Ransacking your password reset tokens

Ransacking your password reset tokens - Author not stated, positive.security.

- Published: date not stated
- Original: <https://positive.security/blog/ransack-data-exfiltration>
- Preserved from: https://positive.security/blog/ransack-data-exfiltration (live) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Ransacking your password reset tokens | Positive Security

-- MARKDOWN --

- Integrating the "Ransack" library in its default configuration into your Ruby on Rails project poses a major security risk that can likely be exploited to extract sensitive information or fully compromise the application
- The issue arises because the recommended way to use the library exposes powerful conditional parameters to end users which allow character by character brute-force of arbitrary attributes on associated database objects
- Other technologies such as Hasura (GraphQL Engine) or older versions of Sequelize (Node.js ORM) are vulnerable to similar attacks when query filters with arbitrary conditional operators are configured
- We have reported internet-exploitable critical and high severity security issues based on this in six different applications. Our research identified several hundred more applications that are potentially affected

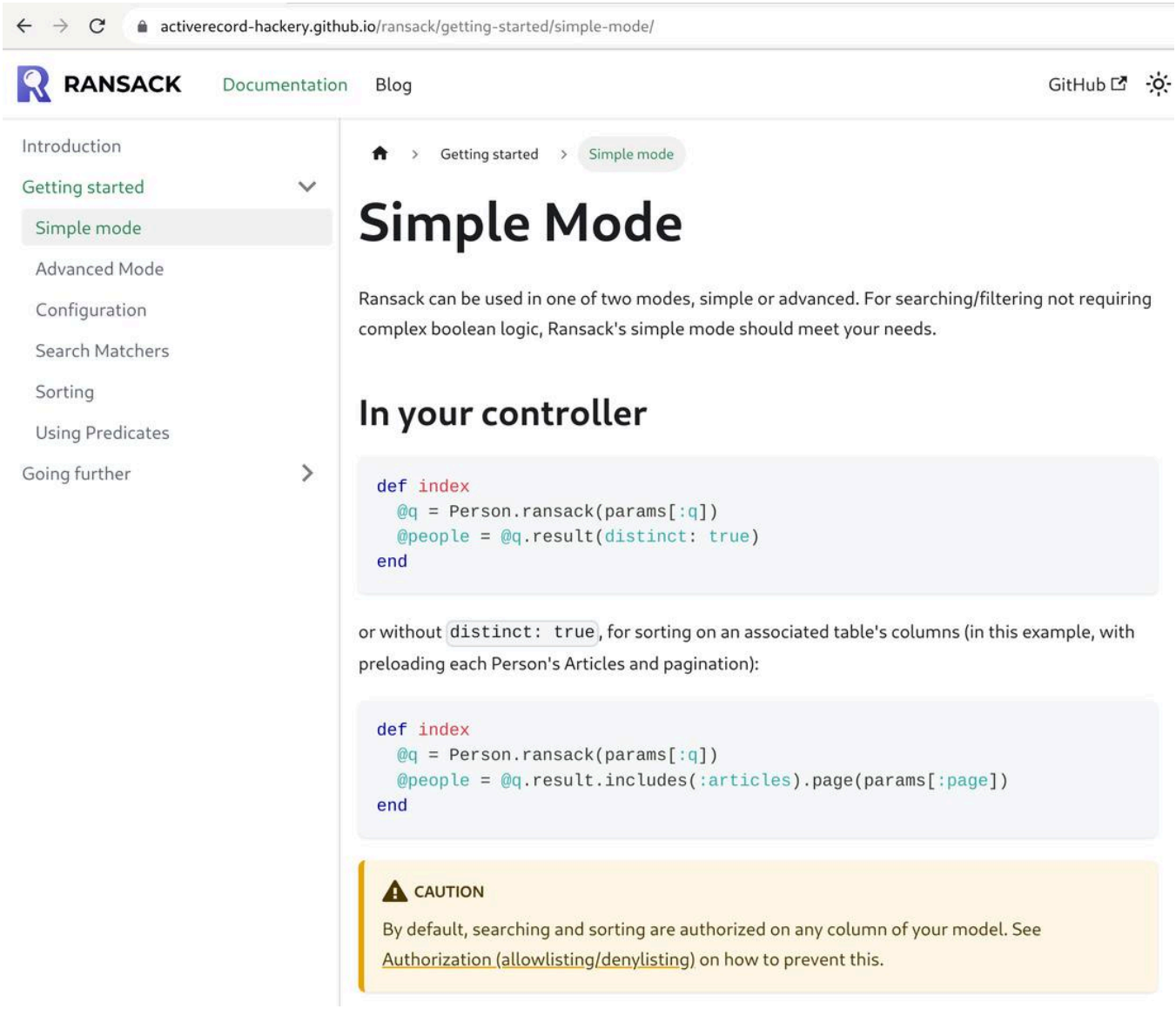
Update 2023-02-10: Ransack 4.0.0 has been released which addresses this issue by changing the default behavior of the library to enforce the use of explicit allow lists for searchable attributes and associations.

Table of Contents

- Searching for vulnerable applications
  - GitHub/searchcode
  - Common Crawl
- Other technologies
  - Hasura (GraphQL)
  - Sequelize (Node.js)
- Responsible disclosure overview

Exploiting Ransack

The popular Ransack Ruby library provides a very powerful feature set around object-based database searching in Rails applications. One of its main appeals is the ease with which it can be utilized to implement public facing search functionality on a website. As is often the case when using a very powerful and complex tool for a rather simple use case, this can lead to problems.



Official Ransack documentation suggests processing query parameters from unrestricted user input. The warning was added to the documentation on 2022-11-03, possibly in reaction to our suggestion in an o

pen GitHub issue

In its default configuration, Ransack will allow for query conditions based on properties of associated database objects. An application is potentially vulnerable if it exposes a search/filtering function which processes an unrestricted query object, usually represented by a Ruby hash constructed via the `q` GET parameter. Search matchers are Ransack's syntax to specify how user provided query values are compared to real records from the database. A straight-forward and less problematic search matcher is `*_eq` (exactly equals), whereas other search matchers like `*_start` (starts with), `*_cont` (contains) or `*_gt` (is greater than) can be abused to exfiltrate potentially sensitive attribute values of associated database objects via character by character brute-force or binary search for numbers. An attacker can perform character by character brute-force by repeatedly guessing a prefix of the value they want to extract using the `*_start` search matcher, starting with a prefix length of just one character, and extending that prefix by an additional character whenever their previous guess was successful. The attacker can tell that a guess was successful whenever their Ransack search returns at least one result. With a case sensitive database collation a single bcrypt password hash for example can be extracted with fewer than 2000 requests on average, all within a few minutes.

Vulnerable example application

Consider a blogging platform implemented in Rails:

It can have different users which authenticate themselves via passwords and have access to an email based account recovery feature:

```
class User < ActiveRecord::Base
 validates :email, :username, presence: true
 attr_accessor :password_hash, :reset_password_token

 has_many posts
end
```

These users can publish posts on the platform (Note how the `User` and `Post` classes are associated via the `has_many` and `belongs_to` keywords, which instruct the framework to set up database relationships in the background):

```
class Post < ActiveRecord::Base
  validates :title, :content, presence: true

  belongs_to :user
end
```

The platform offers a search feature to help look for posts containing specific keywords, conveniently implemented with the `Ransack` library on the backend side:

```
def index
 @q = Post.ransack(params[:q])
 @posts = @q.result(distinct: true)
end
```

The search feature is accessible via a simple HTML form from the frontend, intended to allow finding posts based on their title:

```
<form id="post_search" action="/posts" method="get">

 <label>Search by title:</label>
 <input name="q[title_cont]" type="text" value="">

 <input type="submit" name="submit" value="Search">

</form>
```

Intended usage of the search feature via the HTML form causes the browser to issue a GET request to `/posts?q[title_cont]=hacking`, which is met with an HTML response containing a list of matching posts.

Character by character brute-force

The technique used here is similar to known exploitation techniques for (boolean-based) blind SQL injections. In the fictional example, an attacker can determine the first character of the password reset token of a post's author by submitting a series of search queries until the application returns a non-empty list of posts. Note the use of the `Post` -> `User` association and the `*_start` search matcher.

```
GET /posts?q[user_reset_password_token_start]=0 -> Empty results page
GET /posts?q[user_reset_password_token_start]=1 -> Empty results page
GET /posts?q[user_reset_password_token_start]=2 -> Results in page
```

Once the first character has been recovered, we can start guessing the next character by extending the prefix given in the query filter:

```
GET /posts?q[user_reset_password_token_start]=20 -> Empty results page
GET /posts?q[user_reset_password_token_start]=21 -> Empty results page
GET /posts?q[user_reset_password_token_start]=22 -> Empty results page

[...]

GET /posts?q[user_reset_password_token_start]=2c -> Empty results page
GET /posts?q[user_reset_password_token_start]=2f -> Results in page
```

And so on and so forth until the entire token has been recovered.

Automating this process yields excellent entertainment value, as it is a rare example of an exploit that can sensibly be visualized in true Hollywood hacking fashion:

Hacking in progress...

The speedup gained from this technique over regular brute force is enormous: The worst case for required attempts to brute-force a random string of length `N` generated from a pool of `M` different characters becomes `M * N` instead of `M ^ N`. Taking a randomly generated 32-character hex string as an example (comparable to a UUID, assuming fully random generation), character by character brute force requires a maximum of `512` (`16 * 32`) attempts vs `340282366920938463463374607431768211456` (`16 ^ 32`) for regular brute force. At 1000 attempts per second, this equates to a time of roughly half a second vs ten octillion (`10 ^ 28`) years.

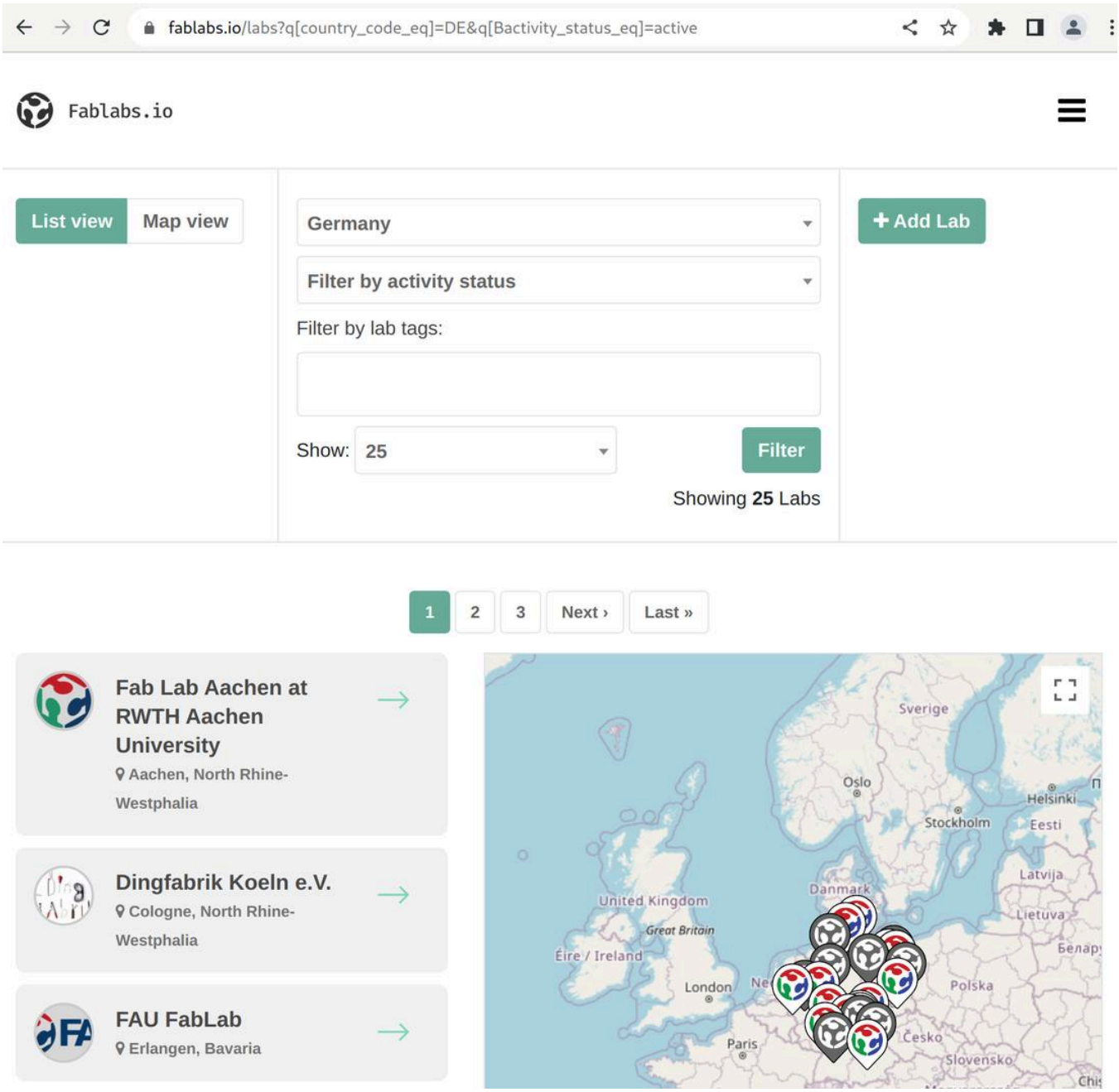
Mitigation

The Ransack library allows restricting which associations and attributes are respected via a whitelist. We recommend setting `ransackable_attributes`, `ransackable_associations` to an empty array in your ORM base class (usually `ApplicationRecord < ActiveRecord::Base`). Next, you can explicitly allow the specific attributes and associations that should be available for Ransack queries by explicitly overriding those variables in the respective ORM subclasses. From version 4.0.0 the library enforces the use of these allow lists (See discussion on GitHub)

Case Study: Becoming superadmin on fablabs.io

The following is a real world example of a full compromise of an application via this issue. The underlying vulnerability has since been fixed.

1. Finding a susceptible endpoint In our search for potentially vulnerable Rails applications in the Common Crawl dataset (see section below) we came across the following URL pointing to a site with fairly high popularity according to our data: `https://www.fablabs.io/labs?q%5Bcountry_code_eq%5D=US`. Closer inspection quickly revealed that the endpoint constituted an unrestricted entry point to `Ransack` queries for finding Fablabs (makerspaces that are part of the "Fablab Network").



Vulnerable fablabs.io lab search page

2. Finding an association chain While exploitation in the fictional example above only requires the use of a single association, different applications might require chaining multiple associations together. As far as we know, database performance is the only limiting factor to how many database relationships can be used as hops to get to a targeted the sensitive attribute. The application hosted on fablabs.io is open source, allowing us to review the relevant model files rather than having to spend time guessing the names of interesting attributes and associations of the `Lab` database object. The 2 important association chains we ended up exploiting to take over a superadmin account are:

- `creator_roles_name` (`Lab` class -> `User` class -> `Role` class -> `name` attribute): Allows targeting a valuable superadmin account for takeover
- `creator_recoveries_key` (`Lab` class -> `User` class -> `Recovery` class -> `key` attribute): Our goal - the password reset token of a superadmin account

A good strategy for guessing association chains in closed source projects is to test them with a condition you expect to be false in conjunction with no or any other filtering rules that you know return results. This is because a condition based on a non-existing association chain is ignored by `Ransack` and does not affect the result set. For example: Adding the parameter `q[creator_recoveries_key_eq]={long_random_string}` would never return any labs (unless you straight up guessed the token on the first try), while the parameter `q[nonExistentAssociation_recoveries_key_eq]={long_random_string}` is ignored and will return a list of labs solely based on the rest of the given query conditions.

3. Brute-forcing a superadmin's password reset token Code review and testing also showed that password reset tokens on fablabs.io did not expire and could even be re-used, so any user who had ever reset their password in the past could be targeted for takeover without having to trigger a new password reset mail to them, which might have raised suspicion. With the two association chains from above, we can craft a query encapsulating the instructions "Show me the list of labs where: 1. The lab creator has a role called 'superadmin', and 2. The lab creator has a password reset token whose key attribute starts with '0'": `/labs?q[creator_roles_name_cont]=superadmin&q[creator_recoveries_key_start]=0` We used this query as the base for a brute force script which ended up extracting the token with fewer than 600 HTTP requests in less than 15 minutes (this could have been even faster, but we rate limited ourselves to not cause too much server load).

4. Finding a less case sensitive password reset token At this point we realized that the case insensitive collation of the database backing this application only allowed us to extract a case insensitive version of the password reset token. The password reset endpoint though required us to submit the token with the correct capitalization. The password reset token was a randomly generated alphanumeric string of length 22. Luckily, character case only matters for the 26 different letters of the English alphabet, but not for the 10 Arabic numerals. The token we extracted was made up of 19 letters and 3 digits, resulting in `524288` (`2 ^ 19`) different case combinations. Brute forcing this many combinations at 1 request per second would take 6 days in the worst case - not unthinkable, but certainly not convenient either.

The total set of characters that the case sensitive alphanumeric string was made up of is 52 letters (26 * 2 for different casing) plus 10 numerals. Therefore, when generating a random string, we can expect around 16% (10 / 62) of the characters to be digits. This translates to an expected value of ~3.5 digits in a random string with length 22, meaning that we got slightly unlucky with our first token only containing 3 digits. If we could instead extract a different token with 4 or more digits, it would cut our character case brute force time in half for each additional digit.

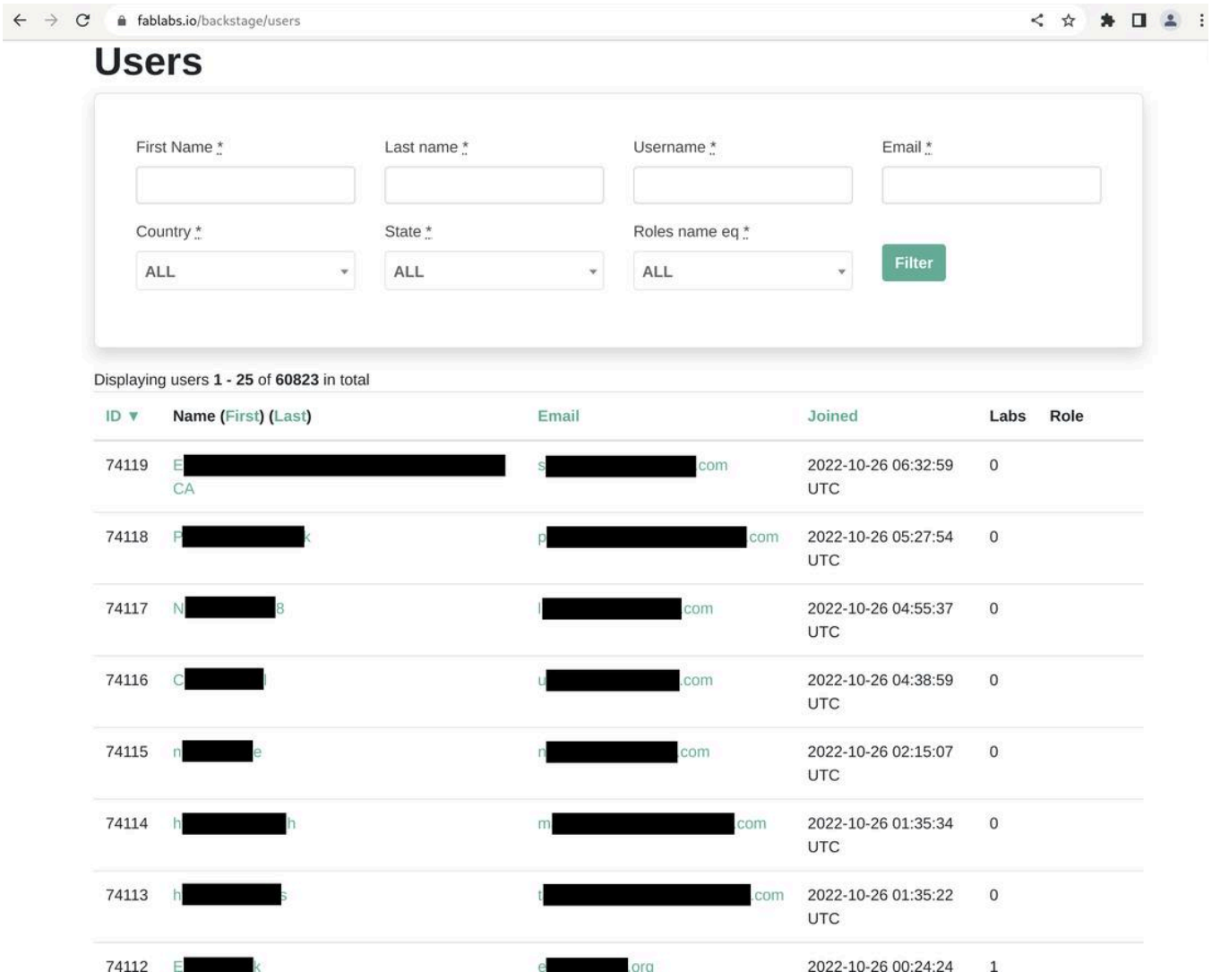
The site had multiple superadmins with old but still valid password reset tokens, so we did not even have to trigger new password reset emails to any of them to get more attempts at extracting a less case sensitive one. While trying to come up with a structured strategy to pre-filter for tokens with a high number of digits, we started casually playing around with the Ransack `*_matches` search matcher, which allows you to craft the equivalent of an SQL `LIKE` query with an arbitrary amount of wildcard characters. We managed to find a token with at least 4 digits by starting out with an empty query into which we repeatedly inserted random `%{digit}%` segments at random locations to find the next bigger valid query. `{digit}` is not a functional wildcard itself and used as a placeholder only in the context of this article, meaning we had to brute-force an actual sequence

of digits with `%` wildcards in between. Before we were able to write a script to do this faster or come up with a more structured approach, we decided to pull the trigger on extracting this token in parallel. To ensure that our character by character brute force targets the correct token, we could just add the `q[creator_recoveries_key_matches]=%25{digit}%25{digit}%25{digit}%25{digit}%25` parameter to our base brute force query from the previous step and rerun the script. Much to our delight, 15 minutes later we had extracted a token which contained not only 4 but 5 digits and deemed it good enough to move on to the next stage.

5. Brute-forcing case distribution Brute forcing the case distribution went very smoothly. We pregenerated all `131072` (`2 ^ 17`) possible combinations, shuffled that list for good luck, and let a script probe the combinations at the `/recoveries/<token>` endpoint. The correct combination was found after just over `60000` requests, beautifully aligned with when you would expect to find it on average. As friendly security researchers we (repeatedly) left a short message in their server logs via an additional GET parameter with our brute force requests to let them know that we will get in touch with them to send over a vulnerability report very soon and would not continue creating server load for them if they banned our one attacking IP address.

More hacking in progress...

6. Resetting the password and gaining access The account recovery page happily let us choose a new password for our victim's account, and we were able to log in as superadmin. We used this to promote one of our personal accounts to superadmin and took some screenshots to showcase the extent of sensitive data we could have stolen and how we could have manipulated any crucial information published on the website.



fablabs.io admin interface: phone numbers, email addresses and password hashes of ~60000 users can be downloaded here

The total turnaround time from first probing of the website to gaining superadmin privileges was just under 24 hours, in which the targeted webserver had to serve ~65000 HTTP requests for us. We spent around 4 hours of human time on the project, during which we also extracted an API token that could be used to download sensitive user data from the fablabs.io API. Half an hour after taking over the superadmin account we sent our vulnerability report to the two people who had been most active on the fablabs.io GitHub project. Thanks to our own superadmin privileges, we could just look up the email addresses of their user accounts on fablabs.io.

Searching for vulnerable applications

When we first encountered this issue during a client engagement in August of 2022, we had the strong suspicion that similar vulnerabilities would be quite prevalent in other Ruby on Rails projects relying on the Ransack library. To test this theory, we needed a structured approach to search the internet for other potentially vulnerable applications.

GitHub/searchcode

A straight-forward way to finding potentially problematic usages of the library is of course to search for a relevant code snippet (i.e. `ransack(params[:q])`) in open source projects. There are some services which let us do this pretty conveniently, such as:

- GitHub (requires login)
- searchcode (covers multiple sources incl. GitHub, GitLab and Bitbucket)

This approach led us to Active Admin, which in turn led us to finding the vulnerability in Pageflow (see disclosure overview below). We ended up investigating Spree Commerce and a large number of its forked versions for some time, but did not find anything wrong with the current state.

Pros (+)

- No or miniscule setup time
- Free

Cons (-)

- Only covers open source projects
- Nontransparent index behavior/reach (We found that repeating the same code search on GitHub at different times during the course of a week might yield different results even when there have not been any changes to the relevant code)
- Sifting through many similar results based on forked repositories can be tedious

Common Crawl

Common Crawl is a very cool project which crawls the internet and publishes massive data dumps of as much of the HTTP(S) web as it can reach roughly every two months. In addition to the raw data dumps (~460 TB uncompressed, mostly HTML), they also regularly provide an index to all 3 billion+ crawled URLs in a columnar format. This ~300GB URL index can be queried via AWS Athena without having to download it.

Pros (+)

- Vast coverage of internet exposed websites, including closed source projects
- Results are immediately available in machine readable (CSV) format, making further filtering easy

Cons (-)

- Cannot find anything that's hidden behind a login
- Setup requires more effort
- Searching does incur some cost

Building an SQL query for AWS Athena Once Athena is set up on Common Crawl data, the following query can be used to retrieve a list of ~700 URLs to potentially vulnerable websites:

```
SELECT url_host_registered_domain, MAX(url)
FROM "ccindex"."ccindex"
WHERE crawl = 'CC-MAIN-2022-40'
 AND subset = 'warc'
 AND url_query LIKE '%^%5D=%' ESCAPE '^'
 AND url_query NOT LIKE '%^_^_in%5D=%' ESCAPE '^'
 AND url_query LIKE '%^%5B%' ESCAPE '^'
 AND (url_query LIKE '%^_eq^%5D=%' ESCAPE '^' OR url_query LIKE '%^_eq^_any^%5D=%' ESCAPE '^' OR url_query LIK
GROUP BY url_host_registered_domain
```

This query is the result of an iterative process where we tried a few slightly different approaches. Some notes on this final version:

- We're only looking for URL encoded versions of `[` (`%5B`) and `]` (`%5D`) here because we saw in an earlier query that the non encoded versions do not return anything related to our search
- The exclusion of `__in]` is to filter out a particularly common false positive related to Wordpress sites
- We do not assume that the relevant GET parameters start with `q[`, but instead included the full list of available search matchers (`*_cont`, `*_in`, `*_eq`, etc.). This gives us some extra results such as applications that call the parameter `search` instead of `q`. The query might look ridiculously inefficient to some, but Athena happily executes it in about 2 minutes for a few cents which is very much acceptable for a one-off query like this
- The results are grouped by `url_host_registered_domain` and we only pick one URL per domain as an example to have a more focused output (if one of the URLs for a domain is a false positive, it seems likely that all of them are)

Target selection With ~700 potentially vulnerable domains, we were not able to manually probe each one to see if they actually have an exploitable Ransack implementation. It also seemed like implementing an accurate automation for further probing would be quite difficult as it would need to be able to distinguish between genuinely sensitive and deliberately public information, while operating on freeform HTML responses of hundreds of different applications in 20+ different languages. Instead, to make sure we wouldn't miss any high profile targets, we wanted to enrich our data with a rough measure for the popularity of each domain. We did this by matching the potentially vulnerable domains to the Majestic Million dataset. For ~150 of out of our ~700 potentially vulnerable websites the host or super domain could also be found in the majestic million dataset, with one hit at a double digit rank and a handful of triple digit ranked domains.

We ended up manually probing ~20 domains, most of which could be shown to have an unrestricted `Ransack` query in the backend by confirming that different attributes and search matchers could be used. To demonstrate exploitable vulnerabilities, an association chain to some truly sensitive data also needed to be found for each application. We were able to do so for 3 applications within a reasonable time frame: openSUSE TSP, fablabs.io and PrepMod (see disclosure overview below).

While the URL index search yielded pretty extensive results for us already, it is likely that many more vulnerable applications could be found by searching the raw HTML from Common Crawl for relevant form input element snippets like `name="q[` or `_cont]"`. Searching through the entirety of their ~460 TB dataset would however require a lot more setup work and computational cost.

Other technologies

The issue of powerful query conditions that can be abused for character by character brute force by end users is not limited to Ransack.

Hasura (GraphQL)

Hasura is a GraphQL server implementation that can provide your application with a lot of (individually generated) boilerplate code. Similarly to the Ransack examples given here, during a 2021 penetration test we were able to fully compromise a Node.js/Hasura application by extracting a session token of an admin. The exploit involved finding a multi-step association chain from a store product to an admin user's session refresh token and character by character brute forcing the token via a ColumnExp expression.

At the time we searched for other potentially vulnerable projects on the internet, but did not find anything interesting.

Sequelize (Node.js)

Sequelize is a library providing an ORM interface to craft SQL queries for TypeScript and Node.js applications. Older versions by default allowed string comparison operators (`operatorAliases`) in query parameters that would commonly be provided by user input. This allowed for character by character brute force in a similar fashion, as well as a host of other potential problems. The feature was disabled by default in 2017, and removed entirely in 2019.

Responsible disclosure overview

We reported vulnerabilities based on this issue in the following projects:

CodeOcean - Educational execution and development environment for practical programming exercises, 100k+ users

- Impact: Full application compromise via extracted authentication token
- Initial report: 2022-08-20
- Communication: Found during a commissioned penetration test. Issue was identified through monitoring and fixed a few hours after exploitation before we were able to formally report it
- Fixed: 2022-08-20

Pageflow - CMS for multimedia storytelling and interactive web stories, 17k+ users

- Impact: Extracting email address and password hash (all lowercase) of any user
- Initial report: 2022-09-11
- Communication: Response in one day through contact email listed on website

- Fixed: 2022-09-12
- Security advisory published on 2022-09-14 (GHSA-wrrw-crp8-979q)

Active Admin - Ruby on Rails framework for creating website administration backends

- Impact: Extracting user data like email address and password hash (all lowercase) of any user. Seems less critical in most deployments since access to the endpoint should be reserved to admins, but was cause of the vulnerability in Pageflow (see above)
- Initial report: 2022-09-13
- Communication: Reported issue to official security contact at Tidelift and followed up two more times. Still no response at time of writing

openSUSE Travel Support Program - Application to manage the requests and reimbursements from travel help programs of free software organizations

- Impact: Extracting bank details, email address and password hash (all lowercase) of any user
- Initial report: 2022-10-24
- Communication: Official project and organization contacts never responded via email. Another GH user actively working on the project responded within one day and promptly fixed the issue
- Fixed: 2022-11-29
- Security advisory published on 2023-01-09 (GHSA-2wwv-c6xh-cf68)

fablabs.io - Organizational platform for FabLab Makerspaces, 60k+ users

- Impact: Full application compromise via extracted password reset token
- Initial report: 2022-10-26
- Communication: No contact listed on GH, contact email on website defunct, response only after multiple communication attempts via email and LinkedIn
- Fixed: 2022-11-30

PrepMod - Searching and booking vaccination appointments through different US state level health departments (e.g. Minnesota Department of Health)

- Impact: Extracting name, race, email address, phone number and appointment details of any patient
- Initial report: 2022-11-01
- Communication: Software vendor not reachable via official contact email, contact form or LinkedIn. After reporting the issue to one of their clients, a fix was deployed to the tested domains within a week
- Fixed: ~ 2022-12-21

-- MARKDOWN --

Follow us on Mastodon (@positive_sec) to keep up to date with our posts.