Security Research of a Social Payment App

Ben Kraft, Eric Mannes, Jordan Moldow

14 May 2014
Appendix C added 7 July 2014

Abstract

We have performed a security audit of Venmo, a social payments application. We were interested in analyzing Venmo because of its position at the intersection of payment processing (which must be implemented securely) and social networking. We think it is important to know if secure payment is possible in a social networking environment. As such, we searched for both technical and social vulnerabilities, in addition to reverse-engineering the private API used by Venmo's apps. We looked for bugs that would allow us to steal money from others, and also ways to trick people into voluntarily giving money to the wrong person. We found several issues, including some which could allow some adversaries to steal other users' money, and some which leak information which should not be public. We recommend that Venmo they improve their user interface, remove certain user actions, and change their default privacy settings.

1 Introduction

1.1 About Venmo

Venmo is a social payments application. It provides users with an interface for easily paying the people they pay most frequently—their friends—for pizza, their share of rent, or anything else. Venmo allows users to connect with their friends on Venmo, send payments, charge other users, and publicize their transactions to other users under various privacy settings. Venmo has web, iPhone, and Android versions, as well as a public API that allows for integration with third-party applications.

In 2012, Venmo was purchased by Braintree, an online payments platform, for $26.2 million. Venmo expected to process $250 million that year [11]. Braintree was later purchased by PayPal, owned by eBay, for $800 million. Venmo has nearly 1.5 million distinct user accounts.

Venmo prominently advertises its security on its website. Venmo encrypts all connections using SSL and "uses bank-grade security systems and data encryption to protect you and prevent against any unauthorized transactions or access to your personal or financial information." [7]

1.2 About our project and paper

We were interested in analyzing Venmo because of its position at the intersection of payment processing (which must be implemented securely) and social networking. We performed a security analysis of Venmo and reverse-engineered their private API. In Section 2, we describe the methods we used to analyze Venmo's client-side code and private API. In Section 3, we describe the issues we found in Venmo that violate its security, such as enabling fraudulent transactions or leaking information to users whose access to it should be restricted. In Section 4, we provide recommendations to improve Venmo's security. Finally, in our appendices, we describe the private API used by the Venmo app as well as other API calls used in our work.

2 Methods

2.1 Android App

Venmo publishes applications for Android and iOS devices to allow users to take nearly all actions they can take on a computer from their mobile device. While the source of the apps is not public, we were able to easily decompile the Android version by downloading the APK and using the `dex2jar` utility [8] according to instructions available online [1]. While the decompiler was unable to decompile a few functions, the vast majority of the app's code came out in a readable format, from which we could reverse-engineer its functionality.

The app works about as one might expect. It makes requests to an API similar to Venmo's public API (detailed in Section 2.3), and caches some data in a local database. Fortunately, the app appears to keep the local database secure using the appropriate Android APIs, and it appears that Venmo's security model does not assume any integrity of the data therein.

2.2 Javascript

For the Venmo web application, Javascript code gets shipped to the client as gzipped, minified Javascript. It is possible to beautify this code in order to make it easier to inspect for potential vulnerabilities. We used the Python package `jsbeautifier` [2] to generate a human-readable version of the Venmo web application code.

The code appears to be partially obfuscated. Function arguments are shortened to single letters, but global variables and object attributes are not changed at all. This makes the code similar enough to the original source that it is not hard to read and make sense of it. For example, we were able to find that the code was calling a private API (see Section 2.3). We also noticed that the site was not calling the API to get lists of friends for the search autocomplete, but instead that this list was being hardcoded into the HTML page. Other than these two things, we did not find any bugs while inspecting the code, although it is possible that more bugs could be found with a more complete review.

2.3 Private API

Venmo publishes an API [6] allowing a few simple lookups of user and transaction information, along with transaction creation. Both the Android app and some parts of the Javascript webapp use a similar but distinct API; instead of URLs beginning with `https://api.venmo.com/v1/`, they use URLs beginning with `https://venmo.com/api/v5/`. This "v5" API has calls similar to those in the public API, but also allows much more: users can manage settings and friends, search people, view public or friends' transactions, and view certain Venmo metadata. This API makes it possible to build a third-party Venmo client with all of the functionality of the official one, which is not possible with the public API. We have documented many of the v5 API calls in Appendix A.

3 Vulnerabilities

3.1 Leaking friends-only posts

Venmo transactions can have one of three privacy settings, or "audiences": public (visible to everyone), friends (visible only to the friends of the payer and payee), and private (visible only to the payer and payee). However, the Venmo API contains a call that violates their security policy: it publicly leaks information about transactions that are set to an audience of friends.

Each transaction on Venmo has a corresponding ID. These IDs (an example one being 10743167) are ordered sequentially. If `[paymentid]` is a number corresponding to a public or friends-only transaction, then a GET request to `https://api.venmo.com/payments/[paymentid]` returns a JSON response containing information about the transaction, whether or not the person making the API call is friends with either party to the transaction. The call returns the ID of the user receiving money in the transaction, a description of the charge, and timestamps for the transaction. (See Appendix B.)

3.2 SMS confirmations of charges

When Alice charges Bob on Venmo, Bob receives a text message of the form "Alice requests $[amount] - for [message]. To pay, reply with '[three-digit code]'." If Bob replies to Venmo with the confirmation code corresponding to an existing charge, Venmo processes the transaction and notifies both Alice and Bob of the result. If the code is invalid, Venmo sends a text message to Bob that reads "No one has requested money from you with that confirmation code." If Alice knows Bob's cell phone number and is able to spoof an SMS message to Venmo that appears to come from him and contains the correct 3-digit code, she can cause Venmo to transfer money to her from Bob through a fraudulent transaction.

Forging SMS messages is possible, and the sending address of an SMS may not be trustworthy. In late 2012, security researcher Jonathan Rudenberg was able to attack Twitter, Facebook, and Venmo using such a vulnerability. Twitter allowed users who had connected their phones to make tweets by text. Venmo allowed users to initiate (not just confirm) payments by text. By not authenticating text messages beyond looking at the untrustworthy sender field, these services made were vulnerable to attackers doing anything that a user could do by text message. [9]

In the case of Venmo charge confirmation codes, Alice doesn't need to know the correct confirmation code for any charge to Bob that she wants confirmed. As far as we could tell, there were no negative consequences to sending an incorrect code. So, Alice could send many SMS messages to Venmo appearing to come from Bob, each guessing a different confirmation code, until she reached the correct one. Alternatively, if sending fake text messages is expensive, she could make a large number of charges and send a smaller number of spoofed texts guessing confirmation codes. Either of these attacks could succeed quickly, due to the small confirmation code space of only 1000 possibilities.

We were unable to implement this attack in practice. Several SMS gateways that we tried, including the one used by Rudenberg in 2012, no longer appear to support users sending SMS messages that come from a number not purchased by the user for use specifically with the gateway. However, we think that trusting the integrity of an SMS message is a bad idea when it has been untrustworthy so recently in the past, and we believe that it is likely still possible to forge SMS messages, potentially for a small fee, using a method or service that we didn't try.

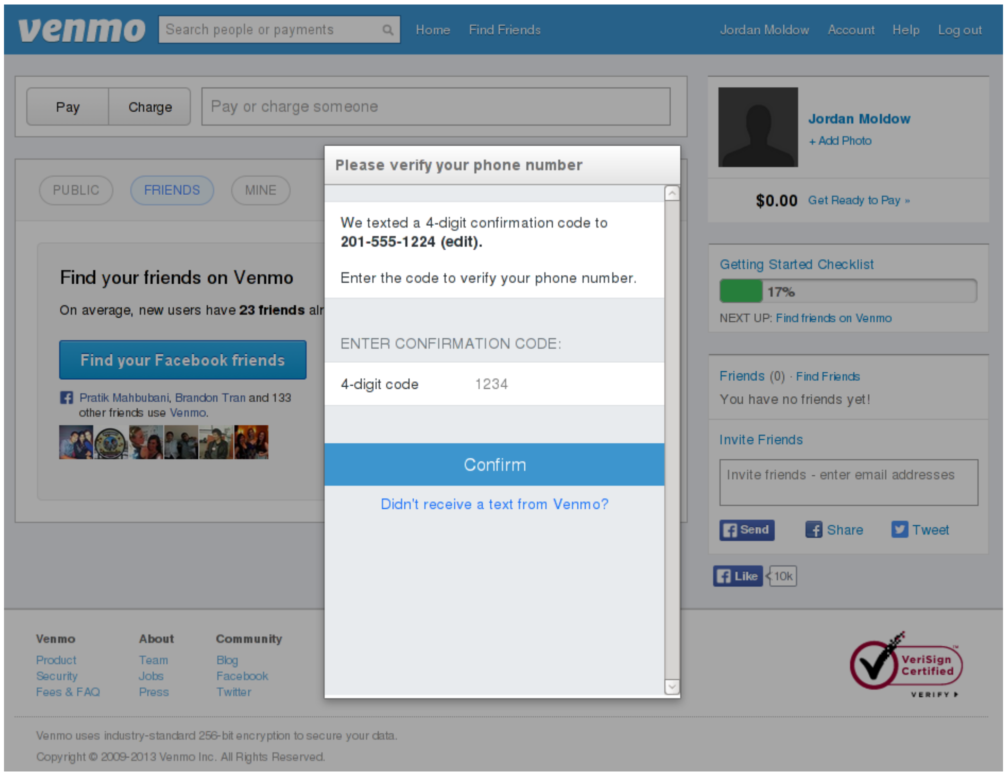
We describe a potential fix to this vulnerability in Section 4.1.

3.3 Security Policy Vulnerabilities

The Venmo account creation process implies that there is a security policy dictating that all accounts must be tied to verified identities. Creating accounts requires entering a unique phone number and email address (the Venmo server does not allow duplicates), and requires that both be verified (via SMS and email, respectively). Additionally, Venmo users are strongly encouraged to further identify themselves by verifying themselves via Facebook or entering their Social Security number; the benefit of this is an increased spending limit.

The verification process can be circumvented. You do not need to enter your own phone number; you can enter one of a friend who is not yet a Venmo user, or choose a random phone number, or choose a phone number that is guaranteed to be fictitious. It also easy to create a throwaway email address to use. After creating the account, you are directed to a screen where the application tries to force you to verify your phone number. However, this page is implemented as a modal dialog on top of the normal Venmo application (see Figure 1). In many modern browsers, you can manually modify the DOM to delete the modal and continue using the site. We did not find any site features which were explicitly disabled by the API for unverified accounts.[1] We were able to send a charge request from, and receive a payment at, an unverified account.

[1]Creating an app that can use the public API is disabled, but due to the insecurity of the client secrets discussed in Section 3.5, this is unnecessary.



Please verify your phone number

We texted a 4-digit confirmation code to **201-555-1224 (edit).**

Enter the code to verify your phone number.

ENTER CONFIRMATION CODE:

4-digit code: 1234

Confirm

Didn't receive a text from Venmo?

On average, new users have **23 friends** al

Pratik Mahbubani, Brandon Tran and 133 other friends use Venmo.

You have no friends yet!

Venmo uses industry-standard 256-bit encryption to secure your data.

Copyright © 2009-2013 Venmo Inc. All Rights Reserved.

Figure 1: Venmo phone number verification modal dialog.

From our standpoint, this does not seem to be a cause for concern. However, since it seems that Venmo does care about accounts being verified, that they might consider this to be a violation of their account security policy.

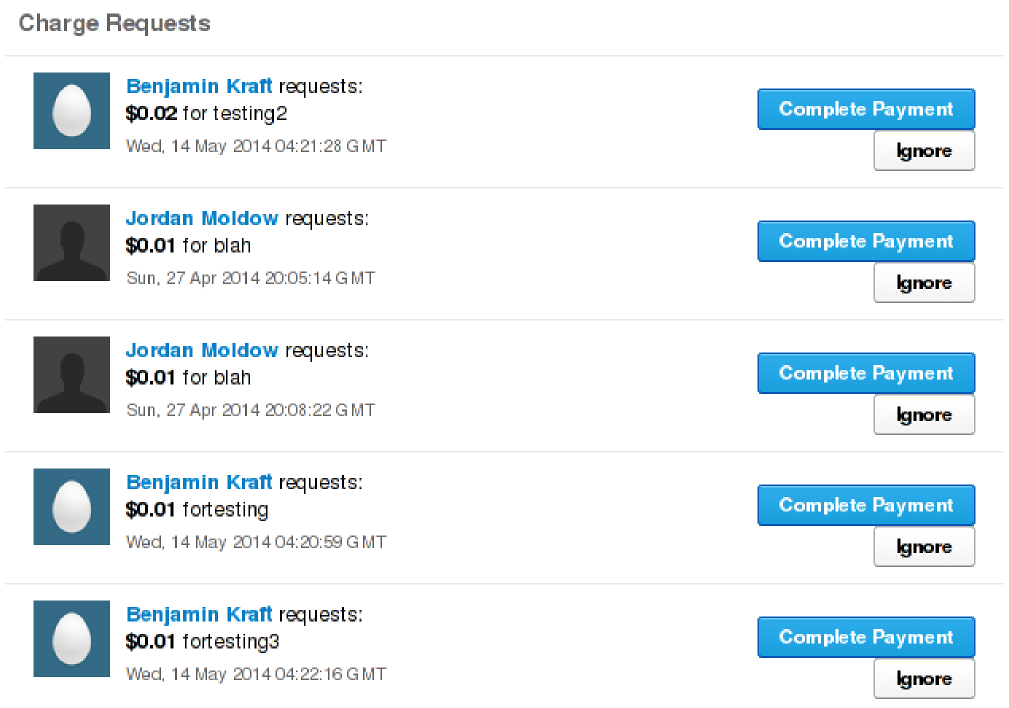
3.4 Social engineering attacks

As with any web application, Venmo is vulnerable to social engineering attacks. However, some of these attacks could be mitigated with appropriate changes to the UI.

Venmo has a notion of friendships. This is implemented in a similar fashion to those of other popular social networks such as Facebook (in fact, Venmo can import your Facebook friends and automatically create Venmo friendships with them). Within the application, you can search for all users by name, send friend requests to any of them, and accept friend requests. On any user's page (including your own), you can view their friends. Friends can be trusted (more on that later) and autocomplete in searches. You can also set stories in your feed to be available only to your friends (instead of public).

For example, one might expect that the application would only allow friends to charge each other, and only allow users to pay users who are their friends. This is not the case; any Venmo users can send charges and payments to any other Venmo user. Additionally, the process for approving a charge sent by a non-friend is exactly the same as for approving a charge sent by a friend. There are no technical barriers to tricking someone into giving you money.

The user interface makes it very feasible to carry out such an attack. The interface for pending charge requests does not make any distinction between charges from friends and charges from others (see Figure 2). Ideally, every request would be clearly marked as either being from a confirmed friend or a non-friend, so that the user can better scrutinize potentially false charges. Without this, it is harder to tell at a glance if a charge request is legitimate or from a fake account with no prior history with you.


Charge Requests

Benjamin Kraft requests:
$0.02 for testing2
Wed, 14 May 2014 04:21:28 GMT

Jordan Moldow requests:
$0.01 for blah
Sun, 27 Apr 2014 20:05:14 GMT

Jordan Moldow requests:
$0.01 for blah
Sun, 27 Apr 2014 20:08:22 GMT

Benjamin Kraft requests:
$0.01 fortesting
Wed, 14 May 2014 04:20:59 GMT

Benjamin Kraft requests:
$0.01 fortesting3
Wed, 14 May 2014 04:22:16 GMT

Figure 2: Venmo charge requests. Ben Kraft is a friend, Jordan Moldow is not. The views for both types of requests are the same.

This sort of attack is made even easier by the fact that setting and changing your identity on Venmo is incredibly easy. At any time you can change your name, profile picture, username, and biography, and have those changes instantly go into effect. If you want to steal money from Alice, and you know that she needs to use Venmo to pay Bob for something, you can change your name and profile picture to match Bob's, send a charge request to Alice, and hope that she approves it. Unless Alice has memorized Bob's username, inspects the link to your profile, and notices that the URL is different than Bob's (duplicate usernames are not permitted by the application, but you can still choose something which appears legitimate), there is nothing in the request that distinguishes it from a legitimate charge from the real Bob.

In the presence of allowed name changes, an improvement in the interface to clearly mark friends and non-friends does not even prevent this attack in all cases. If an attacker is a real-life acquaintance of Alice who is also Venmo friends with Alice, or has previously tricked Alice into accepting a friend request (perhaps also via a name change), they can later change their name and profile picture to that of her friend Bob, and execute the same attack described above. Then you would be marked as a friend, and Alice would have a fifty-fifty chance of guessing whether the charge request is from the real Bob. Therefore, even with a great interface, it seems to be an exploitable security flaw to allow users to change their names at will.

These attacks can also be carried out passively. An attacker can create a fake Venmo account with a name similar to that of a real person and hope that someone accidentally pays them. While testing the name changing feature, Eric modified his account to match Ben's. Later, our exploit was accidentally verified when our mutual real-life and Venmo friend Gurtej accidentally paid Eric's account for something instead of paying Ben.

A clever, data-driven attacker could mount an improved version of this attack. The private Venmo API we found provides a method, GET /public, for downloading a list of public transactions, which comprise as much as 50% of all Venmo transactions. The data returned by this method includes monetary amounts, payers, payees, and user-provided text descriptions of the transactions. By filtering this data, you can look for recurring spending habits, and try to hijack those. For example, you can try to find users who are receiving rent payments on Venmo, create a fake account for the rent collector, and perform active or passive attacks against the renters to attempt to collect payments.

3.5 API Authentication Vulnerabilities

In order to allow users to authorize third-party applications to use the Venmo API on their behalf, Venmo's API allows authentication using OAuth 2.0 [5]. In addition, Venmo uses other services' OAuth APIs to allow Venmo users to authorize Venmo to access those services. While many of the faults are in the OAuth standard itself, both Venmo's OAuth implementation and its usage of other services' OAuth APIs do have potential vulnerabilities.

Venmo's API implements a version of the OAuth 2.0 standard [5]. The standard itself is flawed in many ways, some of which affect how Venmo uses it. (The details and flaws of the OAuth standards are beyond the scope of this paper, but for more information, see for example the blog of the editor of the OAuth 1.0 standard, who resigned his role as the editor of the OAuth 2.0 standard [3].) The Venmo OAuth API has two possible request flows. The "client-side flow" uses only a client ID to authenticate the client, and gives the client tokens valid for only 30 minutes, while the "server-side flow" uses a client ID and client secret, and gives the client tokens valid for 60 days, after which they may be renewed. The intention seems to be that apps which have no way of securing a client secret (such as mobile apps) must use the former flow. However, even Venmo's own app violates this policy. In particular, the Venmo Android app's client ID and client secret are stored in the app, so that anyone who decompiles the app can easily find them. This is the case even though it appears that since users log into the app with their username and password rather than through OAuth, the app does not actually need the client secret at all. Other third-party apps may do the same. While compromising the client secret does not compromise user accounts, it does break the intended guarantee that the client is in fact the client it claims to be, and trivializes the distinction between the two authentication flows.

Venmo also uses the Twitter OAuth 1.0 API. This implements the OAuth 1.0 standard [4], and is somewhat similar to Venmo's "server-side flow", although it is somewhat more complex. Venmo stores its client key and secret for the Twitter API in its Android app. Again, this does not directly compromise user credentials, but it means that from Twitter's perspective, anyone can claim to be Venmo, and redirect users to a page where they will be asked to authorize Venmo to access their Twitter account, but actually authorize whomever redirected them there. This opens an attack vector where an attacker Eve could put up a web page asking users to authorize Venmo to tweet as them, and linking them to this hijacked auth page. The users would see that they were authorizing Venmo, and therefore accept the authorization, but Eve would get the OAuth tokens for the request. Fortunately, it appears that Venmo's settings in the Twitter API are such that the user will always have to confirm the authorization; otherwise, the same attack would be possible against any logged-in user who has already authorized the real Venmo client, without the user even seeing the authorization page or having a chance to deny the authorization.

Unfortunately, Twitter offers no help here; even a Twitter developer suggested that not much more could be done to secure the client secret [10]. In this case, it might be possible for Venmo to proxy the requests through its servers, storing the secret only on Venmo servers, but that would be more difficult, and it seems Twitter does not expect it. Venmo also connects to the Foursquare and Facebook APIs, and these connections appear to be more secure, as their secrets are not stored within the app. Fortunately, the Venmo OAuth API is not vulnerable to the same attack even if keys are compromised, because it does not allow OAuth clients to set an arbitrary redirect URL. Twitter could also implement a similar system, but it does not.

4 Recommendations

4.1 Rate-limiting charge confirmations and capping charges

In Section 3.2 above, we described an attack in which an attacker who can spoof SMS messages that appear to come from an arbitrary sender (a plausible threat) can steal money from any user whose cell phone number he knows. The attack works because there are only 1000 possible 3-digit confirmation codes a Venmo charge can have, there is no limit on the number of charges one Venmo user can be charged, and there are no consequences for submitting incorrect confirmation codes by SMS. Thus, an attacker can spam a victim with charges to fill up their confirmation code space and then send a few guesses via spoofed SMS messages.

We suggest that Venmo do the following:

- Use 6-digit confirmation codes instead of 3-digit ones;
- Only allow a user to have 20 charges to them pending at once;
- Not accept any confirmation codes submitted over SMS for some period of time if a user submits 10 incorrect confirmation codes;
- Decrease the number of incorrect confirmation codes Venmo will accept from individual if a large number of incorrect codes come in globally.

It is unlikely that anyone using Venmo normally will run into these limits. (If not, these numbers can be tweaked.) If an attacker tries to make the maximum number of charges against someone's account and then tries to guess the maximum number of text messages, they'll succeed with probability $1 - \left(\frac{99980}{1000000}\right)^{10} \approx .04\%$. To successfully perform this attack even once, an attacker will likely have to submit an enormous number of wrong confirmation codes, by which time Venmo would have throttled SMS confirmations and otherwise taken notice.

4.2 OAuth Secrets

Some of the vulnerabilities in the OAuth APIs are very difficult to fix. In particular, it is more or less impossible for any standalone app to authenticate itself to a service without potentially leaking its authentication information to users. Thus, in such a situation, it should be assumed that any client identifier should only be used for the purpose of tracking or analytics, and not to guarantee security. In addition, the implementation should ensure that a user's tokens are only ever given to the application the user thinks they are giving access, such as by ensuring that redirect URLs specified in the OAuth standards are on the correct domain. (Venmo's OAuth API already does this; Twitter's does not.) Finally, if client authentication is desired, requests can still be proxied through an external server on which the secret is stored. However, this just passes the buck—now the client must authenticate to that server, which remains difficult.

4.3 Plugging leak of friends-only transactions

The API call described in Section 3.1 should be fixed so that it doesn't display information about friends-only transactions to a Venmo user who is not a friend of either party to the transaction.

4.4 Preventing impersonation attacks

Venmo should clearly indicate when a user is about to pay a non-friend or when a user receives a charge from a non-friend. Otherwise, there is no obvious way for a user to distinguish a payment to or charge from their friend from a transaction with an attacker using the friend's name and profile picture to impersonate the friend, as described in Section 3.4.

5 Conclusion

On the whole, we found Venmo to be reasonably secure. We were unable to actually steal any money with the exploits we found, although it may be possible to do with the SMS spoofing attack. However, we did find some bugs that can cause users' assumptions to be violated: OAuth authorizations that are not what they seem to be, one Venmo user masquerading as another, and transactions that are supposed to be shared only with friends being publicly accessible. In addition, we learned just how easy it was to reverse-engineer an API; while Venmo could have done more to hide their private API, we expect that it would only have made our job more time-consuming, without changing the outcome.

To that end, we have learned a lot about the security of mobile and web applications. In writing apps, developers should assume that any their code that is ever on a user device is public, including all APIs and any secrets stored in apps or Javascript. In addition, developers should consider how users will know that the action they are taking is the action they think they are taking, and whether an adversary can circumvent that. Rate-limiting is very easy to do, generally doesn't affect normal users, and is a great defense against many attack vectors. Finally, developers should be careful which external services, networks, and organizations they trust. And, of course, users should pay attention to defaults, and to what they share with whom.

6 Acknowledgements

The authors would like to thank the entire course staff of 6.857, especially Prof. Ron Rivest and Anand Oza, the TA who advised our project. We would also like to thank the Venmo security team for giving us permission to do this project.

7 References

[1] Jerod Brennen. *Step-by-step guide to decompiling android apps*. Dec. 2013. URL: `https://infosecguide.wordpress.com/2013/12/17/step-by-step-guide-to-decompiling-android-apps/`.

[2] Stefano Sanfilippo et al. Einar Lielmanis. *jsbeautifier*. 2014. URL: `https://pypi.python.org/pypi/jsbeautifier`.

[3] Eran Hammer. *OAuth 2.0 and the road to hell*. July 2012. URL: `http://hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell/`.

[4] E. Hammer-Lahav. *The OAuth 1.0 Protocol*. RFC 5849 (Informational). Obsoleted by RFC 6749. Internet Engineering Task Force, Apr. 2010. URL: `http://www.ietf.org/rfc/rfc5849.txt`.

[5] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749 (Proposed Standard). Internet Engineering Task Force, Oct. 2012. URL: `http://www.ietf.org/rfc/rfc6749.txt`.

[6] Venmo Inc. *Venmo OAuth API*. 2014. URL: `https://developer.venmo.com/docs/oauth`.

[7] Venmo Inc. *Venmo — Security*. 2014. URL: `https://venmo.com/info/security`.

[8] Bob Pan. *dex2jar*. 2013. URL: `https://code.google.com/p/dex2jar/`.

[9] Jonathan Rudenberg. *SMS Vulnerability in Twitter, Facebook and Venmo*. Dec. 2012. URL: `https://titanous.com/posts/twitter-facebook-venmo-sms-spoofing`.

[10] Taylor Singletary. *How to protect my Consumer Key and Consumer Secret Key*. 2012. URL: `https://dev.twitter.com/discussions/5456`.

[11] Jenna Wortham. *Braintree, a Payments Company, Buys Venmo for* $26.2 *Million*. Aug. 2011. URL: `http://bits.blogs.nytimes.com/2012/08/16/payments-start-up-braintree-buys-venmo-for-26-2-million/?smid=tw-share`.

Appendix A Venmo v5 API

In general, the v5 API is a REST API similar to the public v1 API. The HTTP requests used are GET, POST, PUT, and DELETE; for GET and DELETE parameters should be URL-encoded, while for POST and PUT they should be form-encoded in the body of the request. The response will be JSON, usually a dictionary, often with a 'data' key containing most of the data. Errors are returned using HTTP status codes with a JSON response body including another error code and an error message.

We did not document every call in the API. For example, we did not document all of the calls involved in setting up a bank account or debit/credit card and confirming a phone number.

A.1 Authentication

GET /oauth/authorize Authorizes an app to use the API on your behalf. As far as we can tell, this works identically to the documented v1 OAuth API [6] authorization process. The client ID and secret from a v1 API app may also be used for the v5 API.

GET /oauth/access_token Gets an access token. This may either be used as a part of the OAuth server-side flow with GET /oauth/authorize (as in the v1 API), or may be used alone to log in using a username and password. If used alone, requires arguments 'phone_email_username' and 'password'. Note that while the URL implies that this implements OAuth, it in fact does not; it simply sends a username and password.

A.2 Transactions

GET /public Gets 20 recent public transactions. Does not require authentication. Optional arguments ‘since’ and ‘until’ allow paging through the list of all public transactions by timestamp. Returns the details of each transaction, including the payment ID, payer, payee, timestamp, message, permalink, client used, and any comments or likes.

GET /feed Gets recent transactions in the user’s newsfeed. Optional parameters ‘limit’, ‘until’, and ‘since’ allow paging through the feed. Returns information similar to GET /public, along with transaction amounts and the user’s current balance and number of notifications.

GET /users/[id]/feed Gets recent transactions in another user’s newsfeed. Optional parameters ‘limit’, ‘until’, and ‘since’ allow paging through the feed. Returns information similar to GET /public, along with the requestor’s balance and number of notifications.

GET /news Gets recent transactions by a user and their friends. Otherwise similar to GET /public and GET /feed; only includes amounts on the user’s own transactions.

GET /stories/[id] Gets the details of a story by its ID (a 96-bit hex string, returned by any of the above calls). Includes amount only for the user’s own transactions.

GET /stories/[id]/likes Gets the likes of a story by its ID.

GET /transactions Identical to GET /feed, except omits the user’s notification count.

GET /users/[id]/transactions Identical to GET /users/[id]/feed, except omits the user’s notification count.

GET /pending Gets a user’s pending transactions.

POST /transactions Makes a transaction. Similar to /payments in the v1 API. Required parameters are ‘uuid’, ‘note’, and ‘transactions’; ‘transactions’ should be a JSON-encoded list containing a single dictionary. The dictionary should have a key ‘amount’ with value a number (negative to charge instead of paying) and one of the keys ‘user_id’, ‘username’, ‘phone’, and ‘email’. Optional parameters are ‘audience’, ‘sharing’ (a JSON-encoded list of strings like “facebook” and “twitter”), ‘location’ (a comma-separated pair of doubles), and ‘app_id’ (which controls the “via [app]” line in the transaction display). Returns your current balance, a success message, and the transaction’s ID.

POST /likes Likes a transaction. The only parameter is ‘story_id’.

POST /comments Comments on a transaction. Required parameters are 'story_id' and 'comment_text'.

A.3 Users & Friends

GET /me Gets information about the current user.

GET /users/[id] Gets information on a particular user, including whether they are a friend of the current user, and whether they are trusted by the current user. Returns additional information and settings if the user is the current user.

POST /users Creates a user. Required arguments are 'first_name', 'last_name', 'phone', 'email', and 'password'. If the phone or email are already used, it will return appropriate errors; if the account is created it will still return an HTTP 400 Bad Request, but the only error will be 'Bad Request'. A 'facebook_id' and 'facebook_access_token' may also be included. A 'phone_claim_secret' may also be included; we are unsure of its purpose.

GET /recents Gets users with whom the current user has recently interacted, possibly only including friends.

GET /friends Gets the current user's friends (and whether they are trusted).

GET /friends/typeahead Identical to GET /friends.

GET /users/[id]/friends Gets another user's friends.

PUT /friends/[id] Makes or approves a friend request. Returns whether you are now awaiting their approval or now friends.

DELETE /friends/[id] Removes a friend (or an outgoing friend request).

PUT /friends/[id]/trust Makes or approves a trust request. Returns whether you are now waiting their approval or now trusted.

DELETE /friends/[id]/trust Untrusts a friend (or removes an outgoing trust request).

A.4 Social & Sharing

PUT /linked_accounts/facebook Links a Facebook account. Required parameters are 'facebook_access_token' and 'facebook_id'. Optionally 'return_friends' may be set to 1 to return those Facebook friends who are also on Venmo or 0 to return an empty response. If the access token is incorrect, it will still return that Facebook user's friends who are on Venmo if that information is available from Facebook. Optionally 'facebook_expires_in' is also optional; it is unclear what its value should be.

PUT /linked_accounts/facebook/refresh Relinks an expired Facebook account. It is unclear how this differs from PUT /linked_accounts/facebook.

GET /friends/facebook Gets the current user's facebook friends who are also on Venmo.

PUT /linked_accounts/twitter Links a Twitter account. Required parameters are 'twitter_access_token' and 'twitter_access_token_secret'. Optionally 'follow_venmo' may be set to 0 or 1.

PUT /linked_accounts/foursquare Links a Foursquare account. The only parameter is 'foursquare_access_token'.

POST /contacts Adds contacts (e.g. from a phone) as friends. The parameter 'contacts' should be a JSON-encoded list of dictionaries, each containing keys 'n' (for name), 'e' (for email), and 'p' (for phone number).

POST /invites Invites users. Exact syntax is unclear, but requires a parameter 'invite_list', which is likely a JSON-encoded list of strings.

POST /stories/each Modifies all of a user's transactions in the same way. The only known parameter is 'audience', which sets the privacy of all past transactions.

POST /stories/[id]/audience Changes the audience of a particular past transaction. The only parameter is 'audience'.

A.5 Settings

GET /settings/notifications Gets notification settings.

PUT /settings/notifications Updates notification settings. Parameters are those returned by GET /settings/notifications.

GET /users/me Gets general settings and information about the current user.

PUT /users/me Updates general settings. It is unknown exactly which of the parameters returned by /users/me may be updated; 'audience', 'autofriend', and 'sharing' are definitely included. Returns an empty response.

PUT /users/[id] If the ID is that of the current user, equivalent to PUT /users/me. Otherwise, a permission error.

A.6 Other

GET /notifications Gets notifications for the current user.

POST /notifications Responds to a charge request. Parameters 'id', 'action', and 'type' are required. 'action' is 1 to approve, 0 to reject; 'type' should be 'charge'.

GET /search Searches for users. The only required parameter is 'q'; 'limit' is optional.

GET /ab_testing Gets some kind of data related to A/B testing options. The exact details of how this is used are unknown, but it appears to allow changing the text in certain places and certain default messages. User authentication is not required, so by omitting it and querying repeatedly one can see all possible responses.

GET /metadata Gets some kind of metadata about the client. Required parameters are 'client_id' and 'client_secret', but user authentication is not required. Appears to include information on referral credits, possible client permissions, and more.

Appendix B Other API Calls

GET https://api.venmo.com/payments/⟨id⟩ Gets details about a payment or charge. It returns the status (e.g. 'CHARGE_COMPLETED'), timestamps for creation and completion, the aforementioned ID, a note, the audience, the action type (payment or charge), and the 'target_user_id.' It also includes the amount of the transaction, although if the user making the call is not a party to the transaction, the amount field is null.

There are two types of transaction IDs. One type is a decimal number (up to 8 digits long, such as `10743167`), numbering completed transactions in sequential order. Another type is a longer hexadecimal string (such as `5372ef2bd546b84342cb223a`), which corresponds to a pending charge. For the first type, the 'target_user_id' is the ID of the user receiving the money (the recipient of a payment or the person making the charge). For the second type, the 'target_user_id' is the ID of the user receiving the charge.

This call returns payment information for all transactions that have a public or friends-only audience, even if the person making the call is not friends with either party to the transaction. Since transaction IDs are sequential, it is easy to just iterate through all of them to get this information. This is a violation of Venmo's security policy (see Section 3.1).

Appendix C Venmo's Response

We researched Venmo security issues under a responsible disclosure policy, which was established with the company before we began our project. At the conclusion of our investigation, we sent the above paper to Venmo, to allow Venmo engineers to address the issues we found before publishing the paper and publicly disclosing our findings.

Venmo's written response to our paper was the following:

> *Venmo engineers addressed the following issues from the research findings:*
>
> - *Leaking friends-only posts: The API endpoint was taken down.*
> - *SMS confirmation of charges: Confirmation code now 6 digits long and rate-limited.*
> - *OAuth Secrets: We don't rely on the client secret for purposes of security for mobile app binaries we ship.*
>
> *In addition, we are constantly improving how we mitigate risks around social engineering and identity fraud. We currently minimize these risks by using various rate limits, transaction caps, and internal monitoring tools to detect and eliminate abuse.*