So I reverse engineered two dating apps...

And I got a zero-click session hijacking and other fun vulnerabilities

May 02, 2020 (Last Modified: May 05, 2020) security

PAGE CONTENT

In this post I show some of my findings during the reverse engineering of the apps *Coffee Meets Bagel* and *The League*. I have identified several critical vulnerabilities during the research, all of which have been reported to the affected vendors.

Introduction

In these unprecedented times, more and more people are escaping into the digital world to cope with social distancing. During these times cyber-security is more important than ever. From my limited experience, very few startups are mindful of security best practices. The companies responsible for a large range of dating apps are no exception. I started this little research project to see how secure the latest dating apps are.

Responsible disclosure

All high severity vulnerabilities disclosed in this post have been reported to the vendors. By the time of publishing, corresponding patches have been released, and I have independently verified that the fixes are in place.

I will not provide details into their proprietary APIs unless relevant.

The candidate apps

I picked two popular dating apps available on iOS and Android.

Coffee Meets Bagel

Coffee Meets Bagel or CMB for short, launched in 2012, is known for showing users a limited number of matches every day. They have been hacked once in 2019, with 6 million accounts stolen. Leaked information included a full name, email address, age, registration date, and gender. CMB has been gaining popularity in recent years, and makes a good candidate for this project.

The League

The tagline for *The League* app is "date intelligently". Launched some time in 2015, it is a members-only app, with acceptance and matches based on LinkedIn and Facebook profiles. The app is more expensive and selective than its alternatives, but is security on par with the price?

Testing methodologies

I use a combination of static analysis and dynamic analysis for reverse engineering. For static analysis I decompile the APK, mostly using *apktool* and *jadx*. For dynamic analysis I use an MITM network proxy with SSL proxy capabilities.

The majority of the testing is done inside a rooted Android emulator running Android 8 Oreo. Tests that require more capabilities are done on a real Android device running Lineage OS 16 (based on Android Pie), rooted with Magisk.

Findings on CMB

Both apps have a lot of trackers and telemetry, but I guess that is just the state of the industry. CMB has more trackers than *The League* though.

See who disliked you on CMB with this one simple trick

The API includes a `pair_action` field in every `bagel` object and it is an enum with the following values:

```
ACTION_NOT_CHECKED = 0
ACTION_LIKED = 1
ACTION_PASSED = 2
ACTION_CHECKED = 3
```

There exists an API that given a bagel ID returns the bagel object. The bagel ID is shown in the batch of daily bagels. So if you want to see if someone has rejected you, you could try the following:

- After the call to get daily suggested bagels, save the bagel ID of someone you liked.
- Wait until your profile is shown to your match.
- Call the API to get bagel, using the bagel ID.
- If you see the `pair_action` changes from `0` (unseen) to `2`.

This is a harmless vulnerability, but it is funny that this field is exposed through the API but is not available through the app.

Geolocation data leak, but not really

CMB shows other users' longitude and latitude up to 2 decimal places, which is around 1 square mile. Fortunately this information **is not real-time**, and it is only updated when a user chooses to update their location. (I imagine this must be used by the app for matchmaking purposes. I have not verified this hypothesis.)

However, I do think this field could be hidden from the response.

Findings on The League

Client-side generated authentication tokens

The League does something pretty unusual in their login flow:

- The app sends a POST request with user's phone number
- User receives the one-time password (OTP) via SMS and punches it into the app
- The app sends a POST request with the phone number, the OTP, and a `bearer` value, which is a 16 byte UUID.
- Server receives the request, and if the OTP matches the phone number, the `bearer` becomes user's login token.
- From this point, subsequent requests to endpoints that require authentication would include the header `authorization: bearer sms:{the_uuid}`

The UUID that becomes the `bearer` is entirely client-side generated. Worse, the server does not verify that the `bearer` value is an actual valid UUID. It might cause collisions and other problems.

I recommend changing the login model so the bearer token is generated server-side and sent to the client once the server receives the correct OTP from the client.

Phone number leak through an unauthenticated API

In *The League* there exists an unauthenticated API that accepts a phone number as query parameter. The API leaks information in HTTP response code. When the phone number is registered, it returns `200 OK`, but when the number is not registered, it returns `418 I'm a teapot`. It could be abused in a few ways, e.g. mapping all the numbers under an area code to see who is on *The League* and who is not. Or it can lead to potential embarrassment when your coworker finds out you are on the app.

This has since been fixed when the bug was reported to the vendor. Now the API simply returns 200 for all requests.

LinkedIn job details

The League integrates with LinkedIn to show a user's employer and job title on their profile. Sometimes it goes a bit overboard gathering information. The profile API returns detailed job position information scraped from LinkedIn, like the start year, end year, etc.

```
"title": "Something something intern",
"end_year": 2014,
"end_month": 8,
"is_current": false,
"start_year": 2014,
"start_month": 6,
"identity_provider": "linkedin"
```

While the app does ask user permission to read LinkedIn profile, the user probably does not expect the detailed position information to be included in their profile for everyone else to view. I do not think that kind of information is necessary for the app to function, and it can probably be excluded from profile data.

Picture and video leak through misconfigured S3 buckets

Typically for pictures or other asserts, some type of Access Control List (ACL) would be in place. For assets such as profile pictures, a common way of implementing ACL would be:

- User uploads an image
- Server generates a random key
  - It is essential that the key has high entropy and is difficult to guess
- Server uploads the image under the key to the storage service

- Server puts the URL to the image with the key in user's profile

The key would serve as a "password" to access the file, and the password would only be given users who need access to the image. In the case of a dating app, it will be whoever the profile is presented to.

I have identified several misconfigured S3 buckets on *The League* during the research. **All pictures and videos are accidentally made public**, with metadata such as which user uploaded them and when. Normally the app would get the images through Cloudfront, a CDN on top of the S3 buckets. Unfortunately the underlying S3 buckets are severely misconfigured.

The photo and video buckets are unauthenticated and **even allow ListObjects**. The keys in the bucket are in the format `{profile_uuid}-{timestamp}-{filename}`, so there is plenty of abuse potential. For one, a competitor can dump the keys and find out the exact number of users on the app. Or an attacker can see all the photos uploaded by a victim once the profile UUID is known.

> Side note: As far as I can tell, the profile UUID is randomly generated server-side when the profile is created. So that part is unlikely to be so easy to guess. The filename is controlled by the client; the server accepts any filename. However in the client app it is hardcoded to `upload.jpg`.

The vendor has since disabled public ListObjects. However, I still think there should be some randomness in the key. A timestamp cannot serve as secret.

IP doxing through link previews

Link preview is one thing that is hard to get right in a lot of messaging apps. There are typically three strategies for link previews:

- Sender-side link previews
  - When a message is composed, the link preview is generated under the sender's context.
  - The sent message will include the preview.
  - Recipient sees the preview generated by sender.
  - Note that this method could allow sender to craft fake previews.
  - This strategy is typically implemented in end-to-end encrypted messaging systems such as *Signal*.
- Recipient-side link previews
  - When a message is sent, only the link is included.
  - Recipient will fetch the link client-side and the app will show the preview.
  - Note that this method could leak some context about the recipient, such as the IP address, user agent, and other headers.
- Server-side link previews
  - In this strategy the preview is generated server-side.
  - Sender simply sends the link. Recipient gets the preview from server.
  - Server can fetch the link for preview either on message sent, or when message is opened.
  - An attacker controlled external server could return a different response if the request comes from the link preview server, thus sending a fake preview to recipient.

The League uses **recipient-side link previews**. When a message includes a link to an external image, the link is fetched on user's device when the message is viewed. This would effectively allow a malicious sender to send an external image URL pointing to an attacker controlled server, obtaining recipient's IP address when the message is opened.

A better solution might be just to attach the image in the message when it is sent (sender-side preview), or have the server fetch the image and put it in the message (server-side preview). Server-side previews will allow additional anti-abuse scanning. It might be a better option, but still not bulletproof.

Zero-click session hijacking through chat

The app will sometimes attach the authorization header to requests that do not require authentication, such as Cloudfront GET requests. It will also gladly give out the bearer token in requests to **external domains** in some cases.

One of those cases is the external image link in chat messages. We already know the app uses recipient-side link previews, and the request to the external resource is executed in recipient's context. The authorization header is included in the GET request to the external image URL. So the bearer token gets leaked to the external domain. When a malicious sender sends an image link pointing to an attacker controlled server, not only do they get recipient's IP, but they also get their victim's session token. This is a critical vulnerability as it allows session hijacking.

Note that unlike phishing, this attack **does not require the victim to click on the link**. When the message containing the image link is viewed, the app automatically leaks the session token to the attacker.

It seems to be a bug related to the reuse of a global OkHttp client object. It would be best if the developers make sure the app only attaches authorization bearer header in requests to *The League* API.

Conclusions

I did not find any particularly interesting vulnerabilities in CMB, but that does not mean CMB is more secure than *The League*. (See *Limitations and future research*). I did find a few security issues in *The League*, none of which were particularly difficult to discover or exploit. I guess it really is the common mistakes people make over and over. OWASP top ten anyone?

As consumers we need to be mindful with which companies we trust with our data.

Vendor's response

I did receive a prompt response from *The League* after sending them an email alerting them of the findings. The S3 bucket configuration was swiftly fixed. The other vulnerabilities were patched or at least mitigated within a few weeks.

I think startups could certainly offer bug bounties. It is a nice gesture, and more importantly, platforms like HackerOne provide researchers a legal path to the disclosure of vulnerabilities. Unfortunately neither of the two apps in the post has such program.

Limitations and future research

This research is not comprehensive, and **should not be seen as a security audit**. Most of the tests in this post were done on the network IO level, and very little on the client itself. Notably, I did not test for remote code execution or buffer overflow type vulnerabilities. In future research, we could look more into the security of the client applications.

This could be done with dynamic analysis, using methods such as:

- Hooking Android APIs to see what functions the app is calling (with *Xposed framework*)
- Attaching a debugger to the app (Android Studio can do this)
- Scanning memory for interesting stuff (using tools such as *scanmem*)

Appendix

Special thanks to…

- XDA-Developers and the open source Android community, for the amazing tools.
- David Siu, VP of Engineering at *The League*, for the quick turnaround when the vulnerabilities were reported.

References

1. **Coffee Meets Bagel**. coffeemeetsbagel.com.
2. Williams, Chris. **620 million accounts stolen from 16 hacked websites now for sale on dark web, seller boasts**. *The Register*.
3. Ha, Anthony. **The League brings its picky dating app to Android**. *Tech Crunch*.
4. **Xposed Framework Hub**. *XDA-Developers*.
5. **Hacker One**. hackerone.com.
6. **Network security configuration**. *Android Developers*.

APPS RE