?** If not, mark that a token should be generated.
2. **Does the request requires validation?** If so, validate and mark that token has been used.
3. **If validation was required and failed,** then short circuit the response and halt processing. If validation was successful, then process the request.
4. **If a token does not exist** or was marked as used, then generate a new token and add a cookie for it to the response.

It's worth noting that generating a new token every time, even when one is not validated, doesn't add any security or open up a new attack vector. If it's easier, you can build your implementation around generating a new token on each request. You'll get no additional protection but the performance loss should be negligible. The token is added to the cookie here as a method of providing the value to the browser in a location that JavaScript will be

capable of reaching and the browser will automatically persist. It's important to ensure that the HttpOnly flag is never used for this cookie. Doing so will break the implementation, however there is no security issue as the only threat would be from XSS which will already provide the necessary conditions to defeat the CSRF protection anyways.

```
String generateToken(int userId, int key) {
  byte[16] data = random()
  expires = time() + 3600
  raw = hex(data) + "-" + userId + "-" + expires
  signature = hmac(sha256, raw, key)
  return raw + "-" + signature
}
```

Above is an example of a simple generation routine for creating new tokens. This is just the four parts concatenated with hyphens. The HMAC is taken of the first three parts to ensure the authenticity of each of them and is appended to the end to be the fourth. Hyphens were chosen as the delimiter because colon is not a valid character for Cookie version 0 cookies. Using it will force an upgrade to version 1 which may break some compatibility with older browsers.

```
bool validateToken(token, user) {
   parts = token.split("-")
   str = parts[0] + "-" + parts[1] + "-" + parts[2]
   generated = hmac(sha256, str, key)
   if !constantCompare(generated, parts[3]) {
   return false
  }

  if parts[2] < time() {
    return false
  }

  if parts[1] != user {
    return false
  }

  return true
}
```

The next code block is a sample validation routine which takes a token and computes validity. The token is split into its components and the first step is to verify the HMAC by regenerating it from the three components and comparing it with the expected HMAC. Make sure to use a constant time compare here to avoid introducing any timing attacks. If that succeeds, we then verify that the token isn't expired and the user matches. This is

fundamentally all there is to the generation and validation of the tokens. The real challenge is getting the user's browser to automatically submit the token with all necessary requests.

Most modern frameworks take care of this for you when building applications. They have libraries to handle XHR that insert the token into requests and template helpers for including the current token in forms. This is the sort of functionality we're going to replicate without depending on a framework to provide it for us. Instead, as part of our response interception we're going to prepend or append a small snippet of JavaScript into responses. While definitely not spec compliant through rigorous testing, nearly every browser will properly handle the script tag with JavaScript both prepended or appended to opening or closing HTML tag respectively. We'll target specifically responses with an HTML content type, to ensure that we're only injecting this into responses that we won't break, and that we're only modifying responses that are non-XHR. This will avoid us loading our snippet multiple times into a browser or a JSON response.

There are two parts to implement this, one to handle form submissions and another to handle XHR. The first snippet is a minified version of the callback

that is attached to document for the onclick event. It's important to attach this to document rather than attempting to attach to individual forms or clickable elements because it's very possible that the forms or elements won't exist in the DOM at the time of attaching resulting in the callback not firing for it. Instead we attach to the document, which always exists, and delegate to the elements that we care about. We also need to use onclick rather than onsubmit because the onsubmit does not bubble in all browsers and versions, meaning we can't attach to document and get called for the event.

```
var target = evt.target;
while (target !== null) {
  if (target.nodeName === 'A' || target.nodeName === 'INPUT' ||
target.nodeName === 'BUTTON') {
    break;
  }

  target = target.parentNode;
}
```

```
// We didn't find any of the delegates, bail out
if (target === null) {
  return;
}
```

This first section grabs the target element of the fired event. This is the element that was clicked on by the user. Due to the DOM's tree structure and event bubbling system, this element might not be the one that we're interested in. Instead, we must walk up the DOM looking for an element that is something that could submit a form; in this case an a, input, or button tag. If we get to the top of the DOM before discovering one, then we bail out because it was a click event on an element that wasn't submitting a form.

```
// If it's an input element make sure it's of type submit
var type = target.getAttribute('type');
if (target.nodeName === 'INPUT' && (type === null ||
!type.match(/^submit$/i))) {
  return;
}
```

```
// Walk up the DOM to find the form
var form;
for (var node = target; node !== null; node = node.parentNode) {
  if (node.nodeName === 'FORM') {
    form = node;
    break;
  }
}
```

```
if (form === undefined) {
  return;
}
```

Next we check if the tag is an input tag. If it is, then we want to make sure it's a submit button. Otherwise it's not submitting a form—it's just causing the browser to focus on the element. Once we've identified that the target is causing a submit to occur, then continue to walk up the DOM looking for a form tag. If we reached the top of the DOM but didn't find a form tag, then the element is not submitted unless it's using XHR which will be handled by the XHR relevant code section of this approach.

```
var token = form.querySelector('input[name="csrf_token"]');

var tokenValue = getCookieValue('CSRF-TOKEN');
if (token !== undefined && token !== null) {
  if (token.value !== tokenValue) {
    token.value = tokenValue;
  }
  return;
}
```

```
var newToken = document.createElement('input');
newToken.setAttribute('type', 'hidden');
newToken.setAttribute('name', 'csrf_token');
newToken.setAttribute('value', tokenValue);
form.appendChild(newToken);
```

Once the form is found, the only step left is to add the token into the form as a hidden input element. The first step is to check if the element already exists from a previous submit. If it does, then check the value and change it if necessary. If not, then create a new element and append it into the form. Due to the way bubbling works, this handler fires before the form is submitted and adds the element into the form before the handler returns, causing the browser to submit the request with the new element added to the form and subsequently adding the token to the body of the request.

For non-form based requests, a way to get the token into XHR requests is required. Most libraries that provide abstractions around this, including jQuery, make this significantly easier because they provide callbacks that can modify a request as various points of the request being made allowing for this to be added. Unfortunately we can't assume a specific library will be present and will need to create our own hooks for the standard XHR API. To do this

we'll wrap and monkey patch the object itself to add the additional functionality.

```
XMLHttpRequest.prototype._send = XMLHttpRequest.prototype.send;
XMLHttpRequest.prototype.send = function() {
  if (!this.isRequestHeaderSet('X-Requested-With')) {
    this.setRequestHeader('X-Requested-With', 'XMLHttpRequest');
  }
  var tokenValue = getCookieValue('CSRF-TOKEN');
  if (tokenValue !== null) {
    this.setRequestHeader('X-CSRF-Header', tokenValue);
  }
  this._send.apply(this, arguments);
};
```

By taking advantage of JavaScript's prototypal inheritance and dynamic nature, we save a copy of the original send method onto the object so that we can keep a reference to it to use later. We then create a new function that gets attached to the send prototype that pulls the token from the cookie and adds a header to the request with the value. The real send method is called via the saved reference with the original arguments passing them along to work as intended. As far as the code in browser is concerned the API hasn't changed

and the XHR object is not different but we've now forced all requests to submit a CSRF token with them in the header that the server can read.

One specific note here about this implementation is that it only works back to Internet Explorer (IE) 8 due to prototype support and XHR availability. XHR was introduced to Internet Explorer 7 but proper prototype support didn't exist, allowing the functionality to be patched by this solution. The form based solution should be usable by at least IE6. There may be a way to add additional support for older IE versions via custom ActiveX controls but that falls outside the scope of this post. Another solution for dealing with the lack of support is to simply not perform CSRF checking for older Internet Explorer browsers identifiable via the User Agent. While it's true these may not always be present and can be falsified, doing so would require the victim to actively be subverting the security or already be compromised via malware or XSS. All other browsers seem to have good support as shown by the below chart. The above code may be supported by browsers older than the ones listed below, however, finding copies for testing proved difficult and this covered the vast majority of what most users would have access to.

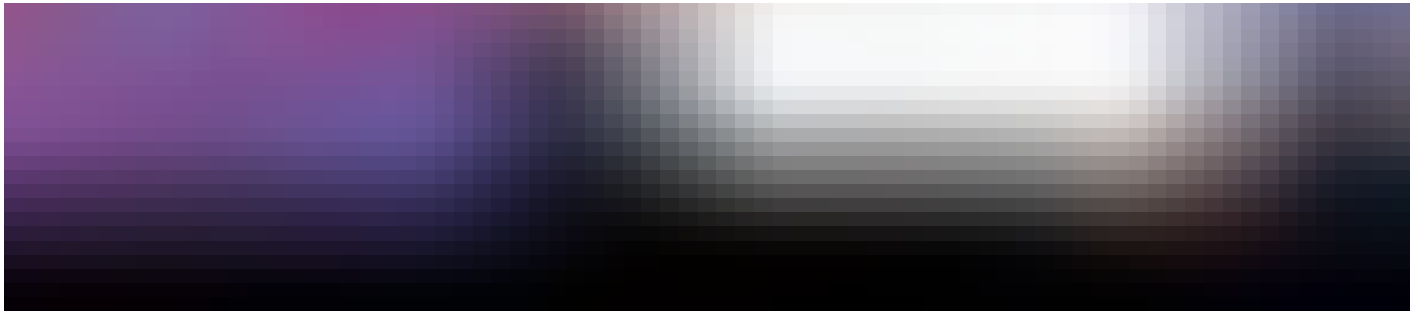
The Future

Now that we've covered building a solution that works for ancient to modern browsers based on current best practices, it's time to look at a new solution that may be the final nail in the coffin for most cases of CSRF. This comes in the form of an extension to cookies called Same-Site that adds support for origin checking to cookies. Same-Site largely replaces the need for synchronizer tokens by allowing the browser to restrict cookies sent with only requests that originate from a host matching the domain. There are two variations, strict and lax enforcement. Strict enforcement enforces this check for safe and unsafe requests while lax only supports the check for unsafe requests. The most likely configuration most applications will require will be

lax because protecting safe requests will not allow the session cookie to be sent with the original GET request to the site.

At the time of writing, browser support is pretty minimal with mostly Chrome supporting the feature. The table below outlines the support as taken from https://www.caniuse.com. However, as an extension, Same-Site does not break compatibility of cookies in older browsers that do not support it. Older browsers will degrade gracefully and simply ignore the additional field treating it as it if doesn't exist.

As of the time of writing, Same-Site is still in draft form and I am unaware of any cookie libraries with support out of the box for this feature. Until it becomes stabilized and accepted, the feature can be added in the same manner as we added token support by rewriting cookies in the responses going down to the client. It could be used in conjunction with a synchronizer token to support both newer and older browsers. The one drawback to the Same-Site approach is the lack of CORS support. As of the time of writing the spec makes no mention of adding support for whitelisting specific origins as safe to send the cookies with. This would break CORS requests that depend on cookies that provide state information to the server. A potential workaround for this would be drop a second cookie used only for the external sites that don't use Same-Site and perform origin verification as discussed above.



Content originally presented at DEF CON 25 in Las Vegas, NV

Csrf | Web Security | Hacking | Defcon | Owasp

Like what you read? Give Joe Rozner a round of applause.

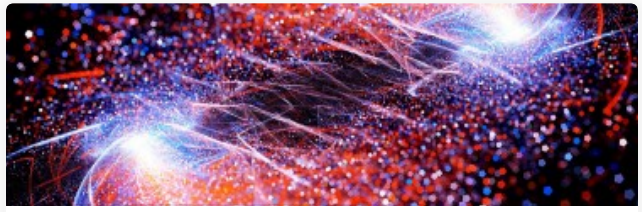
From a quick cheer to a standing ovation, clap to show how much you enjoyed this story.

32

Joe Rozner

Startups, infosec, snowboarding. I build shit in Los Angeles

Follow



More from Joe Rozner

Holiday Hack Challenge 2015 Complete Writeup

Joe Rozner
11 min read

6

Also tagged Hacking

Quantum computers could crack Bitcoin by 2027 — but don't worry!

Nauticus Blockc...
3 min read

1.9K

Related reads

SickOS 1.2 CTF VM — Vulnhub.com

Andrew Hilton
9 min read

6

Responses

Write a response...