A Study of SSL Proxy Attacks on Android and iOS Mobile Applications

John Hubbard, Ken Weimer, Yu Chen
Department of Electrical and Computer Engineering, Binghamton University, SUNY, Binghamton, NY 13902
E-mail: {jhubbar1, kweimer1, ychen}@binghamton.edu

***Abstract*—According to recent articles in popular technology websites, some mobile applications function in an insecure manner when presented with untrusted SSL certificates. These non-browser based applications seem to, in the absence of a standard way of alerting a user of an SSL error, accept any certificate presented to it. This paper intends to research these claims and show whether or not an invisible proxy based SSL attack can indeed steal user's credentials from mobile applications, and which types applications are most likely to be vulnerable to this attack vector. To ensure coverage of the most popular platforms, applications on both Android 4.2 and iOS 6 are tested. The results of our study showed that stealing credentials is indeed possible using invisible proxy man in the middle attacks.**

***Keywords*-Mobile Devices, SSL, TLS, Security, Proxy, Man-in-the-middle, Android, iOS.**

I. Introduction

In todays hyper-connected world, mobile computing has become a pervasive technology. Everyone from CEOs of large companies down to children carry smart phones and tablet computers with always on convenience and connectivity to the global Internet. On these devices, small, internet enabled mobile apps reign king. We use apps to check our bank accounts, secure our homes, hold our photos, and store the contact information of everyone we know. We use our apps for social networking, holding our documents, finding directions, online shopping, and other new functions almost every day, but how often do we consider what trusting our phone with all of this private information means? Many people probably know the risk of what could happen if they were to lose their smart phone or tablet, but what about the information that can be stolen while the device remains safely in our hands? It is easy to believe that competent developers from recognizable companies would be effective at securing mobile data, but this may not always be true. The trust put into these apps goes far beyond simple data exchange on the Internet - peoples financial and personal information is on the line.

In this paper, we will explore the claims of the ArsTechnica.com article [3] that inspired this project by putting a slew of iOS 6 and Android 4.2 applications to the test against man in the middle invisible proxy attacks. Section 2 contains detail on our motivations for researching this topic, section 3 reviews some previous work in the field of mobile application security, section 4 explains how our attack was performed, and section 5 and 6 will go over our analysis and the conclusions we have drawn as a result of this research.

II. Motivation

In this project, we seek to answer the question: *Should we trust our mobile applications to protect our data?* Mobile apps typically use the Internet to provide much of their functionality, gigabytes of data are sent from our phones through the air every month. This data contains all sorts of sensitive information, and to add to the problem, software developers are in a race to provide application based services for this data, and ultimately, to extract a paycheck from it.

New applications are created, iterated, and sold at lightning speed, and it is often discovered that in the rush to grab revenue, security has taken a back seat to fast deployment. We trust that in transit, our data is protected using Secure Sockets Layer (SSL)/Transport Layer Security (TLS) encryption [7], but due to the complex nature of software and the multitude of operating system versions it must run on, research suggests flaws may exist in many applications implementation of this security.

An inexperienced programmer may use application-building frameworks that help them get past the most daunting aspects of coding and focus on design and function. What they might not realize is that the code theyre pulling in from a library may not implement a function the way they expect. If an encryption scheme is not implemented correctly at the framework level, or the correct settings are not used, an entire group of applications may be left open to attack. It may seem like a needless exercise to understand prewritten code that provides a service such as SSL encryption, but if this code is left unreviewed, undesirable settings may go unnoticed and uncorrected in the software development cycle. They also may relax security controls for application testing, but if these controls are not put back at the correct settings before the software is released, vulnerabilities could remain.

The SSL/TLS encryption layers are generally the only means of defense used against collecting Internet traffic in plain text. If an attacker could intercept this traffic, username and password key pairs could easily be skimmed, credit card numbers, home addresses, phone numbers, social security numbers, PINs, and birthdates could also potentially be trivially recovered. The extended danger of this is that research has shown over the years that this information is not usually unique across sites. In the event that one application implements these security layers incorrectly, opening itself up for attack, the information could be used in alternate applications even if they protect themselves from similar exploits.

Over the course of this project, we attempted to test for the mistake of incorrect SSL certificate security validation across the Apple iOS and Google Android platforms. We gathered a wide sample of programs with backings large and small and ran them through a realistic attack scenario. We believe this is a valuable area of research due to the prevalence of free Wi-Fi hotspots in public areas, and the relative ease in which this type of attack is carried out. We also believe it represents one of the most likely threat scenarios when it comes to users data, and seek to answer the question of if we should trust our apps when it comes to the closed source applications so many of us download, use, and trust on a day to day basis.

III. Related Work

A. *ActiveSync Wipe*

Microsoft Exchange servers require the ability to control policy on mobile devices. Changes to settings such as password complexity, encryption, and screen timeouts are pushed to a device from the server through ActiveSync at the time of account creation. A user must accept these settings to enable the connection to Exchange server and therefore generally are allowed by the user. This policy push can occur whenever a request is made to the ActiveSync server.

Hannay has developed an attack using a rogue Wi-Fi network that presents a self-signed certificate to any devices that connect to it [4]. Most mobile devices use SSL handshakes as a means of authentication. Since the rogue network does not have the private key for the server the device is attempting to connect to, the server makes use of a self-signed certificate to complete the SSL connection. Masquerading as an exchange server, it listens for a request to provision and replies with an HTTP 449 error until the device eventually issues the request. Once a connection is established and the provisioning request is sent, the server responds with the binary encoded XML required to initiate a remote device wipe.

This attack was successfully executed using both the Android Operating System and Apple iOS resulting in a complete self-erase of the device. The author notes that this could be tailored to extend the attack to more than just remote wiping a device. Actions such as intercepting emails, stealing credentials, and forcing syncs with a rogue machine would be possible.

B. *Analysis of Android SSL (In)Security*

An analysis of the state of SSL used in Android apps was performed in [2]. For the experiment 13,500 Android apps were downloaded from the Google Play Market and studied by researchers to learn how they used SSL connections. Their vulnerability against Man-in-the-Middle (MITM) attacks was targeted by testing each app for inadequate or incorrect use of SSL certificates.

They found that of the 13,500 apps tested that 1,074 (8%) contained potential vulnerabilities to MITM attacks. Of these apps with a potential vulnerability, 100 were subjected to manual tests to discover what data could be stolen. Forty-one of the apps were shown to provide sensitive data over a connection secured with a bad SSL certificate.

Not only could user data be intercepted, but it was found that code could be injected back into a program. An anti-virus scanner was susceptible to having its virus signature database manipulated to detect any app as a virus, or to completely disable virus detection.

The study went on to test the human factor involved in mobile security, finding that over 50% of 754 participants incorrectly judged the security of their mobile connection. By forcing unprotected connections and using incorrect certificates to invoke warning messages by the operating systems, users were quizzed on whether the connection was safe. They typically misinterpreted or didnt understand the visual clues and warnings that were displayed, opening themselves up to further assisting sensitive data being stolen.

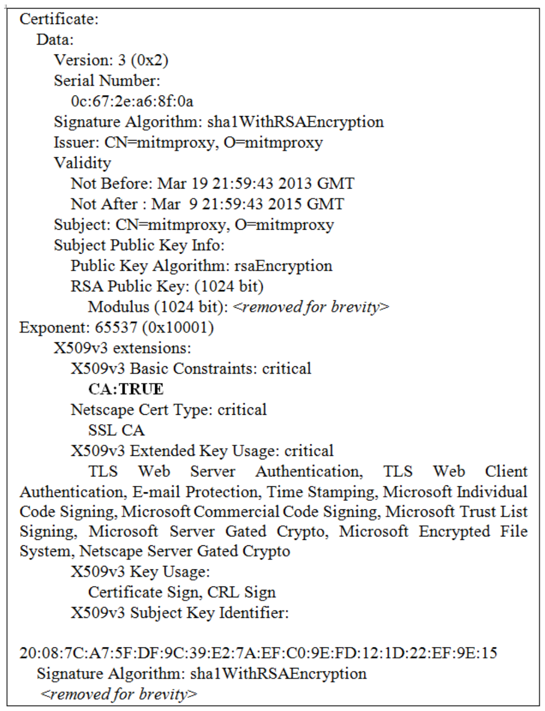
IV. TESTING METHODOLOGY AND RESULTS

In order to perform our tests, we divided attempts to steal credentials into two styles: invisible SSL proxy attacks, and manual proxy setup sniffing. The first of these methods is the main focus of our research because it is what an attacker would use in a typical scenario to surreptitiously steal login info on a shared Wi-Fi network with the victim. The second method, although not likely to be used as a surprise attack, was studied in order to see how different applications react under a realistic proxy use case, which will likely involve accepting an untrusted certificate, and implies potentially exposing a users credentials to whoever is in charge of the proxy.

A. *Invisible SSL Proxy Attack*

The first type of attack we performed was a MITM attack combined with an invisible SSL enabled proxy. This attack we believe is an extremely realistic scenario of how someone trying to steal credentials would operate. The first step of this attack is to generate a Certificate Authority (CA) marked certificate that can sign the various other certificates that our tool will be dynamically generating and serving to the victim device. The contents our generated certificate can be seen in Figure 1.

The second step of the invisible proxy attack is to make sure the attacking machine has traffic forwarding turned on so that our proxy will be able to receive forwarded victim traffic. In addition, a firewall rule must be created to route all victim traffic with a destination port of 443 to the port



```
Certificate:
   Data:
      Version: 3 (0x2)
      Serial Number:
         0c:67:2e:a6:8f:0a
      Signature Algorithm: sha1WithRSAEncryption
      Issuer: CN=mitmproxy, O=mitmproxy
      Validity
         Not Before: Mar 19 21:59:43 2013 GMT
         Not After : Mar  9 21:59:43 2015 GMT
      Subject: CN=mitmproxy, O=mitmproxy
      Subject Public Key Info:
         Public Key Algorithm: rsaEncryption
         RSA Public Key: (1024 bit)
            Modulus (1024 bit): <removed for brevity>
Exponent: 65537 (0x10001)
      X509v3 extensions:
         X509v3 Basic Constraints: critical
            CA:TRUE
         Netscape Cert Type: critical
            SSL CA
         X509v3 Extended Key Usage: critical
            TLS Web Server Authentication, TLS Web Client
Authentication, E-mail Protection, Time Stamping, Microsoft Individual
Code Signing, Microsoft Commercial Code Signing, Microsoft Trust List
Signing, Microsoft Server Gated Crypto, Microsoft Encrypted File
System, Netscape Server Gated Crypto
         X509v3 Key Usage:
            Certificate Sign, CRL Sign
         X509v3 Subject Key Identifier:

20:08:7C:A7:5F:DF:9C:39:E2:7A:EF:C0:9E:FD:12:1D:22:EF:9E:15
   Signature Algorithm: sha1WithRSAEncryption
   <removed for brevity>
```

Figure 1. A fake CA certificate.

that *sslsniff*, our invisible SSL proxy tool [5], is set to accept traffic on.

The third step of the attack is to perform the actual MITM attack to gain visibility of the users traffic. We accomplished this task with the help of the *ettercap* tool [6]. It allows us to reroute a victims traffic through our computer using ARP requests in a way that will be totally invisible to the average user. The main way a victim of this attack could check if this were occurring would be to check their ARP table. In order to spot an active attack, they would need to know the MAC address of the router, and notice that the current one listed is incorrect. Given that we are assuming our victim is on a mobile device, it is exceedingly unlikely that this step would be noticed by the victim, especially since there is no built method in iOS or Android to check this. This is one area of possible continued research, methods could be developed to protect a user if a MAC address suddenly changes for the gateway on a know Wi-Fi network. Armed with this knowledge, the user may be able to make an intelligent decision to cease communicating or at least verify settings to ensure an attack is not underway.

The final step of this attack is to set up the proxy server to hijack all of the victims SSL traffic. The *sslsniff* tool listens on the forwarded port chosen earlier for victim traffic. When the victim attempts to make a secure connection, our proxy silently intercepts the request and replies with a certificate dynamically generated by *sslsniff* for the intended website, signed by our fake CA certificate. The secure tunnel is now broken at our computer so that we can see all traffic in plain text, and all transactions are logged to a text file. This attack

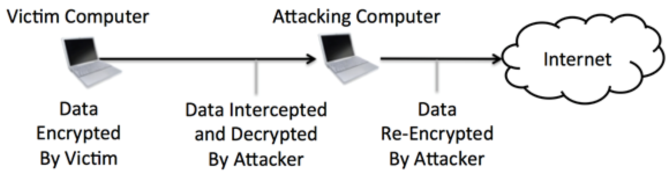

Victim Computer
Attacking Computer
Internet
Data Encrypted By Victim
Data Intercepted and Decrypted By Attacker
Data Re-Encrypted By Attacker

Figure 2. Invisible SSL proxy traffic path.

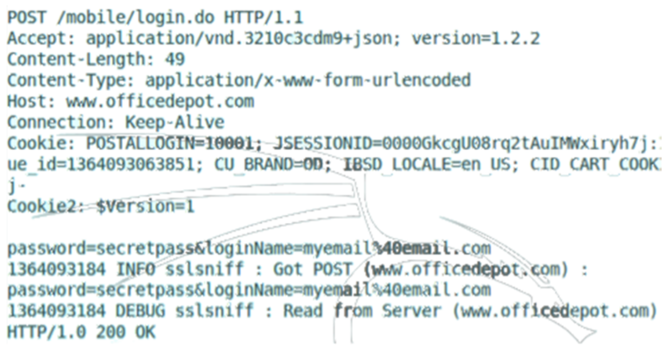
will only succeed if, and only if our victim application does not mind that the certificate it receives was signed by an untrusted CA. An illustration of the attack traffic path can be found in Fig. 2.

In our tests, once the attack was set up, applications from both iOS and Android were tested. We attempted this attack on 24 iOS 6 applications and 41 Android 4.2 applications, all using the most recent versions of the application that were released as of 5/3/2013. In terms of iOS applications, we were unable to retrieve any credentials from all that were tested. Every attempt was met with errors of various types indicating the connection was not safe, or that there had been some sort of general network error.

Android applications however, were a different story. Although many applications rejected our attempts, surprisingly, there were multiple applications from well-known and established companies that had no problem continuing traffic with our invalid certificates. There was no indication of an error presented to the user, and the applications continued to operate as if nothing was wrong. Figure 3 shows an example of this attack leading to the captures of credentials from the Office Depot Android application.

B. Manual Proxy Setup

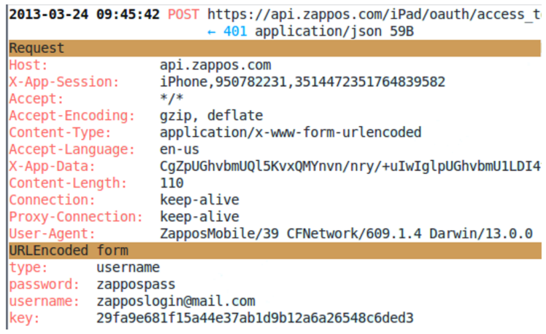
One interesting situation we found was that of when a manual proxy setup was entered for a Wi-Fi network. On iOS, for example, we observed that some applications would continue to refuse to connect. This could be due to the application rejecting the certificate, or due to the application just plain being incompatible with proxies, there was no sure way to tell. However some applications would, in this type of setup, begin to accept the certificates signed by our untrusted CA.



```
POST /mobile/login.do HTTP/1.1
Accept: application/vnd.3210c3cdm9+json; version=1.2.2
Content-Length: 49
Content-Type: application/x-www-form-urlencoded
Host: www.officedepot.com
Connection: Keep-Alive
Cookie: POSTALLOGIN=10001; JSESSIONID=0000GkcgU08rq2tAuIMWxiryh7j:
ue_id=1364093063851; CU_BRAND=OD; IBSD_LOCALE=en_US; CID_CART_COOK
j-
Cookie2: $Version=1

password=secretpass&loginName=myemail%40email.com
1364093184 INFO sslsniff : Got POST (www.officedepot.com) :
password=secretpass&loginName=myemail%40email.com
1364093184 DEBUG sslsniff : Read from Server (www.officedepot.com)
HTTP/1.0 200 OK
```

Figure 3. Office Depot application traffic showing exposed user credentials.



```
2013-03-24 09:45:42 POST https://api.zappos.com/iPad/oauth/access_t
                         ← 401 application/json 59B
Request
Host:                  api.zappos.com
X-App-Session:         iPhone,950782231,3514472351764839582
Accept:                */*
Accept-Encoding:       gzip, deflate
Content-Type:          application/x-www-form-urlencoded
Accept-Language:       en-us
X-App-Data:            CgZpUGhvbmUQl5KvxQMYnvn/nry/+uIwIglpUGhvbmU1LDI4
Content-Length:        110
Connection:            keep-alive
Proxy-Connection:      keep-alive
User-Agent:            ZapposMobile/39 CFNetwork/609.1.4 Darwin/13.0.0
URLEncoded form
type:          username
password:      zappospass
username:      zapposlogin@mail.com
key:           29fa9e681f15a44e37ab1d9b12a6a26548c6ded3
```

Figure 4. Zappos API credentials captured with mitmproxy through manual network proxy setup.

This sort of proxy configuration could potentially exist if a user is tricked into installing a device profile preloaded with proxy server settings. A file containing the profile could be generated by a malicious user and distributed to unknowing victims. Once installed, all traffic from proxy supporting applications on that device would be routed through the specified proxy. In the proxy this data is unencrypted and could be analyzed and used with ill intent.

One example of this behavior is with the Zappos application. Zappos iOS app would give a network error during the MITM attack using the invisible proxy. However, the application gave no complaints when the program *mitmproxy* [1] was used to proxy the traffic via a typical SOCKS proxy connection. A screenshot of captured example credentials is shown in Fig. 4.

This behavior isnt necessarily a vulnerability per se, but it is interesting to see that some applications will accept proxied connections and some will continue to refuse. A well-informed user utilizing a proxied connection after all should know and expect that their SSL connections must be broken and encrypted again by the proxy. Assuming this, the user should also be aware that they are accepting the risk that the proxy could be maliciously modifying their traffic in transit, or store their credentials, if those in charge desired to do so.

Our research showed that there was no easy way to predict which applications were likely to work with a proxy. Doing so is likely to be debated in app design because on one hand, an authors application will not work under proxy situations, leaving the users stranded if this is the only type of connection available. On the other hand, it ensures that no matter what, a users credentials will not be able to be stolen by any malicious users operating the proxy. An application creator who wanted to enable proxy use while still holding user data safe could mitigate this problem entirely by using another layer of encryption at the application data level. This would allow a user to send credentials through a proxy and continue to prevent them from being exposed or modified in transit.

V. ANALYSIS

Of the 24 iOS applications that were analyzed, the only vulnerability we ran into was that of flyertalk.coms application, which sent login info in plain text via a regular HTTP POST. No applications were found that were willing to accept our rouge CA signed certificates, a full list of tested applications can be found in the Appendix. Searching as to why this may be turned up one possible answer, and that is that it appears that a class that is commonly used by iOS developers to establish https connections hides the variable that controls the ability to accept invalid certificates.

The *allowAnyHTTPSCertificateForHost* parameter in the *NSURLRequest* class[1] is this hidden Boolean value, and appears to be in control of iOSs willingness to accept untrusted CA signed certificates. The variable is not mentioned anywhere on the developer API reference page for the *NSURLRequest* class, and it seems plausible that due to this, many developers may never know of its existence and thus leave it at its default setting, causing iOS to reject certificates signed by untrusted CAs.

Android applications, however, showed a different response to our attacks, of the 41 applications tested, 11 were willing to accept invalid certificates, and 3 didnt use SSL at all (one of them being the application from flyertalk.com that also wasnt protected on iOS, the other 2 were a option trading program and frequent flyer point tracker.) The full list of Android applications tested and their security findings can also be found in the Appendix. Of the 11 applications that were vulnerable, three were financial applications for stock and option trading from smaller companies, four were popular name commercial brands, one was a popular travel deals site, one was a coffee rewards card number aggregator and two were small brand ticket brokers that did not expose user data due to a lack of login functionality, but nonetheless were willing to proceed with our certificate.

One interesting thing that was noted about applications that were found to be vulnerable to SSL attacks on Android is that the same companys iOS application was not. The list of these apps is as follows: Zappos, Newegg, OfficeDepot, Staples, TravelZoo, AnyOption, EZTrader, GlobalOption, and TradeRush. Of these applications, EZTrader and GlobalOption had applications that seemed to have identical Android GUIs and traffic structure. They also seemed to match on iOS as well as seen in Fig. 5.

This leads us to believe that one of the main reasons for insecurity may be the use of application building frameworks that default to poor security selections. This also makes us believe that even though some framework was behind both of these applications iOS versions, whichever framework was chosen for iOS must not have the SSL security issue.

[1] $https://developer.apple.com/library/ios/\#documentation/Cocoa/Reference/Foundation/Classes/NSURLRequest_Class/Reference/Reference.html\#//apple_ref/doc/uid/TP40003762.$

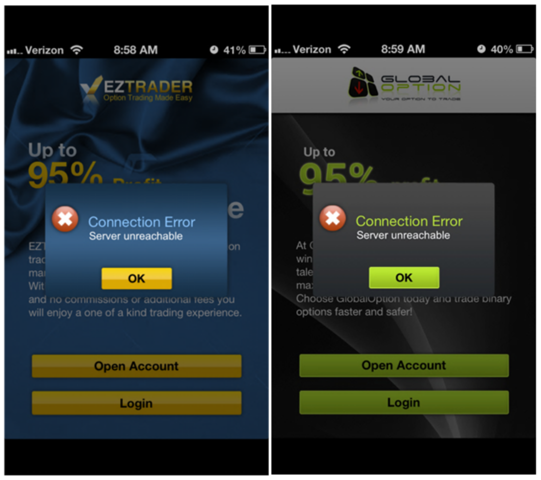


Figure 5. Identical GUIs on iOS for two trading programs.

For other Android applications, we did not find similar looking matches, but due to the small number of Android applications we were able to test, it is entirely plausible that many applications may exist that match those too. This is another area that deserves further research, and if particular Android frameworks could be identified that are at fault for these errors, it may be a very simple fix to ensure that future applications developed with them do not default to accepting invalid certificates.

Analyzing the Android API that controls certificate validation shows a different situation than that of iOS documentation. The Android java class *HttpsURLConnection* controls https connections and with it comes the ability to assign an application specific *TrustManager* that implements a *KeyStore* that can accept self-signed certificates. In addition, in the documentation, there is mention of an *X509TrustManager* and *CertPathValidator* class that, when implemented incorrectly could allow the accepting of all certificates[2]. Our research seems to back up the case that at least some Android application authors are indeed implementing this class incorrectly, allowing certificates not signed by a trusted CA to pass, and therefore allowing traffic to continue and potentially become exposed.

VI. Conclusions

Our research into the topic of mobile application security points to several conclusions. One is that some Android applications are most certainly vulnerable to an invisible proxy MITM attack. The users could be at any public access point, have someone on the same network perform the identical attacks we used for our research, and without the victims knowledge or any warning from their device whatsoever, have their credentials stolen. This is a very real danger and any user who plans on using applications that require a login or personal information on a public network may want to utilize a VPN to ensure their credentials stay safe.

The good news however, is that most mobile applications, in our experience, protect a user from this attack. Generally, Android applications from big names such as Facebook, Twitter, Amazon, and everything from Google itself will deal with certificates correctly and not allow credentials to leave the device without a trusted CA signed certificate. It was also good news that all applications that we tested on iOS have rejected any network connections while under invisible proxies. Differences in accessibility to the classes that drive the acceptance or rejection of untrusted certificates between the two operating systems seem to be one possible reason for this discrepancy, but investigation of this is a suggested area of continued research. Identifying the application creation frameworks that we believe were used to create vulnerable Android applications and changing their default settings is something that may go a long way in securing applications on the Android platform.

Developers on these platforms need to be conscious and knowledgeable in this realm. From our analysis of the iOS and Android API, we speculate that the difference in ease of use in the Android development framework may be a factor in the difference in security posture. The multiple settings and little guidance on how to properly implement a secure connection could be confusing for a novice programmer. The advice given in many public forums is to turn SSL certificate verification off during development instead of going through the trouble of obtaining a legitimate certificate, and later to turn those safeguards on only when the project is nearing completion. In a long development cycle this could be forgotten or overlooked, especially with critical deadlines and launch dates approaching.

Ultimately, mobile application security is a new field and it comes as no surprise that some applications come with poor security. Identifying and fixing these applications however should be made a priority for researchers and developers alike. Finding the root cause for the development of applications that accept untrusted CA signed certificates is likely not a difficult task and if frameworks are to blame, getting the poor default choices changed is one simple action that would result in a significant increase in application security across the board.

[2] $http://developer.android.com/reference/javax/net/ssl/HttpsURLConnection.html.$

References

[1] A. Cortesi, “mitmproxy: a man-in-the-middle proxy,” *http://mitmproxy.org/*, 2013.

[2] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben, “Why eve and mallory love android: An analysis of android ssl (in) security,” in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 50–61.

[3] S. Gallagher, “Mobile app security: Always keep the back door locked,” *http://arstechnica.com/security/2013/02/mobile-app-security-always-keep-the-back-door-locked/*, 2013.

[4] P. Hannay, “Exchanging demands,” *Blackhat USA 2012*, 2012.

[5] M. Marlinspike, “sslsniff tool,” *http://www.thoughtcrime.org/software/sslsniff/*, 2013.

[6] A. Ornaghi and M. Valleri, “Ettercap,” *http://ettercap.github.io/ettercap/*, 2013.
[7] E. Rescorla, *SSL and TLS: designing and building secure systems*. Addison-Wesley Reading, 2001, vol. 1.

Appendix

List of Applications and Vulnerability Status

A. *iOS 6 Applications Tested Secure from SSL Attack*

Alien Blue
AnyOption
Craigslist
EZTrader
Facebook
Flyertalk* (does not use SSL)
GlobalOption
Goodreader
Newegg
OfficeDepot
RedBox
Snapchat
Splitwise
Spotify
Staples
ToodleDo
TradeKing
TradeRush
Travelzoo
Twitter
Verizon Wireless
Waze
Wordpress
Zappos

B. *Android 4.2 Applications Tested Secure from SSL Attack*

Amazon
CloudMagic
Currents
Dominos Pizza
Dropbox
EasyJet
Ebay
Expedia
Facebook
Google Play Store
Groupon
Kik Messenger
Living Social
Mint
Netflix
Pandora
Pinterest
Qatar Airways
Skype
Snapchat
Southwest Airlines
Splashtop
Twitter
United Airlines
Walmart
Zinio

C. *Android Applications Vulnerable to MitM SSL Proxy*

EZTrader
GlobalOption
MyCoffeeCard
Newegg
Office Depot
PeakSeats
Staples
Stub Nut
TradeRush
TravelZoo
Zappos

D. *Android Applications No use of SSL*

AnyOption
Flyertalk
MilePoint