Reverse Engineering the Instagram API to Create a Messaging Bot
by Vala Bahrami

Motivations

I frequently spend time on Instagram browsing posts, sending them to friends for laughs as many people do. I had recently encountered an Instagram account bot which allowed individuals to send them posts, and the bot would respond by sending them the post's uploaded media, be it a photo or a video. Essentially, the bot allowed users to save media from other posts to their own phones without needing to screen record or screenshot. I found it very interesting that the creators of this bot account were able to create such a tool despite Instagram providing no such API to developers, to discourage the creation of robotic accounts which may damage the reputation of their platform.

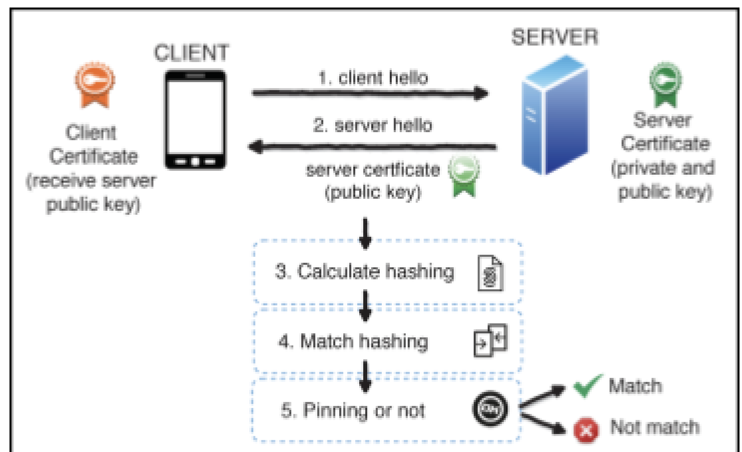
```python
# Creates a new message thread with the provided username or user ID.
def createMsgThread(self, text, username=None, userid=None):
    if not userid and username:
        recipient = self.getUserID(username)
    elif userid:
        recipient = userid
    else:
        raise Exception("Neither username nor userid provided.")
    if not recipient.isdigit():
        return print('Unable to use user id.')
    payload = {
     "recipient_users": f"[[{recipient}]]",
     "action": "send_item",
     "is_shh_mode": "0",
     "send_attribution": "message_button",
     'client_context': random.randint(111111, 999999),
     "text": text,
     "device_id": self.device_id,
     "_uuid": self.uuid
    }
    return self.send('/direct_v2/threads/broadcast/text/', data=self.sign(payload))
```

A snippet of the message sending function from the finished product

I decided to attempt to create my own botted account both as a way of having a private tool which would do this for me quickly (the accounts I had originally stumbled upon had many users and as such took a while to send you the media you wanted), and as a learning experience into the inner workings of a massive platform which I believed would be both interesting and valuable in the future.

Unexpected Barriers

I knew from past experiences that the best way to get started would be to run all the Instagram mobile app's traffic through a proxy. If I wanted to recreate functionality only available on the app elsewhere, such as in a Python program, I had to see how the app was performing these actions itself. I had done this previously to get information from websites, namely from YouTube, so I could build a tool that downloaded YouTube videos. In that situation, I was able to use my browser's developer tools window to monitor network traffic. It would be no such walk in the park with the Instagram app. For starters, I tried to route my iPhone's traffic through a proxy on my laptop. Upon opening and using the Instagram app, my proxy would log requests to Instagram, but none of the requests would be successful, nor was the data being transmitted even readable. A quick Google search later revealed the culprit: "SSL/Certificate pinning", the process of baking into an application the only acceptable certificate (or set of certificates) for a valid connection to be made to a remote server, in this case Instagram's servers. In short, the Instagram app would only allow requests to be



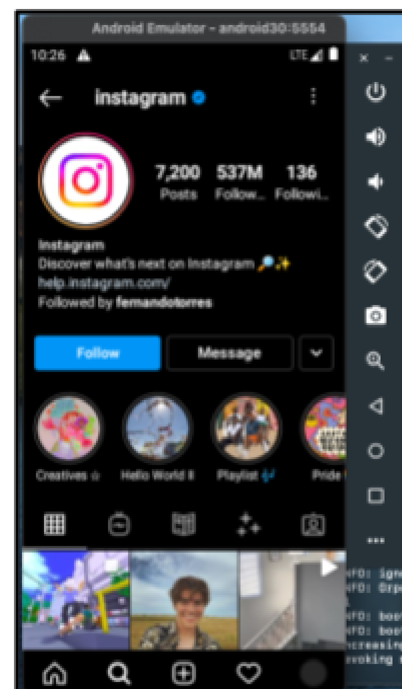
SSL Pinning implementation by means of certificate hashing
By Apurv Pandey (mailapurvpandey.medium.com)

made to the Instagram server if the traffic was in no way being spied on by an outsider (which could only have had a non-Instagram certificate).

Certificate Pinning is a neat security mechanism to prevent “Man in The Middle” attacks, which can be done with dubious certificate authorities installed on the victim’s device, but in my case it only impeded my research. If I myself was the one trying to view my own personal traffic, but the app had been designed to keep anyone from reading traffic as a way to keep attackers out, how could I still find a way to view the traffic for research and reverse engineering? The answer was… I had no idea. Again resorting to Google as one does, I found that there were two solutions. The first: A patched Instagram APK. An APK is an Android app, and some other clever hackers and researchers had patched the Instagram Android app to remove the certificate pinning feature. The problem? Well, there were two: The first was that the only publicly available patched APKs that I could find were of very old Instagram app versions- so old in fact, that they lacked many of the key features that I was trying to reverse engineer. The second issue was that I didn’t even have an Android device! Even if I did find a new version with a patched APK, it would be worthless with no Android device to install it on. But- there had to be a reason that no one else was making these patched APKs anymore. I dug a little deeper and found out that Facebook (now Meta) actually had begun providing security researchers access to a setting on their registered accounts which allowed them to turn off certificate pinning from within the app - as easy as that! *Too* easy… what was the catch?

The Catch

This feature was only available on Android devices. Oh come on! It seems as though all of the security researching tools, both official and those created by independent community members were catered to the Android researching audience, which makes sense, as I suppose it is easier to tinker with unofficial content on Android than iOS. Now, what was I to do? From my previous projects I remembered working with an IDE called Android Studio, and I remembered that Android Studio allowed you to emulate an Android device for testing APKs. I quickly downloaded the emulator command-line portion of Android Studio and after a long while of figuring out which system images were compatible with my MacBook’s processor, I had an emulated version of Android up and running on my laptop. I then registered my Instagram test account as a researcher account and was able to disable certificate pinning. Now, I had everything I needed to begin reverse engineering.



Android emulator running on macOS

Finding Endpoints

In order to be able to recreate the functionality of the Instagram app in my bot, such as sending messages, I needed to see exactly what endpoints the app used to communicate with the Instagram servers. For example, in order to send a text direct message, the app would send an HTTP POST request to the URL:

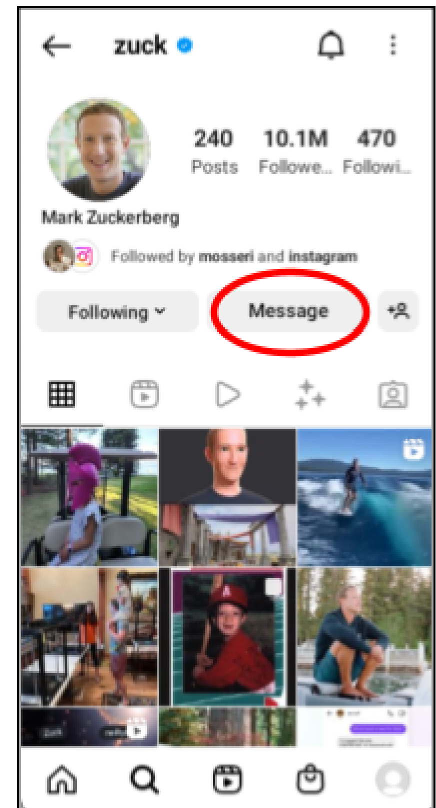
```
https://i.instagram.com/api/v1/direct_v2/threads/broadcast/text/
```

Inside the request’s body would be a JSON payload containing the desired recipient and message, along with a plethora of other data about the event, such as how the message was sent. Such parameters in the

request body include: *send_attribution*, *recipient_users*, *is_shh_mode*, and of course *text* for the actual message itself. These are only a select few among the many other parameters the app sends for quality assurance and usage statistics. I spent a long while exploring all direct message features with the proxy running to scrape all the endpoints I would need to build my bot.

Automation

With all the information I needed, I opened up a new Python file and started experimenting with sending some requests. The first bit of app functionality that I needed to recreate was logging in. If I couldn't find a way to "log in" through the API, I would not be able to puppeteer any account. Luckily, the log in endpoint worked like a charm and all I had to do was mimic the app's login flow with manual requests and use the correct user agent. I now had a valid session cookie that I could send with each request to verify my authenticated status to Instagram, the same way one's browser remains logged in for a long time after they first input their credentials to a given site such as Facebook. Authentication out of the way, I could now begin working on the meat of the project, sending direct messages. When sending a direct message POST request, it is required to also provide a valid `send_attribution` field. This field communicates how the message thread was created, and as such is valuable to Instagram's servers in determining the nature of a message. If, for example, the message thread is being created through the "New Message" button inside the direct message menu, Instagram may enforce a stricter rate limit as it is not likely a real person is continuously typing various usernames manually and then composing a new message to them. On the other hand, if the thread is created through the "Message" button on a user's profile, this is an action which may occur more commonly and thus is subject to a more lax rate limit policy. Of course, I knew none of this at first, and had to experiment with various `send_attribution` values until I stumbled upon the best for my purposes: "`message_button`".



Location of the "message_button" in the app (circled in red)

Sending Photos & Videos

Despite all the work I had put into the project, nothing could prepare me for just how difficult the key part I had yet to complete would be. I knew that in comparison to sending text, sending media would be no small task, but I never could have anticipated it would've been as difficult as it was. Instagram's media processing flow, built to handle tens of thousands (likely more) of simultaneous media uploads to their encoding servers, was much more complex than anything I had ever dealt with, and as it was not meant to be "reverse engineered" by an outsider, it had a shroud of cold unfriendliness surrounding it. I decided to start with sending photos, figuring it would be easier. First thing of note when implementing photo sending functionality was that when a photo was sent in the app, two POST requests were made (as opposed to the one request made for sending text). One of these was for uploading the photo, and the other for "sending" it. As an example, the photo could be uploaded with the first request, but if it was

never “sent” with the second request, it would never actually be messaged. If one tried to use only the second request without uploading the photo using the first, an error message would be returned. Additionally, the upload request went to a strange top level endpoint (no “`/api/v1/`” path as the other endpoints had) with the URL:

`https://i.instagram.com/rupload_igphoto/`[upload_name]

(where [upload_name] is replaced with a specially formatted name for the uploaded file). Sidenote: I have no idea what the “r” in “rupload_igphoto” is meant to refer to, I can only guess it may have something to do with Redis? Regardless, I will spare you the painfully boring details of the second request, but just know it required me to go through endless nested JSON hell. Moving on to the video endpoint, I expected to have a better grasp on things now that I had dealt with the photos- a naive assumption. Sending videos required 3 requests, one of which was a GET which had to occur before the upload request that “initialized” the upload. There were also a ton more parameters which had to be sent in the POST requests with identifying information about the video, and it took me a while to recreate the functionality in Python, but eventually after much trial and error I had done it.

```
upload_id = str(int(time.time() * 10000))
upload_name = f"{random.randint(1111111111111111, 9999999999999999)}_0_-{video_len}"
rupload_params = {
    "retry_context": '{"num_step_auto_retry":0,"num_reupload":0,"num_step_manual_retry":0}',
    "media_type": "2",
    "xsharing_user_ids": "[]",
    "upload_id": upload_id,
    "upload_media_duration_ms": str(int(duration)),
    "upload_media_width": str(width),
    "upload_media_height": str(height),
    "direct_v2": "1"
}
```

The required parameters for uploading a video (this took a lot of trial and error!)

Completion

After polishing off some aspects of my implementation and adding a few more bits of functionality (sending profiles, GIFs, clickable links, etc) I amassed my loose functions into a Python class. I could now reuse all of my code easily if I desired by importing and initializing my API class and calling my own functions, so if I ever desired to send DMs programmatically I could easily do so in no more than a few lines. As an example:

```
from myInstagramAPI import myInstagramAPI

api = myInstagramAPI(“username”, “password”)
api.login()
api.createMsgThread(“Message Text”, username=“recipient_username”)
```

This project took much longer than I thought it would, but I greatly enjoyed the satisfaction that came with finally recreating functionality that was giving me the most trouble, such as sending videos. I learned a lot not only about reverse engineering, but also the internal design and operations of massive social media networks. The API wrapper I wrote would prove to be very lucrative for me in the future, when I began creating specialized Instagram bots and selling access to individuals desiring to market their social media profiles. Thank you for reading!