INSTRUMENTING WEBCAPSULE ON ANDROID WEBVIEW

by

SNEHA TANAJI DALVI

(Under the Direction of Roberto Perdisci)

ABSTRACT

WebCapsule is a record and replay forensic engine for web browsers. It can assist forensic analysts to reconstruct and analyze real world web security attacks such as social engineering and phishing attacks. WebCapsule is always on, lightweight, portable and collects critical information. WebCapsule is designed and implemented as a self-contained instrumentation layer around Google's Blink web rendering engine and V8 JavaScript engine. Blink is already embedded in a variety of browsers and can run on different platforms, which makes WebCapsule portable. In this research, we instrument WebCapsule on Android WebView to verify the portable nature. WebView allows us to display web pages in Android Apps. We build Android System WebView with embedded WebCapsule and show that it is possible to record and replay web contents in WebView of Android applications. We evaluate the efficiency of the System WebView with embedded WebCapsule on a self-developed app and few real-world apps.

INDEX WORDS: Forensic Engine, Web Security, Browsing Replay,
Android WebView, Android App

INSTRUMENTING WEBCAPSULE ON ANDROID WEBVIEW

by

SNEHA TANAJI DALVI

BE, University of Mumbai, India, 2011

A Thesis Submitted to the Graduate Faculty of the University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2017

INSTRUMENTING WEBCAPSULE ON ANDROID WEBVIEW

by

SNEHA TANAJI DALVI

Major Professor:	Roberto Perdisci
Committee:	Kyu Lee Kang Li

Electronic Version Approved:

Suzanne Barbour
Dean of the Graduate School
The University of Georgia
May 2017

DEDICATION

I dedicate my work to my family and friends and all my mentors, academic or otherwise, who gave me motivation to push the boundaries of my achievements.

ACKNOWLEDGEMENTS

I would like to thank Dr. Roberto Perdisci for his guidance and perseverance throughout my research work and writing of thesis. I would also like to thank Dr. Kyu Lee, Christopher Neasbitt and Bo Li for their support and insightful comments throughout my research.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

1.1 Problem

It is critical to perform forensic analysis of web-based security incidents. Because, security researchers can then understand past incidents better and develop stronger defenses against future attacks. Unfortunately, analyzing real world web attacks that directly target users, such as social engineering and phishing attacks, is an extremely challenging and time-consuming task. Generally, forensic analysts rely on browser's history, cache files and system logs to analyze these attacks. But these sources lack adequate details to reconstruct the past incidents of social engineering and phishing attacks. Another approach is to leverage access to full network packet traces, which may provide more information on how a security incident happened. Still, there is a large semantic gap between the web traffic and the detailed events such as page rendering, mouse movements, and key presses that occurred within the browser. Due to the semantic gap, it is very difficult to precisely reconstruct victim interactions, the steps of the attack, and the impact of the attack.

To address this problem, we can use a record and replay engine which collects critical information such as network traces, DOM tree, user interactions. The record and replay approach is generally used for software debugging and testing in order to reconstruct and analyze bugs. There are few record and replay solutions for different types of software for example multi-threaded programs, web browsers and mobile

applications. However, the existing solutions for recording and replaying applications are not portable. There are not enough record and replay solutions available for mobile applications. There are few record and replay solutions available for Android applications but not for other platforms like iOS or Windows. Additionally, most of the existing solutions for Android applications are only suitable for functional or usability testing and not perfect for forensic analysis.

Mobile phones have been increasingly popular in recent years and millions of applications (apps) are available in different platforms. Moreover, more websites are now loaded on smartphones and tablets than on desktop computers. A recent study by a web analytics company StartCounter shows that internet usage by mobile and tablet devices exceeded desktop worldwide for the first time in October 2016 [25]. More numbers of mobile apps are being developed and used by people. Most of the apps use WebView to display web contents inside the app. WebView is a system component used in mobile platforms such as Android, iOS and Windows. It is called UIWebView in iOS. It enables smartphone and tablet apps to embed a simple but powerful browser inside them. WebView provides a number of APIs, allowing code in apps to invoke and be invoked by the JavaScript code within the web pages, intercept their events, and modify those events. According to a study [26] in the year 2011, 86 percent of the top 20 most downloaded apps in 10 diverse categories in Android market use WebView.

Therefore, there are equal or probably more chances of security incidents such as social engineering or phishing attacks in mobile apps. A study presents possible attacks on WebView in the Android System such as JavaScript Injection, event sniffing and hijacking [26]. They explain how two essential pieces of the Web's security infrastructure

are weakened if WebView and its APIs are used: the Trusted Computing Base (TCB) at the client side, and the sandbox protection implemented by browsers. As a result, many attacks can be launched either against apps or by them. Another study presents cross-site scripting attacks on Android WebView [27]. Such types of attacks result in stealing of cookies and other sensitive information such as contacts from the Android phone. They also result in Session Hijacking and impersonating user using stolen cookies. The attacks are a result of breach in the same origin policy of Android browsers. XSS attacks are easy to execute, but difficult to detect and prevent. Then, there is a blog by Dan Wallach [28], which describes possible threats of mobile advertising in Android WebView. Virtually all the advertising we see in Android apps is hosted inside a WebView. It's all HTML, JavaScript, and images, and it's all generally loaded over HTTP. That means there's no encryption and no authentication, violating Google's security advice. Just like web advertising, mobile advertising uses HTTP redirects, so the content can come from a variety of different sources. Attackers may well be able to craft malicious content and use any of the many advertising services to deliver it directly into apps. A WebView runs with the same privileges as the app that's hosting it. Maliciously crafted advertising content will then inherit all these privileges. Even with full Internet privileges and nothing else, malicious ad content connected to the Wi-Fi behind your firewall could use your phone as a launching pad for further attacks. Therefore, we need a tool to detect and analyze such attacks. A record and replay tool can be helpful to reconstruct such incidents in order to analyze and find the source of the attack.

1.2 Proposed Solution

To better address this problem, WebCapsule [1] is proposed. It is an always on, lightweight, portable, record and replay forensic engine for web browsers. It collects critical information such as network traces and snapshot of DOM tree which allows the forensic analyst to analyze exactly how the page was structured at every significant user interaction with the page's components. Using WebCapsule, an analyst can later replay previously recorded browsing sessions in a separate controlled environment, without providing any new external user inputs or network transactions. This enables detailed analysis of security incidents that are unexpected. It allows reconstructing detailed information about incidents that may follow new attack patterns. Since, WebCapsule can replay all non-deterministic inputs, including all content provided by the server, it enables a full forensic investigation of incidents involving short-lived phishing or social engineering attack pages.

To make WebCapsule portable, it is designed and implemented as a self-contained instrumentation layer around Google's Blink web rendering engine, which is already embedded in a variety of browsers such as Chrome, WebView, Opera, Amazon Silk, etc. and can run on different platforms like Linux, Android, Windows, and Mac OS. WebCapsule is implemented around Chromium [2] codebase. Chromium is an open source browser project behind the Google Chrome browser, which uses Blink rendering engine [3] and V8 JavaScript engine [4]. Due to the portable nature, WebCapsule can be used in browser and apps in mobile devices as well.

In this research, we show that WebCapsule can also be embedded in Android WebView [5] and we can record and replay web contents in WebView of Android apps.

Android WebView is a system framework component and is used to display web pages in Android apps. Since Android 4.4 (KitKat), the WebView component is based on Chromium code and from Android 5.0 (Lollipop), the WebView is moved to an APK as Android System WebView, so it can be updated separately. It is possible to test modifications done in Chromium codebase by building full browser or simple test shells for different platforms. We built Android WebView Shell and System WebView APK from Chromium with WebCapsule. Then, we performed record and replay on web contents in WebView of Android Apps. This proves the portable nature of WebCapsule forensic engine. The record and replay capabilities of WebCapsule are implemented by extending's Chrome's DevTools. Hence, we had to enable WebView debugging in real world apps in order to test the record and replay functionality.

In CHAPTER 2, we will look at the implementation and portable nature of forensic engine WebCapsule and the motivation behind this research. We will then talk about some past works of record and replay engines for debugging software in CHAPTER 3. Then, we will describe the process of building Android System WebView with embedded WebCapsule and how we tested record and replay capabilities in Android apps in CHAPTER 4. Finally, we will present the evaluation results and discuss about native library in CHAPTER 5 and conclude in CHPTER 6.

CHAPTER 2

BACKGROUND

2.1 WebCapsule

WebCapsule is a record and replay forensic engine for web browsers. It transparently records enough information to enable forensic analysts to reconstruct a user's historic browsing activities. WebCapsule records corresponding HTML and a snapshot of the current DOM tree. It also records all non-deterministic inputs to the rendering engine, and all input events such as mouse location coordinates, keypress codes, etc. It also records responses to network requests, and return values from calls to the underlying platform API. This allows analysts to replay previously recorded browsing sessions without any external user input or network transaction. This enables investigation of incidents such as short-living phishing or social engineering attack pages.

WebCapsule can function as an always-on system by configuring to start recording at browser startup. Moreover, WebCapsule is highly portable, can be embedded in a variety of web-rendering applications, and can run on a variety of platforms. To make WebCapsule portable, it is implemented by injecting lightweight instrumentation shims around Google's Blink web rendering engine and its tightly coupled V8 JavaScript engine without altering their application and platform APIs as shown in figure 1. Blink web rendering engine is embedded in a variety of browsers like Chrome, WebView, Opera, Amazon Silk, etc. and can run on different platforms e.g. Linux, Android, Windows and Mac OS. Hence, WebCapsule inherits Blink's portability.

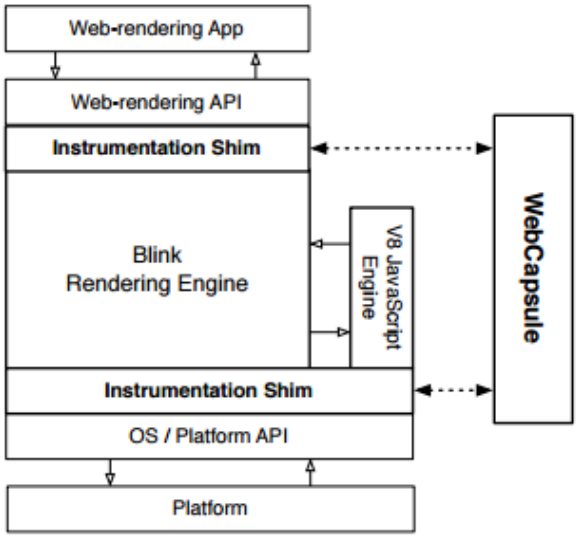

Figure 1: Overview of WebCapsule's Instrumentation Shims

(Source: WebCapsule [1])

2.2 Motivation

WebCapsule is implemented around Chromium, which is an open source project and it uses Blink rendering engine and V8 JavaScript engine. We can test modifications done in Chromium by building full browser or test shells for different platforms. Chromium project provides instructions to build Chromium on different platforms such as Linux, Windows, Mac, ChromeOS, iOS, Cast and Android. Therefore, Chromium for Linux and ChromeShell APK for Android was built from Chromium code with WebCapsule instrumentation and the record and replay capability was tested successfully.

We found that, we can even build Android WebView from Chromium code with WebCapsule instrumentation. Android WebView is used to display web pages in Android apps. Hence, with WebCapsule embedded in Android WebView, we will be able to

record and replay web contents in Android apps. This will help forensic analysts to reconstruct previously recorded incidents in Android apps. There are other existing record and replay systems but most of them are not portable and cannot be used in Android apps. Hence, in this research, we will build Android WebView with embedded WebCapsule and test its record and replay capabilities in our self-developed test app and few real world apps. This will also prove the portable nature of WebCapsule.

2.3 Types of mobile apps

Mobile apps can be divided in three types: native app, web app and hybrid app. Native and hybrid apps are installed from an app store, whereas web apps are mobile optimized webpages that look like an app. Both hybrid and web apps render HTML web pages, but hybrid apps use app-embedded browsers like Android WebView to do that. Hence, in this research, we will be focusing on hybrid apps that use Android WebView.

Native apps are installed on the device through an application store such as Google Play or Apple's App Store and are accessed through icons on the device home screen. They are developed specifically for one platform and can take full advantage of device features such as camera, GPS, compass, list of contacts and so on.

Mobile web apps are mobile-optimized websites and look like a native app. They are accessed by a URL in mobile browser. They are developed sing technologies such as JavaScript or HTML5. But, web apps cannot take advantage of all the native features of device.

Hybrid apps are installed from an app store and run on the device. They contain both native as well web components. Like native apps, they can take advantage of many device features available. Like web apps, they render HTML web pages in a browser

embedded within the app like Android WebView. Hybrid apps are written with web technologies like HTML, CSS and JavaScript and they can be developed as cross-platform. There are tools such as PhoneGap/Cordova that allow people to develop hybrid apps across platforms.

CHAPTER 3

RELATED WORK

3.1 Record and Replay Tools

During debugging, a developer needs to reproduce a bug repeatedly and adjust breakpoints in order to find the root cause of the bug. This is a time-consuming process and record and replay can be a very useful feature to reconstruct a bug. Replay can be used for a variety of reasons such as failure analysis using debugging tools, performance evaluation, usability analysis and forensic analysis. Many efforts have been done to implement record and replay feature in different ways.

John Vilk, James Mickens and Mark Marron propose a gray box approach for high-fidelity and high-speed Time-Travel Debugging (TDD) [6]. Time-travel debuggers allow developers to step forward as well as backward through a recorded program's execution. To increase fidelity in time-travel debuggers, they propose a gray-box approach, in which external or black box components like GUI state is also recorded. Based on this approach, they implement REJS, a new time-traveling debugger for client-side web applications that uses the JavaScript runtime as the virtualization layer. REJS leverages gray-box techniques to capture important state like animation metadata that resides in the rendering engine, which was formerly a black box component. REJS has less than 6% overhead during program recording, negligible overhead during replay, logs less than 1.5KB/s uncompressed on a wide variety of programs, keeps the GUI and other runtime state live during time-travel, and can migrate web applications in less than one

second over a 10Mbps connection. REJS has been incorporated into Microsoft's open-source Chakra-Core JavaScript engine. This gray-box approach can also be used in other managed runtimes like the JVM and CLR.

Jong-Deok Choi and Harini Srinivasan present DejaVu [32], which provides deterministic replay of Java multi-threaded application. Threads and concurrency constructs in Java introduce non-determinism to a program's execution, which makes it hard to understand and analyze the execution behavior. This non-determinism also makes it impossible to use execution replay for debugging, performance monitoring, or visualization. DejaVu is a record and replay tool which provides deterministic replay of program's execution. It is implemented as an extension to the Sun Microsystem's Java Virtual Machine. It records the logical thread schedule information of the execution while the Java program runs and replays the same execution behavior of the program by enforcing the recorded logical thread schedule. In addition to thread schedule information, DejaVu also records other non-deterministic attributes such as network events and windowing inputs/events. This is designed for generic multi-threaded applications and not for web applications. To record and replay web applications, there are many other non-deterministic inputs that needs to be recorded.

There are a variety of tools for debugging web applications, which are used by forensic analysts. For example, Fiddler is a web proxy that allows the local user to inspect, modify, and replay HTTP messages. The commercial Selenium, a Firefox extension records user activity for later playback. Recording can only be done in Firefox, but playback is portable across browsers using synthetic JavaScript events. Because Selenium does not log the full set of nondeterministic events, it is suitable for automating

tests, but it cannot reproduce many nondeterministic bugs. The commercial products ClickTale and CS SessionReplay capture mouse and keyboard events in browser-based applications. The services provide click analytics and a movie of client-visible interactions. However, neither of these products expose a full, browser-neutral environment for logging all sources of browser nondeterminism. They do not provide the underlying internal state of the JavaScript heap and the browser DOM tree.

James Mickens, Jeremy Elson, and Jon Howell propose Mugshot [7], a system that captures every event in an executing JavaScript program so that developers can deterministically replay past executions of web applications. The advantage of Mugshot is that its client-side component is implemented entirely in standard JavaScript, providing event capture on unmodified client browsers. It is always-on and imposes low overhead in terms of storage (20-80KB/minute) and computation (slow-downs of about 7% for games with high event rates). It only records JavaScript component of the application which includes DOM events, interrupts, non-deterministic functions, text selection. Mugshot has a server-side proxy which stores the external objects like images fetched by the application so that at replay-time, requests for the objects can access the log-time versions. If an application fetches external content that does not pass through the proxy, Mugshot cannot guarantee faithful replay of its data or its load time.

Silviu Andrica and George Candea presents WaRR [9], a tool for high-fidelity web application record and replay. The WaRR recorder extends WebKit and is embedded into the Chrome web browser. It records and replays interactions between users and web applications only. It only records user inputs such as mouse clicks, UI-element drags and

keystrokes. It works as “always-on” system and incurs low overhead. It can assist in finding bugs in web application and generating user experience reports.

Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim and Todd Millstein propose RERAN [10], a timing and touch sensitive record and replay for Android. This tool captures and playbacks GUI events such as different touchscreen gestures (e.g., tap, swipe, pinch, zoom) and other sensor inputs (e.g., accelerometer, light sensor, compass). Its implementation is heavily based on Android’s Software Developer Kit (SDK) tools. This is specially designed for Android apps to capture complex gestures and sensors with microsecond accuracy in order to replay it later for testing. It does not capture other sources of non-determinism. They successfully recorded and replayed 86 out of the top 100 free Android apps on Google play store.

Yongjian Hu, Tanzirul Azim and Iulian Neamtiu propose a tool called VALERA [33] (VersAtile yet Lightweight rEcord and Replay for Android). It records and replays smartphone apps, by intercepting and recording input streams and events with minimal overhead and replaying them with exact timing. It records all kinds of inputs such as network, GPS, camera, microphone, touchscreen with accurate timing. VALERA records the data sent and received, timing of network API and any exception occurred for each HTTP/HTTPS connection. It is implemented as instrumented app that intercepts communication between the app and Android Framework to produce log files. It is evaluated on 50 popular Android apps showing low overhead of around 1% during record and replay.

Zhengrui Qiny, Yutao Tangy, Ed Novak, and Qun Li present a novel system, called MobiPlay [11], which aims to improve record and replay testing for mobile

applications. It records all input data, including all sensor data, all touchscreen gestures, and GPS. It records the inputs at application layer and not at the Android Framework or Linux kernel layer. It is able to record and replay on the mobile phone as thin client as well as on the server. The client app on mobile phone intercepts all input data to target app and transmits to the server, so it the input data can be recorded on mobile as well as server. It uses SVMP, secure mobile application platform developed by MITRE2, based on thin client technology and cloud computing technology to implement client-server platform. Again, it does not record network requests and responses. It requires large number of resources such as virtual machine and high-speed internet connection.

Brian Burg, Richard Bailey, Andrew J. Ko and Michael D. Ernst presents TimeLapse [8], a developer tool for creating, visualizing, and navigating program recordings during debugging tasks. It utilizes Dolos, a novel record/replay infrastructure for web applications. To ensure deterministic execution, Dolos captures and reuses user input, network responses, and other nondeterministic inputs as the program executes. TimeLapse is based on Apple's WebKit rendering engine. Thus, it is not portable and only works on MacOS+Safari+WebKit. It deeply modifies the internals of WebKit which does not allow for transparent recording and it does not work as "always on" system.

C Neasbitt, B Li, R Perdisci, L Lu, K Singh and K Li propose WebCapsule [1], a record and replay engine for web browsers. It captures all user inputs, network requests and responses and other non-deterministic calls to underlying system. WebCapsule is implemented as instrumentation layer around Blink rendering engine, which is already embedded in variety of browsers and can run on different platforms. The record and replay capabilities are implemented by extending Blink's built-in instrumentation facility

known as Dev-Tools. Since WebCapsule records all the non-deterministic inputs including all previously rendered web content, it can record and replay even short-lived phishing pages, which is very important for forensic analysis.

Table 1: Summary of past Record and Replay techniques

Name	Year	Record and Replay	Implementation
Gray-box TDD/REJS	2016	External program state such as GUI	Gray-box virtualization
DejaVu	1998	Logical thread schedule information	JVM extension
Mugshot	2010	Client-side JavaScript	JavaScript program
WaRR	2011	Only user inputs to web application	WebKit instrumentation
RERAN	2013	Touch-screen gestures and sensor inputs to Android phone	Based on Android SDK tools
VALERA	2015	Different inputs to Android phone such as network, GPS, camera, microphone, touchscreen	API interception
MobiPlay	2016	All user inputs such as sensor data, touchscreen gestures, and GPS	Intercepting app and client-server platform
TimeLapse	2013	User input, network responses	WebKit instrumentation

WebCapsule	2015	User inputs, network transactions, and non-deterministic calls to the underlying system platform	Blink instrumentation

3.2 Summary

The primary aim for most of the above approaches is to assist in debugging, and they record specific components such as JavaScript or user inputs to web application or smartphone. WebCapsule records enough detailed information to enable a full reconstruction of web security incidents, including phishing attacks. Additionally, some of the approaches only work for particular type of applications or smartphone. WebCapsule is highly portable than all the other approaches. It allows us to record and replay web contents in browsers as well as mobile apps on different platforms.

CHAPTER 4

BUILD AND TEST

4.1 Building Chromium

WebCapsule is implemented as an instrumentation layer around Chromium, which is an open source project. It uses Blink rendering engine and V8 JavaScript engine. We can test modifications done in Chromium by building full browser or test shells for different platforms. Chromium project provides instructions to build Chromium on different platforms such as Linux, Windows, Mac, ChromeOS, iOS, Cast and Android. We can test record and replay capabilities of WebCapsule by building Chromium browser from Chromium codebase having WebCapsule instrumentation. For our research, we will focus on build instructions for Android [12]. In Android, we can have Chrome browser or WebView, where WebView is used to display web pages in Android apps. We can build Chrome Shell APK or full Chromium browser APK for Android. We will focus on building WebView Shell APK and Android System WebView APK in order to test modifications to WebView. This building process consists of many steps as shown in figure 2.

4.1.1 Pre-requisites

A Linux machine is required to build Chrome or WebView for Android. Other platforms (Mac/Windows) are not supported for Android. We used a desktop Dell Optiplex 980 with a Core i7 870 CPU and 8GB of RAM running 64-bit Ubuntu 14.04 Linux. At least 100GB free disk space is required. We also need Git and Python installed.

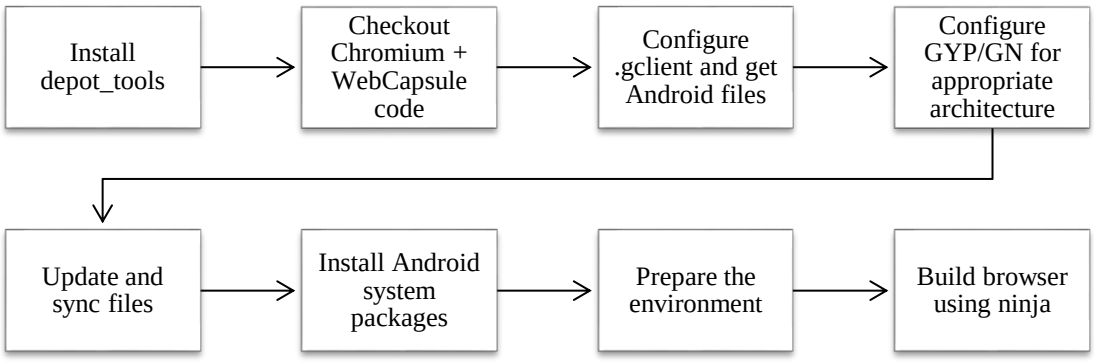

Figure 2: Common steps in building Chromium browser

4.1.2 Get the code

Foremost, we need to checkout depot tools and Chromium code using Git subversion tool. Depot tools is a package of scripts to manage checkouts and code reviews. It contains many utilities such as gclient, gcl, git-cl, repo and others. This depot tools should be added in PATH. Then, we checkout Chromium with WebCapsule instrumentation codebase using git. This step will be different from the instruction provided by Chromium, as we will clone the WebCapsule repository and not the original Chromium. This will take around 30 minutes to download the code.

After this, we add Android support by appending target_os = ['android'] to the .gclient file and run “gclient sync” to get the Android related files checked out. Gclient is a python script to manage a workspace of modular dependencies and the dependencies can be specified on a per-OS basis.

4.1.3 Configure the build

We can specify environment variables that configure the way Chromium is built using GYP (Generate Your Project) [13] or GN. GYP is intended to support large projects that need to be built on multiple platforms e.g. Mac, Windows, and Linux. By

modifying these parameters, we can build chrome for different platforms and architectures, or speed up the build process [14]. For example, for android and arm based 32-bit architecture build, the GYP_DEFINES environment variable is set as "{'GYP_DEFINES': 'OS=android target_arch=arm'}". The default architecture is “arm” and can be omitted. GN also uses the same target values.

Table 2: GYP/GN Targets for different architectures

Device Architecture	Target
Arm 32-bit	arm
Arm 64-bit	arm64
X86	ia32
MIPS	mipsel

We used GYP to build, as GN is not fully supported yet. But, GN incremental builds are the fastest option and GN will soon be the only supported option. GYP and GN are both meta-build systems that generate ninja files for the Android build. Both builds are regularly tested on the build waterfall.

After this, we update and sync project using “gclient runhooks” and “gclient sync” command. This will download more things and prompt to accept Terms of Service for Android SDK packages. We also need to install build dependencies and prepare the environment by running few scripts. We need to make sure that the PATH environment variable is set to latest build tool package in Android SDK.

4.1.4 Build the browser

Now, we can use ninja command “ninja –C out/Release android_webview_apk” to build test shell WebView i.e. WebView Shell APK. To build full WebView i.e. System WebView APK, the command is “ninja –C out/Release system_webview_apk”.

Ninja is a build system written with the specific goal of improving the edit-compile cycle time. It is included in depot tools. Also, we can build either debug or release version of APK. After this, we can install the APK in an Android emulator or device and test the any modifications done to WebView.

4.2 WebView Shell

Android WebView is a system framework component that allows Android apps to display web content. Since Android 4.4 (KitKat), the WebView is implemented using Chromium code. It is possible to test modifications to WebView using a simple test shell, called WebView shell. It is a view with a URL bar at the top and is independent of the WebView implementation in the Android system. The WebView shell is essentially a standalone unbundled app. We built WebView Shell APK from the Chromium having WebCapsule instrumentation code and installed in an Android device. Then, we opened the app, entered a web URL and tested record and replay functionality successfully. We will explain the process of recording and replaying web contents later in this chapter.

4.3 System WebView

4.3.1 Build

The WebView Shell is sufficient to run tests and for certain development tasks. But, if we want to run WebView code as an Android system component which is useful when working on performance or application compatibility, we need to build System WebView APK [15]. System WebView is the complete Android WebView framework component and is required to test the capabilities of WebView with WebCapsule in an Android app which uses WebView. Since Android 5.0 (Lollipop), the WebView is moved to an APK so it can be updated separately to the Android platform. Thus, we built

System WebView APK from the modified Chromium source code. To test the record and replay capabilities of WebCapsule in an Android app with the new System WebView APK requires additional steps as shown in the figure.

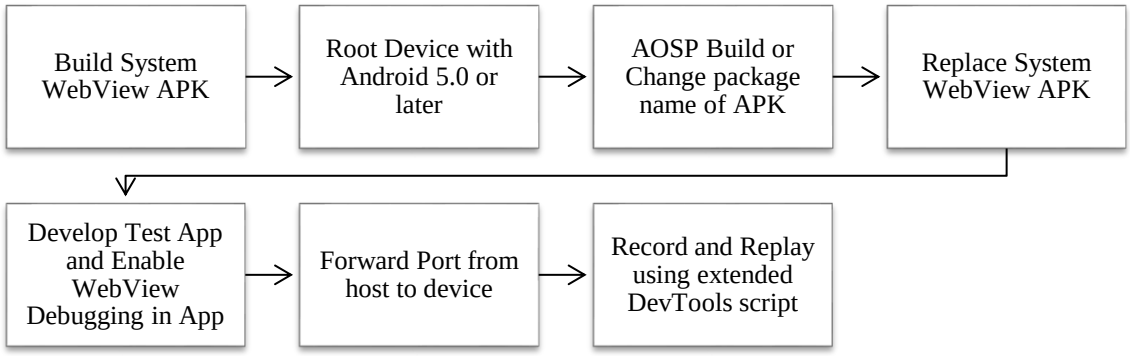

Figure 3: Steps to test record and replay capabilities of modified System WebView

4.3.2 Install

Now, we need to replace the default System WebView App in an Android device and install this new System WebView APK in order to test record and replay WebView contents of an Android app. But, System WebView is a system app and cannot be removed unless the device is rooted. Thus, this experiment requires a rooted device with Android 5.0 or above. For our experiment, we rooted Google Pixel C Tablet with Android version 6.0 [16].

Moreover, AOSP (Android Open Source Project) builds are required for testing the new System WebView APK as per the given instructions. AOSP build means an operating system built from standard Android source code. In AOSP build Android device, the System WebView package is named as “com.android.webview” by default and is called by this package name when any app tries to open a WebView. The System

WebView built from the Chromium source code also has package name "com.android.webview".

But, the Android devices in market have released builds of Android and are shipped with Google applications with the Google-specific version of the WebView called "com.google.android.webview". These partner devices have product builds of Android and are shipped with device specific drivers, software and Google apps. The AOSP builds and product builds of android, use two different package names because only the Google-specific package can be updated via the Play Store. The icons for the two different System WebView can be seen in Figure 4. The first one is the released System WebView and second one is the developer build, which is built from modified Chromium source code.

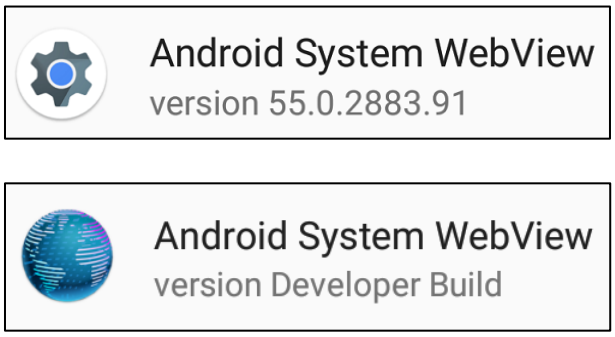

Figure 4. Two different builds of Android System WebView

But, there is a workaround if we do not have AOSP build in Android device. We can simply change the package name of System WebView APK to "com.google.android.webview". We can decompile the APK using apktool, edit package name in AndroidManifest.xml file, compile again using apktool and sign the APK using apksigner. We used this workaround because flashing AOSP build on Pixel C [17], was not allowing to root the device and we had to use modified stock kernel.

After this, we uninstalled the default System WebView and removed its related files from the device. Then we installed the modified System WebView.

4.3.3 Develop Test App

Now, we need an Android app that uses WebView in order to test record and replay feature of the new WebView. We developed a simple app which accepts a URL and opens that webpage in WebView [18]. The test app opens the entered URL in a WebView without crashing. Note that, the entered URL is not opened in Chrome or any other browser, but it is opened in WebView inside the test app. It indicates that the System WebView is built properly.

4.3.4 Enable WebView Debugging

The record and replay capabilities of WebCapsule are implemented using Chrome's DevTools. We will explain it in detail later in this chapter. But, because of this, we need to enable WebView debugging in the testing Android app, in order to remotely debug WebView [19]. To enable WebView debugging, call the static method setWebContentsDebuggingEnabled on the WebView class in onCreate() method of the app's application class or launcher activity. This setting applies to all of the application's WebViews. One thing to note here is that WebView debugging is not affected by the state of the debuggable flag in the application's manifest. After this, once the APK is built and installed in Android device, connect the device to the computer by USB cable and enable USB debugging. Then, open the app and we can see a list of running debug-enabled WebViews of the app in the desktop Chrome browser by navigating to "chrome://inspect" as shown in figure 5.

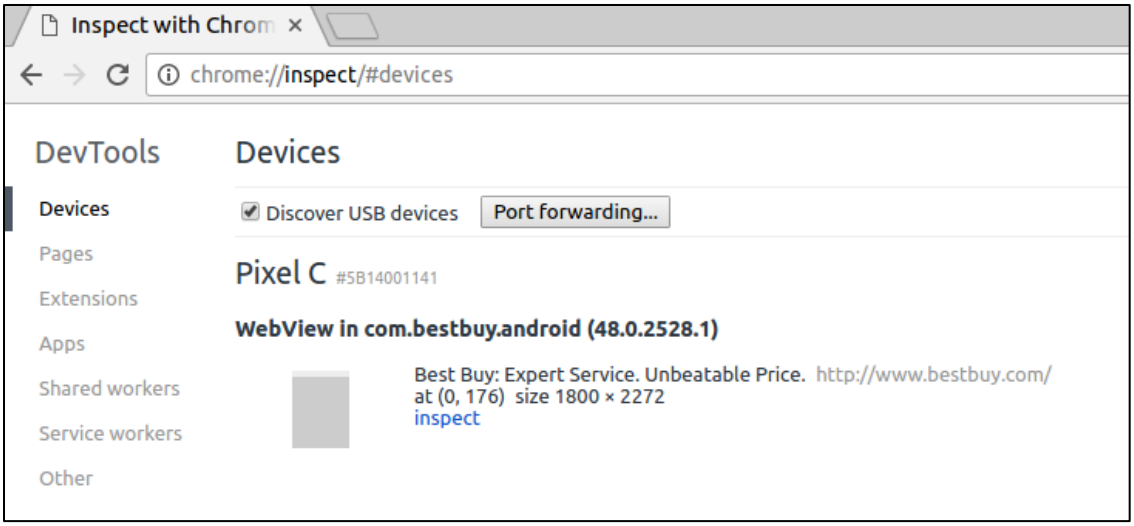

Figure 5. Chrome inspect page showing running WebView

4.3.5 Port Forwarding

We can use port forwarding to remotely debug Chrome on Android. ADB (Android Debug Bridge) has forward command that forwards requests on a specific host port to a different port on a device. This makes a port on the android device accessible from the computer and allows us to remotely debug an app on Android device. The port for Chrome on Android is chrome_devtools_remote. Similarly, we found that the port for WebView on Android is webview_devtools_remote_<process_id> where <process_id> is the process id of the app which is running the WebView. This process id can be obtained by using “adb shell ps” command. This command shows all the running processes and their process ids.

The command for port forwarding is “adb forward tcp:<port_number> localabstract:webview_devtools_remote_<process_id>”. This tells adb to forward any requests on localhost:<port_number> to the WebView port of the specified app on the

android device. Now, if we navigate to “http://localhost:<port_number>/json” in Chrome browser on the desktop, we can see the running WebView details in JSON format.

4.3.6 Record and Replay

The Chrome DevTools [20] are a set of web authoring and debugging tools built into Google Chrome. DevTools is designed to provide developers information about DOM elements, network traffic, and JavaScript execution. The collection and presentation of information from each category is implemented via an ‘InspectorAgent’. Users can retrieve the desired information collected by DevTools using either a graphical interface, called “Developer Tools” in Chromium, or via a JSON-based protocol over a WebSocket connection. The existing functionalities of DevTools are extended by hooking events we want to record. This is done by modifying ‘Inspector-Instrumentation.idl’, which is written using a mix of IDL and C++ code. This allows us to define a special InspectorAgent, which we use to add WebCapsule’s instrumentation shims around the web-rendering API.

To allow for the communication between WebCapsule and the external agent, we extend the DevTools JSON-based network protocol. This extension is developed as Python script, which defines new commands such as StartRecording, StopRecording, StartReply, etc. to remotely control WebCapsule operating mode.

To start recording web contents of any app, first we open WebView in the test app, use port forwarding to forward any requests on “localhost:8080” to the WebView port of the app. Then, we execute the DevTools script and we can see the URL opened in the webview of the app. We select the page which we want to record and execute StartRecording command to start the recording. In this way, we recorded web contents in

WebView of the test app and were able to replay the recorded session successfully. The commands used for port forwarding and performing record and replay on BestBuy app using DevTools extension script can be seen in following figure 6.

```
dalvi@via:~/WebCapsule/src/webcapsule_tools$ adb shell ps com.bestbuy.android
USER      PID   PPID  VSIZE  RSS     WCHAN              PC  NAME
u0_a111   23236 184   2593484 300392 SyS_epoll_ 0000000000 S com.bestbuy.android
dalvi@via:~/WebCapsule/src/webcapsule_tools$ adb forward tcp:8080 localabstract:
webview_devtools_remote_23236
dalvi@via:~/WebCapsule/src/webcapsule_tools$ python devtools_client.py http://lo
calhost:8080/json
Select the page. (Enter the page's number)
(0) Title: Cart Url: https://www-ssl.bestbuy.com/cart?appvi=2C166915851D1A02-400
0015100004148 ID: D5408990-BC4C-49E4-A287-BE39012766F2
:> 0
:> StartRecording
Sent:  StartRecording
:>
Received: {"id":0,"result":{}}
:> StopRecording
Sent:  StopRecording
:>
Received: {"id":1,"result":{}}
:> StartReplay
Sent:  StartReplay
:>
Received: {"id":2,"result":{}}
:> exit
Exiting
```

Figure 6: Port forwarding and Record/Replay commands

One of the commands developed using the DevTools script is EnableJavaScript. In this build, JavaScript is disabled by default and a function is provided to enable JavaScript using the command. We used this command before recording BestBuy app and PhoneGap Browser app. We could see the difference before and after enabling JavaScript. Since, JavaScript is disabled by default and can be enabled only after we connect to the opened WebView, we were not able to test few apps which require JavaScript to open a WebView at first.

4.4 Real-world Apps

As we were successful in recording and replaying WebView contents in a test app, we then tried to record and replay WebView contents in real-world apps. Most of the Android apps we use everyday use WebView in some way. But, in order to detect WebViews in real-world apps and use Chrome's DevTools, we need to enable WebView debugging for the app. This was a challenging task, since we have to reverse engineer the APK and modify source code of the app, to enable WebView debugging. We found a hack to accomplish this task.

We can get APK of an app using ApkExtractor app or we can download APK from websites such as apkmirror.com or apkpure.com. Then, we can get source code from APK by converting it into zip file. The zip file contains classes.dex (Dalvik bytecode), which can be converted to jar file (Java bytecode) using dex2jar tool. The .class files in the jar file can be opened and converted to Java source code by using Java decompilers such as Jd-gui. But, these Java files will not be exactly same as they were in the developer's Android project, because they are the decompiled short-form classes. Hence, it will not be possible to modify the source code and recompile the APK for a complex Android app. Also, the AndroidManifest.xml [21] file will not be in plain-text format. But, we can understand the code of the app and how it works.

Another way to decompile and recompile an APK is by using apktool. This way we get AndroidManifest.xml in readable format and .smali files. Smali/baksmali is an assembler/disassembler for the dex format used by dalvik, Android's Java VM implementation. Smali is assembly based language and after careful inspection we can understand and modify the code. So, we can find the Android app's application class or

the launcher activity from AndroidManifest.xml file. Then, we can edit the corresponding .smali file and add smali code to enable WebView debugging at appropriate place. Then, we can recompile the APK, sign it using apksigner. Now, we have the real-world app with WebView debugging enabled and we can simply record and replay WebView contents using the extended DevTools script, developed for WebCapsule. We can see all the steps to enable WebView debugging in real world apps in following figure 7.

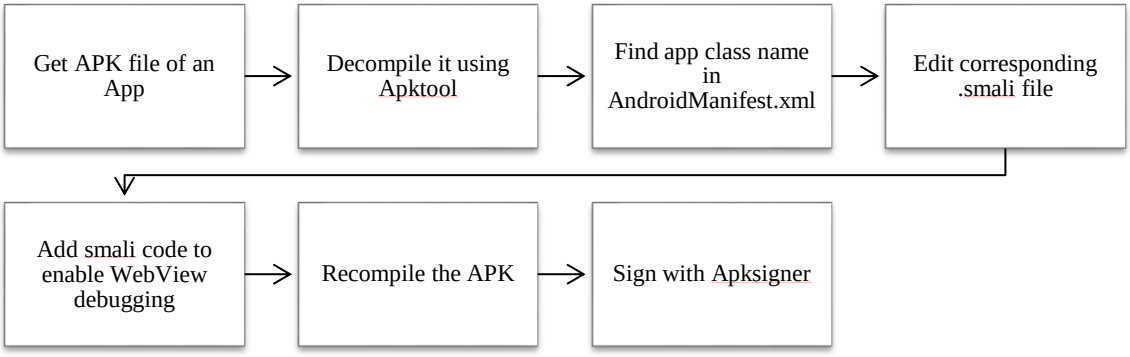

Figure 7: Steps to enable WebView debugging in a real-world app

4.5 Apache Cordova and Adobe PhoneGap Apps

Apache Cordova is an open-source mobile development framework. It allows developers to build applications for mobile devices using CSS3, HTML5, and JavaScript instead of relying on platform-specific APIs like those in Android, iOS, or Windows Phone. It enables wrapping up of CSS, HTML, and JavaScript code depending upon the platform of the device. It extends the features of HTML and JavaScript to work with the device. The resulting applications are hybrid, meaning that they are neither truly native mobile application nor purely Web-based. This is because all layout rendering is done via Web views instead of the platform's native UI framework. Also, they are not just Web

apps, but are packaged as apps for distribution and have access to native device APIs. It is important to note that Cordova apps are ordinarily implemented as a browser-based WebView within the native mobile platform. It means for Android devices, they use System WebView. There are several components to a Cordova application. The following figure 8 is inspired from Apache Cordova documentation [22], which shows a high-level view of the Cordova application architecture.

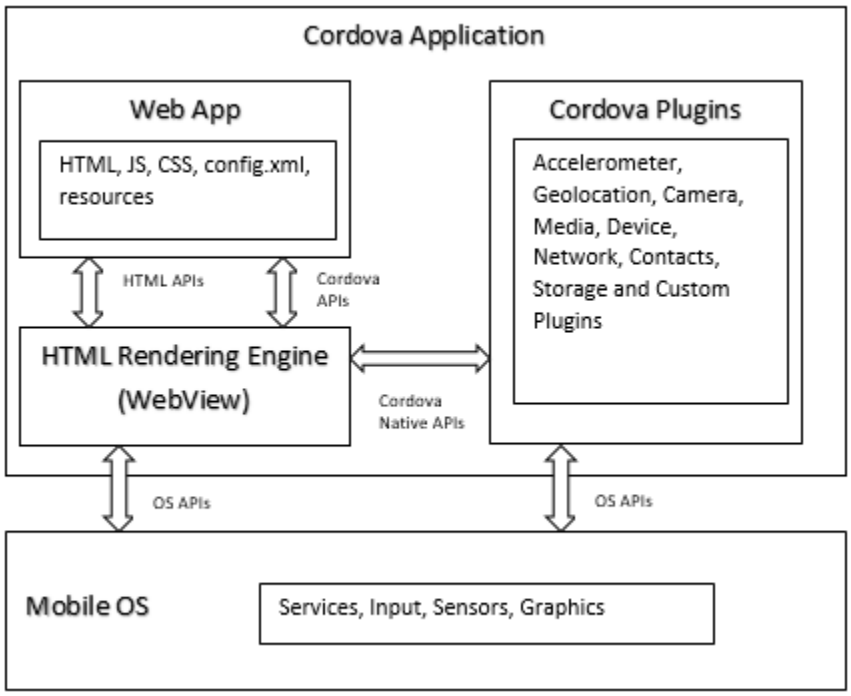

Figure 8: Cordova Application Architecture

(Source: Apache Cordova Documentation [22])

Apache Cordova was formerly known as Adobe PhoneGap [24]. Basically, Adobe PhoneGap framework is an open source distribution of Apache Cordova and also

includes access to the PhoneGap toolset. Over time, the PhoneGap distribution may contain additional tools that tie into other Adobe services.

PhoneGap can be used to build both types of hybrid apps, web hybrid and native hybrid apps. Web hybrid mobile apps are wrapped in a webview with a thin native container which is just used as the bridge to native. It is simply a wrapper for native-to-webview communication and there are no UI components provided from the native side. Native hybrid mobile apps include different native controls as well as one or more webviews. Native controls can be used to provide the navigation and transitions with the main content wrapped in webviews. The following figure 9 is inspired from PhoneGap Blog [30], which shows types of hybrid apps and how PhoneGap provides access to native APIs.

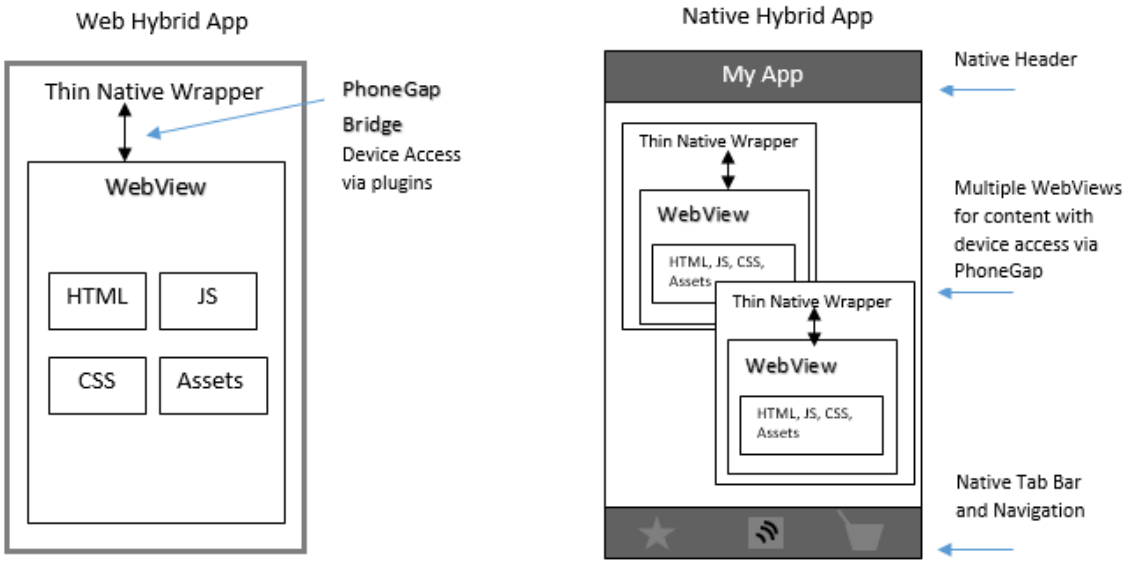

Figure 9: Types of hybrid apps and PhoneGap framework components

(Source: Adobe PhoneGap Blog [30])

It is easy and quick to develop hybrid apps using PhoneGap framework. PhoneGap is being used by hundreds of thousands of developers and thousands of apps built using PhoneGap are available in mobile app stores and directories. Therefore, we tried to record and replay web contents in PhoneGap apps and apparently it is easier to enable webview debugging in PhoneGap apps. In traditional android app, we need to enable webview debugging inside the application class. We have to reverse engineer the APK and add few lines of code in the application or launcher activity class as explained in figure 3. But, this is already supported in PhoneGap apps which use Cordova 3.3 or higher. For such apps, we only need to set debuggable flag to true in AndroidManifest.xml of the app. To do this, we need to decompile an APK using Apktool. Then, we can edit AndroidManifest.xml and set android debuggable to true. Then, we can recompile the APK using Apktool and sign the APK using Apksigner. After this, we can install the PhoneGap app in an Android device having modified System WebView and perform record and replay inside the app using WebCapsule tools as explained in section 4.3.5 and 4.3.6.

For Cordova 3.2 or lower, the webview debugging is not enabled by default and we will need to enable webview debugging inside the application class as shown in figure 3. Once, the webview debugging is enabled, we can record and replay web contents in PhoneGap apps using WebCapsule tools. But, nowadays most of the PhoneGap apps will have Cordova 3.3 or higher version.

CHAPTER 5

EVALUATION

5.1 Experimental Setup

We performed our experiments on two Android tablets - Nexus 9 with Android 5.0.1 and Google Pixel C with Android 6.0. We rooted these Android devices, in order to replace the default System WebView with the modified one. Both tablets have ARM-based 64-bit architecture. Hence, we specified target architecture as ‘arm64’ in GYP environment variables while building WebView Shell APK and System WebView APK. These APKs were built from Chromium codebase having WebCapsule instrumentation. A Linux machine is required to build Chrome or WebView for Android. Other platforms (Mac/Windows) are not supported for Android. We used a desktop Dell Optiplex 980 with a Core i7 870 CPU and 8GB of RAM running Ubuntu 14.04 Linux. We also required this machine to remotely send record/replay commands of WebCapsule to Android device.

We performed some of our initial experiments with WebView Shell on Nexus 5 which has ARM-based 32-bit architecture and Android Virtual Device (AVD) also called as emulator. WebView Shell is a standalone app which encapsulated the WebView libraries and is independent of the System WebView component. Thus, to test the modification to the WebView in WebView Shell does not require replacement of System WebView and can be performed in non-rooted Android device.

5.2 Results and Evaluation

We initially tested record and replay functionality of WebCapsule in the new built WebView Shell APK with embedded WebCapsule. All the record and replay tests on different webpages opened in WebView Shell APK were successful. We recorded browsing session of different websites such as Google, Wikipedia, etc. inside WebView Shell and were able to replay them with negligible difference.

Table 3: Record and replay tests on popular websites inside WebView Shell

Website	Comments
Google	Successful Record and Replay
Wikipedia	Successful Record and Replay
Flickr	Successful Record and Replay
IMDB	Successful Record and Replay
Ebay	Successful Record and Replay

Similarly, when System WebView APK with embedded WebCapsule was installed in the device, all the record and replay tests on WebView of our self-developed test app were successful. This was a simple test app that accepts a web URL and opens the URL in WebView. We also performed record and replay test on few real-world apps. There are many real-world hybrid mobile apps that show web contents because they need to update the contents frequently, for example apps related to news and magazines, shopping, social media, weather, entertainment or books and reference.

Adobe PhoneGap framework allows mobile developers to quickly and easily build cross-platform hybrid mobile applications with web technologies. These apps are ordinarily implemented as a browser-based WebView within the native mobile platform. It means for Android devices, they use native component System WebView. There are

thousands of apps developed using PhoneGap. Therefore, we developed a simple test app using PhoneGap, which opens a given URL in WebView inside the app. Then, we tested this self-developed app and few more PhoneGap apps from Google play store to test record and replay functionality. The summary of record and replay tests on real-world apps and PhoneGap apps from Google Play store is listed in Table 2.

Table 4: Record and Replay tests on real-world apps

Android App	Category	Results
IMDB	Entertainment	Successful record and replay with no divergence
NYTimes	News and Magazines	
BestBuy	Shopping	
Craigslist (Postings)	Shopping	
Expedia	Travel and Local	
HowStuffWorks	Entertainment	
Cordova Demo	Tools	Successful record and replay (PhoneGap apps)
PhoneGap Browser	Productivity	
WikiHow	Books and Reference	Successful record and replay with the issue of opening Chrome browser
Flickr	Photography	Unable to record and replay, as modified/recompiled APK do not work
Wikipedia	Books and Reference	
Google	Tools	
Amazon	Shopping	Unable to record and replay, due to absence of 32-bit library in WebView
Walmart	Shopping	

Once we enabled webview debugging for the above apps as explained in section 4.4 and section 4.5, we were able to perform successful record and replay on some of the apps listed in the table. IMDB app provides information about movies, TV shows, celebrities, reviews and recent entertainment news. After enabling WebView debugging for the app, we successfully performed record and replay on different web contents including featured items on IMDB such as The Oscars, Awards Central, IMDB Picks as

well as IMDB sign in, conditions of use and other related web pages. The only problem with this app was that, the app uses multiple webviews on some if the screens with no title which makes it difficult to identify specific webview. Some of the webview displayed advertisements.

Similarly, we were able to record and replay some of the web contents in NYTimes app such as news, blogs, pages related to terms and services, subscription. The news on certain screens were not detected in webview list. This may be because the app uses customized webviews or the debugging for those webviews is not enabled.

Then, we performed record and replay of BestBuy shopping app. When the recording is started, the app asks whether to open with Chrome or the app. We selected the app and continued recording. After finishing with recording, we started replaying, again the app asks whether to open with Chrome. Here, we manually select the app and then it continues replaying the same recorded sequence. We were able to record and replay most of the contents such as cart details, product list, my best buy program, and other details regarding order, shipping, etc. The product details page is not detected in webview list. Again, this may be because the app uses customized webviews or the debugging for those webviews is not enabled.

Craigslist Postings app is easy to use and visually appealing Craigslist search application. It uses native UI components as well as provides option to open the website in app. We were able to record and replay different pages such as search results, categories, help and support.

Expedia app is used for hotel and flight booking. The app uses native components as well as webview. Functionalities such as hotel or flight booking, location of the hotel

are developed in native UI. But, the app also supports the same functionality through webview, which can be recorded. The app also uses webview for career section, terms and conditions and links to entire website in webview. We were able to record and replay all of these functionalities.

HowStuffWorks app provides articles, videos, podcasts, quizzes and more. We were able to record and replay articles in the app. Sometimes, the app opens articles in multiple webviews. For example, clicking on a link within a webview opens the link in a new webview, which is not selected for recording. In this case, we could not continue recording the contents in the new webview. Otherwise, if the link opens in the same webview, then we could record and replay the contents.

PhoneGap browser is an app that can be used to display any URL on the internet in PhoneGap app. It is linked to the demo page of major frameworks for example Bootstrap, Ionic. Thus, we can check the compatibility of the framework and PhoneGap. For PhoneGap apps, webview debugging is enabled by default for debug version. Thus, we only had to set debuggable flag to true in AndroidManifest.xml file of the app. After that, we performed record and replay in the PhoneGap browser app successfully. Similarly, we recorded and replayed Cordova demo – nativeDroid app, which provides demonstration of jQuery Mobile theme nativeDroid v0.2.2.

The record and replay tests with some of the apps like WikiHow were successful with a particular issue. When we started recording these apps, they opened a new tab in Chrome browser with no URL. This was a different behavior, since the other apps we tested successfully, do not open Chrome browser. Then, if we close the Chrome browser, we could continue recording inside the app and could replay afterwards. During replay,

the app again opens a Chrome browser at the start and we had to close so that the replay continues in the app.

The tests with real-world apps were limited, because of the requirement of enabling WebView debugging. There were some apps for which we could not enable webview debugging. This is because to enable webview debugging, we had to decompile and compile the APK again and some apps do not work after re-compiling with or without modifications. For example, Facebook app does not re-compile due to errors. Wikipedia app gives parse error when tried to install after recompilation. Similarly, Google app does not work after recompiling. Some of the apps crash when we modify or recompile the APK.

Some of the apps did not work with the modified System WebView, because they were 32-bit apps and do not use 64-bit libraries. We can identify 32-bit apps by decompiling its APK and checking its ‘lib’ folder. If the APK does not contain any ‘lib’ folder, the app will run in both 32-bit and 64-bit devices. Inside ‘lib’ folder, ‘armebi’ or ‘armebi-v7a’ contains 32-bit libraries and ‘arm64-v8a’ contains 64-bit libraries. If an APK does not have ‘arm64-v8a’ folder, then the app won’t work with 64-bit System WebView.

The modified System WebView we built, is for 64-bit devices and contains only 64-bit libraries. The main library here is “libwebviewstandalonechromium.so”. After further research, on Chromium Google group [23], we found that we can only build either 32-bit or 64-bit System WebView APK by setting target architecture in building process using GYP. There is a script ‘apk_merger.py’ which can be used to merge 32-bit and 64-bit APKs. We need to build both 32-bit and 64-bit APKs separately, and then merge them

together afterward with “src/android_webview/tools/apk_merger.py” in single APK. We built such System WebView APK, it runs 64-bit apps but it still couldn’t run 32-bit apps. Therefore, we couldn’t test 32-bit apps such as Amazon, EBay or Walmart shopping apps. But, we could see in 32-bit device that these apps use WebView to display most of the web contents.

We have recorded two videos that show a representative demonstration of how WebCapsule can be used to record and replay web contents in Android apps. We recorded two Android apps – BestBuy and PhoneGap Browser. Following are the YouTube links where you can view these videos.

Table 5: YouTube links for the demo videos

App	YouTube Links
BestBuy	https://youtu.be/Cj5T8YeDkHw
PhoneGap Browser	https://youtu.be/9Nukdotll_U

5.3 Discussion

While performing record and replay tests on real-world apps, we noticed that, we could not record and replay some parts of specific apps. This is because, these are hybrid apps and they contain native UI components as well. We can only record and replay web contents inside WebView. We also noticed that, many apps (for example Reddit) use Chrome Custom Tabs [31] to display web pages within the app. In such apps, when a user opens a link in the app, it loads in Chrome Custom Tabs inside the app. The transition between native and web content more seamless. Developers can customize Chrome’s look and feel to match to the app, including changing the toolbar color,

adjusting the transition animations, and even adding custom actions to the toolbar, overflow menu and bottom toolbar to let the user interact with the app. The following figure shows Google webpage in Chrome Custom Tab inside Google app and the overflow menu.

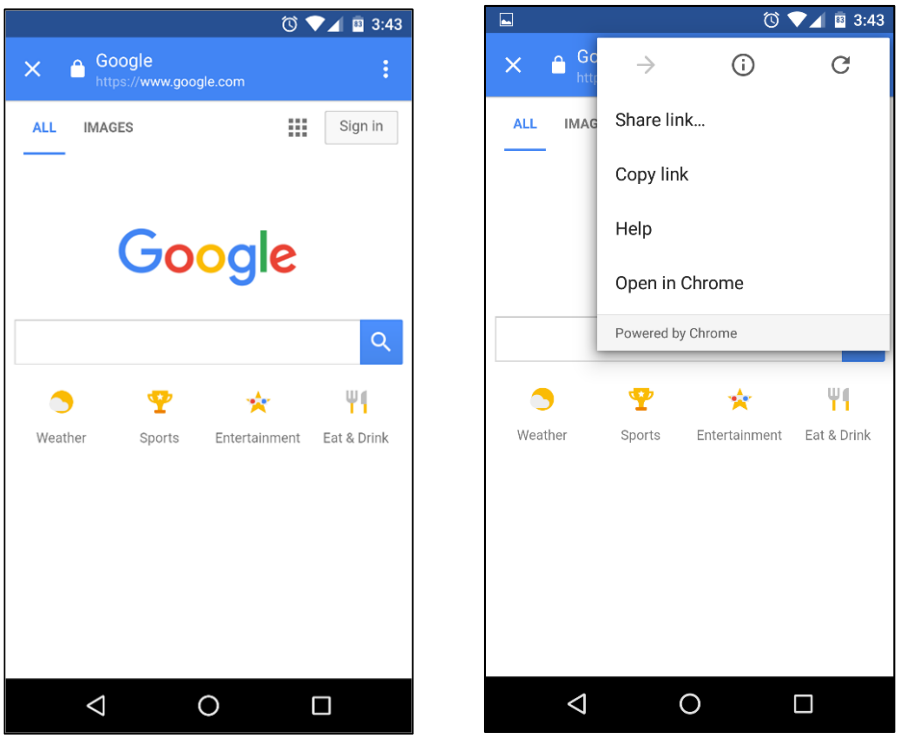

Figure 10: Google webpage in Chrome Custom Tab and overflow menu

The contents of Chrome Custom Tab may also be recorded and replayed using WebCapsule, as it is a Chrome component and uses same rendering engine Blink. But, to verify this, we will need to build full Chrome browser for Android with WebCapsule instrumentation in latest Chrome version which supports Chrome Custom Tab. The process of record and replay remains the same, except that the port for Chrome Custom Tab will be same as Chrome, which is “localabstract:chrome_devtools_remote”. It is

recommended to use WebView if the app hosts its own web contents. If the app directs people to URLs outside the app domain, it is recommended to use Chrome Custom Tab.

Another thing we noticed is the common native library inside all the WebView APKs named, "libwebviewstandalonechromium.so". This library basically bundles all the native code. In our early experiments, we replaced this library from an unmodified WebView Shell APK by the same library of modified WebView Shell and compiled the APK. We tested this new unmodified WebView Shell having modified native library and we were able to record and replay network traces. This implies that the native library was sufficient to include WebCapsule functionality in WebView Shell.

5.4 Limitations

The WebView Shell and System WebView APK were built from past versions of Chromium codebase and initial instrumentation of WebCapsule. They are prototype for record and replay functionality of WebCapsule and does not contain full-fledged implementation. This is the reason the scrolling UI event could not be recorded and replayed in WebView. Otherwise, WebCapsule records all user interactions and UI events such as touch on screen, network requests and responses.

WebCapsule is implemented as instrumentation layer around Blink web rendering engine and user interactions to the browser tools outside of Blink cannot be recorded. For instance, events such as a user’s click on the “back button” on the browser toolbar or touch on back button in Android cannot directly be recorded, because the input event is handled outside of Blink.

The record and replay tests on real-world apps were limited because of the requirement of enabling webview debugging. The process to enable webview debugging

in real world apps is complicated and may not be successful for some apps. This can be simplified if the feature of enabling webview debugging can be added as a flag in AndroidManifest.xml of the app. Just like, we set debuggable flag to true to get debug version of the app.

Chromium is an open source browser project and there are many tools and scripts which are still in development phase or do not have enough documentation. We found some of our answers through chromium google groups, issue tracker and stack overflow website.

CHAPTER 6

CONCLUSION

6.1 Conclusion

WebCapsule is a record and replay forensic engine, which aims to work as an always-on and lightweight forensic data collection system that enables a full reconstruction of web security incidents, including phishing and social engineering attacks. To make it portable, it is implemented as instrumentation layer around Google's Blink web rendering engine, which is already embedded in a variety of browsers and can be run on different platforms. In this research, we instrumented WebCapsule on Android WebView, a system component which is used to display web contents in Android apps. We built WebView Shell APK and System WebView APK from Chromium with WebCapsule instrumentation. We successfully recorded and replayed different websites in the modified WebView Shell APK. We installed the modified System WebView APK in an Android device and tested record and replay functionality on web contents in a self-developed app. Our experiments show that, WebCapsule can record and replay web contents in real-world apps and apps developed using Adobe PhoneGap framework. In this way, we show that, unlike other record and replay techniques, WebCapsule is highly portable and can be used for forensic analysis of real-world phishing attacks in websites as well as mobile apps.

6.2 Future Work

We performed our tests on the prototype implementation of WebCapsule, but more features can be implemented to solve the challenges or limitations occurred during recording and replaying. For example, support for tracking and recording multiple active WebView can be added. Then, similar to PhoneGap apps, support for enabling WebView debugging by debuggable flag in AndroidManifest in Android app can be added. This will make it easy to debug WebView in Android app and we will not have to decompile and modify the app. This will not only help for WebCapsule recording but also in debugging and testing of hybrid apps in general.

As we discussed, WebCapsule is implemented with Google's Blink web rendering engine, which is embedded in a variety of browsers like Chrome, WebView, Opera, Amazon Silk, etc. and can run on different platforms e.g. Linux, Android, Windows and Mac OS. Thus, WebCapsule is portable and its record and replay functionality is already tested in Chromium browser for Linux and Android and we tested it on Android WebView. It can be further tested by building Chromium for other platforms such as Windows, Mac OS X, Chrome OS and iOS [29].

REFERENCES

[1] C Neasbitt, B Li, R Perdisci, L Lu, K Singh and K Li. WebCapsule: Towards a Lightweight Forensic Engine for Web Browsers. In Proceedings of CCS. ACM, 2015.

[2] Chromium. https://www.chromium.org/Home.

[3] Blink web rendering engine. http://www.chromium.org/blink.

[4] V8 JavaScript engine. https://developers.google.com/v8/.

[5] Android WebView. https://developer.chrome.com/multidevice/webview/overview.

[6] J Vilk, J Mickens, M Marron. A Gray Box Approach for High-Fidelity, High-Speed Time-Travel Debugging. research.microsoft.com. 2016

[7] JW Mickens, J Elson, J Howell. Mugshot: Deterministic Capture and Replay for JavaScript Applications. NSDI, 2010. usenix.org

[8] B Burg, R Bailey, AJ Ko, MD Ernst. Interactive record/replay for web application debugging. In Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (New York, NY, USA, 2013)

[9] S Andrica, G Candea. WaRR: A tool for high-fidelity web application record and replay. In Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (Washington, DC, USA, 2011)

[10] L Gomez, I Neamtiu, T Azim, T. Millstein. RERAN: Timing- and Touch-Sensitive Record and Replay for Android. In Proceedings of the 2013 ICSE (2013).

[11] Z. Qin, Y. Tang, E. Novak, and Q. Li. MobiPlay: A Remote Execution Based Record-and-Replay Tool for Mobile Applications. In Proceedings of the 2016 International Conference on Software Engineering, 2016.

[12] Chromium build instructions for Android. https://www.chromium.org/developers/how-tos/android-build-instructions

[13] GYP. https://gyp.gsrc.io/

[14] GYP environment variables. https://www.chromium.org/developers/gyp-environment-variables

[15] System WebView build instructions. http://www.chromium.org/developers/how-tos/build-instructions-android-webview

[16] Root Pixel C. http://androiding.how/root-pixel-c/

[17] AOSP build for Pixel C. https://developers.google.com/android/images#ryu

[18] Building Apps in WebView. https://developer.android.com/guide/webapps/webview.html.

[19] Remote Debugging WebView. https://developers.google.com/web/tools/chrome-devtools/remote-debugging/webviews

[20] Chrome DevTools. https://developers.google.com/web/tools/chrome-devtools/

[21] Android Manifest. https://developer.android.com/guide/topics/manifest/manifest-intro.html

[22] Apache Cordova Overview. https://cordova.apache.org/docs/en/latest/guide/overview/index.html

[23] System WebView APK with 32 and 64-bit libraries. https://groups.google.com/a/chromium.org/forum/#!topic/android-webview-dev/hwmFs-L8u5I

[24] Brian LeRoux, Adbe PhoneGap Blog (2012, Mar. 19). Apache Cordova and Adobe PhoneGap. http://phonegap.com/blog/2012/03/19/phonegap-cordova-and-whate28099s-in-a-name/

[25] Mobile Internet Usage. http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide

[26] T Luo, H Hao, W Du, Y Wang, H Yin. Attacks on WebView in the Android system. In Proceedings of the 27th Annual Computer Security Applications Conference Pages 343-352. Orlando, Florida, USA. December 05 - 09, 2011.

[27] Bhavani A B. Cross-site Scripting Attacks on Android WebView, IJCSN International Journal of Computer Science and Network, Vol 2, Issue 2, April 2013, ISSN:2277-5420

[28] Dan Wallach, Freedom to Tinker (2015, Jan. 28). Android WebView Security and the mobile advertising marketplace. https://freedom-to-tinker.com/2015/01/28/android-webview-security-and-the-mobile-advertising-marketplace/

[29] Building Chromium for different platforms. https://chromium.googlesource.com/chromium/src/+/master/docs/get_the_code.md

[30] Holly Schinsky, Adobe PhoneGap (2015, Mar. 12). Choosing a mobile strategy for PhoneGap Apps. http://phonegap.com/blog/2015/03/12/mobile-choices-post1/

[31] Chrome Custom Tabs. https://developer.chrome.com/multidevice/android/customtabs

[32] Choi, J.-D., and Srinivasan, H. Deterministic replay of java multithreaded applications. In Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (New York, NY, USA, 1998)

[33] Y Hu, T Azim, I Neamtiu. Versatile yet Lightweight Record-and-Replay for Android. ACM SIGPLAN Notices, 2015 - dl.acm.org