MBSD Technical Whitepaper

Identifier based XSSI attacks

Takeshi Terada / Mitsui Bussan Secure Directions, Inc.

March 2015

Table of Contents

1. Introduction 1

2. Attack techniques / vulnerabilities 2

2.1. (A) Simple IE's bug leaks runtime errors 2

2.2. (B) JSON and other data theft using UTF-16 4

2.3. (C) Harmony proxy bug in Firefox / Chrome 6

2.4. (D) Exhaustive search 7

2.5. (E) CSV with quotations theft 9

3. Conclusion 11

4. References 13

5. About us 14

1. Introduction

Cross Site Script Inclusion (XSSI) is an attack technique (or a vulnerability) that enables attackers to steal data of certain types across origin boundaries, by including target data using SCRIPT tag in an attacker's Web page as below:

```
<!-- attacker's page loads external data with SCRIPT tag -->
<SCRIPT src="http://target.example.jp/secret"></SCRIPT>
```

For years, XSSI has been known among Web security researchers that JavaScript file, JSONP and, in certain old browsers, JSON data are subject to this type of information theft attacks. In addition, some browser vulnerabilities, that allow attackers to gain information via JavaScript error messages, have been discovered and fixed in the past.

In 2014, we conducted research on this old topic and discovered some new attack techniques and browser vulnerabilities that allow attackers to steal simple text strings such as CSV, and more complex data under certain circumstances. In the research, we mainly focused on a method of stealing data as a client side script's identifier (variable or function name).

In this paper, we first describe these attack techniques / browser vulnerabilities in the next section and then discuss countermeasures for these issues.

 ©2015 Mitsui Bussan Secure Directions, Inc. All rights reserved.

2. Attack techniques / vulnerabilities

In the research, we found five XSSI-related attack techniques or browser vulnerabilities:

(A) Simple IE's bug leaks runtime errors

(B) JSON and other data theft using UTF-16

(C) Harmony proxy bug in Firefox / Chrome

(D) Exhaustive search

(E) CSV with quotations theft

The details of each item are explained in this section.

2.1. (A) Simple IE's bug leaks runtime errors

In order to prevent cross-origin information leakage via JS error messages, browsers present only fixed JS error messages like "Script error." to the pages loading the JS, when errors occur in the loaded JS file of different origin. However, in IE9 and IE10, it was not necessarily the case.

Technically, in the case of syntax errors occurring in the external JS, these browsers provide the Web pages with fixed error messages, like other non-vulnerable browsers do. However, in the case of runtime errors, these browsers provide the Web pages with the detailed error messages, such as "'foobar' is not defined", even if the loaded JS comes from another origin.

This means that, if a target Web site serves data that is syntactically valid as JavaScript, the web page loading the data can possibly gain some information about the data through JS error messages, even though the data is not supposed to be interpreted as JavaScript. Typical example of such data is plain CSV (Comma-Separated Values).

Suppose the target Web site serves CSV data shown below.

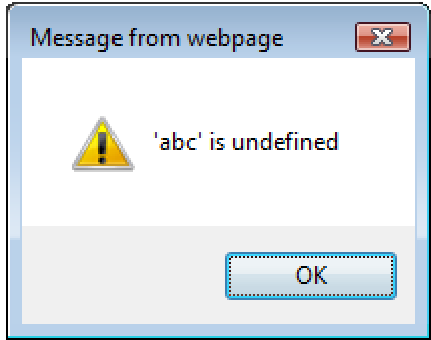
```
HTTP/1.1 200 OK
Content-Type: text/csv
Content-Disposition: attachment; filename="a.csv"
Content-Length: 13

1,abc,def,ghi
```

Attacker's Web page sets an error handler and loads this CSV as JavaScript using SCRIPT tag, and then induces a victim user to visit the attacker's page.

 ©2015 Mitsui Bussan Secure Directions, Inc. All rights reserved.

```
<!-- set an error handler -->
<SCRIPT>window.onerror = function(err) {alert(err)}</SCRIPT>
<!-- load target CSV -->
<SCRIPT src="(target data's URL)"></SCRIPT>
```

Then a popup `"'abc' is undefined"` appears, that means the attacker's Web page can successfully gain information of the target CSV data from different origin.



The error occurs, because the strings in the CSV, that are "abc", "def" and "ghi", are regarded as identifiers (global variable names) when interpreted as JavaScript, and they are not declared anywhere in the CSV or the attacker's JavaScript code. In addition, as mentioned earlier, these browsers allow a Web page to capture runtime errors of different origin.

For a successful attack, the stolen data must basically be valid JavaScript identifiers like "abc", "foo_bar", "たちつてと" and so on. In addition, the data, as a whole, must be a valid JavaScript code. Otherwise the attack fails due to a syntax error.

We reported this IE's issue to Microsoft in July 2014, and the vendor released the fix (MS14-080) in December 2014. CVE-2014-6345 was assigned to this bug [1][2]. By their fix, runtime error message from external JS was changed to a fixed one like other browsers. Note that only IE9 and IE10 were found to be vulnerable, that means neither IE7, 8, 11 nor other major browsers were vulnerable, when we tested them in July 2014.

As some researchers may have already noticed, this technique itself is not new. Back in 2008, Chris Evans disclosed a same bug in Firefox 3 [3]. To be exact, his exploit code is a bit more complex than ours, as his code utilizes redirection to trick the browser, while ours do not use any such tricks.

Another quite relevant research is one made by Yosuke Hasegawa with @masa141421356 in 2013 [4]. According to Hasegawa, Microsoft addressed a bug of IE6/7/8 that could lead to JSON array theft through VBScript error messages in MS13-037; meanwhile the exactly same bug of IE9/10 was left unfixed because the vendor regarded it as "behavior" then. But

 ©2015 Mitsui Bussan Secure Directions, Inc. All rights reserved.

this undesirable behavior seems to have been fixed in MS14-080, probably at the same time as the bug we reported was fixed. This implies the cause of our bug was completely same as that of his.

2.2. (B) JSON and other data theft using UTF-16

In the above bug, only data of limited formats, such as CSV, can be exploitable. Thus we made more research to expand exploitable types of data, and discovered a trick using UTF-16 encoding to achieve it. Using this trick, attackers can gain information on data of more various formats such as JSON under certain conditions.

The trick itself is quite simple. The following is an example of attacker's Web page, the emphasized part in which is only different from the attacker's page in (A).

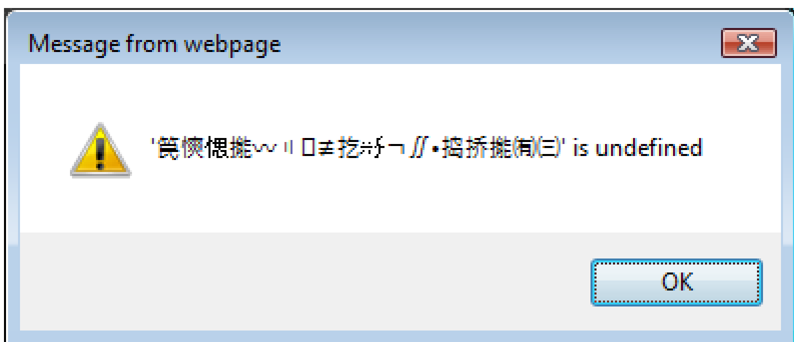
```
<!-- set an error handler -->
<SCRIPT>window.onerror = function(err) {alert(err)}</SCRIPT>
<!-- load target JSON -->
<SCRIPT src="(target data's URL)" charset="UTF-16BE"></SCRIPT>
```

An example JSON data being stolen is shown below.

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Disposition: attachment; filename="a.json"
Content-Length: 39

{"aaa":"000", "bbb":"111", "ccc":"222"}
```

When the response lacks charset specification, by adding charset attribute to the SCRIPT element in the attacker's page, he can designate in which charset the loaded JS file is processed on the browser. In this example, the attacker is loading the external JSON in UTF-16 (UTF-16BE), and it basically succeeds in many browsers including IE9.

In this attack, a text looks garbled shows up in a popup box.



 ©2015 Mitsui Bussan Secure Directions, Inc. All rights reserved.

The reason why you see the garbled text is that the JSON bytes are decoded as UTF-16 by the browser, since the SCRIPT element has `charset="UTF-16BE"` attribute. The decoding process is shown in the table below:

Original String:	`{"`	`aa`	`a"`	`:"`	`00`	`0"`	...	`22`	`2"`	`}`
HEX:	7B22	6161	6122	3A22	3030	3022	...	3232	3222	7d--
Decoded as UTF-16:	箢	慡	愢	㨢	〰	〢	...	㈲	㈢	(Ignored)

For example, the first two bytes of the JSON string is `'{"'`, 0x7B22 in hex. When browsers decode these bytes as UTF-16, they turn into a character U+7B22 (箢). Obviously, you can regain the original JSON string from the garbled string, just by reversing the process.

One important caveat is that the attack succeeds only if the garbled string constitutes a valid JavaScript identifier. In this example, for instance, the original JSON string begins with `'{"aaa"...'` (0x7B22 0x6161 0x6122...), and any of U+7B22, U+6161, U+6122 and the rest is valid as JS identifier, so that you can successfully gain the value. If it is not the case, a syntax error, rather than a runtime error, occurs and the attack fails.

Regarding this restriction, IE is quite permissive and is a convenient platform for attackers, because IE treats more characters as a part of a valid JS identifier, than ECMAScript specification [5] and other browsers do. For instance, `'3q'` (U+3371, ㍱) is in the Unicode category "Symbol, Other [So]", and any character in the category should not occur in an identifier according to ECMA specification, while IE generously accepts the character. It seems that the same is true of characters in "Number, Other [No]" and "Punctuation, Other [Po]".

We researched on what character combinations can be a part of a valid identifier in some browsers, when decoded as UTF-16. As a result, 99.3% of alphanumeric combinations (3,816 out of 3,844, 3,844 = 62^2) are confirmed to be valid in IE9. As shown in the table below, the percentage in IE is considerably higher than that in Chrome or Firefox.

	Alphanumeric `[a-zA-Z0-9]{2}`	Non-ctrl ASCII `[\x20-\x7E]{2}`
IE9	99.3% (3,816 / 3,844)	91.7% (8,274 / 9,025)
Chrome/Firefox	96.7% (3,716 / 3,844)	83.4% (7,526 / 9,025)

The table also shows that the percentage in the alphanumeric case is higher than that in the non-ctrl ASCII case; therefore attacks to a string containing fewer symbols are more likely to succeed in general.

Note that this attack is applicable not only for IE but also for other browsers, if you combine

 ©2015 Mitsui Bussan Secure Directions, Inc. All rights reserved.

this technique with others such those explained later in this paper. This technique, however, would be less practical in other browsers, because character restriction in them is stricter than that in IE as shown in the table above.

One more thing to note is that, in IE10 or later, this attack may not work because they apparently refuses to regard scripts without NUL byte nor BOM (Byte Order Mark) as encoded in UTF-16.

2.3. (C) Harmony proxy bug in Firefox / Chrome

Harmony is a new version of ECMAScript (ver.6), and its standardizing process is in progress as of this writing. It introduces some new language features, one of which is a JavaScript proxy [6]. Firefox and Chrome support this feature at the time of the research.

Similar to Java's reflection Proxy class, JS proxy can customize fundamental operations such as property lookup, assignment, function call etc. In the research, we found this can be utilized to steal simple text strings such as CSV.

An attacker's Web page is shown below:

```
<!-- set proxy handler to window.__proto__ -->
<SCRIPT>
var handler = {
    has: function(target, name) {alert("data=" + name); return true},
    get: function(target, name) {return 1}
};
window.__proto__ = new Proxy({}, handler);
</SCRIPT>

<!-- load target CSV -->
<SCRIPT src="(target data's URL)"></SCRIPT>
```

The key part is highlighted in yellow, where a proxy object is set to `window.__proto__`. Because of this, every access to an undefined global variable (i.e. property of window object) triggers function call defined in the handler.

The CSV file being stolen is:

```
HTTP/1.1 200 OK
Content-Type: text/csv
Content-Disposition: attachment; filename="a.csv"
Content-Length: 13

1,abc,def,ghi
```

When a victim user using Firefox accesses the attacker's page, popup boxes saying

 ©2015 Mitsui Bussan Secure Directions, Inc. All rights reserved.

`"data=abc"`, `"data=def"`, `"data=ghi"` shows up. This means the harmony proxy can be used for cross-origin information theft in certain situations like this. A similar attack was confirmed to be possible in Chrome too.

We reported this issue to both Mozilla and Google in August 2014. As for Chrome, at the time of report, JS proxy was disabled by default and can be enabled by changing a flag (chrome://flags/#enable-javascript-harmony) in the setting screen. In response to our report, the proxy feature was isolated from the setting flag [7].

As for Firefox, the bug is not yet fixed at the time of writing, but it was made open to public by Mozilla itself [8], because Erling Ellingsen (re)-discovered the same bug and publicized it on Twitter in November 2014 [9]. As already stated, the fix is not yet released, and no CVE number is assigned to this bug so far.

As you can learn in the bug tracking board of both Mozilla and Chrome [7][8], it is a bit arguable whether this is really a security bug needs addressing or just a behavior inherent in JS proxy. The latter assertion is supported by the fact that, even without JS proxy, attackers can possibly conduct exhaustive search attacks against external CSV and other syntactically JS-compliant data. The exhaustive search attack is explained in the next section.

2.4. (D) Exhaustive search

Let us use a CSV data as an example again. Suppose the attacker's Web page loads this CSV as JavaScript with SCRIPT tag.

```
HTTP/1.1 200 OK
Content-Type: text/csv
Content-Disposition: attachment; filename="a.csv"
Content-Length: 8

1,xyz123
```

If the attacker's Web page declares a JS variable whose name is "xyz123" before loading the JS (CSV), no "'xyz123' is undefined" error occurs, while it occurs without the variable declaration. This means that he can gain information on whether the data is "xyz123" or not, if he has a way to detect if the error occurred or not.

Generally, browsers do not provide Web pages with detailed error messages when the error occurred in external JS. But the page can still gain one bit information, that is if an error occurred or not, just by setting an error handler for instance. Therefore the attack that

 ©2015 Mitsui Bussan Secure Directions, Inc. All rights reserved.

checks the target value on each by each basis (i.e. brute-forcing) is possible.

There are three techniques that can sophisticate this attack.

The first technique is binary search. For instance, if you know the target value is any of the four values that are "xyz121", "xyz122", "xyz123", and "xyz124", you can first verify if the target is either of the first two values (i.e. "xyz121" or "xyz122") by defining these variables, then loading the data as JavaScript, and then checking if an error occurs. If an error fires, it tells that the target is either of the rest, that are "xyz123" and "xyz124". One more time trial can reveal which one the target is. This way, data of N bits can be identified by N times of data loading trials in general.

The second technique is to utilize JavaScript getter. With JS getter, you do not need to employ binary search, which means you can pinpoint the target value with only one-time data loading.

A sample program of getter is shown below:

```
<!-- set getters -->
<SCRIPT>
Object.defineProperty(window, "xyz121", {get: function() {alert("value=xyz121")}});
Object.defineProperty(window, "xyz122", {get: function() {alert("value=xyz122")}});
Object.defineProperty(window, "xyz123", {get: function() {alert("value=xyz123")}});
Object.defineProperty(window, "xyz124", {get: function() {alert("value=xyz124")}});
</SCRIPT>
<!-- load target CSV -->
<SCRIPT src="(target data's URL)"></SCRIPT>
```

A popup window "value=xyz123" shows up, when a victim using recent browsers visits the attacker's page. This happens because, the getter automatically invokes "get" method, when an access to the corresponding property (`window.xyz123` in this case) takes place.

The third technique is to use VBScript for JSON array theft on IE. The basic idea of using VBScript and JSON array in combination comes from Hasegawa's work [4]. What we found is that they can be applied to exhaustive search attacks.

The target JSON array is as below:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Disposition: attachment; filename="a.json"
Content-Length: 12

[1,"xyz123"]
```

You can identify the value by using the VBScript code:

 ©2015 Mitsui Bussan Secure Directions, Inc. All rights reserved.

```
<!-- set VBScript procedures -->
<SCRIPT language="vbscript">
Sub [1,"xyz121"]: MsgBox "value=xyz121": End Sub
Sub [1,"xyz122"]: MsgBox "value=xyz122": End Sub
Sub [1,"xyz123"]: MsgBox "value=xyz123": End Sub
Sub [1,"xyz124"]: MsgBox "value=xyz124": End Sub
</SCRIPT>
<!-- load target JSON as VBScript -->
<SCRIPT src="(target data's URL)" language="vbscript"></SCRIPT>
```

When a victim navigates to the attacker's page, a popup window "value=xyz123" appears. This indicates that, like the getter attack, the attacker can pinpoint the target JSON array.

The reason why it works is that VBScript's identifiers can be placed inside square brackets, and symbols can occur in identifiers as long as they are inside square brackets. This means something like `[1,"xyz123"]` is, as an entire string, a valid identifier representation in VBScript. The identifier in the target data invokes the corresponding procedure when it is loaded as VBScript, simply because VBScript does not need parentheses to invoke procedures. Due to this attack mechanism, only plain JSON arrays, that are both surrounded by square brackets and containing no newline nor right square bracket (`"]"`) character inside the outer square brackets, can be a target, however.

In either of these attacks, exhaustive search is possible if the unknown part of the data has considerably small amount of entropy. Note that no browser bug is necessary to conduct this attack, and thus it basically works on all major browsers, VBScript attack works only on IE though.

2.5. (E) CSV with quotations theft

The above attack examples targeting CSV do not work if quotation characters enclose the strings in the CSV. A simple trick, however, enables successful data theft attacks, when an attacker controls some parts of the CSV.

Let us suppose the following CSV is served:

```
HTTP/1.1 200 OK
Content-Type: text/csv
Content-Disposition: attachment; filename="a.csv"
Content-Length: xxxx

1,"___","aaa@a.example","03-0000-0001"
2,"foo","bbb@b.example","03-0000-0002"
...
98,"bar","yyy@example.net","03-0000-0088"
99,"___","zzz@example.com","03-0000-0099"
```

Suppose attackers can inject arbitrary strings in each of the highlighted parts, but injected strings will be escaped according to the RFC regarding CSV (RFC 4180 [10]), when output to the CSV. In short, simply doubling quotation character (" → "") is defined in the RFC as the escaping method, which is different from that for JavaScript string.

In this scenario, what the attacker wants to obtain is the content between the two highlighted parts. The attacker can successfully reach the goal, taking advantage of the difference in escaping method. The following CSV actually does the trick.

```
1,"\"",$$$=function(){/*","aaa@a.example","03-0000-0001"
2,"foo","bbb@b.example","03-0000-0002"
...
98,"bar","yyy@example.net","03-0000-0088"
99,"*/}//","zzz@example.com","03-0000-0099"
```

In the first highlighted part, one quotation character supplied by the attacker is doubled to two. When the CSV is parsed as JavaScript, the backslash and the two quotation characters allow you to safely get out of the quoted string, so that you can inject whatever JS expressions or statements you like.

A tricky part is gaining multiline contents in JavaScript, because multiline string literal is illegal in JavaScript. In the above example, `$$$=function() {/*…*/}` is used to do the task. Then the attacker can call `$$$.toString()` that returns the source code of the function including the comment content, that is what the attacker wants. This attack works on all major browsers.

Another technique to handle multiline contents is to use ECMAScript6 template that is supported by Firefox and Chrome. In these browsers, a template string literal surrounded by backquote characters can grab multiline content as well.

 ©2015 Mitsui Bussan Secure Directions, Inc. All rights reserved.

3. Conclusion

As shown in the previous section, some browser vulnerabilities and attack techniques can result in leakage of certain types of data such as CSV and JSON, through XSSI attacks using identifiers. These attacks are not universally applicable to Web contents, but they matter and need addressing in certain conditions.

The primary countermeasure for these attacks is to set `X-Content-Type-Options: nosniff` in the response header of the data you serve. With the header, browsers reject CSV and other data with non-script MIME type designation when they are loaded as JavaScript.

Obviously, an appropriate `Content-Type` field with appropriate charset designation must be in place to prevent XSS and other attacks. Charset field is for protecting your contents from charset-based attacks that use UTF-16 and other exotic encodings including UTF-7 [11].

One thing should be noted here is that this sort of attacks or vulnerabilities is not specific to a certain browser, namely IE. As you saw in the previous section, some types of attacks affect browsers other than IE, e.g. (C) affects Firefox and Chrome, (D) and (E) affect all major browsers.

Taking that into account, it is unfortunate for us that `X-Content-Type-Options` header is supported only in IE8+ and Chrome, as far as we know. In other words, it is not available in Firefox, Safari and other legacy browsers including IE7, and attacks of type (C), (D) and (E) cannot be prevented by this header in these browsers.

In the case of Firefox, there has been a discussion about whether or how to implement the header since 2008, but they have not yet reached a conclusion to proceed at the time of writing [12]. One of the reasons for the prolonging discussion is that the header was originally a Microsoft's own expansion and there has not been a clear and established standard of it, the MIME sniffing standardizing seems to be in process though [13].

Considering these, to eliminate windows of nasty attacks like (C), (D) and (E), it is recommended that you implement measures, in addition to setting proper response headers (`Content-Type` with charset and `X-Content-Type-Options`).

The additional measures would be:

- Reject requests using GET method
- (and/or) Use hard-to-guess parameters in URL or request body

 ©2015 Mitsui Bussan Secure Directions, Inc. All rights reserved.

- (and/or) Use custom header when content is supposed to be retrieved via XHR

Just reject the HTTP request that does not satisfy the condition(s). For instance, you should just stop serving the content whenever you find the method is GET, or the request does not contain the valid parameter or header.

These additional measures are necessary only when you provide highly sensitive data potentially valid as JavaScript (or VBScript) against your intention, and when your data needs protection against attacks of type (C), (D) and (E). Specifically, CSV, JSON array and other custom-formatted data could be in the scope of these measures.

Finally, we would like to emphasize the significance of browsers adopting or backporting `X-Content-Type-Options`, because we think the header is the most simple-to-apply solution ensures that Web contents are completely immune from XSSI and other content sniffing attacks.

 ©2015 Mitsui Bussan Secure Directions, Inc. All rights reserved.

4. References

[1] https://technet.microsoft.com/en-us/library/security/dn820091.aspx

[2] http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6345

[3] http://scary.beasts.org/security/CESA-2008-011.html

[4] http://www.slideshare.net/hasegawayosuke/owasp-hasegawa

[5] http://www.ecma-international.org/ecma-262/5.1/#sec-7.6

[6] http://wiki.ecmascript.org/doku.php?id=harmony:direct_proxies

[7] https://code.google.com/p/chromium/issues/detail?id=399951

[8] https://bugzilla.mozilla.org/show_bug.cgi?id=1048535

[9] https://twitter.com/steike/status/533198334547468288

[10] http://tools.ietf.org/html/rfc4180

[11] http://sebug.net/paper/charset/BlackHat-japan-08-Hasegawa-Char-Encoding.pdf

[12] https://bugzilla.mozilla.org/show_bug.cgi?id=471020

[13] https://mimesniff.spec.whatwg.org/

 ©2015 Mitsui Bussan Secure Directions, Inc. All rights reserved.

5. About us

About MBSD

MBSD (Mitsui Bussan Secure Directions, Inc.) is the Japanese leading security company in managed security services, vulnerability assessment and testing, GRC (Governance, Risk, Compliance) consulting, incident response and handling, digital forensics, and secure programming training services. The MBSD services are provided by its personnel including the leading security experts in the field of secure programming, application security, penetration testing and threat analysis who have in-depth knowledge and understanding of attackers' methodologies. MBSD is working for the Internet infrastructure companies, cyber commerce and media giants, financial institutes, global enterprise, and government agencies in Japan to support their strategies against rapidly increasing threats from cyber space.

About the author

Takeshi Terada, MBSD Professional Service Dept., Senior Security Specialist, CISSP

TT-1 Building 6F, 1-14-8, Nihonbashi Ningyo-Cho, Chuo-ku, Tokyo, 103-0013, Japan

+81-3-5649-1961 | http://www.mbsd.jp/

 ©2015 Mitsui Bussan Secure Directions, Inc. All rights reserved.