Empirical Investigation of the Web Browser Attack Surface under Cross-Site Scripting: an Urgent Need for Systematic Security Regression Testing

Erwan Abgrall, Sylvain Gombault, Yves Le Traon, Martin Monperrus

2014 IEEE International Conference on Software Testing, Verification, and Validation Workshops

Empirical Investigation of the Web Browser Attack Surface under Cross-Site Scripting: an Urgent Need for Systematic Security Regression Testing

Erwan Abgrall
Telecom-Bretagne - RSM
University of Luxembourg - SNT
DGA-MI
Email: erwan.abgrall@telecom-bretagne.eu

Yves Le Traon
University of Luxembourg & SNT
Kirshberg, Luxembourg
Email: yves.letraon@uni.lu

Sylvain Gombault
Institut Mines-Telecom & Telecom-Bretagne - RSM
Rennes, France
Email: sylvain.gombault@telecom-bretagne.eu

Martin Monperrus
University of Lille & INRIA
Lille, France
Email: martin.monperrus@univ-lille1.fr

***Abstract*—One of the major threats against web applications is Cross-Site Scripting (XSS). The final target of XSS attacks is the client running a particular web browser. During this last decade, several competing web browsers (IE, Netscape, Chrome, Firefox) have evolved to support new features. In this paper, we explore whether the evolution of web browsers is done using systematic security regression testing. Beginning with an analysis of their current exposure degree to XSS, we extend the empirical study to a decade of most popular web browser versions. We use XSS attack vectors as unit test cases and we propose a new method supported by a tool to address this XSS vector testing issue. The analysis on a decade releases of most popular web browsers including mobile ones shows an urgent need of XSS regression testing. We advocate the use of a shared security testing benchmark as a good practice and propose a first set of publicly available XSS vectors as a basis to ensure that security is not sacrificed when a new version is delivered.**

I. Introduction

Back in 2000, the CERT released an advisory on Cross-Site Scripting (XSS) attacks, stating that XSS will be a growing threat for the next 10 years. Nowadays, a decade later, XSS attacks are still the major threat for web clients. The question we ask in this paper is whether the constant evolution of browsers leads to an overall improvement of final clients security. In this paper, we analyze six different families of web browsers and their evolution in terms of threat exposure to XSS.

Cross-Site Scripting (XSS) is a polymorphic category of attacks that may infect web applications as well as their clients, in many different direct and indirect ways. Many countermeasures can be deployed to face this threat: these security mechanisms are located in the internal parts of web applications (e.g. validation checks), on external security components (reverse-proxies, web application firewalls (WAF) like ModSecurity [1]) or even on client-side web browsers. One major difficulty to protect web clients against XSS is the technical nature of each XSS: unusual HTML and JS mechanisms are triggered. Predicting which HTML and JS mechanisms may be exploited by an attack is not trivial. To ensure the robustness of a web browser against XSS, test cases must be selected to evaluate the quality and efficiency of security mechanisms.

The technical contribution of this paper is a method to systematically test the impact of a large set of XSS vectors on web browsers, including mobile browsers (e.g. on Android). Our test driver, called XSS Test Driver executes a code within the web browser equivalent to the one ran by victims under XSS attacks. This allows us to measure the attack surface of a given web browser wit respect to XSS [2]. Using this tool, we assess two hypotheses related to the attack surface of web browsers:

H1. Browsers belonging to two different families have different attack surfaces. In other words, they are not sensitive to the same attack vectors. This first hypothesis is crucial to understand whether there is a shared security policy between web browser vendors headed against XSS attacks to protect clients against web attacks.

H2. Web browsers are not systematically tested w.r.t. their sensitivity to XSS vectors. This second hypothesis explores whether there is a clear continuity or a convergence in the attack surface of a given web browser over time. The validation of this hypothesis would mean that web browser providers do not have a systematic regression strategy for improving the robustness of their web browser from one version to the next one.

To assess those two hypotheses, we analyze the releases of six families of web browsers over a decade. Our results validate both H1 and H2. This shows there is an urgent need of systematic non-regression testing with respect to XSS. We

978-0-7695-5194-4/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSTW.2014.63

advocate the use of a shared security testing benchmark and propose a first set of publicly available XSS vectors to ensure that security is not sacrificed when a new version is delivered.

The paper is organized as follows: section 2 presents the background related to XSS. Section 3 describes the security test philosophy, related work and XSS Test Driver logic. Section 4 defines the metrics we propose for the empirical investigations and experimental setup. Section 5 motivates the development of XSS Test Driver based on technical considerations. The empirical studies of section 6 try to answer the hypotheses and address other related issues. In particular, the paper tackles new research issues in the domain of XSS test selection and regression testing of web browsers.

II. Background on Cross-site Scripting (XSS)

An XSS attack typically proceeds in two main steps: putting XSS code on a web server and then propagating the attack to the clients by making them executing malicious code on their browsers. In this paper, we focus on the second step and more specifically clients' exposure to XSS attacks. A XSS attack is composed of an attack vector (to penetrate the system) and of a payload (to perform the effective attack). Given the dynamic nature of today's web applications, and the variety of browser's implementations when it comes to interpreting HTML and JavaScript (JS), it is hard to determine if a XSS vector passing through a web application is a major threat or not for users. Most security mechanisms work well against basic XSS but tend to fail against sophisticated ones. Those advanced XSS exploit rarely known behaviors from peculiar interpretations of the HTML and other web page resources, in order to evade known Intrusion Detection System (IDS) signatures and put JavaScript (JS) calls in unexpected properties of some tags like:

```
<DIV STYLE="width:expression(eval(
String.fromCharCode(97,108,101,114,
116,40,39,120,115,115,39,41,32)));">
```

In this example, one must know that a regular CSS (Cascading Style Sheet) property *expression* executes *JavaScript* (JS) code when it is evalated in an *Internet Explorer* (IE) browser. The CSS expression calls the JS function eval() that itself calls String to convert data from decimal ASCII and produce this simple and non-destructive payload:

```
<script>alert(xss)</script>
```

Two main techniques are used to block the propagation of an XSS attack from the server to a client: signature-based and behavior-based like the SWAP approaches [3]. While the first approach cannot block new attacks (no known signature), the second fails in detecting browser specific XSS (a known limitation for SWAP), like the ones shown in [4]. In particular, an XSS detection engine relying on a specific HTML and JS parser cannot detect a browser specific XSS if it doesn't behave like the targeted web browser, since each browser have a specific web engine. As a consequence, it is necessary

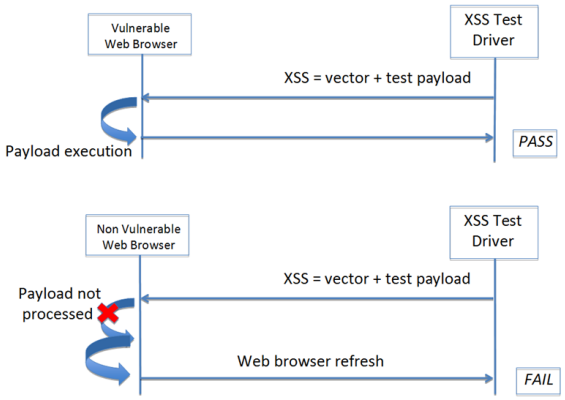

Fig. 1. XSS Test Driver Testing Logic

to focus on web browsers by estimating and analyzing their exposure degree to XSS. As highlighted by this example, one major difficulty to protect web clients against XSS is the technical nature of each XSS: unusual HTML and JS mechanisms are triggered. Predicting which HTML and JS mechanisms may be exploited by an attack is not trivial. Beyond this technical nature of the problem, testing is thus helpful as a way to estimate the discrepancies in browsers facing XSS in a systematic way.

III. Assessing the XSS Attack Surface over Time

A. Testing Methodology

A payload usually contains JavaScript code for the browser to execute. It can be innocuous or it can be noxious, by executing a redirection to an attack website. It then exploits a flaw inside the browser, leading to arbitrary code execution on the client like in the Aurora attack against Google employees [5]. The payload is executed if the browser "understands" the vector, meaning that it interprets it as expected by the attacker. In our context, a test case is composed of an attack vector carrying a non-destructive payload. As shown in Figure 1, a test fails if the browser does not execute the payload or if it crashes or hangs endlessly, preventing the JavaScript to be executed and thus the attack. In our context, a XSS test case "passes" if the vector is executed by the browser. This means that a passing test case reflects a real threat for the browser. "*pass*" thus means "possibly vulnerable" (where the use of a *pass* verdict usually corresponds to an absence of error in the testing domain). This is especially important since it accurately pinpoints the exact attack surface a web browser offers to the attacker. It also allows launching accurate test cases that will challenge server-side countermeasures. This testing methodology allows determining the overall security of a system, and also to measure each layer's contribution to security.

B. Metrics

To provide an overview of the sensitivity of a given web browser submitted to a set of XSS attack vectors, we define

some metrics.

Let TS be an XSS test case set. Let $verdict(tc, wb)$ be the verdict of the execution of test case tc of TS against a web browser wb. $verdict(ts, wb)$ returns either *Pass* (the XSS succeeds) or *Fail*. Let TR be the tests results of the execution of TS against a web browser wb, represented as a n-dimension vector.

The relative threat exposure of a web browser is equal to the number of Pass verdicts. The Threat exposure degree is defined as follows:

1) The Threat Exposure Degree $ThExp(wb, TS)$: The degree of threat exposure of a web browser wb to a XSS test set TS is defined as the rate of "Pass" verdicts when executed against the elements of TS:

$$ThExp(wb, TS) = \frac{|\{verdict(tc, wb) = `Pass', tc \in TS|}{|TS|} \quad (1)$$

A value of 1 means that all the test cases (XSS attack vectors) are interpreted: the web browser is thus potentially vulnerable to the full XSS test set. On the contrary, a value of 0 means that the web browser is not sensitive to this XSS test set.

Symmetrically to the analysis of the exposure degree of a particular web browser, one can be interested to study the impact of a given XSS attack vector on a set of web browsers. The degree of noxiousness of a test case is thus related to the percentage of web browsers it potentially affects.

2) The Degree of Noxiousness $Nox(tc, WB)$: of a XSS test case tc, against a set of web browsers WB is defined as the percentage of "Pass" verdicts among the number of tested browsers:

$$Nox(tc, WB) = \frac{|\{verdict(tc, wb) = `Pass', wb \in WB|}{|WB|} \quad (2)$$

Nox equals 0 if the XSS attack vector is not interpreted by any web browser, and equals 1 if all web browsers interpret it.

To focus on the evolution of a family of web browsers, we need to estimate the convergence or divergence of the attack surface from one version to another. The attack surface distance is defined to measure how much a version differs in behavior from another. Two versions may have the same exposure degree while not being sensitive to the same attack vectors.

The browser attack surface is defined, in this paper, by the set of passing test results on a given browser. Since we want to compare evolutions between browsers, we also need a similarity measure:

3) The Attack Surface Distance: is defined as the hamming distance between browsers attack surface:

$$ASD(WB_1, WB_2) = Hamming(TR_1, TR_2) \quad (3)$$

Attack surface distance equals 0 if the two versions of a browser have exactly the same attack surface. Note that exposure degrees may be the same while two web browsers do not have the same exact attack surface. For instance, if $Pass(1) = \{tc1, tc4, tc5\}$ and $Pass(2) = \{tc1, tc2, tc3\}$, $ASD(1, 2)$ equals 4, while the threat exposure degrees are the same. Indeed, the version 2 is no more impacted by tc4 and tc5 but is now affected by tc2 and tc3. The *attack surface distance* thus reveals the number of differences between two versions in terms of sensitivity to a set of XSS attack vectors.

C. Experimental Design

The empirical study requires executing a set of XSS test cases on a large set of web browsers. This raises the question of the selection of the test cases.

1) XSS Vector Set: The XSS vector set was built from the XSS Cheat Sheet [6], the HTML5 Security Cheat Sheet [7] and UTF-7 XSS Cheat sheet [8], and a few "discovered" vectors using a n-cube test generation. To find new vectors, we exhaustively combined HTML4 tags and property sets with JavaScript calls and used the scalar product of those {tag,property,call} sets to generate XSS vectors.

From this approach, only 6 vectors were effective out of the 44 000 generated test cases, after retrieving variations of already known vectors. With such a systematic test cases generation, we neither consider the inter-dependencies between tags nor the related constraints to be satisfied in order to obtain a valid vector. The resulting vectors thus are sometimes invalid, such as calling HTML5 or SVG tags without the proper document type/content-type declared.

We then proceeded in three steps:

- union of the referenced sets
- manual filtering of redundant test cases
- replacement of the default payload with one payload dedicated to *XSSTestDriver* (for facilitating the elaboration of the oracle verdict).

Test cases are different when they are exercising different JS mechanisms. It is possible to artificially multiply the total number of XSS test vectors; however we wanted to get the smallest number of different test cases. This point is crucial for the internal diversity of the test benchmark we propose. Similar test cases would not be efficient to exhibit different behaviors for web browsers.

The XSS test cases we use represent a large variety of dissimilar XSS vectors. We adapted them to have a payload dedicated to results interpretation. The resulting test set contains 87 test cases, among them 6 generated by our systematic test generation method (which were unreferenced).

2) Browser Set: The browser set consists of various versions of the browser families from July 1998 to March 2011. The qualified browsers are: Internet Explorer, Netscape, Mozilla, Firefox, Opera, Safari and Chrome. When available, we also consider and compare mobile versions of the web browsers.

Browser installers were collected from oldapps.com. Installation and execution was automated using the AutoIT Framework running in several Windows XP virtual machines for compatibility purposes. Mobile versions were installed manually either within emulators or real smartphones when available.

3) Threats to Validity: The validity of the experiments relies on the relevance of the test cases. As far as we know, we have proposed the most comprehensive and compact set of different XSS test vectors. However, as shown in section 2, it is extremely difficult to be exhaustive: new attacks are difficult to find since they exploit very particular aspects of JS interpreters. New attack vectors can be found everyday by hackers, or may be still unreferenced in the literature and the security websites. To overcome this problem, we tried to generate new, still unreferenced, XSS test vectors: the results were quite disappointing (6 success out of 44000 trials) showing that the problem is similar to finding a needle in a haystack.

D. Technical Issues and Details

Existing frameworks, such as JS unit and JS test driver, do not meet fundamental requirements for systematic testing of web browsers: being non intrusive (the test environment must not impact on the test results), being compatible with any web browser (for systematic benchmarking) and allowing the test results to be easily interpreted (test oracle). The developed testing framework for XSS is called XSS Test Driver. Anyone can test his/her own browser here: [9] and source code is available here on github [10].

An XSS execution comes in two parts: the browser parses the HTML, identifying the parts of the Document Object Model and building an internal representation of it. Then it calls the identified JavaScript (from < script >tags or tags properties) and executes it if necessary (it is not always the case when it comes to onevent properties such as onload or onmouseover). In the technical report [11], we fully explain the reason why JSUnit and JS TestDriver were inadapted for launching all types of XSS test vectors on any kind of web browsers. We summarize the difficulty met when using JSTestDriver (intrusiveness of the testing framework).

IV. Empirical Results

The empirical study we present targets two objectives:

1) validating the applicability of our testing framework and
2) investigating to what extent main web browser families are tested by their developers with respect to a regression testing policy.

A. Testing Hypothesis H1

To test H1, we execute 84 XSS attack vectors against three categories of web browsers: modern/recent versions, mobile versions and some still used legacy versions of web browsers.

The result is a snapshot of main web browser's threat exposures. Table I and II shows the test results: On table I, we present the results against XSS test cases 3 to 45, and table II presents results from 45 to 87 (result 45 is repeated for presentation reasons). A black cell represents a *Pass* verdict. The three families of browsers appear, and for each of the web browser the threat exposure degree is presented in the first row (30 for IE8 means 30% of threat exposure degree). The noxiousness degree for each XSS test case is given on the last column, right. We provide these degrees considering all browsers in the web browser set. Test cases #53, #54 and #59 are based on HTML5 tags and properties, thus making them ineffective against legacy browsers.

Some XSS vectors *pass* with the majority of the browsers, while others *pass* only with a specific version. This is due to the implementation of various norms, and the quality of parser's behavior toward the norm (Ex: between IE6 and IE7 a significant effort was done toward the implementation of standards). Only few test cases are effective within the whole browser set. Vectors number #3 to #6 are basic < *script*> tag based XSS with various payloads delivery. #12 and #13 are < *body* > tags based XSS with an *OnLoad* event set to execute the payload. #17 is a < *script* > tag with doubled brackets to evade basic filters. Test data #19 offers a very interesting form of evasion based on a half-opened < *iframe*> tag loading the payload from a dedicated HTML page:

```
<iframe src=/inc/16/payload.html <
```

We observed that 29 collected vectors were not executed by any of the selected browsers for the following reasons:

- some browser specific vector affects a precise version, like #15 from the XSS Cheat Sheet[6] which works only with specific version of firefox 2.0 and Netscape 8.1.
- Some failed due to improper test context like the character set used for the test suite, or the wrong DTD or content type, showing that context-dependent and context independent vectors exists.
- Some vectors made the browser unstable or crash, like

```
<DIV STYLE="width:expression(
  eval(String['fromCharCode']
   (97,108,101,114,116,40,39,
     120,115,115,39,41,32)
   ));">
```

  which plunge IE in a some kind of polling loop against the server

Some web browsers have similar behaviors. However, we can remark that all columns are different, meaning that each web browser has a different "signature" when submitted to our testing benchmark. When the signature is very similar, this reveals a JS interpretation engine that is based on the same initial implementation. Most popular web browsers are not exactly sensitive to the same attack vectors, and many of them have very different signatures.

1) Application to test cases selection: This snapshot opens a new perspective for security test cases selection. As shown, each web browser has its own threat exposure, and each attack vector is carrying a potential noxiousness degree. The table offers a very simple way to select a subset of web browsers enabling a maximum number of attacks. We can thus use this matrix to select the test cases that can be used for testing a web application for a given category of web browsers. For instance, test cases (#10, #23, #40, #80) are not noxious for modern web browsers. The fast-paced development of nowadays browsers makes difficult to track the effectiveness of a XSS vector, and

TABLE I
TEST RESULTS FOR VECTORS 1 TO 42

Vector / Browser	Chrome 11.0.696.68	IE 8.0.6001.19048	Opera mobile 11	Opera 11.11 rev2109	ie mobile	Safari Mac OSX	iPhone 3GS	Android 2.2	Firefox 5 Android	Firefox 8.0a1	IE 6.0.2900.2180	Firefox 2.0.0.2	Netscape 4.8	ie 4.01	opera 4.00
3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1
5	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1
6	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
7	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0
8	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
9	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
10	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
11	1	1	0	0	1	1	1	1	1	1	1	1	0	1	0
12	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
13	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
14	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	1	1	1	1	1	0	0	0	1	1	1	1	0	0	0
17	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
18	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
21	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
22	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	1	0	0	0	0	0	1	0	1	1	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
31	1	0	1	1	0	1	1	1	0	0	0	1	0	0	0
32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
33	1	1	0	0	1	1	1	1	1	1	1	1	0	1	0
34	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
35	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
36	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
37	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
38	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0
39	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
41	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
42	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
43	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0
44	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
45	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

TABLE II
TEST RESULTS FOR VECTORS 42 TO 84

Vector / Browser	Chrome 11.0.696.68	IE 8.0.6001.19048	Opera mobile 11	Opera 11.11 rev2109	ie mobile	Safari Mac OSX	iPhone 3GS	Android 2.2	Firefox 5 Android	Firefox 8.0a1	IE 6.0.2900.2180	Firefox 2.0.0.2	Netscape 4.8	ie 4.01	opera 4.00
45	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
46	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
47	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
48	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
49	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
50	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
51	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
52	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
53	1	0	0	0	0	1	1	1	1	1	0	0	0	0	0
54	1	0	0	0	0	1	1	1	1	0	0	0	0	0	0
55	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
56	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
57	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
58	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
59	1	0	1	1	0	1	1	1	1	1	0	0	0	0	0
60	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
61	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
62	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
63	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
64	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
65	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
66	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
67	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
68	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
69	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
70	0	1	0	0	1	0	0	0	0	0	1	0	0	0	0
71	0	1	0	0	1	0	0	0	0	0	1	0	0	1	0
72	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
73	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
74	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
75	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
76	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0
77	0	1	0	0	1	0	0	0	0	0	1	0	0	1	0
78	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
79	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
80	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
81	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
82	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
83	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0
84	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
85	0	0	0	0	0	0	0	1	1	1	0	0	1	1	0
86	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
87	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

when a new vector is discovered, it can be quite tedious to test it against several browsers. XSS Test Driver solves this issue, and eases comparisons.

2) Modern browsers have similar behaviors: With the considered modern browsers, 32 of the 84 test cases *pass*. We observe similar behaviors for some web browsers. Safari and Chrome's behaviors against the 84 test cases are exactly the same except for test #16 and #83. This could be easily explained because Chrome uses *Apple Webkit 534.3* as rendering engine, when the Safari version we tested uses the version *533.21.1* (version depicted by the User-Agent). This confirms that the HTML Parser matters for XSS execution.

3) Mobile versus desktop browsers: For mobile browsers, 43 test cases *pass* among the 84 ones for at least one mobile browser, but the passing test cases are quite different from the set of passing ones for modern web browsers. If we compare the results of the Safari mobile with the desktop version, we can see that the results are the same, since they use the same codebase (table III).

4) Parsing engine and mobile browsers: When comparing mobile and desktop versions of the same browser's family, we can observe slight differences, like between Opera mobile and desktop, or Firefox 4 mobile and desktop (table III). H1 is also verified, meaning that, even with very close browsers, the behaviors are not exactly the same. Between Opera mobile and desktop, only one vector execution changes:

```
<input onfocus=javascript:eval(
   String['fromCharCode'](
     97,108,101,114,116,40,
     39,120,115,115,39,41,
     32)
   ) autofocus>
```

Since they embed the same *Presto* engine, they recognize the same vectors, but the JavaScript events are interpreted differently due to specificity of mobile browsing, here the *onfocus* event. The same behavior can be observed between Firefox desktop and the mobile versions: the results are closed but different. Mobile browsers like Android's default browser

offer a "normal version" browsing function by changing the user-agent for a desktop one. When testing both mobile and standard version on XSS Test Driver, test results are the same, indicating that no specific rendering is done, relying only on the server's behavior. If we modify the mobile browser's options, we can impact its interpretation of vectors. As you can see in figure IV, the IE Mobile browser was set with a loose policy, and so he rendered more vectors than the version used in the table I and table II.

5) Legacy browsers are more exposed: While it is still broadly used in corporate environment, IE6 offers the highest threat exposure, with 45% $ThExp$.

B. Testing Hypothesis H2

Figure 8 presents the evolution of the threat exposures $ThExp$ over time. It clearly appears that no continuous improvements appear; many curves are chaotic and the exposure often increases. Figure 2 presents this evolution for Opera, which is released every six months. The number of XSS vectors that *pass* is given in dark columns. The ASD between the current version and the previous one is presented in grey columns (*attack surface distance*). Between Opera 10.50 (n) and 10.10 (n-1), while the number of passing vectors is close (23 and 17), the $ASD(TR_{Opera10.50}, TR_{Opera10.10})$ is high (12). It reveals a strong instability between these two minor versions instead of a stabilized behavior. It also reveals a lack of systematic regression testing from one version to another. This cannot be explained only by new norms implementations for HTML. As a result, there is no convergence, no strict decreasing or stabilization of the $ThExp$ from one version to another.

The same observations can be done related to Firefox (Figure 5), and IE (Figure 6). For IE, there are distances that are higher than the new number of passing XSS vectors ($ASD(TR_{IE5}, TR_{IE6})$ and $ASD(TR_{IE6}, TR_{IE7})$). It means that, from one version to the next one, the same web browser reacts in a different way to XSS attack vectors. This limit case reveals a lack of systematic regression testing methodology related to XSS attack vectors.

For Android (Figure 7), the evolution seems more straightforward, with a more or less constant threat exposure degree and small variations of distance values. To conclude, since in all cases there is no constant improvement for any web brother, we consider that the hypothesis H2 is validated: web browsers are not systematically tested w.r.t. their sensitivity to XSS vectors.

Web browser attack surface main evolutions from one version to another cannot be due only to external factors, such as changes in HTML standard definition or JavaScript. If these changes force the web browser implementations to evolve, they do not explain the chaotic evolutions of attack surfaces. The attack surface is not strictly decreasing or stabilizing from one version to another.

Most of the validation efforts from w3c are focusing on the HTML standard, but not on the browser's behavior. One reason is the difficulty to automate testing and make it cost-efficient.

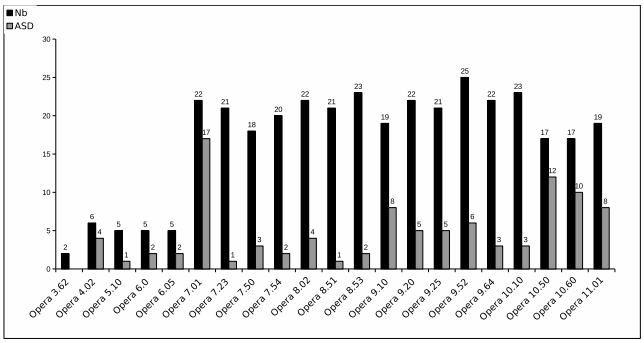

Fig. 2. Opera regression. passing vectors / $ASD(TR_n, TR_{n-1})$

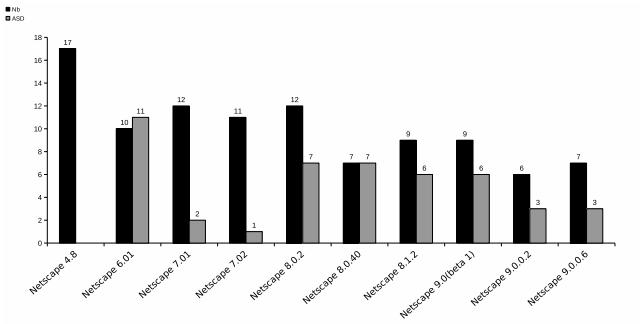

Fig. 3. Netscape regression. passing vectors / $ASD(TR_n, TR_{n-1})$

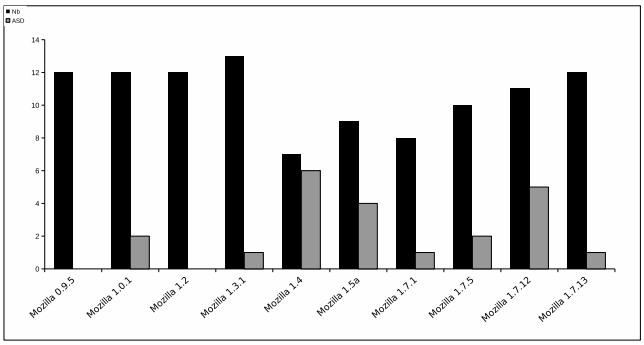

Fig. 4. Mozilla regression. passing vectors / $ASD(TR_n, TR_{n-1})$

XSS Test Driver can be used to ensure such regression testing. It allows determining, for a given web browser:

- its exposure to XSS vectors over time
- its behavioral stability from a version to another.

This experiment shows that systematic regression testing is feasible with XSS Test Driver and opens new research issues for test selection and diagnosis of web browsers.

V. Related Work

As far as we know, no previous work studies how to automatically execute and compare a set of XSS test cases. However, several work, including ones by the authors, proposed techniques and tools for automatically testing the security policies (access control policies) [12],[13],[14],[15]. Others

TABLE III
MOBILE BROWSER VS DESKTOP BROWSER COMPARISON

	0	1	2	3	8	9	10	13	14	16	26	28	30	34	35	36	40	42	43	47	48	50	51	53	56	69	70	72	74
Opera 11 Desktop	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**1**
Opera 11 Mobile	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**1**
Firefox 4 desktop	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**0**
Firefox 4 mobile	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**

TABLE IV
MOBILE BROWSER RESULT COMPARISON

Browser/Vector	3	4	5	6	7	9	11	12	13	14	16	17	18	20	21	22	26	31	33	34	35	36	37	38	41	42	43	46	48	50	53	54	56	59	61	64	68	70	71	74	76	77	83	85
ie mobile	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	0	0	0	0	0	1	1	1	1	0	1	1	1	0
Opera mobile 11 Android	1	1	1	1	0	0	0	0	1	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	1	0	0	0
iPad 2	1	1	1	1	0	0	1	1	1	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	1	0	0	0	0	1	0	1	0
Nokia E65	1	1	1	1	0	0	0	1	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0
iPhone 3GS	1	1	1	1	0	0	1	1	1	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	1	0	0	0	0	1	0	1	0
n810 tablet browser	1	1	1	1	0	0	1	1	1	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	0
Firefox 4.0.2 Android	1	1	1	1	0	0	1	1	1	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	1	0	0	0	0	1	0	0	0
Opera mobile Emulator	1	1	1	1	0	0	0	1	1	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	1	0	0	0
iPhone 3GS	1	1	1	1	0	0	1	1	1	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	1	0	0	0	0	1	0	1	0
Android 2.2 (Htc desire z)	1	1	1	1	0	0	1	1	1	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	1	1	1	0	1	0	1	0	0	0	1	1	0	1	1

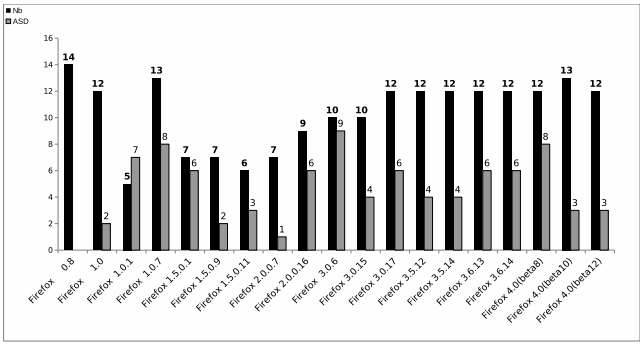

Fig. 5. Firefox regression. passing vectors / $ASD(TR_n, TR_{n-1})$

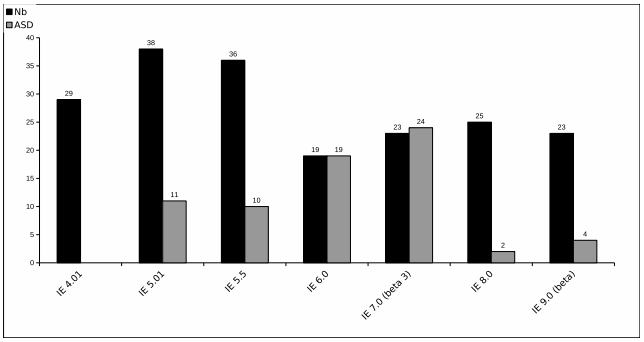

Fig. 6. Internet Explorer regression. passing vectors / $ASD(TR_n, TR_{n-1})$

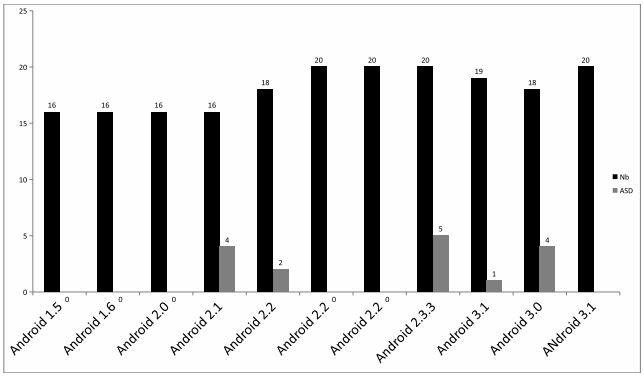

Fig. 7. Android regression. passing vectors / $ASD(TR_n, TR_{n-1})$

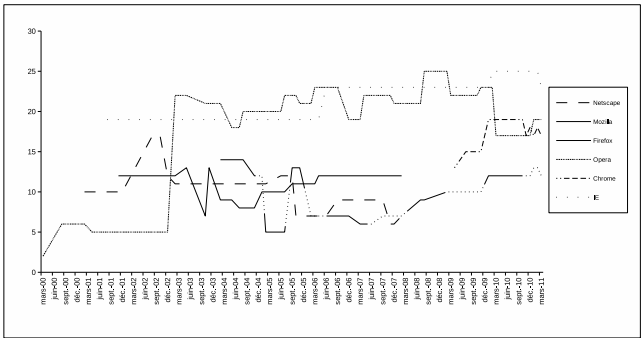

Fig. 8. Browsers' XSS Exposure over Time

offer frameworks and techniques to test the systems from its interfaces [16],[17]. Closer to the XSS testing technique we propose is the approach of bypass testing proposed by Offutt et al. [12][18]. We go along the same lines in this paper, but with a specific focus on XSS test selection and systematic benchmarking through testing (and we do not bypass client-side browser mechanisms since it's a part of the XSS target). Similarly to Su's statements [19], Huang et al. [20] propose to mutate and inject faulty inputs, including SQL injection and XSS against web application (WAVES tool), but do not provide a diagnosis technique to distinguish the various security layers and validate the capacity of an XSS vector to pass in a web browser or not. The only XSS test case evaluation methodology we found was done using mutation based testing [21]: a test data set was qualified by mutating

the PHP code of five web applications. XSS attacks were used to kill the mutants. In their study, they do not consider the impact of the browser on the efficiency of an XSS vector, thus introducing a bias in their experiments. They also used similar sources for the XSS vectors, and used them without adapting them to the specific injection point. Doing this, you introduce a bias in the efficiency of the attacks. Attacks should be tailored to the injection point to be effective like in Duchene et al. approach[22]; otherwise, depending on the injection point, your XSS attack can be rendered useless (while with the same vector, an attacker can succeed). Most of XSS research works focus either on detection of XSS attacks [1],[3], or on finding XSS vulnerabilities [23],[24]. Other related papers study XSS vulnerabilities or XSS worms [25],[26]. A state of the art on XSS issues and countermeasures is available in [25]. Undermining the influence of charset, doctype and browser behavior in an xss attack can lead to false positives in web application vulnerability scanners. Some testing strategies rely on one instrumented web browser [22],[27] to assess XSS Vulnerabilities, thus ignoring vulnerabilities related to XSS vectors bound to a specific web browser. The only exception in this topic is the xenotix XSS testing tool[28] wich embeds 3 different browser engines (Trident from IE, Webkit from Chrome/Safari and Gecko from Firefox) to deal with browser-specific XSS vectors.

VI. Conclusion

In this paper, we present a methodology and a tool for accurately testing web browsers against XSS vectors. The XSS Test Driver framework is a building block to address this issue. To demonstrate the feasibility of the approach, we execute a set of XSS test cases against popular web browsers.

We performed a first experiment that compares current web browsers and leads to the conclusion that they do not exactly behave the same way under XSS, even when they embed the same JS interpretation engine. The second investigation addresses the question of the improvement of web browsers on a 10 years period. We observe that there is neither a clear systematic reduction or stabilization of the attack surface nor any logic in the way the web browsers react to the XSS test cases. This result pleads for a systematic use of security test regression technique. For that purpose, we provide a first set of test cases [9] and a set of practices that can be used both by web browser developers and by their users.

VII. Acknowledgments

Authors would like to thanks KEREVAL's CTO Alain Ribault who gave the XSS Test Driver code to the community; DGA-MI; Ingrid Kemgoum from Telecom-Bretagne. The sla.ckers community, RSnake and .mario for the initial vectors. All the reviewers. This publication is a part of the DALI (Design and Assessment of application Level Intrusion detection systems) project (2008-2012) funded by the French national research agency (ANR ARPEGE 2008).

References

[1] "Mod security," http://www.modsecurity.org/.
[2] P. K. Manadhata and J. M. Wing, "An attack surface metric," *Software Engineering, IEEE Transactions on*, vol. 37, no. 3, pp. 371–386, 2011.
[3] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel, "Swap: Mitigating xss attacks using a reverse proxy," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*. IEEE Computer Society, 2009, pp. 33–39.
[4] E. Nava and D. Lindsay, "Abusing internet explorer 8's xss filters," *BlackHat Europe*, 2010.
[5] "détails de l'attaque aurora," http://fr.wikipedia.org/wiki/Op%C3%A9ration_Aurora.
[6] "Xss cheat sheet," http://ha.ckers.org/xss.html.
[7] "html5 security cheat sheet," http://html5sec.org/.
[8] "Utf7 xss cheat sheet," http://openmya.hacker.jp/hasegawa/security/utf7cs.html.
[9] "Xss test driver demo," http://xss.labosecu.rennes.telecom-bretagne.eu/.
[10] "Xss test driver sources," https://github.com/g4l4drim/xss_test_driver.
[11] "Xss test driver technical report," http://xss.labosecu.rennes.telecom-bretagne.eu/doc/.
[12] J. Offutt, Y. Wu, X. Du, and H. Huang, "Bypass testing of web applications," in *Proc. of ISSRE*, vol. 4.
[13] E. Martin and T. Xie, "Automated test generation for access control policies via change-impact analysis," in *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*. IEEE Computer Society, 2007, p. 5.
[14] Y. Le Traon, T. Mouelhi, and B. Baudry, "Testing security policies: Going beyond functional testing," 2007.
[15] T. Mouelhi, F. Fleurey, B. Baudry, and Y. Le Traon, "A model-based framework for security policy specification, deployment and testing," *Model Driven Engineering Languages and Systems*, pp. 537–552, 2008.
[16] H. Liu and H. Kuan Tan, "Testing input validation in web applications through automated model recovery," *Journal of Systems and Software*, vol. 81, no. 2, pp. 222–233, 2008.
[17] A. Tappenden, P. Beatty, and J. Miller, "Agile security testing of web-based systems via httpunit," 2005.
[18] J. Offutt, Q. Wang, and J. Ordille, "An industrial case study of bypass testing on web applications," in *2008 International Conference on Software Testing, Verification, and Validation*. IEEE, 2008, pp. 465–474.
[19] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *ACM SIGPLAN Notices*, vol. 41, no. 1. ACM, 2006, pp. 372–382.
[20] Y. Huang, S. Huang, T. Lin, and C. Tsai, "Web application security assessment by fault injection and behavior monitoring," in *Proceedings of the 12th international conference on World Wide Web*. ACM, 2003, pp. 148–159.
[21] H. Shahriar and M. Zulkernine, "Mutec: Mutation-based testing of cross site scripting," in *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*. IEEE Computer Society, 2009, pp. 47–53.
[22] F. Duchene, R. Groz, S. Rawat, and J. Richier, "Xss vulnerability detection using model inference assisted evolutionary fuzzing," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 815–817.
[23] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 332–345.
[24] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 2008, pp. 171–180.
[25] D. Jayamsakthi Shanmugam1, "Cross site scripting-latest developments and solutions: A survey," *Int. J. Open Problems Compt. Math*, vol. 1, no. 2, 2008.
[26] M. Faghani and H. Saidi, "Social networks' xss worms," in *2009 International Conference on Computational Science and Engineering*. IEEE, 2009, pp. 1137–1141.
[27] "Owasp xelenium," https://www.owasp.org/index.php/OWASP_Xelenium_Project.
[28] "Owasp xenotix xss exploit framework," https://www.owasp.org/index.php/OWASP_Xenotix_XSS_Exploit_Framework.