Group: Ziling Zhou (zz2320), Oliver Xu (ox207), Jeremy Rivera (jr4222), Yida Zhou (yz4499), Jonathan Tang (jt3135)

Backdoored Android Apps

Abstract- *Software developers frequently use many third-party libraries in projects without verifying the integrity of the plugins they decide to use. Because the plugins are outside the scope of developers, they may employ unsecure methods that compromise the security of their whole project. In order to minimize security loopholes in code outside the codebase of the app developer, we suggested a Google API/Library system that aims to limit malicious and poorly written plugins to be used in the app store. The system ensures that data sent from apps cannot be sniffed by third party attackers. It seeks to minimize exploits in code that handles potentially dangerous actions. Furthermore, we suggest certain coding practices (forcing developers to use HTTPS) to prevent leakage of private user data.*

I. Introduction

Most applications require network capabilities in present day. Everytime a packet is sent, there is an opportunity for an attacker to obtain that data. Android apps, in particular, use many JAR libraries that often handle critical tasks. These libraries can contain vulnerabilities that can be exploited by attackers. For example, a poorly designed application may send login information in plain text to a server (e.g. ESPN [1]). An attacker can sniff the packets sent by the app and have access to the user's login information; or, in the case of Igexin, attackers were able to compromise their server and send *spyware* to users who downloaded apps utilizing Igexin. Hence, all networked data should be protected from attack vectors on the internet. The solution to this problem is a Google library system with a whitelist of plugins that developers have to use in order for their app to be hosted in the Play Store. This means all attack vectors that Google are aware of including data sniffing, malicious code injections, etc. can be mitigated once they are discovered.

Traditionally, Google and other app distributors have relied on security researchers to find exploits and help propose solutions in active apps. For example, Igexin was discovered by Lookout, an Android security company that produces antivirus solutions for Android [2]. Doing so usually meant a vulnerability has already been exploited and many victims have been attacked. Google has employed a bounty system for vulnerabilities in android. Researchers who discover exploits can turn in their research to Google for a monetary reward (up to $200,000) [3]. For example, Guang Gong from Qihoo 360 Technology was rewarded $112,500 for his discovery of an exploit chain that could compromise Pixel mobile devices [4].

In this paper, we discussed an efficient solution to solving the unsecure transmission of data problem in our proposed Google library solution. By using HTTPS and modern cryptology in our whitelisted methods in our library, we will effectively force app developers to practice safe data transmission techniques. Doing this, we can drastically protect the integrity of apps on the play store.

II. Background

Security in mobile devices has been a chief concern of developers since smartphones gained massive popularity. A simple naive solution to ensure the integrity of apps on the market would be for Google to hire engineers and verify the code of every app released. In practice, this is impossible and cost inefficient. The sheer amount of apps uploaded to the store per day would overwhelm the engineers in a matter of seconds. Engineers are usually the most expensive resource of any business. Therefore, any solution that involves a human verifying code would be impractical. Over time, most Android exploits have ended up being discovered by third-party researchers, or any security-minded individuals who happen to stumble across an exploit. The security of mobile apps used by billions of people worldwide depends solely on the abilities of those in the security community.

Insecure data transmission has been one of the top contenders in security vulnerability since the inception of the internet [5]. In the beginning, sensitive data was sent in plaintext over the internet. Alice's message to Bob could be intercepted by Eve with little difficulty. It wasn't until 1995 that Netscape included SSL in their browser. Though SSL 2.0 had many vulnerabilities, it was still a major improvement in internet security. A newer version

(SSL 3.0) was designed and released a year later. Today, TLS is the underlying transport layer that HTTPS utilizes[6].

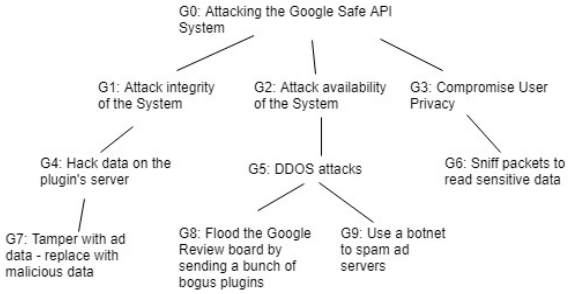
Despite HTTPS being easier to implement than ever before with services such as Let's Encrypt, the lack of encryption is still one of the leading causes of vulnerabilities. This carelessness was translated over to app development. Insecure cryptography and insecure authorization are some of the leading vulnerabilities in mobile apps today [7].

Our solution seeks to encourage good development practices by integrating modern standard cryptography such as HTTPS into mobile development.

III. Threat Model

In the previous paper, we created an attack tree for our Google Library System. Our system is more efficient and secure than the previous paradigm of waiting for a report to be filed by a third-party security researcher.

Let's review the tree and look into how plugins such as Igexin were attacked and how our system would address these problems.



Threat	**Solution Proposed**
G6: Sniff packets	Our plugin forces HTTPS and other modern cryptography. Furthermore, our system has a certified CA that developers can use with minimal extra charge in case their systems do not support HTTPS. HTTPS encrypts data sent so anyone who obtains the data cannot read the contents.
G5: DDoS Attacks	We will implement a review system that utilizes a state-of-the-art DDoS mitigation system. We will employ techniques such as a blacklisting of known compromised IPs, filtering and gateway protection layers. Most DDoS attacks will be unsuccessful because our protection service will have already blacklisted their IP (from suspicious activity in our protection layers).
G7: Tamper with ad data-replace with malicious data	Our proposed solution is to create an authentication system to make sure ads are coming from the correct servers by our registered vendors. Furthermore, all of our vendors will be subject to regular audits of their security methods to make sure they are not easily hacked. If we recognize that a service has been compromised, we will blacklist the service from all users until actions are taken to address the problem.

We believe by deploying our system we can change the Android ecosystem to become more secure.

IV. Design

The proposed solution will consist of a library and a review system built to verify plugins and apps. The verification system will ensure the following security properties: (1) privacy of data, (2) integrity of data, (3) availability of the verification system.

The system will have the following core components:

1. *Google Android Library*: provides plugin developers with a safe and secure way to access network data and other critical tasks; will force HTTPS to enforce data privacy

2. *Review System for Plugins*: provides plugin developers a way to register and submit their plugins to be approved for the Play Store; ensures that plugin developers are held accountable for the quality of their code

3. *Verification System for apps*: screens Android apps and makes sure apps in the play store only use approved and verified plugins, reducing the risk of having a plugin with a vulnerability

Figure 1: Plugin developer using the Google Library system.

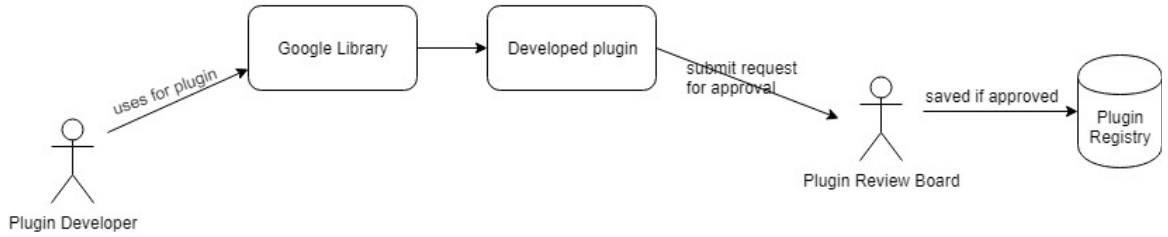
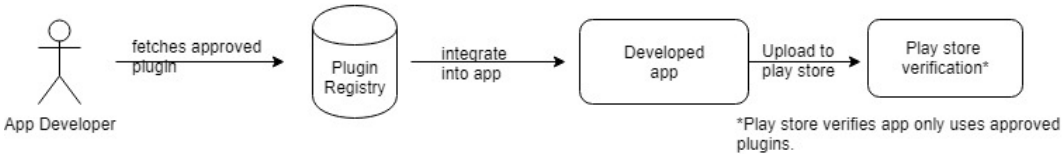


Figure 2: App developer using the approved plugin library.



In Figure 1, the plugin developer pulls the Google Library into their project for use. After their plugin is developed, they submit a request for approval to the Google Plugin Review Board. The review board checks the plugin to make sure there are no obvious security flaws. Once approved, the plugin is uploaded to the Plugin Registry, where app developers can utilize it for their projects.

In Figure 2, the app developer browses the Plugin Registry for a suitable plugin and integrates it into their app. Once the project is finished, the app will be uploaded to the play store, where an automated system will verify that the app only uses approved plugins from the Plugin Registry.

V. Implementation

The plugin will be built in SDKs in major mobile development languages such as Objective-C, Java, Javascript, and others. Some of the major functions that will be implemented in this library is as follows:

- fetchApi(): a function that fetches JSON data from an API; it will parse text and not accept data not in JSON form
- fetchUri(): fetches images, videos and links; it will parse through and will not accept data in any other form
- fetchHtml(): fetches HTML and parses what is fetched to make sure it is valid
- verifyRegistry(): verifies that the plugin developer is registered with the Plugin Registry; returns information about registration info (registration date, registration expiration date), etc.; returns “not registered” if new user

The library will also verify the integrity of received data by parsing through and checking the data to make sure that it is not corrupt. This will be automated using machine learning algorithms to detect suspicious behavior.

The technological engineers hired to be part of the review board will parse submitted plugins to make sure that there is not obvious malicious behavior with the plugin. If the plugin is approved, it will be inserted into the registry. The plugin developer will be given the details about his registration, and verifyRegistry() will now return these details.

The plugin registry will now be available through the regular distribution channels for app developers, such as NPM, Nuget, and Maven. The app developer will create an app and submit it to the play store. The system checks the app manifest and makes sure only approved plugins are used.

VI. Evaluation

Pros: Comparing our system implementation with the previous ones, we created a white-list based security for mobile plugins. All the plugins used in mobile apps are published to the Play Store which have been reviewed and put into the white-list, so that the plugins are more secure than those used before our systems established. At the same time, our system alleviates security burdens from mobile app developers since they no longer need to care much about the security of plugins they are using in their apps, which makes it easier for their apps to meet certain security standards and also easier for app reviewers to review the security of those mobile apps.

Cons: If plugin servers are compromised, our system can not prevent it from happening. However, if this happens, our system can easily get the list of apps using this plugin, and notify the developers and users using these apps in the first time. What's more, our system will bring higher cost to the mobile market owner (e.g. Google, Apple) because we need a team of engineer review board and it costs a lot, but it will prevent conspicuous security bugs and makes the mobile developing environment more secure at the same. On the other hand, there is technological cost as well but will not increase too much, since mature apps market(e.g. Google Play) has existed stably for a long time, and all we need to do is to duplicate the similar technique implementation and policy, using some additional servers and databases to store and publish the approved plugins.

VII. Related Solutions

- Google AdMob

Google AdMob[8] is a ads plugin for iOS and also a build-in plugin for Android that allows developers to monetize their apps. With AdMob, developers can easily implement three kinds of advertisements:

Banner

Interstitial

Rewarded Video

Take Banner ads as example, to implement banner ads, several lines of code[9] will work (See Appendix.A). GoogleMobileAds.Api will handle all the data transferring, with only three parameters for the object *BannerView*: UnityId, AdsSize and AdsPosition. And then, developers can define their own callback functions for successfully loaded ads, clicking events on ads, and returning to apps from ads, etc.

- Why Google AdMob is secure?

For Android, since Google AdMob is an official ads build-in plugin, and there are over one million users and over one million advertisers using Google AdMob, Google should be very responsible for them. Google's reputation and confidence are one part of the security users can trust. Also, HTTPS is used in data transportation in Google AdMob for security[10].

For iOS, ATS[11] is introduced since iOS 9 to enforce app securities, and App Transport Security has blocked a cleartext HTTP (http://) resource load since it is insecure. These restrictions can make sure that the data transport is secure under HTTPS.

VIII. Conclusion

In this paper we discuss bringing over the idea behind HTTPS to mobile applications. The idea is an extension of the Google AdMob in which our idea's implementation would limit the possibility of malicious activity through a registry of whitelisted plugins. These whitelisted plugins can then be used by an app developers who wish to use the google library. Using a plugin that isn't part of the registry will result in the application being denied. The process allows for both human and computer automated verification of plugins and apps. While cost will increase to maintain security, the security standards of the store will increase dramatically as users will know app developers will be using plugins that are approved personally by a Google Plugin Review Board.

References

[1] “Why No HTTPS?” , Troy Hunt, 1 December 2018.
https://whynohttps.com/
[2] “Igexin advertising network put user privacy at risk”, Adam Bauer, Christoph Hebeisen, 1 December 2018
https://blog.lookout.com/igexin-malicious-sdk
[3] “Android Security Rewards Program Rules”, Google, 1 December 2018
https://www.google.com/about/appsecurity/android-rewards/
[4] “Google awards researcher over $110,000 for Android exploit chain”, Charlie Osborne, 1 December 2018
https://www.zdnet.com/article/google-awards-researcher-over-110000-for-android-exploit-chain/
[5] “OWASP Top Ten Project”, OWASP, 1 December 2018
https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf
[6] “The Evolution of SSL and TLS”, Elizabeth Baer, 1 December 2018
https://www.digicert.com/blog/evolution-of-ssl/
[7] “Mobile Top 10 2016-Top 10”, OWASP, 1 December 2018
https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10
[8] “Google AdMob”
https://developers.google.com/admob/
[9] “A minimal implementation of all ad formats”
https://github.com/googleads/googleads-mobile-unity/blob/master/samples/HelloWorld/Assets/Scripts/GoogleMobileAdsDemoScript.cs
[10] “Serve ads on apps securely over HTTPS”
https://support.google.com/admanager/answer/6118579?hl=en
[11] “Information Property List Key Reference”
https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKeyReference/Articles/CocoaKeys.html

Appendix.A Google AdMob Sample Code

```
...
using GoogleMobileAds.Api;
...
public class GoogleMobileAdsDemoScript : MonoBehaviour
{
    private BannerView bannerView;

    public void Start()
    {
        this.RequestBanner();
    }

    private void RequestBanner()
    {
        #if UNITY_ANDROID
            string adUnitId = "ca-app-pub-3940256099942544/6300978111";
        #elif UNITY_IPHONE
            string adUnitId = "ca-app-pub-3940256099942544/2934735716";
        #else
            string adUnitId = "unexpected_platform";
        #endif
```

```
        bannerView = new BannerView(adUnitId, AdSize.Banner, AdPosition.Top);
        // Called when an ad request has successfully loaded.
        bannerView.OnAdLoaded += HandleOnAdLoaded;
        // Called when an ad request failed to load.
        bannerView.OnAdFailedToLoad += HandleOnAdFailedToLoad;
        // Called when an ad is clicked.
        bannerView.OnAdOpening += HandleOnAdOpened;
        // Called when the user returned from the app after an ad click.
        bannerView.OnAdClosed += HandleOnAdClosed;
        // Called when the ad click caused the user to leave the application.
        bannerView.OnAdLeavingApplication += HandleOnAdLeavingApplication;
        // Create an empty ad request.
        AdRequest request = new AdRequest.Builder().Build();
        // Load the banner with the request.
        bannerView.LoadAd(request);
    }
    ...
}
```