Automatically Detecting SSL Error-Handling Vulnerabilities in Hybrid Mobile Web Apps

Chaoshun Zuo
Shandong University
cszuo2013@gmail.com

Jianliang Wu
Shandong University
lucuswu@gmail.com

Shanqing Guo
Shandong University
guoshanqing@sdu.edu.cn

ABSTRACT

Today, there are many hybrid apps in which both native Android app UI and WebView UI are used. To protect the security and privacy of the communications, these hybrid apps all use HTTPS by WebView, a key component in modern web browser. In this paper, we show there is another type of SSL vulnerability that stems from the error-handling code in the hybrid mobile web apps. At a high level, this error-handling code should have stopped the communication but it still proceeds regardless of certificate errors, thereby leading to the MITM attacks. To automatically identify these vulnerable apps, we present a hybrid approach that combines both static analysis and dynamic analysis. We have implemented our approach and evaluated with 13,820 real world mobile web apps from a third party market, of which 645 are confirmed truly vulnerable, with an average overhead of 60.8 seconds per app.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Android Security, HTTPS, SSL, WebView

1. INTRODUCTION

Increasingly, there are hybrid apps that combine both native Android UI and WebView UI, because of the easier development and lower maintaining complexity. Specifically, mobile web apps use WebViews to present web pages and communicate with web servers. Some web pages may transfer sensitive information, like user name and password, to a server, which causes that the communication should be protected. For this reason, they all use HTTPS connections instead of HTTP connections. Under normal circumstances, the attackers couldn't attack HTTPS connections even if they sniffed the network traffic unless they have the cryptographic keys. However, developers' incorrect implementation of HTTPS in WebView can allow Android WebView to present a web page with illegal certificate, which can thus be attacked by Man-in-the-middle [1, 2] or phishing attacks.

ASIA CCS'15, April 14 - 17, 2015, Singapore, Singapore

ACM 978-1-4503-3245-3/15/04 ...$15.00.
http://dx.doi.org/10.1145/2714576.2714583.

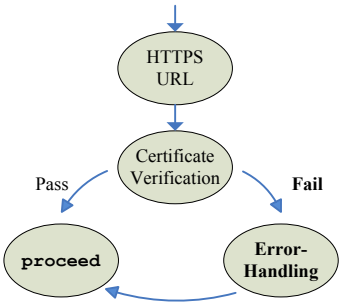

Figure 1: The State Machine of error handling process.

Figure 1 shows how this vulnerability happened. When an app opens an HTTPS web page with an illegal certificate, the app passes the HTTPS URL to Android to verify the certificate. With an illegal certificate Android will get a verification failure, then it will call error handling procedure implemented by the developer. Often times this error handler code just ignores the error and calls `proceed` to show the HTTPS web page (even though the certificate is illegal). This is a particular type of implementation vulnerability we aim to find in this paper.

More specifically, it's possible to analyze this vulnerability manually for a particular app. However, it is impractical to detect this vulnerability on a large scale, given the huge amount of such apps in the market. Meanwhile, unlike native apps, mobile web apps bring new challenges because we have to test not only the normal native app activities but also the web pages. For static analysis, the existing tools such as androguard [3] are not suitable to detect this vulnerability because they cannot track variables. In addition, we can't determine whether an app is vulnerable or not by static analysis only (because it is often an over approximation, leading to false positives). Dynamic analysis is needed to verify whether the WebView would eventually load an HTTPS page and its error handling is vulnerable.

As such, we have designed a new system to automatically identify these vulnerable apps. Our system consists of both static analysis and dynamic analysis. In particular, we first employ static analysis to determine whether these apps are

potential vulnerable or not. If so, the potential vulnerable apps will be further analyzed through our dynamic analysis, which is guided by the static analysis information to drive the native Android UI as well as the WebView UI to trigger the vulnerability. In summary, this paper makes the following contributions:

- We have discovered a new type of SSL vulnerability which could lead to insecure WebView HTTPS connection.
- We have designed a hybrid Android web app test framework using multi emulators. This framework contains both static analysis and dynamic analysis. It can install and run a mobile web app automatically without any user involvement. Besides, not only could the system stimulate jumps between Activities but also it is able to drive jumps between web pages within WebView.
- We have implemented our framework and tested with 13,820 apps collected in July 2014. Experimental results show that our static analysis found 1,360 potential vulnerable apps and our dynamic analysis confirmed that 645 of them are vulnerable.

2. SYSTEM OVERVIEW

2.1 Problem Statement

For hybrid mobile web apps, when an HTTPS URL is passed to WebView, it will first verify the certificate of the HTTPS server: if passed WebView will show the page. If verification failure occurs and the app has rewritten the error handling process, WebView will pass the error handling process to the app and wait for the result. Once the error handling passed to the app, it will handle the error in its own way including ignoring the error and proceed and return the result to WebView. If error handling process has not been rewritten, WebView would shield the page directly. However, programmers often rewrite the error handling process. It's obvious that this is a serious security problem, especially for apps that always use HTTPS to transfer sensitive information such as the login information, user information, payment information, authorization information, etc. With this vulnerability unfixed, attackers could easily get all these information by MITM attack.

2.2 Challenges and Solutions

2.2.1 *Is the potential vulnerable code reachable?*

To identify the app is vulnerable or not, we have to make sure it contains potential vulnerable code. We assume the class inherits from `WebViewClient` which overrides the error handling method (i.e. `onReceivedSslError`) and has the ignore code is potential vulnerable. Any app without this kind of code is invulnerable. It's not easy to make sure if `onReceivedSslError` is reachable because it's called by system callbacks rather than called directly. We locate the method call `setWebViewClient` and find out whether a potential vulnerable `WebViewClient` has been registered or not. If yes, then we find the Activity that loads the WebView which registers this WebViewClient. So we consider this Activity as a target Activity. This helps us to make sure this potential vulnerable code is reachable when the Activity is reachable.

2.2.2 *How to record Activity jump relations with trigger events?*

To confirm if potential app is vulnerable we need to jump to the target Activity from launcher Activity by triggering related events which have been recorded during static analysis. To fulfill this we build an ACG[4] based on which a path from launcher Activity to target Activity is found, which could guide the dynamic test. ACG is a directed Activity Call Graph of which vertexes represents Activities. And we put information on the edge because we need to know what event triggers the jump from one Activity to another.

Vertexes are not hard to find but edges (i.e. how to find the view and event) are not easy to add. We take view and the event triggered by the view that cause Activity jump as an edge. Here goes how we find edges. First, we find all the methods that could cause activity jump and locate these functions in MCG (Method Call Graph). By traveling within MCG we can find method that causes activity jump and which activity jump to. Then we locate event method (such as `onClick`) the method belongs to. This view which owns the event method and the event would be the edge. With ACG, we are able to find a path on which a series of trigger events are recorded from launcher Activity to target WebView.

2.2.3 *How to simulate human operations to both native Android UI and WebView UI ?*

Manual analysis is enough for a particular app, but for large dataset, it's impossible. We need to make it possible to detect automatically thus making large scale analysis possible. Along the process from launcher Activity to load an illegal page, human operations are needed. To make it automatic, we need to simulate human operations. To mitigate this, we have made our own test system Android Tester by modifying the Android framework and we use Robotium developed a general test script app for the target apps . With this framework, we could know which Activity is active, which views are on this Activity, their IDs and how to trigger one specific event all of which other test tools cannot do. Once we jumped to the target Activity, how to jump to the HTTPS page within WebView if the default page is not an HTTPS page? Here in test script we adopt a strategy like a crawler. We first load the default page and extract all the links from the initial page and load every link and extract links again until we have found an HTTPS link or the crawl layer depth is up to 3.

2.3 System Overview

We present the overview of our system in Figure 2. Our system takes APK file as input, and outputs the app is vulnerable or not. First, our system carries out a static analysis to determine if apps are potential vulnerable. Second, we need to dynamically run them and to confirm if it's really vulnerable, which is requisite because of the difficulty of validness verification of the self-implemented error handling process and the uncertainty that if the WebView would load an HTTPS page that cannot be solved during static analysis. So we need to build the app's ACG for dynamic analyzing.

Dynamic analysis starts with installing and running the app on emulator. Then our system would find a path from launcher Activity to target Activity. When the path is found, the system triggers an event and jumps to next Activity till the target Activity. After each jump, our system

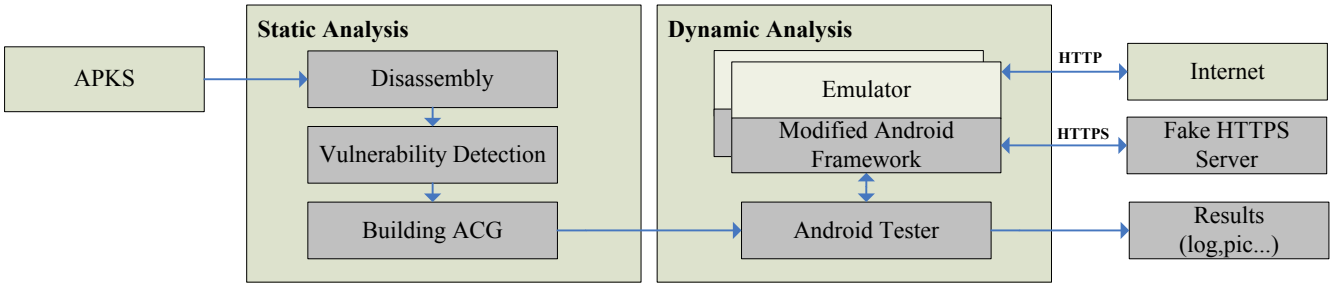

Figure 2: System Overview

calculates the path again in order to avoid the situation that the path found earlier is not applicable because of some extra conditions. During this process, we may not be able to find efficacious path, because there are some views that need conditions to appear (e.g. some app may have an advanced panel that would appear only under advanced mode). Under this condition we jump to target activity directly. At the same time, we built an attack environment (shown in Figure 2) which could redirect HTTPS connections to our fake server who has an illegal certificate. We modified Android framework to print log once an illegal page are presented. At last our system will generate log information and tell us which app is vulnerable and what URL the app has visited.

3. DETAILED DESIGN

3.1 Static Analysis Module

Static Detection. We decompile APK into Smali file by apktool[5]. The static analysis starts once the decompilation process finished. We scan all the smali files to find if there is any class inherits from the `WebViewClient` class. If not found, then we consider the app is free from this vulnerability. We collect all the classes inherit from `WebViewClient` and check them one by one to determine whether they have overridden the method `onReceivedSslError` which would be called by system callback when HTTPS certificate verification failure occurs. The app could trust illegal pages by overriding this method with a rather weak one. According to our research, most app chooses to trust all certificate, and some of them use a simple way to handle this error and the others choose to abort the page.

We have identified three common operations that app choose to perform in `onReceivedSslError`.First, They trust all certificate and returns proceed signal. Obviously this is vulnerable if it's reachable. Second, They reject and return cancel signal. It's free from this vulnerability. Third, They verify certificate by itself. Some of them check hostnames, and some of them use complex algorithm, and some of them even show a dialog for the user to make a choice. We can't determine if the app is vulnerable or not by static analysis, so we need further detect by dynamic analysis.

Build MCG We have seen some apps with unreachable code and most of them are for testing. So we have to make sure the overridden method `onReceivedSslError` is reachable. We would build a Method Call Graph to fulfill this. It is a directed graph representing the calling relationship among methods. Each node in MCG represents one method and an edge from Method `A` to Method `B` means that Method `A` could call Method `B` directly. We employed a conservative approach like[6] to handle virtual method and interfaces while building MCG. A class hierarchy was maintained during the analysis process and all possible assignable classes would be considered when an ambiguous reference occurred.

Algorithm 1 Build Activity Call Graph

Input:
MCG : Method Call Graph
ms : Temporary storage of methods
Output: ACG : Activity Call Graph
function BUILDACG(*MCG*,*AndroidManifest.xml*)
InitACGNodes(*ACG*,*AndroidManifest.xml*)
ms = getParents(*MCG*,"*startActivity*()") ∪ getParents(*MCG*,"*startActivityForResult*()");
for each method *method* in *ms* **do**
Eactivity = getTargetActivity(*method*)
IDs = FindMethodCallerViewId(*MCG*,*method*,ϕ)
for each view ID *viewid* in *IDs* **do**
Sactivity = findActivityByViewID(*viewid*)
ACG = *ACG* ∪ (*Sactivity*-*Eactivity*|*viewid*)
end for
end for
end function

function FINDMETHODCALLERVIEWID(*MCG*,*method*,*IDs*)
if *method* is View Event Method **then**
IDs = *IDs* ∪ FindViewId(*method*,*MCG*)
else
ms = getParents(*method*,*MCG*);
for each method *tmethod* in *ms* **do**
FindMethodCallerViewId(*MCG*,*tmethod*,*IDs*)
end for
end if
return *IDs*
end function

In particular, our system builds MCG based on some prior knowledge. Because there are some method calls in system space where we can't reach such as method `Thread.start` and method `Thread.run`, they do not have any relationship in user space, but from our knowledge `Thread.Start` will invoke `Thread.Run`. So with this knowledge we added some edges in MCG in advance.

Locate Target Activities Native Android app UI consists of several Activities in some of which WebView embedded. The system callbacks would call the methods in the classes which inherit from `WebViewClient` which we define as self-defined-WebViewClient. To make sure `onRe-`

ceivedSslError is reachable code we would find the activity whose WebView uses vulnerable self-defined-WebViewClient.

Once the vulnerable self-defined-WebViewClient is found, the system backtraces through MCG until the Activity which sets the WebViewClient is found. The backtracing would stop when it enters the system callback methods (currently our system could only handle Activity.onCreate and View.onClick) because there is no apparent method invoking these methods. We call these entry methods. Once the entry methods are found, it's easy to determine target Activity. If the entry methods are system callbacks of one Activity, this Activity is the target Activity. The Activity which owns the view is target Activity if the entry methods are system callbacks of a view. For Android app only these Activities that are declared in AndroidManifest.xml could be presented. That's why we check the target Activities set and delete Activities which are undeclared in AndroidManifest.xml.

Build ACG Now, we have got target Activities. Our purpose is to jump to target Activities from the launcher Activity and trigger the vulnerability. We need to find a way from launcher Activity to each target Activity with the help of ACG which is introduced before. We use algorithm 1 to build ACG. Each node in ACG represents one Activity that is declared in AndroidManifest.xml. Native Android UI jumps from one to another Activity because of View Event (such as Button.onClick). So each edge in ACG represents one View ID whose event method triggers native Android UI jumps from edge start Activity to edge end Activity. To our knowledge, there are two system calls to make activity jump, they are startActivity and startActivityForResult. They both need an intent which sets the jump to Activity as parameter. To build the edges, we backtrace the parameters of these two system calls (startActivity and startActivityForResult) to find the Activity (as A1) it starts. At the same time, we would find which View Event calls the system calls directly or indirectly during the backtracing process. Then we could find out the View ID (as ID1) and which Activity (as A2) owns this View. Then we add this edge { A2 - A1 | ID1.event } to ACG. The jump-to Activity sets in intent is not easy to find. There are six constructors of intent[4] and two kinds of intent: explicit intent and implicit intent. Explicit intent needs target Activity name which is recorded in AndroidManifest.xml as parameter however implicit intent just needs an action name which is also defined in AndroidManifest.xml. The Activity name of an explicit Intent could be tracked by method backtracing and register backtracing. For implicit Intents, we first scan the AndroidManifest.xml file and build the correspondence of the Activities and Actions. Once we have got the Action, jump-to Activity could be determined via correspondence built before.

3.2 Dynamic Analysis Module

This module is the most important part of our system. In this module our system automatically runs each app on an emulator and triggering native Android UI to target activity to check whether the app shows an illegal page. We use algorithm 2 to drive UI to target activity.

Dynamic Test Environment. In order to improve the efficiency of our system we apply multi-emulator to run the test. During this phase we need to install and run the app, and to make it automatically we need to simulate human operations on testing app. To meet our needs, the dynamic test environment should have the following features: Being able to understand the UI states, such as which activity is shown on screen, the position and ID of each View, the screen is showing a dialog or not; Being able to get UI objects, such as get the object of the button that is displayed on screen; Being able to perform actions, such as performing click action on a button by specified button id; Being able to get return value, such as whether a click action is successful or not.

Algorithm 2 UI Drive

```
Input:
  ACG : Activity Call Graph
  tas, target activities
  for each target activity act in tas do
     start target APP
     tACG = copy(ACG)
     ca = getCurrentActivity()
     while ca is from target APP and ca is NOT act do
        ViewID = FindNextEdge(tACG,ca,act)
        if ViewID is NOT NULL then
           perform(ViewID)
        else
           perform(return)
        end if
        WaitForJumpOrTimeOut()
        ca = getCurrentActivity()
     end while
     if ca is act then
        TryToOpenHTTPSWebPage()
     end if
     stop target APP
  end for

  function FindNextEdge(ACG,ca,ta)
     path = findPath(ACG,ca,ta)
     if path is φ then
        return NULL
     else
        edg = first edge of path
        ACG = ACG - edg
        return edg
     end if
  end function
```

In order to achieve these features, we have modified Android system tool instrumentation[7] by bypassing the signature verification phase, which allow us to test other apps with our own test script app (APK) though they have different signatures. With modified instrumentation, we don't need to re-sign the target app which may cause app crash. We developed a general test script app of which the configuration file was obtained from static analysis. With these features, we could run the test automatically. The configuration file would be generated automatically from information (e.g. ACG, target activities) obtained during static analysis phase. Then the app would be installed and tested according to the script app automatically.

After the app was installed, our system would drive the app to jump to the target Activities and further to trigger the vulnerability once the test script started by simulating a series of human operations. This driving procedure is divid-

Table 1: Results of Static Analysis

Potential Vulnerable Apps #	1360	9.8%
Free from such Vulnerability #	12203	88.3%
Decompilation Failure #	257	1.9%
Total Apps #	13820	

Table 2: Results of Dynamic Analysis

Vulnerability Confirmed #	645	47.4%
Vulnerability Free #	715	52.6%
Potential Vulnerable Apps #	1360	

ed into two parts: native Android UI driving and WebView UI driving. Native Android UI driving drives the UI to target Activity and WebView driving drives WebView to load an HTTPS web page.

We take a target Activity and calculate the path from current Activity to it based on ACG. If the path exists, we get the first edge in path which represents a View ID and a View event, trigger the View event for this View and delete this edge from ACG to avoid infinite loop. If the path doesn't exist, which means there is no way from current activity to target activity, we return to system or roll back to the previous activity. Once jumped to the next Activity, we do the same thing, calculating paths, triggering events and deleting edges until jump to the target Activity. For the condition that some view is visible on some conditions (i.e. click other button first), we directly jump to the target Activity. Once we have jumped to the target Activity with the WebView, To trigger the potential vulnerable code we have built an attack environment that could redirect to our illegal page when the app tries to load an HTTPS page. But there are some apps which load a static local page or HTTP page first with several links on it. Here we adopt a strategy like a crawler to find the HTTPS link and load it. We first extract all the links from the initial page and load every link and extract links again until we have found an HTTPS link or the craw layer depth is up to 3. It's worthy to note that we don't need to find all HTTPS links because all HTTPS links share the same error handling process.

Confirm vulnerability. While the WebView loads an HTTPS page, it will show a blank page if the WebView rejects the illegal certificate, otherwise it will show the illegal web page. So we check the WebView if it's a blank page or not which determines if this app is vulnerability or not.

With all detailed running information it's easy for us to figure out why this app is vulnerable and what is the function of the HTTPS web page. More over this information helps us analyze the result statistically and more general.

4. EVALUATION

We run experiments on two machines with Ubuntu OS, one for test and another for attack environment. We have downloaded 13,820 apps by its download rank from 360 market as dataset in July 2014.

Static analysis takes 13.5 hours to finish, 3.5 seconds per app which is fast enough to deal with large scale analysis. For decompilation, there are 257 apps can't be decompiled. The result of static analysis is shown in Table 1. From this table, there are 1,360 apps are potential vulnerable from a total number of 13,820. The apps that have its own SSL/TLS certificate verification error handling account for 9.8 percent which are potential vulnerable and need to be further detected in dynamic analysis to confirm if they are truly vulnerable or not. For the rest 12,203 apps checked as free of this vulnerability during static analysis, they either don't have their own `WebViewClient` or the code unreachable or the code reject the page with illegal certificate.

Table 3: Top 3 Categories of Vulnerable Activities

Categories	Count	Percentage
Payment	209	25.0%
Authenticate	280	33.5%
Login&Register	73	8.7%

Table 4: Vulnerable Apps in Ranking Interval

Ranking interval	Count	Percentage
1-1000	136	21.1%
1001-2000	94	14.6%
2001-3000	70	10.9%
3001-4000	50	7.7%
4001-5000	37	5.7%

In dynamic analyzing process, we have been employing 4 emulators with Android 4.2 to run the test apps and it takes 23 hours to run all 1,360 potential vulnerable apps and the average time for each app is 60.8 seconds. The result of dynamic analysis is listed in Table 2 from which we can see that nearly half (645) of the 1360 tested apps are confirmed vulnerable accounted 47.4 percent, which means nearly half of the certificate verification error handlings are not well designed or implemented. Also there are 715 apps are detected potential vulnerable in static analyzing process and confirmed not vulnerable in dynamic analysis because of the effectiveness of their own error handling.

Top 5 categories of vulnerable apps are shown in table 5. According to this table finance and social contain more vulnerable app than other categories and many of these apps employed third party SDK like Tecent Weibo, Sina Weibo and Alipay to fulfill some of their purpose. However, these three SDK are vulnerable themselves which makes all the apps employed these SDKs vulnerable. According vulnerable apps's download rank from the market in table 4, we found that the most popular apps (ranking interval from 1 to 1,000) have the highest vulnerable rate. Besides, the vulnerable rate decreased along with the popularity of the apps, which demonstrates the severity of this vulnerability. However, with the decline in ranking the vulnerable rate also fell, does not mean the apps with lower ranks are more secure. With further study we found that the apps with low rank are less likely to use HTTPS, which means they are more easily to be attacked.

Table 5: Top 5 Categories of Vulnerable Apps

Categories	Count	Percentage
Finance	56	8.7%
Social	56	8.7%
Lifestyle	51	7.9%
Entertainment	44	6.8%
Travel & Local	38	5.9%

We also defined category for each vulnerable activity by their name and function. We show those vulnerable activities in Table 3. The top two kinds are Payment and Authenticate Activity which weighted more than half of the total vulnerable Activities. The reason why so many Activities are these two categories is that, many apps are integrated with Tecent Weibo, Sina Weibo and Alipay SDKs and Tencent Weibo and Sina Weibo SDKs are social SSO SDKs and related to authentication and Alipay is a payment SDK. It's noteworthy that many vulnerable apps share a same vulnerable Activity. We have found a same vulnerable Activity in 128 different apps because of the integration of Alipay SDK.

5. RELATED WORK

Zheng *et al.* in [4] presented a system called SmartDroid which could lead native Android UI to the exposure of sensitive behaviors. But SmartDroid can't deal with web UI. Bhoraskar *et al.* in [8] presented an app automation tool called Brahmastra to test thirdparty components in mobile apps. Brahmastra is powerful enough to do that, but it can't test WebView UI which is necessary in our work.

Recently, a number of efforts have been made to reveal and mitigate SSL security problems. Fahl *et al.* [9] found Android SSL MITM vulnerability and developed a tool called Mallodroid to detect it. But they couldn't confirm the vulnerability automatically for large dataset. Sounthiraraj *et al.* in [1] developed a tool called SMV-Hunter to detect the SSL MITM vulnerability which is able to detect automatically for large scale dataset. Our work is directly inspired by SMV-Hunter. However, our system is very different from it, which is designed for different vulnerabilities with different techniques. SMV-Hunter focuses on app built-in SSL verification weakness, whereas our system focuses on the weakness in HTTPS verification error handling process. Meanwhile, the SSL usage is also different. In SMV-Hunter, it aims to find the apps that use SSL for the backend network communication. In our work, the use of the SSL is UI-based, namely, the web page will show up until that the WebView is show up. This means we have to manage to do more to jump to the target activity and open the HTTPS web page. Tendulkar *et al.* discussed the same problem (`onReceivedSslError`) in [10], and we almost work on it at the same time. They just showed the problem without further study in [10], and we systematically studied on it and developed this tool to detect this problem automatically.

6. FUTURE WORK

There are several limitations of our approach. In static analysis, because of the object-oriented programming diagram there are some virtual method call which is only determined at run time. In dynamic analysis, some activities are reachable on specific conditions. For example, if we want to jump to checkout activity on some shopping apps we have to login and put some goods in the shopping cart. We would get an error if we jump to the activity directly. And some UI elements are visible on particular conditions. For example, a logout button is not visible until you have logged in.

7. CONCLUSION

In this paper, we discovered a new type vulnerability for hybrid Android apps, which could affect Android WebView HTTPS connection making secure connection vulnerable. We have designed a new detection system that uses both static analysis and dynamic analysis to detect this type of vulnerability automatically on a large dataset of apps. Our static analysis discerns potential vulnerable apps and generates essential information to guide the dynamic analysis, which is used to confirm whether the app is vulnerable or not by automatically triggering the vulnerability facilitated for both native Android UI and WebView UI . We have applied our system to test 13,820 apps, and in total we found 645 of them truly vulnerable.

Acknowledgements

This work is partially supported by National Natural Science Foundation of China (61173068, 61173139), Program for New Century Excellent Talents in University of the Ministry of Education, the Key Science Technology Project of Shandong Province (2014GGD01063), the Independent Innovation Foundation of Shandong Province (2014CGZH1106) and the Shandong Provincial Natural Science Foundation (ZR2014FM020).

8. REFERENCES

[1] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps," in *Proceedings of the 19th Network and Distributed System Security Symposium. San Diego, California, USA*, 2014.

[2] J. Clark and P. C. van Oorschot, "Sok: Ssl and https: Revisiting past challenges and evaluating certificate trust model enhancements," in *Proceedings of the Security and Privacy*. IEEE, 2013.

[3] https://code.google.com/p/androguard/.

[4] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications," in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2012, pp. 93–104.

[5] https://code.google.com/p/android apktool/.

[6] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones." in *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[7] https://developer.android.com/reference/android/app/Instrumentation.html.

[8] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, D. Wetherall, D. Langenegger *et al.*, "Brahmastra: Driving apps to test the security of third-party components." in *Proceedings of the 23rd USENIX conference on Security Symposium*, 2014.

[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 *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012.

[10] V. Tendulkar and W. Enck, "An application package configuration approach to mitigating android ssl vulnerabilities," in *Proceedings of the 2014 Mobile Security Technologies Conference*, 2014.