

Understanding the Rails Authenticity Token

Asked 15 years, 6 months ago Modified 2 months ago

Viewed 232k times



What is the Authenticity Token in Rails?

1066

ruby-on-rails

ruby

authenticity-token



Share

Improve this question

Follow

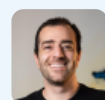
edited Aug 28, 2022 at 20:51



Mateen Ulhaq

27.1k ● 21 ● 117 ● 152

asked Jun 2, 2009 at 20:01



Ricardo Acras

36.2k ● 16 ● 75 ● 112

7 Also see: "Why Does Google Prepend while(1) to their JSON response?" [stackoverflow.com/questions/2669690/...](https://stackoverflow.com/questions/2669690/)
– Chloe Feb 20, 2013 at 3:56

I put this as an edit to the answer as well: a link to the github repo that allows a click-through to reference:

pix.realquadrant.com/authenticity-token – satchel Jul 16, 2021 at 1:50

11 Answers

Sorted by:

Highest score (default)





1538



What happens

When the user views a form to create, update, or destroy a resource, the Rails app creates a random `authenticity_token`, stores this token in the session, and places it in a hidden field in the form. When the user submits the form, Rails looks for the `authenticity_token`, compares it to the one stored in the session, and if they match the request is allowed to continue.

Why it happens

Since the authenticity token is stored in the session, the client cannot know its value. This prevents people from submitting forms to a Rails app without viewing the form within that app itself. Imagine that you are using service A, you logged into the service and everything is OK. Now imagine that you went to use service B, and you saw a picture you like, and pressed on the picture to view a larger size of it. Now, if some evil code was there at service B, it might send a request to service A (which you are logged into), and ask to delete your account, by sending a request to

`http://serviceA.example/close_account`. This is what is known as [CSRF \(Cross Site Request Forgery\)](#).

If service A is using authenticity tokens, this attack vector is no longer applicable, since the request from service B would not contain the correct authenticity token, and will not be allowed to continue.

[API docs](#) describes details about meta tag:

CSRF protection is turned on with the `protect_from_forgery` method, which checks the token and resets the session if it doesn't match what was expected. A call to this method is generated for new Rails applications by default. The token parameter is named `authenticity_token` by default. The name and value of this token must be added to every layout that renders forms by including `csrf_meta_tags` in the HTML head.

Notes

Keep in mind, Rails only verifies not idempotent methods (POST, PUT/PATCH and DELETE). GET request are not checked for authenticity token. Why? because the HTTP specification states that GET requests is idempotent and should **not** create, alter, or destroy resources at the server, and the request should be idempotent (if you run the same command multiple times, you should get the same result every time).

Also the real implementation is a bit more complicated as defined in the beginning, ensuring better security. Rails does not issue the same stored token with every form. Neither does it generate and store a different token every time. It generates and stores a cryptographic hash in a session and issues new cryptographic tokens, which can

be matched against the stored one, every time a page is rendered. See [request_forgery_protection.rb](#).

Lessons

Use `authenticity_token` to protect your not idempotent methods (POST, PUT/PATCH, and DELETE). Also make sure not to allow any GET requests that could potentially modify resources on the server.

Check [the comment by @erturne](#) regarding GET requests being idempotent. He explains it in a better way than I have done here.

Share Improve this answer

Follow

edited Jun 19, 2022 at 11:06



Stephen Ostermiller ♦

25.5k ● 16 ● 94 ● 114

answered Oct 15, 2009 at 11:52



Faisal

19k ● 4 ● 32 ● 34

29 @Faisal, is it possible then, for an attacker to simply read/capture the 'hidden' element of the form for Service A and get that unique token generated for the user - given that they have gotten access to the session started by the user for Service A? – [marcamillion](#) Oct 25, 2010 at 22:18

14 @marcamillion: If somebody hijacked your session at service A, then the authenticity token won't protect you. The hijacker will be able to submit a request and it will be allowed to proceed. – [Faisal](#) Oct 26, 2010 at 7:02

- 14 @zabba: Rails raises an ActionController::InvalidAuthenticityToken exception if a form is submitted without the proper token. You can rescue_from the exception and do whatever processing you want. – [Faisal](#) Jan 31, 2011 at 7:41
-
- 5 re "Also make sure not to make any GET requests that could potentially modify resources on the server." -- this includes not using match() in routes which could potentially allow GET requests to controller actions intended to receive only POSTs – [Steven Soroka](#) Apr 25, 2012 at 18:25
-
- 109 "...and the request should be idempotent (if you run the same command multiple times, you should get the same result every time)." Just a subtle clarification here. Safe means no side-effects. Idempotent means the same side effect no matter how many time a service is called. All safe services are inherently idempotent because there are no side effects. Calling GET on a current-time resource multiple times would return a different result each time, but it's safe (and thus idempotent). – [Eric Turner](#) Aug 18, 2012 at 16:25
-



151



The authenticity token is designed so that you know your form is being submitted from your website. It is generated from the machine on which it runs with a unique identifier that only your machine can know, thus helping prevent cross-site request forgery attacks.

If you are simply having difficulty with rails denying your AJAX script access, you can use

```
<%= form_authenticity_token %>
```

to generate the correct token when you are creating your form.

You can read more about it in the [documentation](#).

Share Improve this answer

edited Jan 31, 2011 at 0:54

Follow



eikes

5,041 ● 2 ● 32 ● 32

answered Jun 2, 2009 at 20:16



Topher Fangio

20.7k ● 15 ● 63 ● 94



What is CSRF?

99

The Authenticity Token is a countermeasure to Cross-Site Request Forgery (CSRF). What is CSRF, you ask?



It's a way that an attacker can potentially hijack sessions without even knowing session tokens.



Scenario:

- Visit your bank's site, log in.
- Then visit the attacker's site (e.g. sponsored ad from an untrusted organization).
- Attacker's page includes form with same fields as the bank's "Transfer Funds" form.
- Attacker knows your account info, and has pre-filled form fields to transfer money from your account to attacker's account.

- Attacker's page includes Javascript that submits form to your bank.
- When form gets submitted, browser includes your cookies for the bank site, including the session token.
- Bank transfers money to attacker's account.
- The form can be in an iframe that is invisible, so you never know the attack occurred.
- This is called Cross-Site Request Forgery (CSRF).

CSRF solution:

- Server can mark forms that came from the server itself
- Every form must contain an additional authentication token as a hidden field.
- Token must be unpredictable (attacker can't guess it).
- Server provides valid token in forms in its pages.
- Server checks token when form posted, rejects forms without proper token.
- Example token: session identifier encrypted with server secret key.
- Rails automatically generates such tokens: see the `authenticity_token` input field in every form.

Follow



Lutz Prechelt

39.2k ● 11 ● 67 ● 89

answered Jun 13, 2012 at 1:54



Rose Perrone

63.4k ● 60 ● 213 ● 248

- 1 Here is a version of this same explanation that is less precise but also less abstract:

stackoverflow.com/a/33829607/2810305 – Lutz Prechelt Nov 20, 2015 at 14:59

I'm not sure but, do modern browsers allow sending not idempotent requests(POST/PUT/DELETE) to another domain? I guess, there must be protection against such in things in browser itself – [divideByZero](#) Sep 22, 2016 at 20:31

@divideByZero (ohh great name!) there is some protection in the form of CORS headers. A site can specify what domains it wishes to receive requests from (and certain browsers/apis are even more restrictive) but I'm not sure when this was adopted or if really old browsers all support it and one probably also wants to have this kind of protection in case the domain left their CORS settings to *.

developer.mozilla.org/en-US/docs/Web/HTTP/CORS

– Peter Gerdes Jun 29, 2021 at 14:14 ✎



71

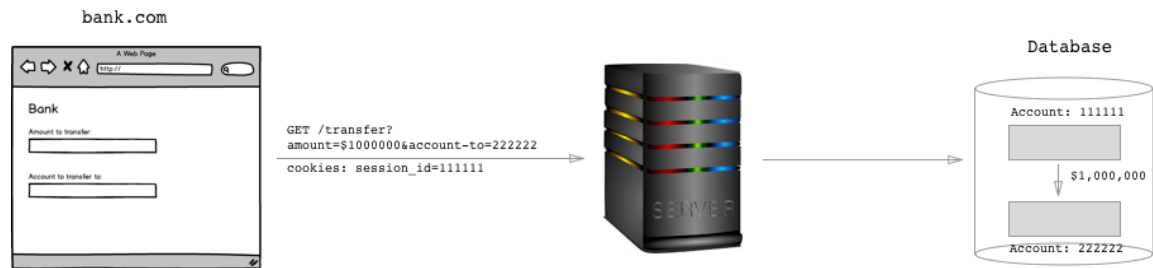


The authenticity token is used to prevent Cross-Site Request Forgery attacks (CSRF). To understand the authenticity token, you must first understand CSRF attacks.

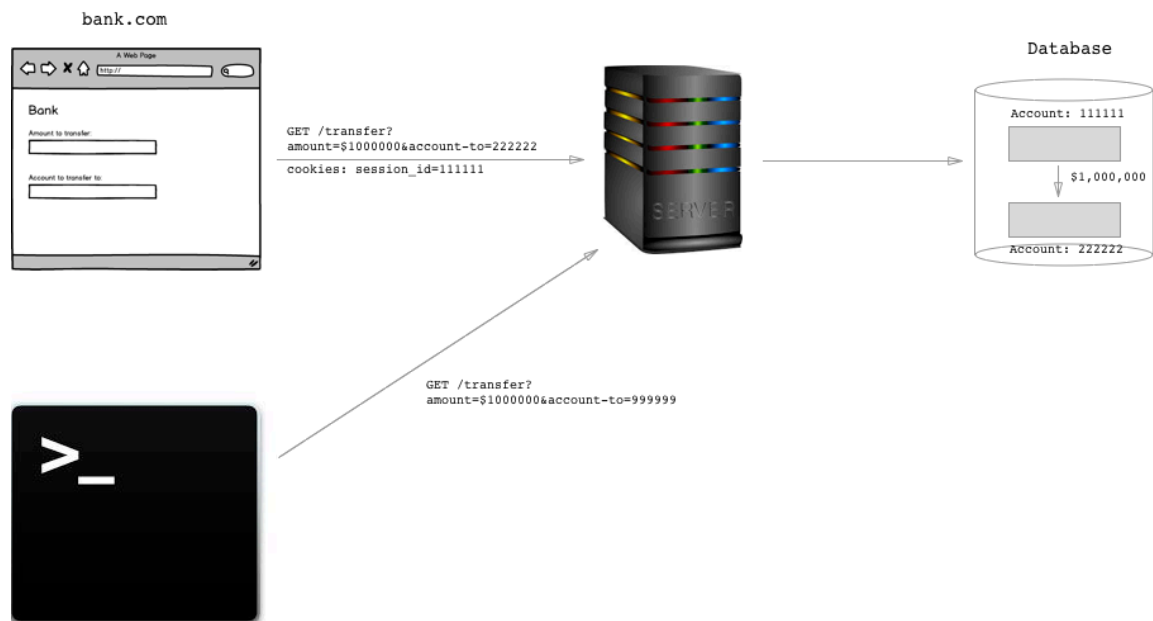
CSRF



Suppose that you are the author of `bank.example`. You have a form on your site that is used to transfer money to a different account with a GET request:



A hacker could just send an HTTP request to the server saying `GET /transfer?amount=$1000000&account-to=999999`, right?



Wrong. The hacker's attack won't work. The server will basically think?

Huh? Who is this guy trying to initiate a transfer. It's not the owner of the account, that's for sure.

How does the server know this? Because there's no `session_id` cookie authenticating the requester.

When you sign in with your username and password, the server sets a `session_id` cookie on your browser. That way, you don't have to authenticate each request with your username and password. When your browser sends the `session_id` cookie, the server knows:

Oh, that's John Doe. He signed in successfully 2.5 minutes ago. He's good to go.

A hacker might think:

Hmm. A normal HTTP request won't work, but if I could get my hand on that `session_id` cookie, I'd be golden.

The user's browser has a bunch of cookies set for the `bank.example` domain. Every time the user makes a request to the `bank.example` domain, all of the cookies get sent along. Including the `session_id` cookie.

So if a hacker could get *you* to make the GET request that transfers money into his account, he'd be successful. How could he trick you into doing so? With Cross Site Request Forgery.

It's pretty simple, actually. The hacker could just get you to visit his website. On his website, he could have the following image tag:

```
To the harmless survey</a>
```

Authenticity Token

When your ApplicationController has this:

```
protect_from_forgery with: :exception
```

This:

```
<%= form_tag do %>  
  Form contents  
<% end %>
```

Is compiled into this:

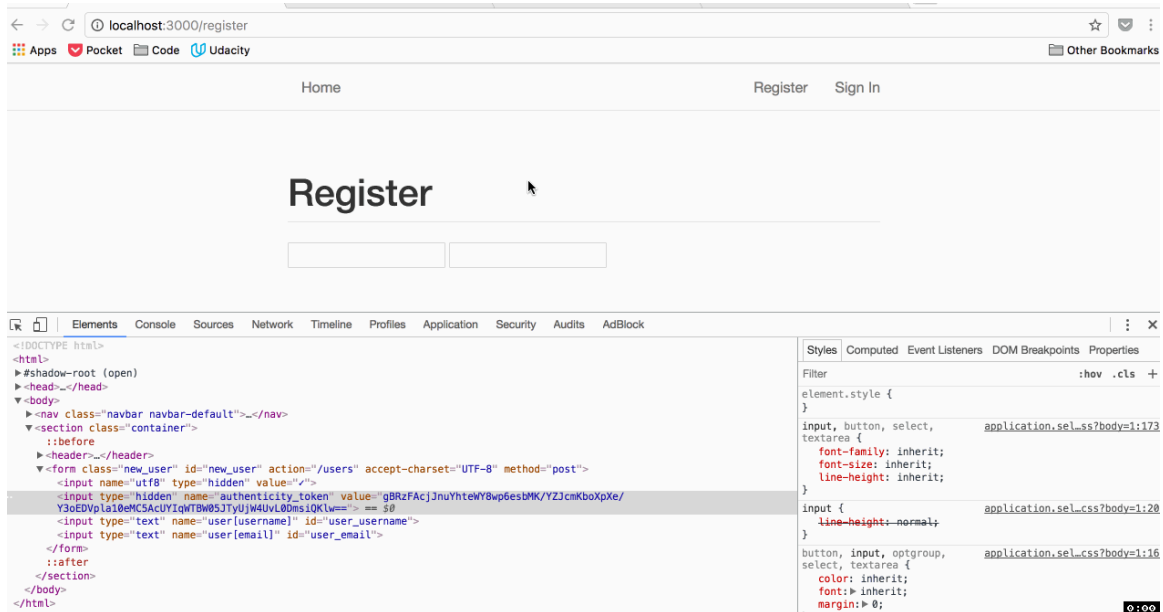
```
<form accept-charset="UTF-8" action="/" method="post">  
  <input name="utf8" type="hidden" value="&#x2713;" />  
  <input name="authenticity_token" type="hidden"  
value="J7CBxfHalt490SHp27hblqK20c9PgWJ108nDHX/8Cts=" />  
  Form contents  
</form>
```

In particular, the following is generated:

```
<input name="authenticity_token" type="hidden"  
value="J7CBxfHalt490SHp27hblqK20c9PgWJ108nDHX/8Cts=" />
```

To protect against CSRF attacks, if Rails doesn't see the authenticity token sent along with a request, it won't consider the request safe.

How is an attacker supposed to know what this token is? A different value is generated randomly each time the form is generated:



A Cross Site Scripting (XSS) attack - that's how. But that's a different vulnerability for a different day.

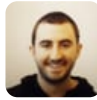
Share Improve this answer

Follow

edited Jun 19, 2022 at 11:08

 **Stephen Ostermiller** ♦
25.5k ● 16 ● 94 ● 114

answered Apr 14, 2017 at 4:33

 **Adam Zerner**
19.1k ● 17 ● 99 ● 168



**Minimal attack example that would be prevented:
CSRF**

49

On my website `evil.example` I convince you to submit the following form:

```
<form action="http://bank.com/transfer" method="post">
  <p><input type="hidden" name="to" value="ciro">
  <p><input type="hidden" name="ammount" value="100"><
  <p><button type="submit">CLICK TO GET PRIZE!!!</butt
</form>
```

If you are logged into your bank through session cookies, then the cookies would be sent and the transfer would be made without you even knowing it.

That is where the CSRF token comes into play:

- with the GET response that returned the form, Rails sends a very long random hidden parameter
- when the browser makes the POST request, it will send the parameter along, and the server will only accept it if it matches

So the form on an authentic browser would look like:

```
<form action="http://bank.com/transfer" method="post">
  <p><input type="hidden" name="authenticity_token"
value="j/DcoJ2VZvr7vdf8CHKsvjdlDbmiiza0b5B8DMALg6s=" >
  <p><input type="hidden" name="to" value="ciro">
  <p><input type="hidden" name="ammount" value="100">
  <p><button type="submit">Send 100$ to Ciro.</button>
</form>
```

Thus, my attack would fail, since it was not sending the `authenticity_token` parameter, and there is no way I

could have guessed it since it is a huge random number.

This prevention technique is called **Synchronizer Token Pattern**.

Same Origin Policy

But what if the attacker made two requests with JavaScript, one to read the token, and the second one to make the transfer?

The synchronizer token pattern alone is not enough to prevent that!

This is where the Same Origin Policy comes to the rescue, as I have explained at:

<https://security.stackexchange.com/questions/8264/why-is-the-same-origin-policy-so-important/72569#72569>

How Rails sends the tokens

Covered at: [Rails: How Does csrf_meta_tag Work?](#)

Basically:

- HTML helpers like `form_tag` add a hidden field to the form for you if it's not a GET form
- AJAX is dealt with automatically by [jquery-ujs](#), which reads the token from the `meta` elements added to your header by `csrf_meta_tags` (present in the default template), and adds it to any request made.

uJS also tries to update the token in forms in outdated cached fragments.

Other prevention approaches

- check if certain headers is present e.g. `X-Requested-With`:
 - [What's the point of the X-Requested-With header?](#)
 - <https://security.stackexchange.com/questions/23371/csrf-protection-with-custom-headers-and-without-validating-token>
 - [Is an X-Requested-With header server check sufficient to protect against a CSRF for an ajax-driven application?](#)
- check the value of the `Origin` header:
<https://security.stackexchange.com/questions/91165/why-is-the-synchronizer-token-pattern-preferred-over-the-origin-header-check-to>
- re-authentication: ask user for password again. This should be done for every critical operation (bank login and money transfers, password changes in most websites), in case your site ever gets XSSed. The downside is that the user has to type the password multiple times, which is tiresome, and increases the chances of keylogging / shoulder surfing.

Follow



Stephen Ostermiller ♦

25.5k ● 16 ● 94 ● 114

answered Nov 12, 2014 at 20:30



Ciro Santilli

OurBigBook.com

380k ● 117 ● 1.3k ● 1.1k

Thank you, but your point about relying on same origin policy to not be able to just read the CSRF token first seems flawed. So first you saying you can POST to a different origin but can't read from it, seems weird but I guess that is correct, but you could inject an image or script tag with a get to the page and link a handler to parse response and get it yes? – [bjm88](#) Feb 28, 2017 at 0:29

- 1 @bjm88 inject the script where? On your site, or on the attacked site? If attacked site, allowing script injection is a well known security flaw, and effectively pawns the website. Every website must fight it through input sanitation. For images, I don't see how they can be used for an attack. On attacking site: you could modify your browser to allow the read, and thus auto pawn yourself at will :-)) but decent browsers prevent it by default, give it a try.
– [Ciro Santilli OurBigBook.com](#) Feb 28, 2017 at 5:42 ✎
-



The `Authenticity Token` is rails' method to [prevent 'cross-site request forgery' \(CSRF or XSRF\) attacks](#).

39



To put it simple, it makes sure that the PUT / POST / DELETE (methods that can modify content) requests to your web app are made from the client's browser and not from a third party (an attacker) that has access to a cookie created on the client side.





Share Improve this answer

answered Jun 2, 2009 at 20:17

Follow



andi

14.5k ● 9 ● 48 ● 46



since `Authenticity Token` is so important, and in Rails 3.0+ you can use

34



```
<%= token_tag nil %>
```

to create



```
<input name="authenticity_token" type="hidden" value="
```

anywhere

Share Improve this answer

answered Aug 1, 2012 at 5:12

Follow



Yuan He

1,163 ● 1 ● 12 ● 14

This was helpful to me. I was actually trying to do `XSS` on the login page, not for nefarious purposes, but to create a new session with pre-filled user name. Now I know I can just use `value="token_value"`. – [Michael come lately](#) Feb 24, 2014 at 16:03



27

Beware the Authenticity Token mechanism can result in race conditions if you have multiple, concurrent requests from the same client. In this situation your server can generate multiple authenticity tokens when there should



only be one, and the client receiving the earlier token in a form will fail on it's next request because the session cookie token has been overwritten. There is a write up on this problem and a not entirely trivial solution here:

<http://www.paulbutcher.com/2007/05/race-conditions-in-rails-sessions-and-how-to-fix-them/>

Share Improve this answer

answered Aug 30, 2011 at 23:52

Follow



jdp

764 ● 8 ● 8



11



Methods Where `authenticity_token` is required

`authenticity_token` is required in case of idempotent methods like post, put and delete, Because Idempotent methods are affecting to data.

Why It is Required

It is required to prevent from evil actions. `authenticity_token` is stored in session, whenever a form is created on web pages for creating or updating to resources then a authenticity token is stored in hidden field and it sent with form on server. Before executing action user sent `authenticity_token` is cross checked with `authenticity_token` stored in session. If

`authenticity_token` is same then process is continue otherwise it does not perform actions.

Share Improve this answer

edited Jan 26, 2016 at 8:31

Follow



Gupta

10.3k ● 5 ● 57 ● 72

answered Feb 26, 2014 at 11:13



uma

2,952 ● 28 ● 20

-
- 3 Actually, isn't it the opposite ? GET is idempotent since its call shouldn't alter the state of the system, where PUT POST and DELETE verbs are NOT idempotent verbs since they alter the system state. I.E : `authenticity_token` is required in case of NOT idempotent methods. – Jean-Théo Jul 10, 2014 at 9:26
-
- 2 @Jean-Daube, uma: idempotent means that if done twice, action only happens once. GET, PUT and DELETE are idempotent: w3.org/Protocols/rfc2616/rfc2616-sec9.html The key property here is not idempotency, but if the method changes or not the data, which is called "Safe method" or not. – [Ciro Santilli](#) [OurBigBook.com](#) Nov 12, 2014 at 20:14
-



What is an `authenticity_token` ?

7

This is a random string used by rails application to make sure that the user is requesting or performing an action from the app page, not from another app or site.



Why is an `authenticity_token` is necessary ?



To protect your app or site from cross-site request forgery.

How to add an authentication_token to a form ?

If you are generating a form using form_for tag an authentication_token is automatically added else you can use `<%= csrf_meta_tag %>`.

Share Improve this answer

Follow

edited Feb 4, 2017 at 15:54



notapatch

7,143 ● 6 ● 43 ● 51

answered Jul 22, 2016 at 2:02



Pradeep Sapkota

2,072 ● 1 ● 18 ● 31



-2



In Ruby on Rails, the authenticity token, often referred to as the CSRF (Cross-Site Request Forgery) token, is a security feature used to protect web applications from certain types of attacks, particularly CSRF attacks.

Share Improve this answer

Follow

answered Sep 21, 2023 at 9:35



Mohsin Amjad

1 ● 2

