Comparing DOMXSS Identification Tools on a Real World Vulnerability

Stefano Di Paola
Minded Security

4/Feb/2015

Index

Introduction

Minded Security has been the first company to launch a commercial tool aimed to identify DOM Based XSS[1] with a runtime approach: *DOMinatorPro*.
In 2012, as a result of our research on DOM XSS, we released the tainting engine on github.com as an open source project and created a commercial version that let users easily identify several kind of JavaScript vulnerabilities with a pretty high rate of accuracy[2].
Since then, some tools, open source and commercial, have been developed and awareness on this very topic grew among application security experts.
This paper will try to give an ***unbiased*** study supported by *objective facts* about precision and accuracy of existing tools that are stated to identify DOM Based XSS vulnerabilities.

Document structure

This document consists of *two main* parts, conclusions and appendix.

- In the *first part*, Tools Comparison Criteria, the scope of this research is described in conjunction with a set of the features that should be implemented by DOM XSS identification tools.
  Finally it is described how the list of tools for the comparison challenge was chosen.
- In the second part, Tools Comparison, a real world DOM based XSS is presented, analyzed and the set of chosen tools is tested against it in order to measure their degree of maturity.
- Conclusions are then presented to sum up the whole research and to give some impression about the status of the art in DOM XSS based identification.
- Finally, for reference, the appendix contains: the remote JavaScript URLs, the minimum HTML code that loaded the flawed code and the flawed code itself.

Tools Comparison Criteria

Scope and Analysis objectives

The scope of this white paper is to shed some light about the status of Client Side XSS (DOM XSS from now on) identification capabilities of existing tools.

A real world DOM Based XSS on pastebin.com was taken as test bed in order to move away from those too simplistic DOM XSS examples such as in:

- http://www.domxss.com/
- http://domxss.me/tests/
- http://www.daspatnaik.com/test/demo/
- others

Which, although their general usefulness, they tend to spread confusion and uncertainty about tool quality on day by day usage in real world web applications.

---

1 Also known as Client Side XSS

2 http://en.wikipedia.org/wiki/Accuracy_and_precision http://en.wikipedia.org/wiki/Precision_and_recall

Expected Features in DOM XSS Identification Tools

In order to identify a vulnerability - DOM XSS in this case - a tool needs to be able to maximize accuracy[3]:

ACC = (TP + TN)/(TP+FP+TN+FN)

It can be mostly done by *minimizing* :

- *False Positives:* which mostly improves user experience.
- *False Negatives:* which can be seen as identification coverage.

From a feature point of view, that means the tool must be able to:

1. Follow the flow
2. Find the execution path that triggers the bug
3. Recognize it as exploitable

Static Analyzers and Runtime Analyzers have different strengths and both approaches can be considered to complement each other.

Before talking about the results of this research, it's important to examine the differences between these two approaches.

The following table gives an overview of the features a tool should have by rating the quantity of implementation effort it has to be done to cover each one.

	Runtime	**Static**
Taint propagation	**Medium Effort** The native JS Parser does most of the work. Although several implementation specific problems must be faced, once the taint flag is set it's quite easy to add flag propagation.	**High Effort** Static analysis is, obviously, strongly language dependent, and implementing a taint propagation engine is not as easy as generating the AST and traversing it. Moreover, JS is a loosely typed language, hence it can be quite hard to propagate the tainting flag, in fact each variable must be tracked for each assignment according to scopes and dependencies. Finally, prototypes, callbacks hell and other language specific subtleties have to be considered.
Code Coverage	**High Effort** This is the ***Achilles' heel*** of runtime approach and it's the main reason people tend to choose static analyzers. Runtime approach only follows the flow of execution path which means that, in order to test another execution path, a program must be executed several times with variations on boundary conditions. Fuzzing, smart fuzzing and event dispatching can improve code coverage in black box situations. TDD[4] testing can also help for code coverage but it requires to spend time for	**Medium Effort** AST traversal, ensure 100% coverage for data collection. Once the whole call flow and tainted data flow are collected, it's quite straightforward to implement a graph traversal algorithm. Alas, most of the times in order to deal with memory minimization and time consuming tasks several approximation are applied, resulting in a lower accuracy rate → higher FP and FN rate. Last but not least, ***JavaScript code must be available for analysis***. In fact, JavaScript loaded at runtime from remote servers could

[3] TP=True Positive; TN=True Negative; FP=False Positives; FN=False Negatives;

[4] Test Driven Development

	its development and is page/application specific.	be missed, lowering the code coverage rate.
Source Coverage	**Low Effort/Intrinsic** Direct source mapping[5] can be quite easily applied when dealing with runtime approach. `*Indirect*`[6] sources, on the other hand, might require more implementation effort. A very important advantage here, is the dynamic real time interaction, which allows identification of client-servers states that are absolutely ***not*** *possible to achieve in static analysis*.	**High Effort** Direct sources mapping, have to always be explicit, since there's no "engine" doing the dirty work for the tool. ***Almost impossible*** to identify `*indirect*` sources unless having tons of FP (intrinsic to static analysis approach).
Filters Coverage	**Medium Effort** It mostly depends on implementation chooses, but they can actually be implemented by exploiting the runtime scenario, also in sophisticated cases, like when replace argument is a function call. In order to correctly identify anti XSS filters an heuristic algorithm must be implemented.	**High Effort** As already stated, "most of the times in order to deal with memory minimization and time consuming tasks several approximation are applied, resulting in a lower accuracy rate → higher FP and FN rate.". *Regular Expressions* based filters are very hard to implement in dynamic situations and often, even the simplest ones are not taken in consideration by static analyzer. The lack of automatic filter analysis results in higher FP rate.
Sink Coverage	**Low Effort/Intrinsic** Sink mapping can be quite easily applied when dealing with runtime approach.	**Medium Effort** Sinks mapping, have to always be explicit, since there's no "engine" doing the dirty work for the tool.
Lower False Positive Rate	**Medium Effort** According to source-filter-sink coverage, in order to lower the FP rate, a context sensitive algorithm implementation can lower FP to near zero. Obviously the source/sink coverage must have been implemented correctly.	**High Effort** Runtime dependent values and input modification filters are very hard to track when dealing with static analysis. That is indeed a known limit of Static Analysis approach.
False Negative Rate	**High Effort** Code execution coverage is the main problem as previously stated on "Code Coverage" row.	**High Effort** Approximation + Source Coverage + filter coverage + Sink coverage are the main problem

Note: the previous rating takes for grant that the creation of a JavaScript Tainting Engine needs an highly skilled technical designer/developer having a deep JavaScript knowledge.

Tools identification

The list of tools that are supposedly able to identify DOM XSS was chosen by our experience

[5] https://code.google.com/p/domxsswiki/wiki/Sources

[6] https://code.google.com/p/domxsswiki/wiki/IndirectSources (partial list). Sources that cannot be directly controlled by an attacker: such as document.title or strings previously stored on server side.

merged with some of the results of our twitter poll[7].

The following tools were requested to be analyzed:

1. **Burp** (native static code analyzer) http://portswigger.net/burp/
2. **DOM Based XSS Burp extensions** (RegExp based)
3. **DOMinatorPro** https://dominator.mindedsecurity.com
4. **JSA (IBM)** http://www.ibm.com/developerworks/downloads/r/appscan/
5. **JSPrime** https://github.com/dpnishant/jsprime/
   ScanJS https://github.com/pauljt/scanjs
   JSPwn https://github.com/Etraud123/JSpwn
6. **ZAP DOMXSS Plugin**
   https://code.google.com/p/zaproxy/wiki/HelpAddonsPlugnhackClientstab
7. **TPJS** https://github.com/neraliu/tpjs/
8. **IronWasp** http://ironwasp.org/
9. **Regexp based standalone DOM Based XSS** identification software
10. **DexterJS** (On line scanner)

From our perspective, we focused on tools with tainting[8] capabilities, so: **2, 6, 8 and 9** were not considered in scope. In fact, tools without tainting capabilities tend to create a lot of noise resulting in tons of False Positives.

Number 10, DexterJS, was requested to be enlisted but since it's an online service we considered as not in scope.

The final list as of 19th January 2015:

#	Name	Static/Runtime Analysis	Commercial[C]/ Open source[OS]
1	**Burp (1.6.09)**	Static	[C]
2	**JSA (IBM) (Appscan 9.0.1)**	Static	[C]
3	**JSPrime / ScanJS / JSPwn**	Static	[OS]
4	**DOMinatorPro (0.9.6.4)**	Runtime	[C][9]
5	**Tainted Phantom JS (TPJS)**	Runtime	[OS]

where **1,2,3** are static analyzers and **4,5** are runtime analyzers.

Plus: **1,2,4** are commercial tools where **3** and **5** are open sourced ones.

---

7 https://twitter.com/WisecWisec/status/553206286201139200

8 Taint Analysis or User Input Dependency checking. http://tanalysis.googlecode.com/files/DumitruCeara_BSc.pdf Pages 10-11

9 Partially Open Source (https://github.com/wisec/DOMinator https://code.google.com/p/dominator/)

Tools Comparison

A Real World DOM XSS Vulnerability

On 6th January 2015 a DOM based XSS[10] was found on pastebin.com.

The URL that created the requirements for the DOM XSS to be identified, can be whatever *pastebin* post like the following:

http://pastebin.com/Tq4Mgkse

That page, requires several external JavaScripts[11].

One in particular is generated from the following code:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
…
<script type="text/javascript">
(function() { function async_load(script_url){ var protocol = ('https:' ==
document.location.protocol ? 'https://' : 'http://'); var s =
document.createElement('script'); s.src = protocol + script_url; var x =
document.getElementsByTagName('script')[0]; x.parentNode.insertBefore(s, x); }
bm_website_code = '2D20BDFEA765412B';
jQuery(document).ready(function(){async_load('asset.pagefair.com/measure.min.js
')}); })();
</script>
…
```

Which, prettified is:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
…
<script type="text/javascript">
(function() {
    function async_load(script_url) {
        var protocol = ('https:' == document.location.protocol ? 'https://' :
'http://');
        var s = document.createElement('script');
        s.src = protocol + script_url;
        var x = document.getElementsByTagName('script')[0];
        x.parentNode.insertBefore(s, x);
    }
    bm_website_code = '2D20BDFEA765412B';
    jQuery(document).ready(function() {
        async_load('asset.pagefair.com/measure.min.js')
    });
})();
</script>
…
```

[10] http://erlend.oftedal.no/blog/research/xss/index.html

[11] See Appendix 1

The JS code above, calls a function (*async_load*) on page load which loads a remotely hosted JavaScript:

- http://asset.pagefair.com/measure.min.js

Its content is attached in the Appendix. The page also uses jQuery 1.8.2.

Fact*:*

- It's a quite common case to have a JavaScript resource "indirectly" available to a HTML parser.
  In such situations the presence of a ***real dynamic*** parser (a.k.a ***browser***) can be discriminant in the identification of such "hidden" resources.
  Anyway, this was not considered during the comparison in order to give some sort of equality at that level. This, in fact, depends on modules and functionalities such as *spidering* and *environment fulfillment*.
  In this case only ***runtime solutions*** can be considered as **ready by design**[12].

JavaScript in measure.min.js can appear quite *intricate* and *very hard to analyze by hand*, but it's actually the browserified[13] version of some bunch of JavaScript that uses *require()*.

It's a closure function called immediately and the first argument is an object whose keys are numbers and each element value is an array which contains a function.

The execution flow starts exactly on the Browserify code, which is shown in order to demonstrate how JavaScript can be complex nowadays:

```
function u(g, d, k) {
    function h(b, r) {
        if (!d[b]) {
            if (!g[b]) {
                var e = "function" == typeof require && require;
                if (!r && e) return e(b, !0);
                if (l) return l(b, !0);
                e = Error("Cannot find module '" + b + "'");
                throw e.code = "MODULE_NOT_FOUND", e;
            }
            e = d[b] = {
                a: {}
            };
            g[b][0].call(e.a, function(d) {
                var e = g[b][1][d];
                return h(e ? e : d)
            }, e, e.a, u, g, d, k)
        }
```

[12] Runtime approach like DOMinator and TPJS cover this kind of problem by design. Anyway, it strongly depends on specific modules implementations.

[13] http://browserify.org

```
        return d[b].a
    }
    for (var l = "function" == typeof require && require, f = 0; f < k.length; f++)
h(k[f]);
    return h
}
```

In order to reach the issue, an execution flow analysis has to be done.

We'll try to skip to the most interesting parts but we also want to leave the *reader* with the ***sense of confusion*** that is quite common when analyzing minimized complex JavaScript.

d(b) is the second call in the execution flow whose code is:

```
function(q) {
            var g = q("./lib/config-measuring"),
                d = q("./lib/detection")(g),
                k = q("./lib/reporting")(g);
            d.H(!0, function(d, g, f) {
                k.G(d, g, f);
                "undefined" !== typeof pf_notify &&
jQuery.isFunction(pf_notify) && pf_notify(d, g)
            })
}
```

d.H loads an image, second argument is a callback function that is called if the image loads successfully.

```
H: function(d, c) { // d ==!0 , c==callback
     t && (t = !1, m = {});
     a = c;
     !0 == d ? (p = 6, l()) : p = 4;
     g();
     b();
     f()
},
```

since *d==!0* , l function is called and an image is loaded. After a couple of nested calls the callback is called and it calls *k.G(d, g, f)*:

```
 G: function(b, g, e) {
      !0 === l.A() && (g.opted_out = 1);
      b = f.p;
      e && (b = g.p_false = 1);
      e = !1;
```

```
      null === f.C && (l.f("bm_monthly_unique", !0, d.b.l), e = !0);
      e && (b = g.new_monthly = 1);
      e = !1;
      null === f.t && (l.f("bm_daily_unique", !0, d.b.i), e = !0);
      e && (b = g.new_daily = 1);
      d.m("pagefair_exp_id") && (g.exp_id = d.e("pagefair_exp_id"));
      if (null !== b && 1 < b) {
                  if (d.random() > 1 / b) return;
                  g.smp = b
      }
      d.q(k.I +"/stats/page_view_event/" + k.L + "/a.js", g, h, !0)
  },
```

where *d.q* is a function that calls an XMLHttpRequest and whose success callback is *h()*.

h() is where the bug lies:

```
  function h(b) {
         b = d.B.parse(b).sample_frequency;
         b != f.p && l.f("bm_sample_frequency", b, d.b.i);
         d.m("pagefair_exp_id") && (b = d.e("pagefair_exp_id"),
jQuery("body").append('<div id="' +
                        b + '" style="display: none;"></div>')
      }
```

jQuery.append is called, and that's a known sink, if b is controllable and unescaped then HTML injection might be possible.

By looking at *d.e("pagefair_exp_id")*:

```
e: function(a) {
        return l(a) ? m[a] : null //l(a) checks if the key a is in the object m
                }
Where m:
m = function() {
        request_params = {};
        for (var a, b =
window.location.href.slice(window.location.href.indexOf("?") + 1).split("&"),
            c = 0; c < b.length; c++)
      a = b[c].split("="), request_params[a[0]] = a[1];
       return request_params
    }()
```

It's important to analyze how m is populated.

m is an object whose keys are the parameter names of the query string and values are their corresponding values.

The operations, in pseudo code, are:

```
var x = By taking location.href as input, extract substring from "?" character on;
var y = x.split("&");

for each element el in y
  var t = el.split("=");
    result[t[0]] = t[1];
```

so, *b* is the value of the parameter ***pagefair_exp_id*** in the query string.

The injection is in:

```
jQuery("body").append('<div id="' +
                        b + '" style="display: none;"></div>')
```

An attacker can just send:

http://pastebin.com/Tq4Mgkse?#&pagefair_exp_id="><img/src='x'onerror='alert(1)'>

right?

Not-at-all.

In fact by looking again at the value extraction code, there's a split("=") which actually forbids the presence of more equals '=' than the first.

That's the result of the previous payload:

```
<div id=""><img/src" style="display: none;"></div>
```

How can this be exploited? jQuery.append comes to our help.

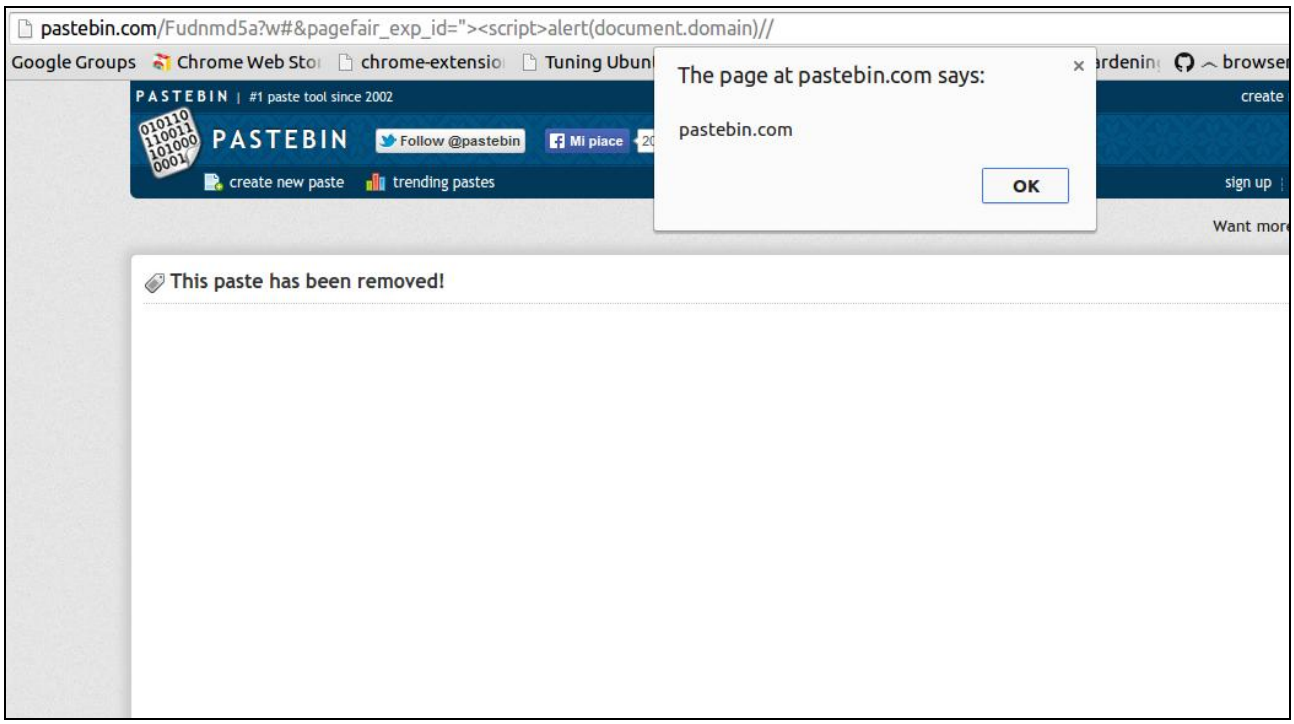
Since jQuery.append uses innerHTML to add new HTML, the browser won't take in consideration the <script> tag.

jQuery solves the problem by searching for "script" tag, extracting the text and sending it to jQuery.globalEval[14] function.

This behavior helps the attacker to exploit the injection using the following payload:

- http://pastebin.com/Tq4Mgkse?#&pagefair_exp_id="><script>alert(1)//

[14] https://code.google.com/p/domxsswiki/wiki/jQuery



As it can be noticed, the <*script*> tag is not closed, and in some situations, that could be also useful to defeat Anti XSS filters.

First phase

By now, every background knowledge has been given to prepare the reader to the tools comparison.

The first assumption is that every JavaScript resource must be available to the environment so that the first phase can be analyzed.

The page that contains all the required resources is:

http://pastebin.com/Tq4Mgkse

Which was used as URL to analyze for runtime analyzers.

On the other side, for static analyzers we created the correct set of resources, according to their needs, in order to give them all the necessary data to find the issue:

- HTML Code
- JS Code: jquery.js (v 1.8.2)
- JS Code: measure.min.js

So that, if any static analyzer or runtime analyzer would identify an exploitable flow the user should have been warned.

That's the results:

#	Name	Static/Runtime Analysis	Identified DOM XSS
1	Burp	Static	**No**

2	JSA	Static	**No**
3	JSPrime / ScanJS / JSPwn	Static	**No**
4	DOMinatorPro	Runtime	**No**
5	Tainted Phantom JS (TPJS)	Runtime	No

Fact:

- **Static Code Analyzers,** such as 1,2 and 3, were not able to identify the DOM XSS.

  But since **they are supposed to cover 100% of the code**, they can be considered as not capable of following the call flow at some point or lost the tainting for some reason.

- **Runtime Analyzers,** such as 4 and 5, were not able to identify any issue when the flawed code is not executed.

Given this results we can see that Static Code Analysis might not be the perfect choice, unless the implementation is mature enough to catch all issues.

About the runtime approach in the first phase, it's important to clarify the fact that if the executed code hit the bug, the tool would have had the chance to analyze it.

In this very case the flawed code is not executed, hence it's actually impossible the get any alert for runtime tools.

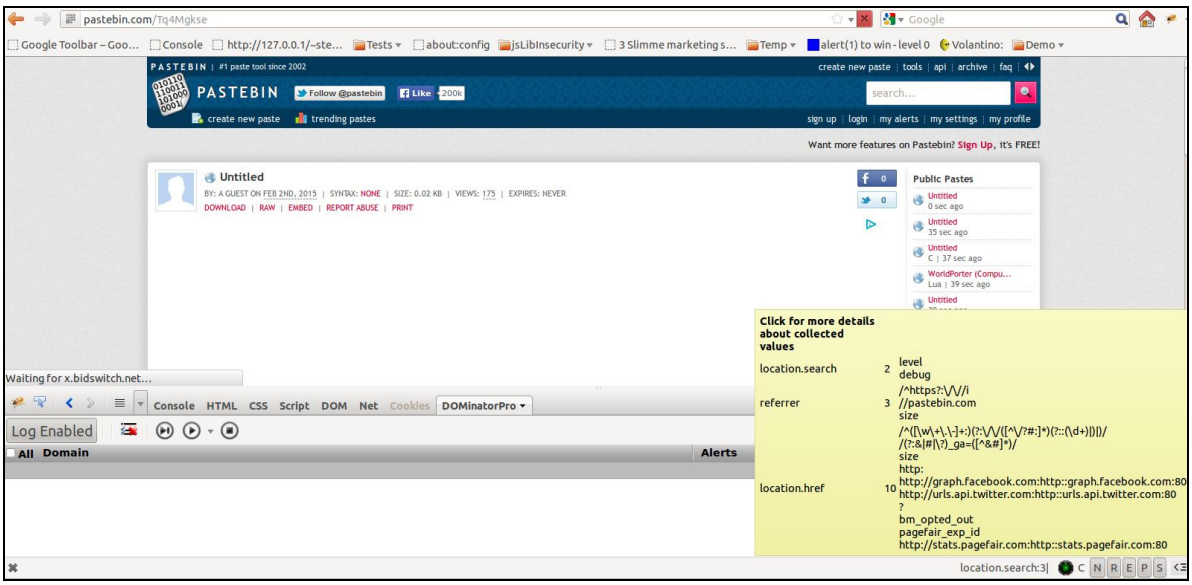
That leads us to the second phase.

Second phase

As static analyzers *1,2 and 3* are,by now, *out of this competition*, the question about code coverage solutions remains open for tools number *4 and 5*, respectively DOMinatorPro and TaintedPhantomJS.

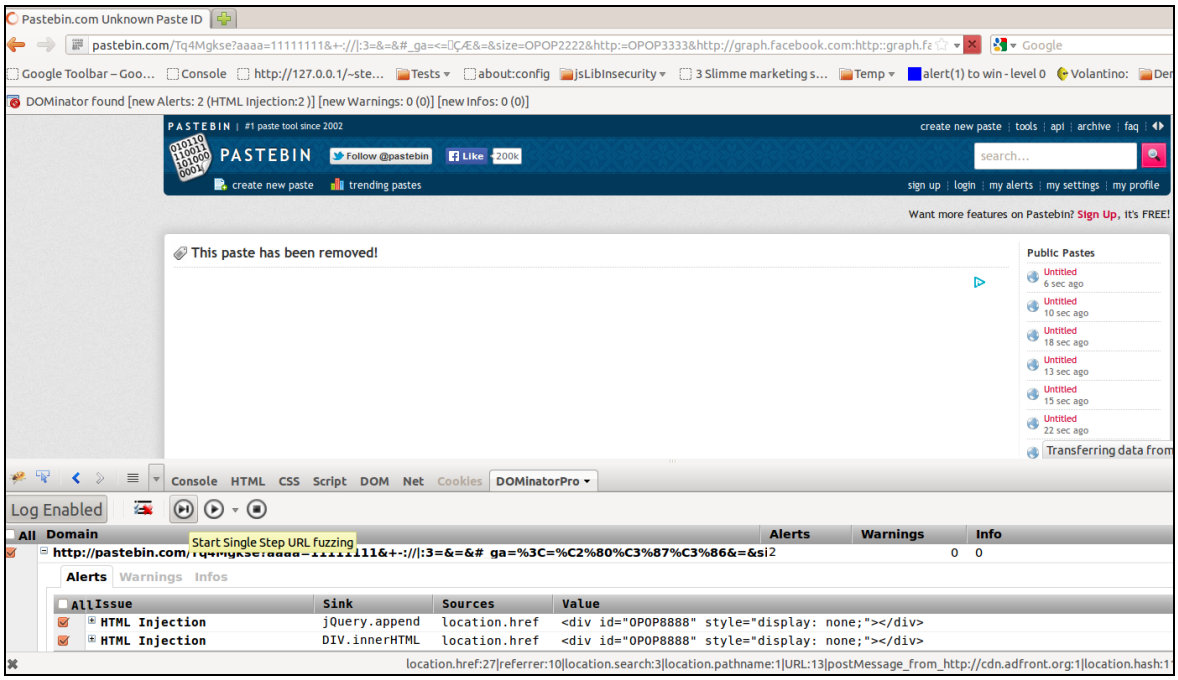
DOMinatorPro has a Smart Fuzzing button that collects all the patterns searched during JavaScript execution and uses them to populate tainted objects when the button is clicked.

This approach was *thought in order to improve False Negatives rate* and *code coverage*. Using Smart Fuzzing, in fact, the execution flow takes other, previously "*unseen*", branches.



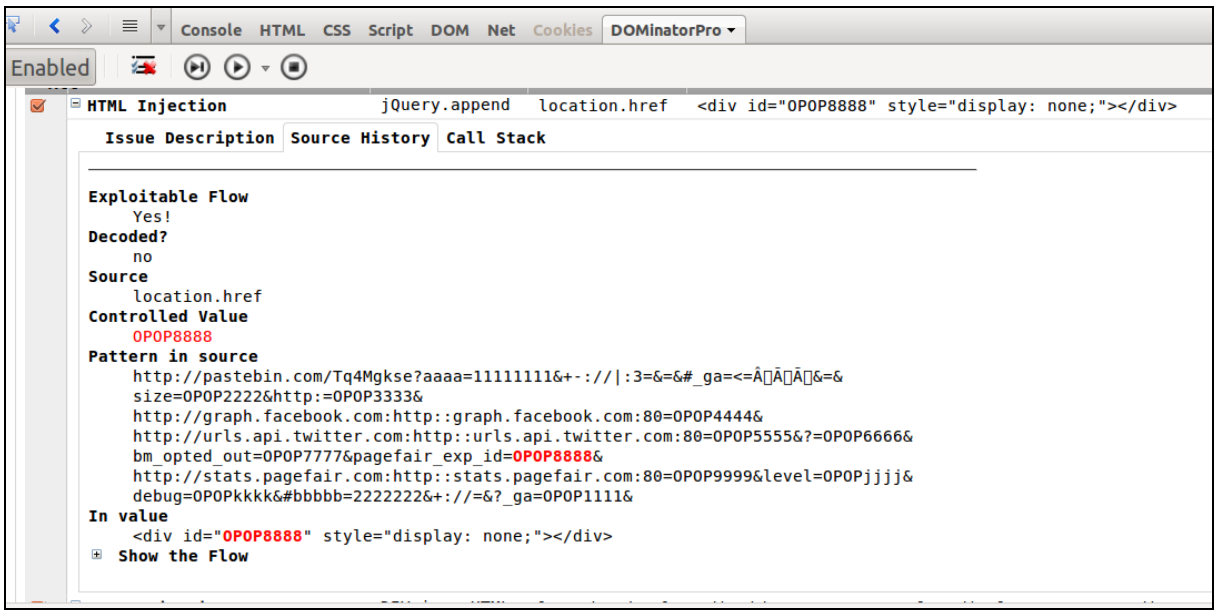
It can be noticed that the collector has identified several patterns, including *pagefair_exp_id* parameter. They will be added to their corresponding tainted object when in fuzz mode.

Here's what happens when clicking the smart fuzzing button:



DOMinatorPro was able to execute the flawed code and identify the problem.

In the next screenshot details are shown:



DOMinatorPro, anyway, does not give any information about the exploitation payload which is left to the tester.

TPJS, number 5 of the tools list is the latest DOM XSS identification tool that remains to be tested.

There's no fuzzing option, the only way to find out if the tool identifies the issue it's to explicitly add the pattern in order to execute the flow and reach the bug:

```
$: TPJS_HOME=. cli/tpjs "http://pastebin.com/Tq4Mgkse?pagefair_exp_id=OPOP8888&"
```

The output is

```
[Tue, 03 Feb 2015 11:01:19 GMT] [TPJS] Running Tainted Phantomjs....

[Tue, 03 Feb 2015 11:01:19 GMT] [TPJS] Running Tainted Phantomjs with URL:
http://pastebin.com/Tq4Mgkse?pagefair_exp_id=OPOP8888&

[Tue, 03 Feb 2015 11:01:19 GMT] [TPJS] Running Tainted Phantomjs with cookie file: cookie.txt

[Tue, 03 Feb 2015 11:01:19 GMT] [TPJS] Running Tainted Phantomjs with verbose: 0

[Tue, 03 Feb 2015 11:01:19 GMT] [TPJS] Running Tainted Phantomjs with timeout: 1000

[Tue, 03 Feb 2015 11:01:19 GMT] [TPJS] Running Tainted Phantomjs with fuzz: 0

[Tue, 03 Feb 2015 11:01:19 GMT] [TPJS] Running Tainted Phantomjs with rendering Path: ./

[Tue, 03 Feb 2015 11:01:19 GMT] [WARNING] cookie_parser: Unable to open file 'cookie.txt'

[Tue, 03 Feb 2015 11:01:19 GMT] [TPJS] --------------------

[Tue, 03 Feb 2015 11:01:19 GMT] [TPJS] TEST #0:1: domxss-db.js(window.onload())

[Tue, 03 Feb 2015 11:01:19 GMT] [TPJS] TEST URL: http://pastebin.com/Tq4Mgkse?pagefair_exp_id=OPOP8888&

[Tue, 03 Feb 2015 11:01:19 GMT] [ERROR] navigate too much
```

There's an error.

By looking at the code it was possible to fix the error and then, by reissuing the command, the following output was displayed:

```
[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] =================

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] source,1,jsLocationHref,location.href,about:blank

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] source,2,jsLocationHref,location.href,http://pastebin.com

...[SNIP]...

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] source,33,jsLocationProtocol,location.protocol,http:
```

```
[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] source,34,jsLocationProtocol,location.protocol,http:

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] source,35,jsLocationHref,location.href,http://pastebin.com

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] source,36,jsLocationHref,location.href,http://pastebin.com

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] source,37,jsDocumentCookie,document.cookie,cookie_key=1;

...[SNIP]...

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] =================

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] sink,162367550,jsHTMLDocumentPrototypeFunctionWrite,document.write,<div

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE]
sink,162382609,jsHTMLDocumentPrototypeFunctionWrite,document.write,<!DOCTYPE

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] sink,84054496,jsHTMLDocumentPrototypeFunctionWrite,document.write,<!DOCTYPE

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] sink,84399408,setJSHTMLElementInnerHTML,HTMLElement.innerHTML,<!--[if

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] =================

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] propagate,162389995,stringProtoFuncReplace,String.replace,function()

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] propagate,162389995,JSString,constructor,function());

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] propagate,162367550,stringProtoFuncReplace,String.replace,function()

...[SNIP]...

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] propagate,24,stringProtoFuncMatch,String.match,cookie_key=1

...[SNIP]...

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] propagate,13,jsString,String.operations,

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] propagate,35,stringProtoFuncSlice,String.slice,http://pastebin.com

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] propagate,35,stringProtoFuncSplit,String.split,pagefair_exp_id=OPO

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] propagate,35,stringProtoFuncSplit,String.split,pagefair_exp_id=OPO

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] propagate,35,stringProtoFuncSplit,String.split,

...[SNIP]...

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] propagate,35,JSString,constructor,<divOPOP8888"

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] propagate,35,stringProtoFuncCharAt,String.charAt,<div

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE]
propagate,46,stringProtoFuncToLowerCase,String.LowerCase,http://pastebin.com

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] propagate,46,stringProtoFuncReplace::exec,RegExp.exec,http://pastebin.com

...[SNIP]...

 [Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] propagate,78,regExpProtoFuncExec,RegExp.exec,http://pastebin.com

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] =================

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] =================

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [CONSOLE] [TRACE] reset,0,globalFuncEncodeURIComponent,encodeURIComponent,_ga

...[SNIP]...

 [Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [RESULT] document.tainted? true

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [RESULT] document.onAlert? false

[Tue, 03 Feb 2015 10:58:33 GMT] [TPJS] [RESULT] document.domxss.vulnerable? true|false
```

The output is really confusing, it took a lot of time to understand it and find out if TPJS identified the issue or not.

It seems that number 35 is the id that has to be followed.

It results in no "sink" with that number and all displayed sinks are False Positives.

The final consideration about TPJS is that the tainting flow was interrupted somewhere, thus preventing the real issue to be found. Also, it was very hard to understand the output of this tool which, as a matter of fact, contains a lot of False Positives.

#	Name	Static/Runtime Analysis	Identified DOM XSS
1	Burp	Static	No*

2	JSA	Static	No*
3	JSPrime / ScanJS / JSPwn	Static	No*
4	DOMinatorPro	Runtime	**Yes**
5	Tainted Phantom JS (TPJS)	Runtime	No

Fact:

- **DOMinatorPro** *was able to find* the issue when activating the smart fuzzing feature.
- **TPJS** does not have any fuzzing functionality. Even by introducing the tool to the correct test case, *TPJS was not able to find* the issue because it lost the tainting flow somewhere.

Conclusions

- DOM Based XSS are very hard to identify.
- It was shown a real world JavaScript code that contained a HTML Injection vulnerability, which was analyzed and an exploit was created by abusing a jQuery feature.
- An unbiased analysis based on several facts demonstrated that the status of DOM based XSS automatic identification is quite immature.
- Apart from classic DOM XSS testbeds where it's expected that each tool scores >90%, a lot of research on real pages must be done in order to really understand where an approach is better than another and where accuracy is really researched and reached.
- It was shown that in complex situations tools based on *static analysis* miss code coverage probably due to the so-called "*callback hell*" or other problems.
- It was pointed out that, even if *runtime analysis* can be very promising as identification approach, it has several limitations that should be addressed and solved by implementing advanced algorithms.
- Finally, all the tools failed the first phase of automatic discovery and the only one that passed the second phase was DOMinatorPro.

DOMinatorPro was the only tool that was able to solve all the challenges that modern, complex JavaScript code can present.

About the author

Stefano Di Paola is the CTO and cofounder of Minded Security, where he is Head of Research and Development Lab. In the last 7 years Stefano presented several cutting edge research topics, such as DOM based XSS runtime taint analysis, Expression Language Injection, HTTP Parameter Pollution and ActionScript Security that led him to be in the Top Ten Web Hacking Techniques initiative for 5 consecutive years (2007-2011). He also published several security advisories and open source security tools and contributed to the OWASP testing guide. Stefano is Research & Development Director of OWASP Italian Chapter.

* The tool gave the same results as in first phase

Appendix

Remote JavaScript Resources on PasteBin Page

- http://pastebin.com/Tq4Mgkse remote JavaScript resources:
  - http://s3.amazonaws.com/po93jnd9j40w/pastebin.min.js
  - http://www.google-analytics.com/analytics.js
  - http://pastebin.com/js/jquery.js (v 1.8.2)
  - http://pastebin.com/js/main_v1.js
  - http://asset.pagefair.com/measure.min.js
  - http://tags.expo9.exponential.com/tags/Pastebincom/Safe/tags.js[16]

Main HTML Content

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<head>
…
<script type="text/javascript" src="/js/jquery.js"></script>
…
</head>
<body>
…
<script type="text/javascript">
(function() { function async_load(script_url){ var protocol = ('https:' ==
document.location.protocol ? 'https://' : 'http://'); var s =
document.createElement('script'); s.src = protocol + script_url; var x =
document.getElementsByTagName('script')[0]; x.parentNode.insertBefore(s, x); }
bm_website_code = '2D20BDFEA765412B';
jQuery(document).ready(function(){async_load('asset.pagefair.com/measure.min.js')});
})();
</script>
…
</body>
</html>
```

Measure.min.js

$ curl "http://asset.pagefair.com/measure.min.js" |gunzip

```
(function u(g,d,k){function h(b,r){if(!d[b]){if(!g[b]){var e="function"==typeof
require&&require;if(!r&&e)return e(b,!0);if(l)return l(b,!0);e=Error("Cannot
```

[16] Script urls taken using:

```
var o={};
for(var i in document.scripts)
  o[document.scripts.item(i).src]=true
dir(o)
```

```
find module '"+b+"'");throw
e.code="MODULE_NOT_FOUND",e;}e=d[b]={a:{}};g[b][0].call(e.a,function(d){var
e=g[b][1][d];return h(e?e:d)},e,e.a,u,g,d,k)}return d[b].a}for(var
l="function"==typeof require&&require,f=0;f<k.length;f++)h(k[f]);return
h})({1:[function(q,g){g.a={r:"v01bcce/"}},{}],2:[function(q,g){var
d=q("../utils"),k=q("../cache_buster");
g.a={g:d.protocol+"asset.pagefair.com",M:d.protocol+"asset.pagefair.net",P:"und
efined"!==typeof
bm_cache_buster?bm_cache_buster:k.r,h:"",O:"",L:bm_website_code,$:d.protocol+("
undefined"!==typeof
bm_static_location?bm_static_location:"pagefair.com"),ba:d.protocol+("undefined
"!==typeof
bm_website_location?bm_website_location:"pagefair.com"),I:d.protocol+("undefine
d"!==typeof
bm_stats_location?bm_stats_location:"stats.pagefair.com"),N:d.protocol+("undefi
ned"!==typeof bm_ads_location?bm_ads_location:
"adfeed.pagefair.net")}},{"../cache_buster":1,"../utils":6}],3:[function(q,g){v
ar d=q("../utils");g.a=function(k){function h(b,r){b=k.h+b;for(var
e=b+"=",p=document.cookie.split(/[;&]/),h="null",l=0;l<p.length;l++){for(var
f=p[l];"
"===f.charAt(0);)f=f.substring(1,f.length);0===f.indexOf(e)&&(h=f.substring(e.l
ength,f.length))}e=h;"null"==h?e=null:("string"==d.type(h)&&(e=h.replace(/___/g
,";")),void
0!==r&&("ARRAY"==r?e=e.split(","):"INT"==r?e=parseInt(e):"BOOL"==r&&(e="true"==
e)));return e}function l(d){d=
k.h+d;document.cookie=d+"=; expires=Mon, 20 Sep 2010 00:00:00 UTC;
path=/"}function f(b,h,e){b=k.h+b;void 0===e&&(e=d.b.v);var
f=h;null===h?f="null":(d.isArray(h)&&(f=h.join(",")),"string"===d.type(f)&&(f=f
.replace(/;/g,"___")));l(b);document.cookie=b+"="+f+";
expires="+e.toUTCString()+"; path=/"}return{d:h,f:f,V:l,X:function(){var
b=d.e("bm_dts");null!=b&&(b=d.K(b),b>d.b.now&&f("bm_donor",1,b));return
null!=h("bm_donor")},A:function(){null!=d.e("bm_opted_out")&&f("bm_opted_out",1
,d.b.o);return null!=
h("bm_opted_out")}}}},{"../utils":6}],4:[function(q,g){var
d=q("../utils");g.a=function(k){function h(){if(d.u(m)===p&&!t){t=!0;for(var
n=!1,c=d.c(m),b=0;b<c.length;b++){var
f=c[b];0!=f.indexOf("ctrl_")&&(0!=f.indexOf("wl_")&&m[f])&&(n=!0)}c=n&&"NOT_BLO
CKING"==s.n||!n&&"BLOCKING"==s.n;e.f("bm_last_load_status",n?"BLOCKING":"NOT_BL
OCKING");void 0!==a&&jQuery.isFunction(a)&&a(n,m,c)}}function l(){function
a(c){var
n=d.c(m);0<d.k(n,"wl_i_hid")||(n=jQuery("#influads_block"),m.wl_i_hid=n.is(":hi
dden")?
1:0,m.wl_i_blk=c,n.remove(),h())}var
c=document.createElement("IMG");d.browser.safari||d.browser.msie?setTimeout(fun
ction(){a(0)},1):(jQuery(c).load(function(){a(0)}),jQuery(c).error(function(){a
(1)}));c.id="influads_block";c.style.width="1px";c.style.height="1px";c.style.t
op="-2000px";c.style.left="-
2000px";document.body.appendChild(c);c.src=k.M+"/adimages/textlink-
ads.jpg"}function f(){function a(n){var
b=d.c(m);0<d.k(b,"s_rem")||(b=jQuery("#"+c),m.s_blk=n,b.remove(),h())}var
c=d.j(),b=document.createElement("SCRIPT");
9>d.w||d.browser.safari||d.browser.mozilla?setTimeout(function(){a(0)},1):(jQue
ry(b).load(function(){a(0)}),jQuery(b).error(function(){a(1)}));b.id=c;b.type="
text/javascript";document.getElementsByTagName("head")[0].appendChild(b);b.src=
k.g+"/adimages/adsense.js"}function b(){var
a=d.j(),c=document.createElement("IFRAME");c.id=a;c.className="ad_iframe2";c.st
yle.border="none";c.style.width="1px";c.style.height="1px";c.style.top="-
1000px";c.style.left="-
1000px";c.src=k.g+"/adimages/ad.html";document.body.appendChild(c);
setTimeout(function(){var
c=jQuery("#"+a);m.if_hid=d.browser.mozilla?0:c.is(":hidden")?1:0;c.remove();h()
```

```
},1)}function g(){function a(b){var
n=d.c(m);0<d.k(n,"i_hid")||(n=jQuery("#"+c),m.i_hid=n.is(":hidden")?1:0,m.i_blk
=b,n.remove(),h())}var
c=d.j(),b=document.createElement("IMG");d.browser.safari||d.browser.msie?setTim
eout(function(){a(0)},1):(jQuery(b).load(function(){a(0)}),jQuery(b).error(func
tion(){a(1)}));b.id=c;b.className="AdHere";b.style.width="1px";b.style.height="
1px";b.style.top="-1000px";
b.style.left="-
1000px";document.body.appendChild(b);b.src=k.g+"/adimages/textlink-ads.jpg"}var
e=q("../cookies")(k),p,s={n:e.d("bm_last_load_status")},m={},t=!1,a;return{H:fu
nction(d,c){t&&(t=!1,m={});a=c;!0==d?(p=6,l()):p=4;g();b();f()},s:s}}},{"../coo
kies":3,"../utils":6}],5:[function(q,g){var
d=q("../utils");g.a=function(k){function
h(b){b=d.B.parse(b).sample_frequency;b!=f.p&&l.f("bm_sample_frequency",b,d.b.i)
;d.m("pagefair_exp_id")&&(b=d.e("pagefair_exp_id"),jQuery("body").append('<div
id="'+
b+'" style="display: none;"></div>'))}var
l=q("../cookies")(k),f={C:l.d("bm_monthly_unique"),t:l.d("bm_daily_unique"),p:l
.d("bm_sample_frequency","INT")};return{G:function(b,g,e){!0===l.A()&&(g.opted_
out=1);b=f.p;e&&(b=g.p_false=1);e=!1;null===f.C&&(l.f("bm_monthly_unique",!0,d.
b.l),e=!0);e&&(b=g.new_monthly=1);e=!1;null===f.t&&(l.f("bm_daily_unique",!0,d.
b.i),e=!0);e&&(b=g.new_daily=1);d.m("pagefair_exp_id")&&(g.exp_id=d.e("pagefair
_exp_id"));if(null!==b&&1<b){if(d.random()>1/b)return;g.smp=b}d.q(k.I+
"/stats/page_view_event/"+k.L+"/a.js",g,h,!0)},s:f,S:l}}},{"../cookies":3,"../u
tils":6}],6:[function(q,g){function d(a){var b="?";for(key in
a)b+=encodeURIComponent(key)+"="+encodeURIComponent(a[key])+"&";return
b.substring(0,b.length-1)}function k(a){a=a.split(".");for(var
b=jQuery.fn.jquery.split("."),c=0;c<b.length;c++){b[c]=parseInt(b[c],10);a[c]=p
arseInt(a[c],10);if(b[c]>a[c])break;if(a[c]>b[c])return!1}return!0}function
h(){return Math.random()}function l(a){return a in m}function f(a){var b=
[],c;for(c in a)a.hasOwnProperty(c)&&b.push(c);return b}function b(a){return
null==a?String(a):s[Object.prototype.toString.call(a)]||"object"}var
r="https:"==document.location.protocol?"https://":"http://",e=function(){var
a=3,b=document.createElement("div"),c;do b.innerHTML="\x3c!--[if gt IE "+
++a+"]><i></i><![endif]--
\x3e",c=0<b.getElementsByTagName("i").length?!0:!1;while(c);return 4<a?a:void
0}(),p=function(){var
a=window.navigator.userAgent,a=a.toLowerCase(),a=/(chrome)[
\/]([\w.]+)/.exec(a)||
/(webkit)[ \/]([\w.]+)/.exec(a)||/(opera)(?:.*version|)[
\/]([\w.]+)/.exec(a)||/(msie)
([\w.]+)/.exec(a)||0>a.indexOf("compatible")&&/(mozilla)(?:.*?
rv:([\w.]+)|)/.exec(a)||[];matched={browser:a[1]||"",version:a[2]||"0"};p={};ma
tched.browser&&(p[matched.browser]=!0,p.version=matched.version);p.Q?p.webkit=!
0:p.webkit&&(p.safari=!0);return p}(),s=function(){var
a={};jQuery.each("Boolean Number String Function Array Date RegExp
Object".split(" "),function(b,c){a["[object "+c+"]"]=c.toLowerCase()});
return a}(),m=function(){request_params={};for(var
a,b=window.location.href.slice(window.location.href.indexOf("?")+1).split("&"),
c=0;c<b.length;c++)a=b[c].split("="),request_params[a[0]]=a[1];return
request_params}(),t=function(){var a={};a.now=new Date;a.i=new
Date(a.now.getTime());a.i.setHours(23,59,59,999);a.l=new
Date(a.now.getFullYear(),a.now.getMonth()+1,0);a.l.setHours(23,59,59,999);a.J=n
ew Date(a.now.getTime());a.J.setDate(a.now.getDate()+1);a.F=new
Date(a.now.getTime());a.F.setDate(a.now.getDate()+
7);a.D=new Date(a.now.getTime());a.D.setDate(a.now.getDate()+14);a.o=new
Date(a.now.getTime());a.o.setDate(a.now.getDate()+28);a.v=new
Date(2030,11,31);return
a}();g.a={protocol:r,w:e,browser:p,R:s,type:b,isArray:function(a){return"array"
===b(a)},now:function(){return(new Date).getTime()},B:{parse:function(a){return
void 0!==JSON?JSON.parse(a):jQuery.parseJSON(a)},stringify:function(a){var
```

```
b;window.Prototype&&(b=Array.prototype.toJSON,delete
Array.prototype.toJSON);a=JSON.stringify(a);window.Prototype&&
(Array.prototype.toJSON=b);return a}},k:function(a,b){return
jQuery.grep(a,function(a){return a==b}).length},c:f,W:function(a){var
b=[],c;for(c in a)a.hasOwnProperty(c)&&b.push(a[c]);return
b},u:function(a){return f(a).length},Z:m,m:l,e:function(a){return
l(a)?m[a]:null},b:t,K:function(a){return new Date(1E3*a)},aa:function(a){return
encodeURIComponent(a)},j:function(){var a=(new
Date).getTime();return"xxxxxxxx".replace(/[xy]/g,function(b){var
c=(a+16*h())%16|0;a=Math.floor(a/16);return("x"==b?c:
c&7|8).toString(16)})},random:h,Y:k,T:d,q:function(a,b,c,e){b=b||{};e=e||!1;c=c
||null;var f=!0;if("undefined"!==typeof XMLHttpRequest&&"withCredentials"in new
XMLHttpRequest&&k("1.5.2")){var
f=!1,g={};g.url=a;g.type="GET";g.data=b;g.cache=!1;g.dataType="text";e&&(g.xhrF
ields={withCredentials:!0});null!=c&&(g.success=c);jQuery.ajax(g)}f&&(null!=c&&
(e=("r"+h()).replace(".",""),b.cbfnc=e,window[e]=c),b._=h(),c=document.createEl
ement("SCRIPT"),c.src=a+d(b),c.type="text/javascript",(document.head||document.
getElementsByTagName("head")[0]).appendChild(c))},
U:function(){return
window.location!=window.parent.location?document.referrer:document.location}}},
{}],7:[function(q){var g=q("./lib/config-
measuring"),d=q("./lib/detection")(g),k=q("./lib/reporting")(g);d.H(!0,function
(d,g,f){k.G(d,g,f);"undefined"!==typeof
pf_notify&&jQuery.isFunction(pf_notify)&&pf_notify(d,g)})},{"./lib/config-
measuring":2,"./lib/detection":4,"./lib/reporting":5}]},{},[7]);
```