Assessing Security of iOS Apps

Spoofing application behavior by monitoring network traffic and overriding methods

David Sukhin dsukhin@mit.edu
Samuel Edson samedson@mit.edu
Santhosh Narayan sannar3@mit.edu
Akshit Dua akidua@mit.edu

1. Introduction

We were interested in exploring network and application security on phone applications seeking to provide various levels of anonymity. Many applications typically require a login username-password combination for each user to access the application. To provide a feeling of anonymity to users, several such applications have deviated from this traditional username-login model. Others have continued to use a login-system with lower user requirements to conceal identity. Examples of such applications that seek to ensure some level of anonymity include Flinch, Tally, Yik Yak, and Snapchat. Of these, Snapchat is the only application that requires a password. Flinch is an anonymous video chatting application. Tally is an anonymous voting application. Yik Yak is an anonymous, location-based message board. Snapchat is an application that allows authenticated users to send each other messages, pictures, and videos that will automatically be deleted in a short period of time. Each of these applications advertise anonymity as if it is an additional level of security; however, our project sets out to show that this belief is not necessarily true.

2. Assessing Vulnerabilities

Mobile applications make many assumptions about the security of compiled code when performing secure tasks. There is an implicit trust that a compiled application will perform every task the designer intended without revealing the details of the implementation. However, man-in-the-middle network attacks, and class dumps and method swizzling can allow us to modify an application's behavior, discover how it authenticates with its API and how we can compromise the general security of the application.

We can assess three different tiers of applications:

- Public Service Apps - apps that do not track your identity
- Pseudo Anonymous Apps - apps that generate a weak identity
- Password Protected Apps - apps that have a strong user account model with a password

The attacks we can easily perform on each type of compiled application are altering app behavior, creating many users to perform mob actions, or repurposing a private API for public or malicious use.

2.1. Flinch

We can start our analysis of how to modify an application by looking a popular video chatting app called Flinch. The premise of the application is to pair two people on a random video call and then enforce that users keep their face in the frame and end the game when one person smiles. These predicates are checked on the client side so if we can alter the runtime of an application, we can get away with modifying the game's behavior such that the manipulating user always wins.

Using a tool which allows us to class dump the header files of a running application and modify method return values, it was trivial to find boolean functions like:

```
(BOOL) faceFound;
(BOOL) smile;
(BOOL) lastFrameDetectedFace;
```

and override their return values using a technique called method swizzling. Method swizzling is the practice of overriding the function name to memory address (implementation) mapping dictionary stored in a predictable place in an application's memory such that your own custom code is run before or optimally entirely replacing an existing implementation. This enables runtime modifications which turned off face tracking and smile detection so that we could never lose the Flinch game.

While method swizzling is easy during runtime for Objective-C applications because Objective-C apps maintain a name to address mapping in memory, overriding methods can be done on the C implementation level as well which can expose much more secure layers of applications such as hash functions and encryption. This is done by injecting a dynamic library into an application which runs specified code to override functions we choose. This can be useful to bootstrap crypto libraries to capture keys that are used on the client side.

2.2. Tally

Tally is an anonymous, group-based voting app. Public interest-based groups exist such as technology, nature, and sports. Private groups for organizations such as universities also exist and enlarge on an invite-only basis. Questions are pre-filled and posed by communities for the members to answer.

Tally hopes to allow users to anonymously vote on topics while still enforcing rules such as only being able to vote once and only seeing results for questions you voted on. The implementation of the app suffers from many security holes however such as using an unencrypted HTTP connection for all communication, using Cordova, a Native to Javascript bridge that runs in a webview, and by allowing a single device to register for many accounts across installs.

2.2.1. Insecure HTTP

The most critical issue with using HTTP for anonymous apps is how trivial it is for someone on a local network to intercept data and tie it to a specific user using device hostnames. iOS devices in particular default to a device name such as "Alice's iPhone" which can be used to identify the voter and spy on the vote. Any identifying information sent by the app like an authorization token can also be intercepted and used to spoof the user. While the implications of these issues are rather small in this particular case (no sensitive info is likely being sent), the use of HTTP made it simple monitor API calls with a network tool like Fiddler or Wireshark and breaks the promise of anonymity the app claims for its users.

2.2.2. Cordova Applications

The next biggest security hole for Tally is the use of Cordova to develop the application. While Cordova is an incredible tool for quickly designing and deploying an app on multiple platforms, it suffers from easy access to source code and easy modification of source code since the front end of the app is entirely based in Javascript. It also allows direct runtime access to any libraries the apps uses and makes method swizzling even easier on the Javascript layer. Inspection of the source code also makes it easy to understand how the application works.

Other application layer frameworks such as Unity make it almost impossible to view the source or override methods by creating a low level interpreter that dynamically loads portions of the app into memory only as needed. This makes it more immune to method swizzling because the locations of the code is constantly in flux. It also make it harder to attach dynamic libraries or modify (patch) the binaries since they are encrypted with a secret key that the device does not have access to.

2.2.3. Tally API

With access to the source code and all network traffic, we could investigate how Tally's API works. Tally authenticates users by creating a random string and requesting a token for this string from the server. While the returned token is not deterministic, the randomly generated string on first launch is the seed for the user id a particular installation of the app gets. Using packet sniffing, this could be pretty easily identified and replayed from outside the application using the python requests library to spoof all the proper headers and authorizations. Thus, we were able to create an indefinite number of users, and target a known group id and question id and repeatedly vote. This is obviously an issue for Tally because they will find it hard to track how many legitimate votes a question got and how many legitimate users actually use their service. It is possible to make all the same requests to the API and create a bot to create users and vote on questions. Tally includes the system time as a parameter in their requests to their API but does not enforce it. Simply removing it does not break the communication.

2.3. Yik Yak

Yik Yak is another application that has foregone the traditional username-password method of authentication. When first released, Yik Yak actually only utilized a user ID that was associated with a user's account (Moskovitz). Given someone's userID, anyone could spoof it and log in to an account and see everything a user has posted in the past breaking the anonymity model. In the past, in order to function, Yik Yak communicated not only over its own servers, but other third-party applications' as well. One of these applications, Flurry, actually did not use HTTPS and sent all information over unencrypted HTTP, including the user ID. Hence, via packet sniffing a user could easily find someone's user ID (Moskovitz) and tie a user to their posts breaking the anonymity model.

2.3.1. Packet Sniffing over HTTPS

Now, as a larger and more established company, Yik Yak uses the HTTPS best practices for any communication that sends the user ID. However, with direct access to a device, we can man-in-the-middle ourselves by installing a malicious root certificate that contains a key we have access to so that we can encrypt and decrypt traffic to and from a device. With a root certificate, the device will trust a previously untrusted encryption chain and make it vulnerable to a man-in-the-middle

attack by a tool that can decrypt and re-encrypt data with the same, compromised, key. This is theoretically possible for MIT to execute on any student device that has the MIT root certificate installed since our devices are set to trust this root certificate. Similarly, the holder of the keys for any other root certificate authority can also spoof any traffic that expects to be verified by that key. Many applications and browsers (like Uber, and Chrome when accessing Google services, or Apple services like the AppStore on iOS devices) use a technique called certificate pinning to ensure only an exact certificate is used when communicating with the expected service. This check is however done on the client side so it can be similarly disabled if we wish to compromise our own device and inspect traffic. We used a tool called SSL Kill Switch which swizzles OS level SSL certificate checking functions to make the device trust the connection.

2.3.2. HMAC message signing

When sniffing traffic, it was clear that the Yik Yak app includes a token with each request so that the server can identify the request's authenticity. The following shows a typical valid request:

> https://us-east-api.yikyakapi.net/api/likeMessage?messageID=R/5553c67430de1858ce0acc40130b3&userID=E3600B32-21CE-11E4-BF59-A79254FB2D44&userLat=42.35908631287494&userLong=-71.09412119932294&version=2.1.2&horizontalAccuracy=405.070757&verticalAccuracy=22.225466&altitude=8.580323&floorLevel=0&speed=-1.000000&course=-1.000000&salt=**1431563487**&hash=**pM8uRHu9moXxsOQRTBLPVo3ZfkA%3D**

The hash value at the end appears to be a 64 character (256-bit) base64 encoded hash string. We were able to verify this by using IntroSpy, an open source crypto function swizzling library to override the entire CommonCrypto Class on iOS so that all parameters are logged. We noticed several calls to CCHmac followed by calls to SHA256 which suggested that requests were being HMACed. Closer inspection of the HMAC system function call revealed that the underlined portion of the url above concatenated with the bold salt (just a unix timestamp) was passed directly into the CCHmac function along with the hardcoded key: **F7CAFA2F-FE67-4E03-A090-AC7FFF010729**. Reproducing this in python and modifying aspects of the request (such as location or the user ID) and signing the message using the an HMAC with the given key proved to be the correct method. The Yik Yak API accepted these requests as valid. We were able to successfully perform API actions with a given user ID that we

could extract from a device. However there was some added intricacy when trying to create new users through the API and accessing data with new, random user IDs.

We monitored the App's traffic after a first install and noticed a POST to a registerUser page. These requests were being signed the same way as the requests above so it was simple to create a user with this. However, any actions (message listing, liking, etc) did not work with the newly generated user. We noticed that this perhaps was because the app additionally registered the user ID with the Parse API with a separate message and most likely checked this registration server side for any future actions.

2.3.3. Creating Users with Parse

It seemed as though it would be harder to spoof the Parse requests because they used private functions for their crypto and message signing so they would be harder to swizzle and inspect. A typical Parse request had many headers set that signed and verified the request presumably with secret keys that were in the application binary. However, before setting out to find these keys, we inspected the request to the parse API to determine how much of the message was being signed. To our surprise, the contents of the request (the user id, timestamps, device id, etc) were all malleable. The header simply held a non-deterministic signature of the origin but did not sign the contents of the request. This meant that it was possible to simply replay the same request with modified parameters and force the Parse API to accept our requests as though they were coming from the Yik Yak app even if we modified the contents of the request.

This is insecure for a few reasons. First, by intercepting a single request, the Parse API can be entirely compromised with a simple replay attack and any keyed data from the entire database can be extracted to any user without privileges. Second, this allows for very simple manipulation of data such as making thousands of fraudulent entries in the database.

2.4. Snapchat

Snapchat advertises itself as an image sharing App with sent images and videos that disappear after they are seen by the recipient. This is successful is because it gives people the feeling that their images and videos will be less likely to be shared with people other than the recipient, so people are more

willing to share things that they don't want others to see. This feeling of security is a facade because Snapchat has had a history of security lapses. In 2013, a group called Gibson Security documented Snapchat's private API, and found "they could easily view, modify or replace snaps sent," and "view all your unread messages, and depending on the situation, modify and even replace the images completely" (Gibson Security 6). To prove their point, they released the usernames, emails, and phone numbers of 4.6 million Snapchat users, and even provided a tool to check if if your account was a part of the release at lookup.gibsonsec.org (my account was not compromised luckily).

The release of Snapchat's private API created many unofficial apps and services that offer functionality like saving Snaps and Snap Videos, that cannot be simply screenshotted. Back when the app was in version 8 and using a Python Snapchat API, I was able to create an automatic account that uploaded any received snap to a public Snapchat Story, which was useful for creating group Stories.

After the release of Snapchat version 9, they stopped supporting requests from any prior version of the app. This broke Gibson Security's hack, so all unofficial APIs no longer work. If you try to authenticate with Snapchat using the keys and method outlined by Gibson Security (all the unofficial APIs and third party apps did), the response message comes with an error, and "We've noticed that you're using a third-party application to access Snapchat, putting yourself (and possibly your friends) at risk. Please change your password and stop using third-party applications when you access Snapchat". Simply changing the HTTP user-agent header to version 9 does not do enough, so Snapchat made changes to their authentication system.

2.4.1. Binary

We began investigating this by taking a look at the Snapchat binary itself. We used a jailbroken iPhone so that we could access it over SSH and download programs outside of the App Store. It would seem to be easy as copying the binary using SCP from the phone to my computer to investigate, but iOS encrypts binaries by default. Instead, we used a program called dumpdecrypted (github.com/stefanesser/dumpdecrypted) to get the binary by dumping the running Snapchat App from RAM to disk, and copying that to my computer with SCP.

With the decrypted Snapchat Binary, we used class-dump to get a dump of Snapchat class definitions, with methods and properties, almost equivalent to stealing their header files. With all the class definitions, we looked for ones that related to authorization or cryptography, and came across SCAPIAuth that had methods names like sha2 and requestTokenForUserToken. Looking at the name,

we thought that maybe Snapchat had rolled its own encryption scheme in version 9 because Objective-C usually precedes class definitions with two letters (NSObject, UITextView, CLLocationManager). It made sense for Snapchat to use SC as their class prefix. We found this to be wrong when we searched fo SCAPIAuth and found the SCAPI library, "an open-source Java library for implementing secure two-party and multiparty computation protocols (SCAPI stands for the 'Secure Computation API')" (Cryptobiu 1). Searching for "SCAPI" in our class dump brought us to a large SCAPI class, so Snapchat started using SCAPI.

```
@interface SCAPIAuth : NSObject
{
}

+ (id)hexadecimalString:(id)arg1;
+ (id)sha2:(id)arg1;
+ (id)userAgentHeader;
+ (id)requestTokenForUserToken:(id)arg1 timestamp:(id)arg2;
+ (id)authenticationParametersForUserWithToken:(id)arg1
      username:(id)arg2 email:(id)arg3;
+ (id)authenticationParametersForEndpoint:(id)arg1 authToken:(id)arg2
      username:(id)arg3 email:(id)arg4 parameters:(id)arg5;

@end
```

2.4.2. Network Traffic

From here, we wanted to see what information was being sent over the network during authentication. To do this, we started by sending our iPhone's network traffic through a proxy on my computer. Most proxy servers work, and on my Mac we used SquidMan. This worked for getting HTTP requests, but any HTTPS traffic from Snapchat was encrypted.

The Snapchat app authenticates with the server using its own, in-App certificates. We downloaded a bad certificate and saved it in iOS System Settings as normal. To force Snapchat to use our bad certificate instead of their app's, we used a utility from Cydia, the jailbroken iPhone's App Store, called

SSL Killswitch that can force Apps to use the System certificate. We made Snapchat use the bad certificate, and made our proxy use the certificate to decrypt all of Snapchat's traffic.
We found a token being sent with Snapchat requests, but were unable to figure out what it was for. It changed every request, and the first few characters seemed to be similar most of the requests. We move on to using Cycript to try to figure out what is being tokenized.

2.4.3. Cycript

Cycript "allows developers to explore and modify running applications on either iOS or Mac OS X..." (Cycript 1). Cycript runs as a program on the iPhone itself, so we SSH into the iPhone to run Cycript while an app is running on the phone's foreground. The command, *cycript -p Snapchat* hooks into the process of the app called Snapchat. This brings up a prompt that allows us to see values of instance variables, navigate and manipulate the UI tree, and create our own objects.

Cycript also has a function called *choose* that you give a class name. The function looks for places in Snapchat running in RAM where there are blocks of memory that match the class you gave, and returns a list of pointers to those blocks of memory which are the objects themselves. Therefore, I could run choose(SCAPIAuth), and it would give me a list of all instances of that class. We had a very tough time actually finding an instance of this class because we couldn't find a time when it was instantiated.

Cycript was very useful for grabbing information out of data fields, so we went back through the class-dump we did with the Snapchat binary, and looked for the class that controlled the login page. As an aside, it is interesting to note that Snapchat does actually use SC as its class prefix, so SCAPI's SC prefix is just a coincidence.

```
@interface LoginViewController : UIViewController
<SCHeaderDataSource, SCHeaderDelegate, SCTextViewDelegate>
{
    BOOL _loggingIn;
    SCButton *_enterButton;
    SCHeader *_header;
    SCTextView *_passwordTextView;
    SCTextView *_usernameTextView;
    UIButton *_forgotPasswordButton;
```

```
}
```

The passwordTextView is right there, so after we have started Cycript, grabbing a password is as easy as running:

```
choose(LoginViewController)[0].passwordTextView.text
```

This finds an instance of the LoginViewController, gets its passwordTextView which is the object on-screen that the user types into, and get the text input. We tried this with a password typed in and were able to get it with this command, no problem. We talk about the feasibility of this kind of attack in 2.4.5.

The last part about this that is extremely peculiar that we have not mentioned is that when you hook in to the login mechanism to log yourself in with Cycript, it comes up with the same warning about third party apps that you get when you try to authenticate. Running the following command with my correct password typed into the textview results in this.

```
[choose(LoginViewController)[0] enterButtonClicked]
```

Some clues as to why this is the case are revealed by using Introspy.

2.4.4. Introspy

Introspy acts as a kind of logging mechanism for common categories of functions. It is meant to identify security issues, so you can set it to log cryptographic functions and networking. We logged our first login in fresh Snapchat App, and found their CCHmac function arguments:


```
{
  "macOut": "5fqQDNVqxsk85ezs5Wt062qhgk3EyOr77LloYPkfWz4=",
                                                          "data":
"username|password|1431295909875|93036c5ba0f1d5d86ef5c3eaf5a6deb486e9
4d8619fefa88c9b4fa14d2c51deb",
  "key": "nBBRj0nmvawrGfsnqgGnXTwGCwpB7rVPL77K1kYpFew=",
  "algorithm": 2
}
```

We looked up the algorithm to find it was sha256. The data was a concatenation of my username, my password, the timestamp with milliseconds, and a random salt presumably. The next item is an *NSURLConnection initWithRequest: delegate: startImmediately:*. The request argument is below:


```
{
  "URL": {
    "parameterString": "nil",
                                                        "absoluteString":
"https://feelinsonice-hrd.appspot.com/loq/login",
    "host": "feelinsonice-hrd.appspot.com",
    "path": "/loq/login",
    "query": "nil",
    "scheme": "https",
    "port": "nil"
  },
  "HTTPMethod": "POST",
                                                                "HTTPBody":
"dsig=E5FA900CD56AC6C93CE5&dtoken1i=00001%3Aj76Jw0hjRPmG2yTkfUbjHbD%2
FP%2BVkEWnbMtMPzHqZBy9W7YjAmzWBIhyCTbtPCv27&height=1136&password=pass
word&req_token=93036c5ba0f1d5d86ef5c3eaf5a6deb486e94d8619fefa88c9b4fa
14d2c51deb&timestamp=1431295909875&username=username&width=640",
  "cachePolicy": 1
}
```


The dsig is where the interesting information is (it is the first 20 hex characters of the message mac shown above) . What struck us next at first is the inclusion of height and width parameters. The height and width are the number of pixels across the iPhone 5S's retina screen. We suspect that Snapchat has designed an authorization scheme that includes the user's tap. It is possible they hash the position of the tap, or something similar, and give that to the server to verify that a user tapped the button instead of a script.

2.4.5. Stealing Passwords

Using Cycript to steal passwords out of Snapchat's input field in the way we described in 2.4.3 is only feasible under the assumption that an adversary can run commands on the victim's phone as root. This is a strict assumption, but considering the fact that all jailbroken iPhones have "alpine" as their default root password, it is not unrealistic. This means that any jailbroken iPhone that does not change its password before installing OpenSSH from Cydia is completely vulnerable to an attacker taking control of your Snapchat app (or most other applications). The other part is that the attacker doesn't need to just steal your password. It can take control of your Snapchat App to do anything, like send Snaps, and it can manipulate any of your other apps. Also, users would be likely to make this mistake because changing root's password requires knowing how to use the Terminal, while installing OpenSSH is almost as easy as installing an app from the App Store coming from Cydia. Younger users would be especially prone to this because they are more likely to use Snapchat and install OpenSSH when they don't fully understand the implications of it.

3. Core Issues

There is an implicit assumption by many app developers is that application binaries and operating systems are secure. However, on a compromised system, it is easy to monitor and modify any level of the application's runtime, network requests, or otherwise to determine how an application works and how to modify it. The most basic issue is that anything a client side application can do, is possible to reverse engineer and spoof externally because the server and app communicate over the narrow HTTP channel.

There is no reliable way to monitor that the desired control flow happened in the application when a request is sent and any hashed or encrypted data is done client side so it is easy to reproduce and fake when talking to the server. Apps like Snapchat and Yik Yak use probabilistic techniques such as seeing different geographic coordinates on two requests (since the coordinate values are very precise) or checking the coordinates of a user's touch as ways to determine with higher certainty that the use of the API is coming from the app instead of being spoofed. These techniques simply make it harder to reverse engineer but cannot prevent the issues.

4. Potential Solutions

As noted above, these issues will be ever present as long as the runtime and network traffic of an application can be compromised. This means that the best we can do from a security standpoint is make it both intellectually or cryptographically hard to spoof the application under normal conditions.

To secure the runtime, some frameworks use closed source layers that decrypt binaries on the fly so that the entire application is never all in memory and the memory space is constantly in flux during runtime. The Unity framework compiles Javascript down to C code and uses a custom loader at runtime that makes it harder to inspect or modify code and impossible to patch the binaries because of validation and encryption by a private key that a user would not have access to. This means that it is cryptographically hard to modify the binary and intellectually hard to inspect the code when it is in RAM for a brief period of time. This is obviously better than the plaintext code stored by Cordova as in the Tally App.

We also see that anonymous apps like Yik Yak and Tally suffer from privacy issues because they create a unique identity for the user. These apps also suffer from spoofing issues because it is simple to reproduce API requests.

To remedy privacy issues, these apps could leverage a serial unlinkable transactions protocol so that user actions do not have to be linked to single ID. Proof of identity (e.g. ownership of a post) can be done by sending a salted hash of the nonce used in the transaction so that the server can know it is talking to the actual poster without creating a strong identity. Token revocation would also be trivial with this method in the case of abuse. While this is an intellectually interesting idea, it is unlikely that any company would implement this because they would forgo access to user information useful in advertising that has historically been the only monetization model for such apps.

In order to issue an initial token, the API could require a proof of work by the client such as finding a nonce that makes the hash of a provided string smaller than a certain reasonable value. This would mean that in the normal workflow of the app, the device would have to solve a cryptographic puzzle so that it can connect with the service. If this can be done asynchronously such that a typical user does not notice, this may be a feasible way to make it hard for a script to create multiple user accounts by

spoofing the API. This could even be a valid method for apps that require the creation of a user account. If the puzzle difficulty is high enough, the cost of abusing the API would be too high.

The other methods employed by apps are just hurdles to make the requests to the API harder to reproduce. Snapchat uses touch coordinates and Yik Yak uses geographic coordinates to sanity check the inputs to the API. Because these checks are done server side, they are secure as long as the requests are not reverse engineered. Closed source application layers keep code encrypted as long as possible to make it harder to dump a decrypted version.

It appears that beyond the cryptographic measures described above, implementing code at the lowest possible level, rolling your own verification functions and making generally unexpected and obscure design choices make it the hardest to reverse engineer these apps and APIs. If code is implemented with primitive operators, it is much harder to swizzle methods at runtime and the only way to reliably determine the output is to decompile and inspect machine code. These are all techniques used to make it intellectually hard to reverse engineer the applications.

Together, intellectual and cryptographic hardness can be used to create more secure applications that are more immune to inspection and modification.

5. References

Freeman, Jay. "Cycript." Cycript. SaurikIT, LLC, 2014. Web. 11 May 2015.

"Snapchat Security Disclosure." Gibson Security. Gibson Security, 27 Aug. 2013. Web. 11 May 2015.

"Welcome to SCAPI." *Scapi 2.3.0 Documentation*. Cryptobiu, 2014. Web. 11 May 2015.

Moskovitz, Sanford. “Yik Yak: Smashing the Yak.” SilverSky Labs. Github, 2014. Web. 11 May 2015.