Where is the Weakest Link? A Study on Security Discrepancies between Android Apps and Their Website Counterparts

Arash Alavi[1], Alan Quach[1], Hang Zhang[1], Bryan Marsh[1], Farhan Ul Haq[2], Zhiyun Qian[1], Long Lu[2], and Rajiv Gupta[1]

[1] University of California, Riverside
[2] Stony Brook University

(aalav003,quacha,hzhan033,marshb,zhiyunq,gupta)@cs.ucr.edu
(fulhaq,long)@cs.stonybrook.edu

Abstract. As we move into the mobile era, many functionalities in standard web services are being re-implemented in mobile apps and services, including many security-related functionalities. However, it has been observed that security features that are standardized in the PC and web space are often not implemented correctly by app developers resulting in serious security vulnerabilities. For instance, prior work has shown that the standard SSL/TLS certificate validation logic in browsers is not implemented securely in mobile apps. In this paper, we study a related question: given that many web services are offered both via browsers/webpages and mobile apps, are there any discrepancies between the security policies of the two?

To answer the above question, we perform a comprehensive study on 100 popular app-web pairs. Surprisingly, we find many discrepancies – we observe that often the app security policies are much weaker than their website counterparts. We find that one can perform unlimited number of login attempts at a high rate (*e.g.,* 600 requests per second) from a single IP address by following the app protocol whereas the website counterpart typically blocks such attempts. We also find that the cookies used in mobile apps are generally more valuable as they do not expire as quickly as the ones used for websites and they are often stored in plaintext on mobile devices. In addition, we find that apps often do not update the libraries they use and hence vulnerabilities are often left unpatched. Through a study of 6400 popular apps, we identify 31 apps that use one or more vulnerable (unpatched) libraries. We responsibly disclosed all of our findings to the corresponding vendors and have received positive acknowledgements from them. This result is a vivid demonstration of "security is only as good as its weakest link".

1 Introduction

Many web services are now delivered via mobile apps. Given that a large number of services already exist and are offered as traditional websites, it is expected that many apps are basically remakes or enhanced versions of their website

counterparts. Examples of these include mobile financial applications for major banking corporations like Chase and Wells Fargo or shopping applications like Amazon and Target. The software stack for the traditional web services has been well developed and tested, including for both browsers and web servers. The security features are also standardized (*e.g.,* cookie management and SSL/TLS certificate validation). However, as the web services are re-implemented as mobile apps, many of the security features need to be re-implemented as well. This can often lead to discrepancies between security policies of the websites and mobile apps. As demonstrated in a recent study [9], when the standard feature of SSL/TLS certificate validation logic in browsers is re-implemented on mobile apps, serious flaws are present that can be exploited to launch MITM attacks. Such an alarming phenomenon calls for a more comprehensive analysis of aspects beyond the previous point studies.

In this paper, we examine a number of critical website security policies that need to be re-implemented in mobile apps. We hypothesize that such security policies in mobile apps are significantly weaker than those in traditional website environment, due to the following observations: 1) mobile devices are much more limited in terms of power and screen size; thus, many of the stringent security features such as CAPTCHAs are likely to be relaxed; 2) many mobile apps are newly developed and may naturally lack the maturity of web services that are developed and tested for a much longer period of time.

To verify our hypothesis, we study the top 100 popular Android apps (each of which has more than 5,000,000 installs) from various categories in Google play, as well as their website counterparts, to perform a comprehensive study about their security discrepancies. The contributions of the paper can be summarized as follows:

- We identify a set of critical security policies that are commonly employed by (app, web service) pairs. Since such pairs represent essentially the same services, the discrepancy in security policies effectively lowers the security of the overall service.
- For the authentication related security policies, we find significant differences in the way their backend services handle login attempts (even when they are essentially the same company, *e.g.,* Expedia app vs. Expedia website). We report 14 high-profile apps without any obvious security layer against failed login attempts while their website counterparts do have security protections. Thus these apps allow unlimited number of login attempts at a high rate that can be used for dictionary attacks. We also find that in 8 apps, the discrepancy allows one to perform an unlimited number of requests and learn whether a user ID has been registered with the service.
- For the cookie management related security policies, we find that cookies managed by mobile apps are generally 1) easier to steal as they are often stored in plaintext and accessible in a number of ways; 2) more valuable to steal as many of them do not expire any time soon; and 3) more usable by an attacker as they can be used from almost any IP address in the world.

- For the use of libraries, we find 2 of the above 100 apps use vulnerable versions of libraries. By extending our study to 6400 apps, we find 31 potential vulnerable apps due to their use of vulnerable libraries.

The rest of this paper is organized as follows. We first introduce the necessary background information for the rest of the paper in section 2. Then we discuss the methodology and implementation details in section 3. We describe our observations for different tests that we have performed in section 4. Section 5 lists the related works and section 6 concludes the paper.

2 Background

In this section, we begin with the introduction to different authentication security policies, and then we discuss the storage encryption methods that are used by different browsers and in mobile apps. Finally, we give a brief overview of library use in Android apps and how it differs from the browser scene.

Authentication Security Policies. We anticipate to see many different forms of authentication security policies in place for both apps and websites. One of the most common forms of authentication policies that can be seen are CAPTCHAs. Others include a mandatory wait period or denial of access either to an account or service. All three of these have potential to be IP/machine-based or global.

*C*APTCHA. Though CAPTCHAs are designed with the purpose of defeating machines, prior research has shown that they can be defeated by machines algorithmically [14] or via speech classification [18]. Due to the possibility of CAPTCHA replay attacks, Open Web Application Security Project (OWASP) recommends that CAPTCHA be only used in "rate limiting" applications due to text-based CAPTCHAs being crackable within 1-15 seconds [16].

*W*aiting Time. A less common method of authentication policy is the usage of waiting periods to limit the number of logins that can be attempted. The response is in the form of an explicit message or disguised through a generic "Error" message. Waiting periods, either for a single IP or for the user account is a very effective method to slow down and mitigate aggressive online credential guessing attacks. Depending on the implementation, it may operate on a group of IPs (*e.g.,* belonging to the same domain).

*D*enial of Access. An extreme policy is the denial of access, where an account is essentially "locked" and additional steps are necessary to regain access (*e.g.,* making a phone call) [19]). If an attacker knows the login ID of an account, then he can lock the account by repeatedly failing the authentication. Though very secure against online password guessing attacks, OWASP recommends that such methods be used in high-profile applications where denial of access is preferable to account compromises [15].

Storage Encryption Methods. Browsers on PCs by default encrypt critical data for long term storage. In the case of Chrome on Windows, after a successful login into a website, by clicking "Save Password", the browser stores the password in encrypted form using the Windows login credential as the key. It is not the same for mobile apps. For instance, the APIs for managing cookies do not require the cookies to be encrypted.

Libraries. Mobile apps use libraries for different functionalities such as advertisements, audio and video streaming, or social media. Previous studies [11,7,1] have shown security and privacy issues that arise by use of some libraries which can lead to leakage of sensitive user information, denial-of-service, or even arbitrary code execution. For services delivered through websites on the other hand, no website-specific native libraries are loaded. Unlike libraries embedded in apps that may be out-of-date and vulnerable, libraries used in browsers (*e.g.,* flash) are always kept up-to-date and free of known vulnerabilities.

3 Methodology and Implementation

In this section we describe our methodology and implementation details of our approach to analyze app-web pairs. We selected the top 100 popular Android apps (each of which has more than 5,000,000 installs) from popular categories such as shopping, social, news, travel & local, etc. in Google play. All apps have a corresponding website interface that offers a similar functionality. For each app-web pair, we created legitimate accounts using default settings. This was done to mimic the processes of an actual user interacting with an app or website.

Login Automation Analysis. We automate logins and logging for apps and websites for the purposes of this study. For each app-web pair, we perform 101 login attempts automatically using randomly generated alphanumeric passwords for the first 100 attempts followed by an attempt with the correct password. 100 attempts was chosen as this was an order of magnitude larger than what an average user would perform within a span of 24 hours [6]. Allowing unlimited number of login attempts is a security vulnerability because it allows an attacker to perform brute force or dictionary attacks. Another security aspect of login attempts is that if the system leaks the user ID (*e.g.,* email) during the login authentication checking, by returning error messages such as "wrong password" either in the UI or in the response message, then an attacker can send a login request and learn whether a user ID has been registered with the service. Therefore, we also compare the servers' responses to login requests, either shown in the UI or found in the response packet, for both apps and websites.

Sign up Automation Analysis. Besides login tests, we perform the sign up tests that can also potentially leak if the username has been registered with the service. Again, we simply need to compare the servers' responses to sign up requests for apps and websites. For both login and sign up security policies, if a service where the website allows for only a limited number of logins/sign-ups before a CAPTCHA is shown whereas the mobile app never prompts with a CAPTCHA, an attacker would be inclined to launch an attack following the mobile app's protocol rather than the website's. Test suites for the purposes of testing mobile apps and websites were created using *monkeyrunner* and *Selenium Webdriver*, respectively.

Authentication Throughput Analysis. From the login automation analysis, we collect the set of app-web pairs where we find different behaviors between the app and the website counterpart, we call this set "discrepancy list". Using the network traffic monitoring tools *Fiddler* and *mitmproxy*, we log network traffic traces for all app-web pairs in the discrepancy list. Using the information in the

network traffic traces, we analyze how authentication packets are structured for each client as well as finding what sort of information is being shared between a client and server. This enables us to determine whether the app-web pair has the same authentication protocol and share the same set of backend authentication servers. In addition, this allows us to construct tools capable of sending login request packets without actually running the mobile app, pushing for higher throughput of authentication attempts. The tool also logs all responses received from a server. To push the throughput even further, we can optionally parallelize the login requests (from the same client) by targeting additional backend authentication server IPs simultaneously. Our hypothesis is that the throughput can be potentially multiplied if we target multiple servers simultaneously.

IP-Changing Clients Analysis. Using *VPN Gate* and a sequence of 12 IP addresses from different geographical locations, including 3 from North America and 9 from other countries, we test the apps and websites regarding their response to accounts being logged in from multiple locations separated by hundreds of miles in a short span of time. The motivation of this analysis was to determine whether app/website has a security policy against IP changes can indicate session hijacks [8]. If not, then an attacker can use the hijacked cookies anywhere without being recognized by the web service. For example an attacker can use a stolen cookie from an app with any IP address to obtain personal and/or financial information pertaining to the user account.

Cookie Analysis. For each app-web pair, we analyze the cookies that are saved on the phone/PC. We collect all the cookies and analyze cookie storage security policies to find whether they are stored in plaintext and more easily accessible. We also perform expiration date comparison testing on 18 shopping app-web pairs from our list of app-web pairs. The hypothesis is that mobile apps run on small screens and it is troublesome to repeatedly login through the small software keyboard; therefore the corresponding app's servers will likely have a more lenient policy allowing the cookies to stay functional for longer time periods.

Vulnerable Library Analysis. While both apps and websites execute client-side code, app code has access to many more resources and sensitive functionalities compared to their website counterpart, *e.g.,* apps can read SMS on the device while javascript code executed through the browser cannot. Therefore, we consider the app code more dangerous. Specifically, vulnerable app libraries running on the client-side can cause serious attacks ranging from denial of service (app crash) to arbitrary code execution. Because of this, for each app, we identify if it uses any vulnerable libraries. We conduct the analysis beyond the original 100 apps to 6400 apps in popular categories. Ideally the libraries should be tagged with versions; unfortunately, we discover that most libraries embedded in Android apps do not contain the version information as part of their metadata. Therefore, in the absence of direct version information, we perform the following steps instead. First, we search the extracted libraries through the CVE database. If there is any library that is reported to have vulnerabilities, we perform two tests to conservatively flag them as vulnerable. First is a simple time test: we check if the last update time of the app is before the release

time of patched library. Obviously, if the app is not updated after the patched library is released, then the app must contain a vulnerable library. If the time test cannot assert that the library is vulnerable, we perform an additional test on the symbols declared in the library files. Specifically, if there is a change (either adding or removing a function) in the patched library, and the change is lacking in the library file in question, then we consider it vulnerable. Otherwise, to be conservative, we do not consider the library as vulnerable.

4 Observations

We present our results obtained from following the methodology outlined earlier.

Security Policies Against Failed Login and Sign up Attempts. By performing login attempts automatically for each pair of app and website, many interesting discrepancies in security policies have been found. Figure 1 summarizes the main results for all 100 pairs, considering their latest versions at the time of experiment. In general, we see that the security policy is weaker on the app side. There are more apps without security policies than websites. We also see that there are significantly fewer apps asking for CAPTCHA, presumably due to the concern about usability of the small keyboards that users have to interact with. Interestingly, in the case when CAPTCHAs are used both by app and website, the CAPTCHA shown to app users is usually simpler in terms of the number of characters and symbols. For instance, LinkedIn website asks the user to enter a CAPTCHA with 2 words while its app CAPTCHA only has 3 characters. Unfortunately, an attacker knowing the difference can always impersonate the mobile client and attack the weaker security policy. We also observe that more apps employ IP block policies for a short period of time. This is effective against naive online credential guessing attacks that are not operated by real players in the underground market. In reality, attackers are likely operating on a large botnet attempting to perform such attacks, rendering the defense much less effective than it seems. In fact, if the attackers are aware of the discrepancy, they could very well be impersonating the mobile client to bypass stronger protections such as CAPTCHA (which sometimes requires humans to solve and is considered additional cost to operate cyber crime).

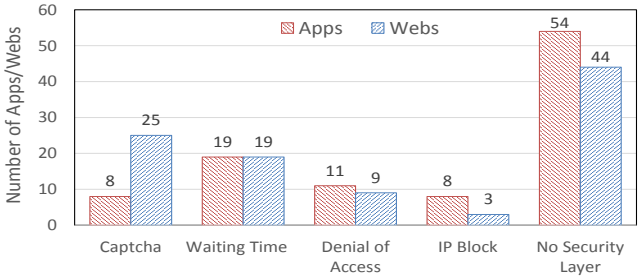

Fig. 1. Security policies against failed login attempts in apps vs. websites (TODO: replace this figure with apps of older versions

Table 1 lists app-web pairs in detail where apps operate without any security protections whatsoever, at least for the version when we began our study but their websites have some security policies. In total, we find 14 such app-web pairs; 8 apps have subsequently strengthened the policy after we notified them. There are however still 6 that are vulnerable to date. We also provide a detailed list of all 100 app-web pairs on our project website [2]. To ensure that there is indeed no security protection for these apps, we perform some follow-up tests against the 14 applications and confirm that we could indeed reach up to thousands of attempts (without hitting any limit). Note that our approach ensures that no hidden security policy goes unnoticed (such as the account being silently blocked), as our test always concludes with a successful login attempt using the correct password, indicating that it has not been blocked due to the failed attempts earlier. In the table, we also list the URLs that correspond to the login requests. Since both the domain names and resolved IP addresses (which we did not list) are different, it is a good indication that apps and websites go through different backend services to perform authentications, and hence there are different security policies.

App-Web	App Security Layer (App Verrsion)	Website Security Layer	App Host	Website Host
Babbel	None(5.4.072011) Account lock(5.6.060612)	Account lock	www.babbel.com/api2/login	accounts.babbel.com/en/ accounts/sign_in
Ebay	None(3.0.0.19) IP block(5.3.0.11)	Captcha	mobiuas.ebay.com/servicesmobile /v1/UserAuthenticationService	signin.ebay.com/ws/eBayISAPI.dll
Expedia	None(5.0.2)	Captcha	www.expedia.com/api/user/signin	www.expedia.com/user/login
Hotels.com	None(12.1.1.1) IP block(20.1.1.2)	Captcha	ssl.hotels.com/device/signin.html	ssl.hotels.com/profile/signin.html
LivingSocial	None(3.0.2) IP block(4.4.2)	Wait time	accounts.livingsocial.com/v1/oauth /authenticate	accounts.livingsocial.com /accounts/authenticate
OverDrive	None(3.5.6)	Captcha	overdrive.com/account/sign-in	www.overdrive.com/account /sign-in
Plex	None(4.6.3.383) IP block(4.31.2.310)	IP block	plex.tv/users/sign_in.xml	plex.tv/users/sign_in
Quizlet	None(2.3.3)	Wait time	api.quizlet.com/3.0/directlogin	quizlet.com/login
Skype	None(7.16.0.507)	Wait time & Captcha	uic.login.skype.com/login/skypetoken	login.skype.com/login
SoundCloud	None(15.0.15) IP block(2016.08.31-release)	Captcha	api.soundcloud.com/oauth2/token	sign-in.soundcloud.com/ sign-in/password
TripAdvisor	None(11.4) IP block(17.2.2)	Captcha	api.tripadvisor.com/api/internal/1.5/ auth/login	www.tripadvisor.com /Registration
Twitch	None(4.3.2) Captcha(4.11.1)	Captcha	api.twitch.tv/kraken/oauth2/login	passport.twitch.tv/authorize
We Heart It	None(6.0.0)	Captcha	api.weheartit.com/oauth/token	weheartit.com/login/authenticate
Zappos	None(5.1.2)	Captcha	api.zappos.com/oauth/access_token	secure-www.zappos.com /authenticate

Table 1. Discrepancy of authentication policies among app-web pairs. In all cases, the above apps have no security policy while their website counterparts do have security policies. This allows attackers to follow the app protocol and gain unlimited number of continuous login attempts (confirmed with 1000+ trials). A subset of them (8) have subsequently patched the security flaw after our notifications.

Impact of online credential guessing attacks. To perform online password guessing attacks, one can either perform a brute force or dictionary attack against

those possibilities that are deemed most likely to succeed. As an example, the recent leakage of passwords from Yahoo [4] consisting of 200 million entries (without removing duplicates). According to our throughput result, at 600 login attempts per second (which we were able to achieve against some services), one can try every password in less than 4 days against a targeted account (if we eliminate duplicate passwords the number will be much smaller). Let us consider an attacker who chooses the most popular and unique 1 million passwords; it will take less than half an hour to try all of them. Note that this is measured from a single malicious client, greatly lowering the requirement of online password guessing attacks, which usually are carried out using botnets. Another type of attack which can be launched is Denial of Service (DoS) attack. By locking large amount of accounts through repeated logins, attackers could deny a user's access to a service. As we mentioned earlier, we find more apps than websites which have the account lock security policy against the failed authentication (11 apps vs. 9 websites). Account lock security policy is a double edge sword: while it provides security against unauthorized login attempts, it also allows an attacker to maliciously lock legitimate accounts with relative ease. The result shows that this kind of attack can be more easily launched on the app side. We verify this claim against our own account and confirm that we are unable to login with the correct password even if the login is done from a different IP address.

To perform online account-ID/username guessing attacks, we report the result of the sign up (registration) security policy testing, which aligns with the login results. We find 5 app-web pairs — 8tracks, Lovoo, Newegg, Overdrive, StumbleUpon — where the app has no security protection against flooded sign up requests while the website has some security protection such as CAPTCHA. We also find that 14 websites leak the user email address during the authentication checking by returning error messages such as "wrong password". In contrast, 17 apps leak such information. The three apps with weaker security policies are AMC Theaters, Babbel, and We Heart It. The discrepancy allows one to learn whether a user ID (e.g., email) has been registered with the service by performing unlimited registration requests. Combined with the password guessing, an attacker can then also attempt to test a large number of username and password combinations.

Throughput Measurement. In throughput testing, we tested authentications-per-second (ApS) that are possible from a single desktop computer. Table 2 shows the throughput results for login testing. An interesting case was Expedia, which allowed ∼150 ApS when communicating with a single server IP and upwards of ∼600 ApS when using multiple server IPs during testing. The existence of multiple server IPs, either directly from the backend servers or CDN, played a role in the amplification of an attack. It is interesting to note that in the case of Expedia, different CDN IPs do not in fact allow amplification attacks. We hypothesize that it is due to the fact that these CDNs still need to access the same set of backend servers which are the real bottleneck. To identify backend server IPs, we perform a step we call "domain name scanning" and successfully locate a non-CDN IP for "ftp.expedia.com". From this IP, we further scan the

App	ApS (Single-server-IP)	ApS (Multi-server-IP)	# of IPs found	CDN/Host
Ebay	~ 77	~100	2	Ebay
Expedia	~150	~600	20	Akamai/Expedia
SoundCloud	~77	~178	2	EdgeCast
We Heart It	~83	~215	5	SoftLayer/ThePlanet.com
Zappos	~84	~188	20	Akamai

Table 2. Throughput results for login testing.

App-Web	App Cookies Expiration Time	Website Cookies Expiration Time
AliExpress	several months	60 minutes
Amazon	several months	14 minutes
Best Buy	several months	10 minutes
Kohl's	several months	20 minutes
Newegg	several months	60 minutes
Walmart	several months	30 minutes

Table 3. Cookies expiration time.

subnet and find 19 other IPs capable of performing authentication. By talking to these IPs directly, we are able to improve the throughput from 150 to 600.

Finally, we also obtain throughput results for 4 of the applications in sign up testing and their average throughput is around 90 to 240 ApS.

Client IP Changing. During IP address testing, we find that 11 app-web pairs have client IP changing detection and associated security policy on the server side. The remaining 89 app-web pairs have no visible security policy. Among them there are 8 app-web pairs for which both the app and the website have the same behavior against IP changing. For the remaining 3 pairs, — Target, Twitch, Steam — the app and website have different behaviors where the website returns an access denied error for some IP address changes (in the case of Target and Twitch) or forces a logout for any change of the IP address (in the case of Steam) but the app allows changing client IP address frequently.

One main consequence is that when an app/website has no security policy against IP changing, an attacker can perform HTTP session hijacking with stolen cookies more easily without worrying about what hosts and IP addresses to use in hijacking. For instance, Steam is a gaming client; it does have security protection in its websites. When a cookie is sent from a different IP, the website immediately invalidates the cookie and forces a logout. However, using the Steam app and the associated server interface, if the attacker can steal the cookie, he can impersonate the user from anywhere (i.e., any IP address).

Cookies. Cookies are commonly used for web services as well as mobile apps. In browsers, cookie management has evolved over the past few decades and gradually become more standardized and secure. However, on the mobile platform every app has the flexibility to choose or implement its own cookie management, i.e. cookie management is still far from being standardized.

We observe that many apps store their cookies unencrypted (47 apps among all 100 apps). An attacker can access the cookie more easily as compared to browsers on PCs. First, smartphones are smaller and more likely to be lost or stolen. Therefore, a simple dump of the storage can reveal the cookies (assuming no full-disk encryption). In contrast, in the case of browsers on PCs, cookies

are often encrypted with secrets unknown to the attacker even if the attacker can gain physical access to the device. For instance, Windows password (used in Chrome) and master password (used in Firefox) are used to encrypt the cookies [20]. Second, if the device is connected to an infected PC (with adb shell enabled), any unprivileged malware on PC may be able to pull data from the phone. For instance, if the app is debuggable then with the help of *run-as* command, one can access the app data such as cookies. Even if the app is not debuggable, the app data can still be pulled from the device into a file with .ab(android backup) format [12].

We also report another type of important discrepancy — cookie expiration time. Here we focus on 18 shopping app-web pairs (a subset from the list of 100 pairs). We observe that app cookies remain valid for much longer time than web cookies. The cookie expiration time in all 18 shopping websites is around 3 hours on average, whereas it is several months in their app counterparts. The result is shown in Table 3. We find that 6 apps have cookie expiration time set to at least 1 month while their websites allow only minutes before the cookies expire. An attacker can easily use a stolen cookie for these apps and perform unwanted behavior such as making purchases as the cookie is not expired. For instance, based on our personal experience, Amazon app appears to use cookies that never expire to give the best possible user experience. We confirmed that a user can make purchases after 1 year since the initial login in.

Vulnerable Libraries. During vulnerable library testing, we find two apps (Vine and Victoria's Secret) use unpatched and vulnerable libraries from FFmpeg [3] framework, which motivates us to look at a larger sample of 6,400 top free apps in different categories. Table 4 summarizes our observation for vulnerable libraries with the number of apps using them. For example, an attacker can cause a DoS (crash the application) or possibly execute arbitrary code by supplying a crafted ZIP archive to an application using a vulnerable version of libzip library [5]. As we discussed before, javascript vulnerabilities are unlikely to cause damage to the device compared to app libraries, especially given the recent defences implemented on WebView [10].

Library	Vulnerabilities	# of Apps	Example Vulnerable Apps(Version)(# of Installs)
libzip	DoS or possibly execute arbitrary code via a ZIP archive	13	com.djinnworks.StickmanBasketball(1.6)(over 10,000,000) com.djinnworks.RopeFly.lite(3.4)(over 10,000,000)
FFmpeg *	DoS or possibly have unspecified other impact	9	co.vine.android(5.14.0)(over 50,000,000) com.victoriassecret.vsaa(2.5.2)(over 1,000,000)
libxml2	DoS via a crafted XML document	8	com.avidionmedia.iGunHD(5.22)(over 10,000,000) com.pazugames.girlshairsalon(2.0)(over 1,000,000)
	Obtain sensitive information	5	com.pazugames.girlshairsalon(2.0)(over 1,000,000) com.flexymind.pclicker(1.0.5)(over 100,000) com.pazugames.cakeshopnew(1.0)(over 100,000)
	DoS or obtain sensitive information via crafted XML data	5	
	DoS via crafted XML data	5	
libcurl	Authenticate as other users via a request	1	sv.com.tigo.tigosports(6.0123)(over 10,000)

Table 4. Vulnerable libraries used by apps.

* FFmpeg includes 7 libraries:
libavutil, libavcodec, libavformat, libavdevice, libavfilter, libswscale, and libswresample.

5 Related Work

As far we know, there are no in depth studies that explicitly analyze the similarities and differences between mobile applications and their website counterparts in terms of security. Fahl et al. [9] understood the potential security threats posed by benign Android apps that use the SSL/TLS protocols to protect data they transmit. Leung et al. [13] recently studied 50 popular apps manually to compare the Personally Identifiable Information (PII) exposed by mobile apps and mobile web browsers. They conclude that apps tend to leak more PII (but not always) compared to their website counterparts, as apps can request access to more types of PII stored on the device. This is a demonstration of the discrepancy of privacy policies between apps and websites. In contrast, our work focuses on the discrepancy of security (not so much privacy) policies between apps and websites. Zuo et al. [21] automatically forged cryptographically consistent messages from the client side to test whether the server side of an app lacks sufficient security layers. They applied their techniques to test the server side implementation of 76 popular mobile apps with 20 login attempts each and conclude that many of them are vulnerable to password brute-forcing attacks, leaked password probing attacks, and Facebook access token hijacking attacks. Sivakorn et al. [17] recently conducted an in-depth study on the privacy threats that users face when attackers have hijacked a user's HTTP cookie. They evaluated the extent of cookie hijacking for browser security mechanisms, extensions, mobile apps, and search bars. They observed that both Android and iOS platforms have official apps that use unencrypted connections. For example, they find that 3 out of 4 iOS Yahoo apps leak users' cookies.

6 Conclusion

In this paper, we identify serious security related discrepancies between android apps and their corresponding website counterparts. We responsibly disclosed all of our findings to the corresponding companies including Expedia who acknowledged and subsequently fixed the problem. The lesson learnt is that, for the same web service (*e.g.,* Expedia), even though their websites are generally built with good security measures, the mobile app counterparts often have weaker or non-existent security measures. As a result, the security of the overall service is only as good as the weakest link — more often than not, the mobile apps.

7 Acknowledgments

We would like to thank our shepherd Kanchana Thilakarathna for his feedback in revising the paper. This work is supported by NSF grant CNS-1617424 to UC Riverside.

References

1. The Hacker News. Warning: 18,000 android apps contains code that spy on your text messages. `http://thehackernews.com/2015/10/android-apps-steal-sms.html`, Retrieved on 10/11/2016.
2. Authentication Policy Table. `http://www.cs.ucr.edu/~aalav003/authtable.html`, Retrieved on 10/11/2016.

3. FFmpeg. https://ffmpeg.org/, Retrieved on 10/11/2016.
4. Hacker Selling 200 Million Yahoo Accounts On Dark Web. http://thehackernews.com/2016/08/hack-yahoo-account.html, Retrieved on 10/11/2016.
5. Red Hat Bugzilla Bug 1204676. https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2015-2331, Retrieved on 10/11/2016.
6. Amber. Some Best Practices for Web App Authentication. http://codingkilledthecat.wordpress.com/2012/09/04/some-best-practices-for-web-app-authentication/, Retrieved on 10/11/2016.
7. T. Book, A. Pridgen, and D. S. Wallach. Longitudinal analysis of android ad library permissions. *CoRR*, abs/1303.0857, 2013.
8. P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. Secsess: Keeping your session tucked away in your browser. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, 2015.
9. S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android ssl (in)security. In *ACM CCS*, 2012.
10. M. Georgiev, S. Jana, and V. Shmatikov. Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks . In *2014 Network and Distributed System Security (NDSS '14)*, San Diego, February 2014.
11. M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *WiSeC*, 2012.
12. S. Hwang, S. Lee, Y. Kim, and S. Ryu. Bittersweet adb: Attacks and defenses. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, 2015.
13. C. Leung, J. Ren, D. Choffnes, and C. Wilson. Should you use the app for that?: Comparing the privacy implications of app- and web-based online services. In *Proceedings of the 2016 ACM on Internet Measurement Conference*, IMC '16, pages 365–372, New York, NY, USA, 2016. ACM.
14. G. Mori and J. Malik. Recognizing objects in adversarial clutter: Breaking a visual captcha. In *Proceedings of the 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2003.
15. OWASP. Blocking Brute Force Attacks. http://www.owasp.org/index.php/Blocking_Brute_Force_Attacks, Retrieved on 10/11/2016.
16. OWASP. Testing for Captcha (OWASP-AT-012). http://www.owasp.org/index.php/Testing_for_Captcha_(OWASP-AT-012), Retrieved on 10/11/2016.
17. S. Sivakorn, I. Polakis, and A. D. Keromyti. The cracked cookie jar: Http cookie hijacking and the exposure of private information. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy, 2016*. IEEE, 2016.
18. J. Tam, J. Simsa, S. Hyde, and L. V. Ahn. Breaking audio captchas. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 1625–1632. 2008.
19. T. Wolverton. Hackers find new way to milk ebay users. In *Proceedings of the 1998 Network and Distributed System Security Symposium*, 2002.
20. J. Wright. How Browsers Store Your Passwords (and Why You Shouldn't Let Them). http://raidersec.blogspot.com/2013/06/how-browsers-store-your-passwords-and.html/, Retrieved on 10/11/2016.
21. C. Zuo, W. Wang, R. Wang, and Z. Lin. Automatic forgery of cryptographically consistent messages to identify security vulnerabilities in mobile services. In *NDSS*, 2016.