JPCERT CC®

Japan Computer Emergency Response Team Coordination Center

Investigation Report
Regarding Security Issues of Web Applications Using HTML5

JPCERT Coordination Center

October 30, 2013

JPCERT CC®

Contents

1. Introduction

In a narrow sense, HTML5 is a specification of a markup language for structuring and presenting contents on the World Wide Web, and it is in the process of formulation by WHATWG and W3C as the next generation of HTML standard which will take over HTML4, the current standard. In a broad sense, HTML5 is a framework for describing web application, including not only the mark-up language but also WebSocket protocol and other related Internet standards. HTML5 is rapidly gathering population according to the inquiry held by Evans Data, an investigation company, in January 2012, in which 58 percent of engineers answered to adopt HTML5 when developing a new web site. A large number of vendors and entities including web browser developers have been working closely on the implementation of HTML5 in order to respond to such strong market needs, however, they sometimes implement incompatible details in their products such as browser because of the insufficient draft-specification still reviewed by the standardization entity.

While HTML5 and its peripheral technologies enable more flexible and convenient web sites to be developed, such as Web Storage storing data in the browser of the web site viewer (herein, “user”), WebSocket for a server and a client to communicate in full-duplex, and Geolocation acquiring geographical location information, , considerations and notifications regarding how attackers may exploit these new features have yet to be properly performed and thus there are concerns that HTML5 may become more and more prevalent without proper security measures being put in place.

This investigation was performed to provide web security researchers and web application developers with the basic information about how HTML5 migration may affect web application security by organizing currently known issues. We delegated this research project to web security experts who mainly focused on vulnerabilities that may be built in as a result of using newly added features in HTML5, and methods on utilizing newly added features when building secure web applications as well as considerations upon using such features. As mentioned above, there are discrepancies in the implementation of HTML5 specifications. The results in this report are based on browser implementations from August, 2012 to February, 2013. The list of browsers used in this investigation is listed in the Appendix.

This report is composed of 6 sections.

Section 1 provides the overview of this investigation and Section 2 will continue to describe the purpose and methods for this investigation.

Section 3 gives an overview of long-lasting web application vulnerabilities and respective countermeasures that must be carefully considered in the development phase of the web applications. This section also describes the new attack vectors introduced by the HTML5 as well as

web application development considerations that were not necessary in the past but have become necessary due to HTML5.

Section 4 contains a description of pitfalls leading to the vulnerabilities in web applications and corresponding mitigations from the point of view of the newly added features in the HTML5 and widely used functions in web application development using the HTML5.

Section 5 contains information on browser security features that are particularly effective for secure operations of HTML5 web applications by focusing on how to configure and implement these features.

Section 6 gives a conclusion of this report.

The contents of this report were created so that it can act as a starting point for discussions on how to develop secure web applications using HTML5. Since this report focuses on HTML5 features used in web applications, basic technologies related to conventional web application security may not be mentioned here. Please refer to the "How to Secure Your Website" document provided by the Information-technology Promotion Agency (herein, IPA) for such information.

Also, not a few specifications of HTML5 still remain undetermined, and for this reason, it is necessary to continuously pay attention to the standardizing processes and their impacts on security. In addition, it cannot be denied that new attack methods specific to HTML5 may be discovered in the future. Therefore, we believe that this report needs to be updated periodically by adding new findings and feedbacks from readers of this report, since this report lists security issues on HTML5 that were discovered at the time of the investigation and the basic mitigations against such issues.

2. Purpose and Method of This Investigation

2.1. Purpose

The purpose of this investigation was to examine the mechanism of the vulnerabilities themselves and basic mitigations to avoid from such vulnerabilities. It was additionally to provide materials that can serve as a basis for technical documentations and guidelines for secure web application development, regarding major vulnerabilities that tend to be accidentally implemented into the HTML5 web applications and frequently utilized in the HTML5 web applications.

2.2. Method

In this investigation, we made the following list of newly introduced features in the HTML5, and offered web application security experts to investigate security risks and mitigations.

- HTML elements that have been newly added or functionally expanded:
  meta
  a
  iframe
  video
  audio
  source
  canvas
  input
  button
- Newly defined JavaScript API
  WebSocket
  Web Storage
  Offline Web Application
  Web Workers
  Cross Document Messaging

In this investigation, we first listed up security risks of each feature and studied them, then inspected them as much as possible, and finally compiled this report. In addition, we also investigated other security issues and included them into this report.

3. Vulnerabilities that need to be noted especially in HTML5

In HTML5, new elements and attributes were added in order to expand features to present contents in web browsers. Misuse of such elements and attributes may cause vulnerabilities to be built into web sites. In addition, web applications that had been secure in the past may became vulnerable since features newly enhanced to the HTML5 may become available for attackers to attack. Of those vulnerabilities, we focused and deeply analyzed the following 4 vulnerabilities that web application developers should especially keep in mind:

- Cross Site Scripting (XSS)
- Cross Site Request Forgery (CSRF)
- Open Redirect
- Lack of Access and Authorization Control

These vulnerabilities have existed prior to HTML5 and considered as features to be used with special care, but this section will describe any behavior that is specific to HTML5 and the methods to prevent them. For those who would like to read overviews of vulnerabilities that are specific to HTML5, please refer to section 4.

“Cross-Origin Communication” is an important mechanism that became available in HTML5 for the first time. “Origin” identifies the source of the contents. According to “The Web Origin Concept” in RFC6454, it is also defined as the combination of scheme, host and port components from which the browser loads the contents. If two different contents have the same scheme, host, and port, it means that they belong to “the same origin”. If a particular content has at least one of the scheme, host, or port component different from the other, it means they belong to a “cross origin”. If a content such as JavaScript communicates another content belonging to a cross origin, it is called “Cross-Origin Messaging”.

3.1. Cross Site Scripting

Cross Site Scripting (herein, ”XSS”) is an attack method where a malicious script is injected against a web site by an attacker and is sent from the web site to the user browser unintentionally. Vulnerabilities used in the XSS attacks (herein, XSS vulnerabilities) are built in when the web applications or JavaScript process various input data and dynamically generate the HTML document for the web browser, but such generation is not implemented properly so that the generated contents remains malicious scripts provided from the input data. In order to avoid XSS vulnerabilities, it is necessary to pay special attention during the dynamic generation of a context that can be interpreted as a script.

The following XSS vulnerabilities will be introduced here since the following may be built in a different

manner than the past due to the complications of HTML structure in HTML5.

(1) XSS triggered by new elements or attributes
(2) DOM Based XSS
(3) XSS caused by XMLHttpRequest Level2
(4) XSS caused by Ajax data

For (1) and (2), background information and countermeasures will be listed below in the section. (3) will be described in section 4.3.1 and (4) will be described in 4.3.3.

3.1.1. XSS triggered by new elements or attributes

Conventional HTML has had elements that can specify the “onerror” event handler. Error processes could be executed by setting JavaScript as the value for the “onerror” attribute in this element. Thus, attackers could conduct XSS attacks when they embed JavaScript into the value of “onerror” in a dynamically generated HTML document by causing an error condition. Multiple new elements have been added to HTML5. Of those elements, <video> and <source> elements support the “onerror” event handler, so that the specified JavaScript will be executed in the HTML document as shown below:

```
<video onerror="javascript:alert(1)">
    <source onerror="javascript:alert(1)">
    </source>
</video>
```

In addition, XSS attacks could be conducted against conventional HTML when escaping of double quote(") and single quote(') is omitted during the dynamic generation of value attributes even if “<” and “>” have been escaped for elements such as <input>. In the conventional HTML case, event handlers that require a user action such as moving a mouse or clicking in order to raise the event, for example “onmouseover” and “onclick”, are the majority among the event handlers.

```
<input value="" onmouseover="alert(1)">
```

However, the combination of some attributes that have been added to HTML5 and attributes that require user actions mentioned above may enable attackers to conduct XSS attacks without any user actions. For example, the new attribute “autofocus” in the <input> element may cause XSS attacks without user actions. The “autofocus” attribute, which moves the focus to an input field when the web page is loaded, may cause a script to be executed without any user action due to the combination of the event handler specified in the “onfocus“ attribute.

```
<input value="" autofocus onfocus="alert(1)">
```

In conventional HTML, attributes that can specify a script as their values were limited to the ones whose names begin with “on”. Though it is noteworthy that attributes such as “formaction” attribute whose name does not begin with “on” and that can specify a script as their values were added to HTML5. Other new attributes whose names do not begin with “on” may be added because of the web browser’s own dialect or due to new HTML5 specifications. Thus, merely detecting attributes that begin with “on” and mitigating them cannot completely prevent XSS vulnerabilities.

```
<form>
    <button formaction="javascript:alert(1)">
        some text
    </button>
</form>
```

Thus, for the reasons stated above, escaping during HTML generation is recommended over detecting elements or attributes in user input that can specify a script as their values, as protection against XSS attacks.

For the more specific measures, please refer to the sections such as section 1.5 “Cross-site Scripting“ in “How to Secure Your Web Site” published by IPA.

3.1.2. DOM Based XSS

DOM (Document Object Model) is defined as an API that enables scripts executed on web browsers to generate or to update a HTML document displayed in the web browsers without communication with a web server. This allows the implementation of responsive web applications which respond like applications that reside on a local PC. DOM Based XSS is an attack method that forces the user web browser to execute specially crafted scripts written in JavaScript etc. These scripts contain data disguised as scripts that an attacker has injected into input through manipulation of the DOM. DOM Based XSS vulnerabilities have a high probability of being built in since many web applications using HTML5 utilize JavaScript.

Countermeasures against DOM Based XSS attacks will be described in section 3.1.2.1. and section 3.1.2.2.

3.1.2.1. Generate a HTML document through DOM

Although “Escaping during HTML document generation” strategy is also an effective countermeasure

against a DOM Based XSS attack, generating text nodes and manipulating through DOM are a more comprehensive countermeasure for XSS attacks. “During HTML generation” in JavaScript context implies the timing where the DOM structure changes within the JavaScript such as when substitution into innerHTML or when a document is written by document.write is performed.

```
// Example of Vulnerable Scripting
// XSS attack can be conducted by using strings from an external source as HTML
var div = document.getElementById("msg");
div.innerHTML = some_text; // String from external resource
```

```
// Example of Secure Scripting
// Escape strings from an external source to generate a HTML document
function escape_html( s ){
    return s.replace( /&/g, "&amp;" )
        .replace( /</g, "&lt;" )
        .replace( />/g, "&gt; " )
        .replace( /"/g, "&quote;" )
        .replace( /'/g, "&#39;" );
}

var div = document.getElementById("msg");
var escaped_text = escape_html( some_text );
div.innerHTML = escaped_text; // Escaped string
// Example of Secure Scripting (Reccomended)
// Operate text nodes through DOM API to generate a HTML document in security
var div = document.getElementById("msg");
var text = document.createTextNode( some_text ); // Generate a text node
div.appendChild( text );
```

It is necessary to take care of attribute values as well as the text component as follows.

```
// Example of Vulnerable Scripting
// XSS attack can be conducted by using strings from an external source as attribute
values without escape
var f = document.getElementById("form");
f.innerHTML = "<input type='text' value='" + some_text + "'>";
```

```
// Example of Secure Scripting (Recommended)
// Set attribute values through DOM API
var f = document.getElementById("form");
var elm = document.createElement( "input" );
elm.setAttribute( "type", "text" );
elm.setAttribute( "value", some_text ); // Set attribute value
f.appendChild( elm );
```

3.1.2.2. Limit schemes to http or https schemes during handling URL

When handling URL in the context such as the “href” attribute in <a> element, “src” attribute in <iframe> element etc., or substitution into “localtion.href”, XSS attacks may be conducted by specifying arbitrary schemes such as “javascript” scheme instead of “http” and “https”.

```
// Example of Vulnerable Scripting 1
// XSS attack can be conducted by literally using strings from external sources
as URL.
// Suppose that attackers can specify strings such as “javascript:…” in a variable
“url”
location.href = url;
```

```
// Example of Vulnerable Scripting 2
// XSS attack can be conducted by literally using strings from an external source
as URL
// Suppose that attackers can specify strings such as “javascript:…” in a variable
“url”
var elm = document.getElementById("link"); // <a id="link">
var text = document.createTextNode( url );
elm.appendChild( text );
elm.setAttribute( "href", url ); // “javascript:…” is specified as the value of
href
```

When strings from external sources are used as a URL, XSS attacks can be prevented by prohibiting schemes other than “http” and “https”.

```
// Example of Secure Scripting
// Process a script only if URL begins with http:// or https://
if( url.match( /^https?:\/\// ) ){
```

```
    var elm = document.getElementById("link"); // <a id="link">
    var text = document.createTextNode( url );
    elm.appendChild( text );
    elm.setAttribute( "href", url );
}
```

However, even if the schemes are limited to ones that begin with http:// or https://, if these are substituted into location object, a XSS attack can be prevented but this may result in an open redirect issue. The details of open redirect will be described in section 3.3.

For DOM Based XSS vulnerabilities, please also refer to “Technical report regarding “DOM Based XSS” published by IPA.

3.2. Cross Site Request Forgery

“Cross Site Request Forgery Attack” (herein “CSRF Attack”) is an attacking method to induce a user to perform a unintended action on a vulnerable web site by (mis)leading such user to an attacker crafted web site. As a result of XMLHttpRequest (herein, XHR) supporting cross-origin requests, CSRF attacks over Cross-Origin Communication became possible. Sequentially, web sites that were previously not vulnerable to CSRF, may now be vulnerable to CSRF attacks.

Let’s consider the case where an attacker attacks a function for uploading files only with HTML4 features, and such attacks with features newly introduced in HTML5. The file upload function here uses the following HTML and the server does not check the referrer. The data sent by the form below does not contain a token or similar information, therefore this function is considered as vulnerable to a CSRF attack.

```
<!-- Example of Code in http://target.example.jp/ -->
<form method="POST" action="upload" enctype="multipart/form-data">
    <input type="file" name="file">
    <input type="submit">
</form>
```

Without using HTML5 features, cross-origin requests using JavaScript cannot be sent. Thus, an attacker may create a malicious web site that POSTs data to a legitimate site targeted for an attack that is cross-origin when loading the page.

```
<!-- Example of Code in Malicious Web Site -->
```

```
<body onload="document.forms[0].submit();">
<form method="POST" action="http://target.example.jp/upload"
     enctype="multipart/form-data">
    <input type="file" name="file">
    <input type="submit">
</form>
</body>
```

However, even though the user may visit the malicious web site, significant data will not be sent to the web server because the content and file name are sent as ‘blank’ as shown below.

```
POST http://target.example.jp/upload
Host: target.example.jp
...
Content-Type: multipart/form-data; boundary=----abcdefg

------abcdefg
Content-Disposition: form-data; name="file"; filename=""
Content-Type: application/octet-stream



------abcdefg--
```

On the other hand, with HTML5 features, the attacker can send POST requests with XHR since XHR has now supported Cross-Origin Communication. As a result, the attacker can construct the contents of the file to be uploaded within the JavaScript as shown below.

```
var xhr = new XMLHttpRequest();
var boundary = '----boundary';
var file="abcd";   // Content of file to send
var request;

xhr.open( 'POST', 'http://target.example.jp/upload', 'true' );
xhr.setRequestHeader( 'Content-Type',
    'multipart/form-data; boundary=' + boundary );
xhr.withCredentials = true;       // Add Cookie
xhr.onreadystatechange = function(){};
request = '--' + boundary + '\r\n' +
```

```
    'Content-Disposition: form-data; name="file"; ' +
    ' filename="filename.txt"\r\n' +
    'Content-Type: application/octet-stream\r\n\r\n' +
    file +
    '\r\n' + '--' + boundary + '--';
xhr.send( request );
```

As just shown above, in HTML5, an attacker can construct the contents and name of a file within JavaScript, and upload the file to a target web server by leveraging a CSRF vulnerability. Normally, Cookies are not sent in cross-origin communication via XHR, but can be sent by setting the “withCredentials” property as “true”. If XSS vulnerabilities exist where file names or contents are displayed, an attack combining these vulnerabilities may be conducted.

Even if your web site does not allow for XHR Cross-Origin Communication or cannot handle such communications, there still exists a possibility that attack sites may send such requests to your site. Therefore, if your site unconditionally processes all data received, an attack against your site may be conducted.

The method to check the "Origin" header included in requests from other sites does not serve as a comprehensive countermeasure since it does not work against CSRF attacks that leverage a request sent from a form on your own site. For a comprehensive CSRF countermeasure, as has been in the past, include a token or other secret information in the <input type="hidden">.

```
<body onload="document.forms[0].submit();">
<form method="POST" action="http://target.example.jp/upload"
     enctype="multipart/form-data">
    <input type="file" name="file">
    <input type="hidden" name="token" value="9CF89BC43B1B6FEA399A...">
    <input type="submit">
</form>
</body>
```

For more comprehensive countermeasures against CSRF attacks, please refer to section 1.6 of the IPA "How to Secure Your Website"

3.3. Open Redirect

There may be a situation where a web application needs to change redirect destination depending

on the input data values. If the web application is constructed in an inappropriate manner, an attacker may alter the redirect destination so to induce users to 'visit' a malicious web site. This vulnerability is called "Open Redirect" vulnerability. Open redirect vulnerabilities do not cause direct damage to the web site itself, but it may lead to the reputation issue of the web site, since the domain name may be leveraged to redirect to a malicious site. Therefore, it is necessary to take the measures against open redirect vulnerability. When URL of the redirect destination is specified in "location.hash", the URL is not recorded in the web server logs. Therefore, it is necessary to note that the final destination cannot be traced and verified afterward if an open redirect vulnerability exists.
To prevent an open redirect vulnerability from being built-in, it is most effective to hold a static list of redirect destinations, rather than generating redirect URLs using data from external sources.

```
#!/usr/bin/perl
# Example of Secure Coding
# Fix and retain redirect destinations(/foo、/bar、/baz) in advance
use URI::Escape;
my $index = uri_unescape( $ENV{QUERY_STRING} || '' );
my $pages = { foo=>'/foo', bar=>'/bar', baz=>'/baz' };
my $url = $pages->{$index} || '/';
print "Status: 302 Found\n";
print "Location: $url\n\n";
```

If it is necessary to generate redirect URLs dynamically, it is best to check if the domain of the generated redirect URL is within an expected scope for that web site, so that users will not be redirected to an arbitrary site.
The sections below will describe 3 typical methods to redirect a user to a dynamically generated URL in a web application, and points to keep in mind regarding each method.

3.3.1. Return 301, 302, or others as HTTP status code and specify the redirect destination in the Location header

In this method, it is necessary to note not only open redirect vulnerabilities but also HTTP header injection attacks:

```
#!/usr/bin/perl
# Example of Vulnerable Scripting
# HTTP header injection attack as well as open redirect may occur
use URI::Escape;
my $url = uri_unescape( $ENV{QUERY_STRING} || '' );
```

```
print "Status: 302 Found\n";
print "Location: /$url\n\n";
```

3.3.2. Specify redirect URL in location object with JavaScript

In this method, it is necessary to limit schemes to http or https. This is due to XSS attacks being conducted by redirecting to schemes that refer to arbitrary JavaScript. For more details, please refer to section 3.1.2.

```
// Example of Vulnerable Scripting 1
// XSS attack can be conducted by using string from external source to the URL
// Attackers specify string such as "javascript:..." in valuavle “url”
var url = decodeURIComponent( location.hash.substring(1) );
location.href = url;
```

```
// Example of Vulnerable Scripting 2
// Insufficiency of checking strings from external sources can lead to open redirect
// Attacker can redirect users to arbitrary web sites by specifying URL in the format
such as “/\example.com/”
var url = decodeURIComponent( location.hash.substring(1) );
// Check if the first character of URL is “/” and the second character is except
“/”
if( url.match( /^\/[^\/]/ ) ) {
    location.href = url;
}
```

```
// Example of Secure Scripting
// Retain static redirect destination list
var index = location.hash.substring(1) | 0;
var pages = [ '/foo', '/bar', '/baz' ]; // Redirection destination list
if( 0 <= index && index < pages.length ){
    location.href = pages[ index ];
}
```

3.3.3. Redirect using refresh by <meta http-equiv=”Refresh”>

If a web page is redirected to a URL that has been dynamically generated based on user input, preventing open redirect is difficult when using this method. Therefore, it is not recommended to use

<meta> refresh as a redirect method. For more details, refer to 4.1.2.

3.4. Defect in Access or Authorization Control

Information leakage may occur if a web site that handles closed personal information and has defects in access or authorization control. When an attacker leverages features that were added to HTML5, information that may not be accessed previously may now be accessed and then leaked.

A variety of attacking methods to steal sensitive information have been discovered. For example, in a web site with Ajax, an attacker prepares a malicious website where sensitive information within the JSON is accessed. This is done by interpreting the JSON as JavaScript by using the “src” attribute in the <script> element. Third-parties that are normally not permitted to access sensitive information may gain access through Cross-Origin Communication that were not available in the past. Some web sites may inadvertently leak sensitive information since they are designed and operated in an inappropriate manner without consideration of features new to HTML5. Properly implementing access and authorization controls is critical during the development of a web site. Details on such implementations will be described in section 4, since details differ depending on the function that may be leveraged for the attack.

4. HTML5 features that should be noted

This section will describe the ways in which vulnerabilities are built in and measures to prevent such development with respect to numerous HTML5 features shown below that are either new to HTML5 or used frequently

- Newly added HTML elements
- JavaScript API
- XMLHttpRequest

4.1. Newly Added HTML Elements

This section will describe security points to be noted with regard to HTML elements and attributes that are added to HTML5 or changed.

4.1.1. <meta charset>

<meta charset> is used to specify the character encoding in the HTML document.

```
<head>
    <meta charset="utf-8">
</head>
```

XSS attacks leveraging character encoding such as UTF-7 may be conducted when hiding special characters in HTML by utilizing character encoding. Therefore, it is important to prevent unauthorized character encoding changes. For this reason, as a major principle, character encoding names should be required to be specified in the HTTP response header. In addition, specifying the encoding using <meta charset> in HTML should be limited for the purpose of preventing display of corrupted text when contents are saved locally. The character encoding name should be accurately specified without any misspell including confusion between “-“ and “_”. Specific expressions such as “utf-8”, “Shift_JIS” and “EUC-JP” should be used so that web browsers can identify the character encoding correctly.

If <meta charset> must be used to specify the character encoding for some reasons since the HTTP response header cannot be used to specify character encoding, any data input coming from an external source must not be inserted prior to <meta charset> . If a web browser misinterprets the <meta charset> inserted by an attacker prior to the legitimate <meta charset>, the character encoding that is interpreted may differ between the browser and the server, hence an attacker may leverage this difference into conducting a XSS attack.

In the example below, an attacker specifies a string written in UTF-7 using “</title><meta charset="utf-7">" as the <title> element. In this example JavaScript will not be executed, but when combining with other methods, an XSS attack may be conducted.

```
<title>
    +/v8APA-/title+AD4APA-meta charset+AD0AIg-utf-7+ACIAPg-
</title>
<meta charset="utf-8">...
<div>+ADw-script+AD4-alert(1);+ADw-/script+AD4-</div>
```

4.1.2. <meta http-equiv>

<meta http-equiv> is generally used to complement the HTTP response header within HTML. In essence, the format <meta http-equiv="name" content="content"> works in the same manner as a HTTP response header with the field “name: content” added to it. This section will describe the redirect feature referred to “ meta refresh” defined using the following syntax.

```
<!-- Example of Redirection to http://example.jp/ -->
<meta http-equiv="Refresh" content="0;url=http://example.jp/">
```

In this example, the web page will be redirected to” http://example.jp/” as specified in “url=” in 0 seconds or in other words, immediately. If an attacker is able to insert arbitrary strings where the redirect destination URL is specified, it may lead to a XSS attack through a redirect to a “javascript” scheme, or a redirect to an arbitrary site (open redirect attack).
For example of redirect to “javascript” scheme, as shown below, web browsers such as Google Chrome, Opera, Safari execute the script specified in the “javascript” scheme on the original page, so a XSS attack is possible.

```
<!-- Example of Redirection to javascript Scheme -->
<meta http-equiv="Refresh" content="0;url=javascript:alert(1)">
```

Internet Explorer 6 and 7 redirects to the value of the last “url=” when “content” attribute contains multiple “url=” as shown in the next example.For redirects using <meta http-equiv> and when attackers can specify arbitrary redirect URLs, it is difficult to restrict the redirect destination, thus it is difficult to prevent open redirect. Therefore, if redirect is implemented using <meta http-equiv>, the redirect URL should be static so that attackers cannot change this URL arbitrarily.

```
<!-- Example of Redirection to Unintended Web Site -->
<meta http-equiv="Refresh"
 content="0;url=http:example.jp/;url=http://evil.example.jp/">
```

4.1.3. <a ping>

The “ping” attribute in <a> element is used to specify the resource URL to be notified when the user clicks a hyperlink.

```
<!--  User’s clicking the hyperlink will be notified to the URL specified in “ping”
-->
<a href="http://example.jp/" ping="http://example.jp/ping">
Link to example.jp
</a>
```

Web browsers such as Google Chrome and Safari support notifications from the “ping” attribute. Only URLs in the http or https schemes are allowed to be notified, while other schemes such as “javascript:alert(1)” are ignored.

4.1.4. <iframe sandbox>

The sandbox attribute in the <iframe> element can specify conditions that are not allowed for contents within an iframe. An example of such is not allowing execution of JavaScript. Such specification can be used to embed untrusted contents within a web page.

```
<!-- Example of iframe with sandbox attribute -->
<!-- Execution of script, submission of form are prohibited -->
<iframe sandbox src="http://evil.example.com/"></iframe>
```

For iframe contents with the sandbox attribute, web browsers restrict execution of plug-ins, JavaScript, submission forms, and interference with the top-level window. They also handle each of the contents as coming from an original origin.
JavaScript execution can be prevented by embedding an iframe element with a sandbox attribute, but XSS attacks cannot be prevented by embedding your own page into a <iframe sandbox>. This is due to the fact that attackers do not need to use iframes for attacks, but can directly open the page with vulnerability and attack such page.
Sandbox attribute can also specify the conditions of the contents for an <iframe> as its value

```
<!-- Example of iframe with sandbox attribute -->
<!-- Execution of script is allowed -->
<iframe sandbox="allow-scripts" src="http://evil.example.com/">
</iframe>
```

In order to specify multiple values as the condition, separate each value with a space.

```
<!-- Example of iframe with sandbox -->
<!-- Execution of script and form is allowed -->
<iframe sandbox="allow-scripts allow-forms"
 src="http://evil.example.com/"></iframe>
```

In order to prevent pages from being loaded within an <iframe> frame, JavaScript code referred to as 'framebusting' has been used. However, JavaScript cannot run when a page is read in using <iframe sandbox>, therefore framebusting does not run properly and the page is displayed. So in order to prevent clickjacking, it is necessary to use X-Frame-Options instead of framebusting. Details are described in section 5.3.

4.1.5. <video>

The <video> element, similar to the way <img> element embeds images, is used to embed playable videos into HTML. In Internet Explorer 9 and 10, an “onerror” event in the <video> element may occur, which may be leveraged in a XSS attack.

```
<!-- Example of XSS Attack Leveraging video element -->
<video onerror="javascript:alert(1)">
    <source src="#"></source>
</video>
```

4.1.6. <audio>

The <audio> element, similar to the way <img> element embeds images, is used to embed playable audio into HTML. In Internet Explorer 9 and 10, an “onerror” event in the <audio> element may occur, which may be leveraged in a XSS attack.

```
<!-- Example of XSS Attack Leveraging audio element -->
<audio onerror="javascript:alert(1)">
    <source src="#"></source>
```

```
</audio>
```

4.1.7. <source>

The <source> element is used to specify applicable media source in the <video> and <audit> elements. In Firefox, Google Chrome, Opera, Safari and the standard Android browser, an “onerror” event in the <source> element may occur, which may be leveraged in a XSS attack.

```
<!-- Example of XSS Attack Leveraging video and source elements -->
<video>
    <source onerror="javascript:alert(1)"></source>
</video>
<!-- Example of XSS Attack Leveraging audio and source elements -->
<audio>
    <source onerror="alert(1)"></source>
</audio>
```

4.1.8. <canvas>

The <canvas> element is used to define the drawing area (canvas) where JavaScript can draw figures and embed graphics.

JavaScript can draw any figures in the Canvas area, but there is an internal flag called “origin-clean” in order to restrict scripts from the different origins from reading figures that is drawn by the script from other origins. The initial value of the origin-clean flag is true, but once a cross-origin image is rendered, the flag automatically changes to false. When origin-clean is false, images in the canvas cannot be read by JavaScript using the toDataURL, getImageData and toBlob methods.

```
// Code in http://example.jp/
// Security exception occurs due to loading of the image belonging to a cross origin
<img id="img" src="http://other.example.jp/sample-image.png">
<canvas id="canvas"></canvas>
...
var img = document.getElementById( "img" );
var canvas = document.getElementById( "canvas" );
var ctx = canvas.getContext( "2d" );
ctx.drawImage( img, 0, 0 );
```

```
alert( canvas.toDataURL( "image/png" ) );  // Exception occurs
```

This mechanism limits the occurrence of security issues when using Canvas.
Meanwhile, in order to respond to the needs to use images cross-origin, Firefox, Google Chrome, Safari 6, and Opera can allow access to such images by an addition of a Access-Control-Allow-Origin response header from the image provider in accordance to the Cross-Origin Resource Sharing (CORS) framework, in a similar manner as allowing Cross-Origin Communication through XHR. For example, in the response header of an image in other.example.jp, specification of the origin that is allowed to access the resources in Access-Control-Allow-Origin is as follows.

```
Date: Tue, 1 Jan 2013 09:00:00 GMT
Access-Control-Allow-Origin: http://example.jp
Content-Length: 28400
Content-Type: image/png
```

It is also necessary to add “crossorigin” attribute in the <img> element in example.jp, which is the source that forwards the image to Canvas.

```
// Code in http://example.jp/
// Load image belonging to a cross origin
// Security exception does not occur
<img id="img"
  src="http://other.example.jp/sample-image.png" crossorigin>
<canvas id="canvas"></canvas>
...
var img = document.getElementById( "img" );
var canvas = document.getElementById( "canvas" );
var ctx = canvas.getContext( "2d" );
ctx.drawImage( img, 0, 0 );
alert( canvas.toDataURL( "image/png" ) );  // Exception does not occur
```

In <img> elements with a “crossorigin” attribute, the Origin request header, which represents the origin of the page that the request was originated from, is added when the request is made. The server side specifies the origins for which contents can be accesses in the Access-Control-Allow-Origin response header. In the above example, JavaScript can access images that have http://example.jp as an origin. If Access-Control-Allow-Origin does not exist in the image response header, or, if the origin to your own contents is not specified in Access-Control-Allow-Origin,

the image data drawn in Canvas cannot be accessed.

In order to only allow access from specific origins, Cookie must be used in the conventional manner in addition to the specifications of Origin request header and Access-Control-Allow-Origin response header. Cookies need to be used since attackers, with tools, may send a request with an arbitrary Origin request header in place of a browser. This may allow direct access to an image without going through JavaScript.

“Access-Control-Allow-Origin: *” means that JavaScript from all origins can access the image. Therefore, this must not be used if you do not want third-parties to view your images. This is due to an attacker being able to create a malicious page containing <img> element with “crossorigin” attributes and JavaScript in the malicious page resulting in access to the images. Also, when Cookies are enabled, using “Access-Control-Allow-Origin: *” is prohibited.

“Access-Control-Allow-Origin: *” should only be used for public contents and do not require access protection. For other contents, specify the allowed origins through Access-Control-Allow-Origin: <origin>.

4.1.9. <input>

The features of <input> element have been substantially enhanced in HTML5. Restriction of data formats is now made simple. For example, “<input type="email">” allows you to only accept input data in an e-mail address format. Also developers can use a regular expression like <input type="text" pattern="^[0-9a-fA-F]$"> to restrict data input to ones that only match an arbitrary pattern that the developers specify. Implementing such restrictions required the use of JavaScript in the past, but can now be easily implemented with HTML5. This feature brings about other merits such as unified error messages as well as decrease in amount of code.

However, attackers can change the source code of HTML in the browser to send request that does not match the restricted pattern specified in the <input> element. Therefore, one cannot assume that input restrictions are complete by using the <input> element alone.

XSS attacks may occur even if “<” and “>” are escaped in the <input> element, but escape is not performed for “”” (double quote) and “’” (single quote) in the “value” attribute. Many of these XSS attacks were conducted by JavaScript that was specified as the event handler such as “onmouseover” and “onclick”, so they required an explicit user action.

```
<!-- Example of attack that needs user’s action -->
<input value="" onmouseover="alert(1)">
```

The “autofocus” attribute added in HTML5 automatically moves the focus to an <input> element. By combining this with an “onfocus “ event handler, JavaScript execution is possible without an explicit user action.

```
<!-- Example of attack that does not need user’s action -->
<input value="" autofocus onfocus="alert(1)">
```

There was a measure to detect attributes beginning with “on” in order to prohibit the embedding of event handlers into a HTML generation. However, this measure is insufficient to prevent XSS attacks since new attributes such as “formaction” have been added in HTML5, that allow javascript schemes to be specified as values that do not begin with “on”.

```
<!-- Example of attack leveraging attributes not beginning with “on” -->
<form>
    <input type="button" formaction="javascript:alert(1)">
</form>
```

4.1.10. <button>

In a similar manner to the <input> element , HTML5 also added attributes to the <button> element that do not begin with “on” such as the “formaction” attribute. Therefore, just detecting attributes that begin with “on” is insufficient as a countermeasure against XSS.

```
<form>
    <button formaction="javascript:alert(1)">
        some_text
    </button>
</form>
```

4.2. JavaScript API

Modern web browsers have implemented APIs to provide various features through JavaScript. This section will describe the APIs that require security considerations during implementation. XMLHttpRequest(XHR) will be described in section 4.3 separately since there are many points to consider.

4.2.1. WebSocket

WebSocket is a feature to implement interactive communications between the web browser and server. Typical JavaScript code used for WebSocket is shown below.

```
var ws = new WebSocket( "ws://example.jp/" );
ws.onopen = function(){
    console.log( 'connected' );
};
ws.onmessage = function( evt ){
    console.log( evt.data );
};
ws.onclose = function(){
    console.log( 'closed' );
};
ws.send( 'data' );
```

ws URI scheme communications, like the http scheme, are not encrypted. wss URI scheme is used to implement TLS encrypted communications.

```
var ws = new WebSocket("wss://example.jp:443/ ");
```

It is recommended to use wss for communications of critical information not to be affected by interceptions or data tempering.

Note that in the ws and wss URI schemes, Cookies are shared with http and https during the handshake phase of the communication. In other words, the Cookie issued when accessing the web page at http://example.jp is sent to the server with the handshake request when accessing ws://example.jp:8080/ or wss://example.jp:8081/.

```
GET / HTTP/1.1
Host: example.jp
Connection: Keep-Alive

HTTP/1.1 200 OK
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Length: 2524
Content-Type: text/html; charset=utf-8
```

```
Set-Cookie: session=12AFE9BD34E5A202; path=/
....

GET /websocket HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: example.jp
Origin: http://example.jp
Sec-WebSocket-Key: mU6OBz5GKwUgZqbj2OtWfQ==
Sec-WebSocket-Version: 13
Sec-WebSocket-Protocol: chat, superchat
Cookie: session=12AFE9BD34E5A202

HTTP/1.1 101 Switching Protocols
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Accept: IsCRPjZ0Vshy2opkKOsG2UF74eA=
Sec-WebSocket-Protocol: chat
....
```

Cookies issued with the secure flag are shared between the https and wss URI schemes. In other words, the Cookie issued with the secure flag when accessing https://example.jp is sent to the server with the handshake request when accessing wss://example.jp:8081/.

Also, Cookies issued with the httponly flag in order to prevent access from JavaScript are sent when using the ws and wss URI schemes just like when using the http and https schemes.

4.2.2. Web Storage

Web Storage or DOM Storage is a mechanism to store data on the client that can be accessed from JavaScript. “sessionStorage” disposes of data when the session is ended, and “localStorage” keeps data even after the web browser is closed. Read and Write access to this data is limited to the same origin in both types of Web Storage. Thus, contents of Web Storage written from an http scheme web page cannot be accessed from https scheme web pages.

By adding the httponly flag to Cookies, access from JavaScript can be blocked, but there is no way to prevent JavaScript from access to Web Storeage. Thus, one must also use Content Security Policy when saving confidential information in Web Storage.

4.2.2.1. sessionStorage

sessionStorage stores data while the browser window or tab is opened. Contents of sessionStorage are retained when a page is reloaded or another page is displayed and only disposed of when the browser window or tab is closed. Contents cannot be read or written to in other web browser windows or tabs. Contents in sessionStorage are shared among the iframes or frames belonging to the same origin.

The following issues have been known to exist in Internet Explorer 8 and 9:

- sessionStorage is not shared among iframes or frames
- sessionStorage is shared between http and https schemes (only in Internet Explorer 8)
-

4.2.2.2. localStorage

localStorage keeps data even when the browser is closed unless the data is deleted explicitly.

However, the following issues have been known to exist in Internet Explorer 8 and 9.

- localStorage sometimes may not be shared between a newly opened window using "New Session" in "File" menu.
- sessionStorage is shared between http and https schemes (only in Internet Explorer 8)
-

When Safari for both Mac and iPhone are in private browsing mode, data written to localStorage is not saved at all. For example, if the following code is executed when in private browsing mode, "undefined" will be displayed.

```
localStorage.setItem( "foo", "value" );
alert( localStrage.getItem( "foo" ) );
```

The expiration date of data stored in Web Storage does not correspond to the expiration date of a session for a web application managed via Cookie. For this reason, if a web application has a login mechanism and stores information unique to users into Web Storage, this stored information can still be accessed after logging out, unless this data is explicitly deleted. For example, after logging into a web application, if (a) it stores data into localStorage or (b) it performs a process that uses data stored in LocalStorage, this web application may store the data into localStorage during a user X logging in as result of (a), and then may allow to be referenced by a user Y when using a browser on a shared device. To avoid this problem, it is recommended to delete data stored in Web Storage upon logout so that other users cannot access this data.

4.2.3. Offline Web Application

“Offline Web Application” is a feature that enables a user to use a web application even without any network connection. It accomplishes this by storing specific resources as offline cache on the device. If one uses offline cache, it results to use resources saved in the user’s device which will be used for a long while. Thus, a device using a not encrypted, untrusted Wi-Fi network may be targeted by an attacker and save malicious contents as offline cache using Man-in-the-Middle attack. For example, the user of this device will continue to use these malicious contents and may result in confidential information being sent to an attacker's site. Therefore, when using the Offline Web Application feature, it is recommended to use HTTPS communications to the whole site.

4.2.4. Web Workers

Web Workers is a feature that allows JavaScript to perform parallel processing in the background. Typical JavaScript code for using Web Workers is as follows:

```
var worker = new Worker( 'worker.js' );
worker.onmessage = function( evt ){
    console.log( evt.data );
};
worker.postMessage( 'ping' );
```

Worker.js is JavaScript that operates in the background. The typical usage of worker.js is as follows.

```
onmessage = function( evt ){
    if( evt.data == "ping" ){
        postMessage( 'pong' );
    }
};
```

As a method to allow Web Workers to read in external script files, one may use Worker constructor, SharedWorker constructor or the importScripts method. It must be ensured that an attacker cannot specify an arbitrary URL to be read in.

```
// Example of Vulnerable Coding of Worker Constructor
var src = location.hash.substring(1);
var worker = new Worker( src );
```

When the above code accesses the URL http://example.jp/#worker.js, “worker.js” after the “#” in the URL is used as the source for Web Workers.

As of now, the source URI for the Worker constructor is limited to the same origin, but there is a possibility that the “data” scheme may be supported in the future. Firefox and Opera have already supported the “data” scheme for the Worker constructor. For that reason, if an attacker forces a user to access the following URI specified by the attacker, the arbitrary script prepared by the attacker will be executed by Web Workers.

```
http://example.jp/#data:text/javascript,onmessage=...
```

Currently, the source URI for the SharedWorker constructor is also limited to the same origin but there is a possibility that the “data” scheme may be supported in the future. Similar to the Worker constructor, you must ensure that an arbitrary URI cannot be specified as a source.

Import Scripts is not limited to the same origin, and thus Web Workers may read in a script from an arbitrary origin. Therefore, when using the code below, an attacker prepared script may run as Web Workers.

```
// Vulnerable Code. Arbitrary script will be passed to importScripts method
var src = location.hash.substring(1);
var worker = new Worker( 'worker.js' );
worker.postMessage( src );
```

```
// worker.js
onmessage = function( evt ){
    if( evt.data ) importScripts( evt.data );
};
```

However, even if an attacker script is running in Web Workers, browser DOMs cannot be directly manipulated from Web Workers. As a result, this is considered to be less of a threat than a XSS attack.

4.2.5. Cross Document Messaging

Cross Document Messaging (herein “XDM”) is a feature that allows exchanging messages between HTML documents from different origins.
Typical code for using XDM is shown below. In this example, documents that belong to different

origins, http://example.jp and http://example2.jp, are communicating with each other.

```
<!-- Site 1 -->
<!-- http://example.jp/xdm.html -->
<script type="text/javascript">
// Register the handler during reception of messages
window.addEventListener( 'message', function( evt ){
    if( evt.origin == 'http://example2.jp' ){
        alert( 'got:' + evt.data );
    }
}, false );
...
// Send a message
var iframe = document.getElementById( "child" ); // child iframe
iframe.contentWindow.postMessage( 'ping', "http://example2.jp" );
</script>
...
<!-- Communication Target-->
<iframe src="http://example2.jp/xdm-child.html" id="child"></iframe>
```

```
<!-- Site2- -->
<!-- http://example2.jp/xdm-child.html -->

<script type="text/javascript">
// Register the handler during reception of the message
window.addEventListener( 'message', function( evt ){
    if( evt.origin == 'http://example.jp' ){
        evt.source.postMessage( "pong", evt.origin );
    }
}, false );
</script>
```

The “postMessage” method is used to send messages to a HTML document. The origin of the destination document or “*” (asterisk) is specified as the second argument in the “postMessage” method. In the Site 1 example above, since http://example2.jp/xdm-child.html is the URL specified in the iframe, the destination origin for the second argument of the “postMessage” method is specified as http://example2.jp. When the origin specified in the iframe does not match the origin of the target document, the message cannot be sent to the target document.

```
<!-- Extarct from site1 -->
<iframe src="http://example2.jp/xdm-child.html" id="child"></iframe>
...
var iframe = document.getElementById( "child" ); // child iframe
iframe.contentWindow.postMessage( 'ping', "http://example2.jp" );
```

When calling “postMessage” method with "*" to specify the origin of the target document, this message can be received by a document in any origin. Since this message can be received by anybody, it is recommended to specify the target origin instead of using "*" when the message contains confidential information.

In order to receive the messages from external sources, the “message” event handler must be registered in the “window” object. In the “message” event handler, the event details are passed as an argument in the “Event” object.

When a message is received, the origin property of the Event object argument is checked to verify if the source of the message is an expected source. This is performed to ensure that messages from arbitrary documents are not received.

```
<!-- Extract from Site1 -->
window.addEventListener( 'message', function( evt ){
    if( evt.origin == 'http://example2.jp' ){
        alert( 'got:' + evt.data );
    }
}, false );
```

4.3. XMLHttpRequest

XMLHttpRequest (XHR) is an API for JavaScript to communicate with the server using HTTP protocol. At this time, major browsers support XHR Level 2 which enables Cross-Origin Communication through HTML5 and Cross-Origin Access Control (herein, CORS). This section will describe security concerns about XHR Level 2.

Note that the XHR implemented in Internet Explorer 8 and 9 cannot perform Cross-Origin Communication but has a similar function called XDomainRequest to provide Cross-Origin Communication. Details on XDomainRequest will not be a part of this report.

4.3.1. Unintended cross-origin communication (Client-side)

There are cases where existing codes using XHR that was written in assumption that it can communicate among same origin documents only may now be vulnerable since XHR now supports cross-origin access.

For example, suppose we have a piece of code that implicitly assumes the communication is limited within the same origin as itself, specifies the destination of XHR communications with your site in the URL after the # (hash), and displays the document including a part of response from the destination as is.

```
// Code written on the premise of same-origin communication
// A URL like http://example.jp/#/foo
// Use the string after # as the communication destination URL
var url = location.hash.substring(1);
var xhr = new XMLHttpRequest();
xhr.open( "GET", url, true );
xhr.onreadystatechange = function(){
    if( xhr.readyState == 4 && xhr.status == 200 ){
        div.innerHTML = xhr.responseText;
    }
};
xhr.send( null );
```

This code did not have any issue when XHR did not support Cross-Origin Communication, but it now has a problem because major browsers support Cross-Origin Communication via XHR. If an attacker forces a user to browse a URL such as http://example.jp/#//evil.example.com/, the target for XHR communications is not the resources in example.jp but is the resources in evil.example.com, and as a result, a XSS attack or displaying unintended contents may occur. The above example uses “innerHTML”, but even with the substitution into a “text” node or displaying contents after proper escaping, the contents prepared by an attacker may be displayed although the attacker’s script will not be executed.

As a mitigation against this problem, the destination of the requests must be limited so that XHR cannot read data from arbitrary sites. More specifically, keeping a static list of the destination to which requests can be sent and preventing XHR from attacker’s insertion of the request destinations is one of the possible mitigations.

```
// Example of secure coding; Retain the static list of the communication destination
// Specify index of the communication destination in URL like http://example.jp/#1
var pages = [ "/", "/foo", "/bar", "/baz" ];
var index = location.hash.substring(1) | 0;

var xhr = new XMLHttpRequest();
xhr.open( "GET", pages[ index ] || '/', true );
xhr.onreadystatechange = function(){
    if( xhr.readyState == 4 && xhr.status == 200 ){
        div.innerHTML = xhr.responseText;
    }
};
xhr.send( null );
```

It is insufficient to check whether the request destination's domain is within expectation or not within JavaScript, because an attacker can force the user to display the attacker-prepared contents if there is an open redirect vulnerability in the site.

```
// Example of code that may become vulnerable
//  Communication  to  arbitrary  sites  can  be  performed  if  open  redirect
vulnerabilities exist in the web site
var url = url_for_request; // URL for request
if( url.indexOf( "http://example2.jp/" ) == 0 ||
    url.indexOf( "http://example3.jp/" ) == 0 )
{
    // Communicate only if URL for request is example2.jp or example3.jp
    var xhr = new XMLHttpRequest();
    xhr.open( "GET", url, true );
    ....
}
```

In this code above, the XHR request destination is limited to example2.jp and example3.jp, but when either example2.jp or example3.jp has an open redirect vulnerability, this may result in communications with an arbitrary site. There is no way to detect whether a redirect occurred in XHR or to discover the final destination URL. Therefore, there still remain issues with the mitigation to check the request destination for XHR, and it is recommended to keep a static list of request destinations to limit communications as discussed before.

4.3.2. Unintended cross-origin communication (Server side)

Client-side JavaScript for Cross-Origin Communication via XHR is shown below.

```
var xhr = new XMLhttpRequest();
xhr.open( "GET", "http://other.example.jp", true );
xhr.onreadystatechange = function(){
    if( xhr.readyState == 4 && xhr.status == 200 ){
        div.appendChild( document.createTextNode( xhr.responseText ) );
    }
};
xhr.send( null );
```

This JavaScript code itself is nothing different from the code used for same-origin communications.
A sample HTTP request and response when using the above are shown below.

```
GET http://other.example.jp/ HTTP/1.1
Host: other.example.jp
Origin: http://example.jp
User-Agent: Mozilla/5.0(Windows NT 6.0; rv:18.0)
Connection: keep-alive

HTTP/1.1 200 OK
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Length: 1512
Content-Type: text/plain; charset=utf-8
Access-Control-Allow-Origin: http://example.jp

...
```

The HTTP request contains an Origin request header which shows the origin of the page that issued the request. On the server-side the Access-Control-Allow-Origin response header specifies what origin(s) can access the contents. In the example above, JavaScript with an origin of http://example.jp can access the contents.

If the Access-Control-Allow-Origin response header does not exist in the server response or if the Access-Control-Allow-Origin does not contain the origin, contents cannot be accessed through the properties of JavaScript using XHR such as responseText property.

If the contents contain confidential information and must be allowed to access from specific origins, access restrictions must be put in place using Cookies in addition to restrictions using the Origin request header and Access-Control-Allow-Origin response header. If Cookies are not used in addition to the Origin request header and Access-Control-Allow-Origin response header, an attacker may use a tool such as a telnet client other than web browsers to send a request with an arbitrary Origin request header added, so the contents can be read without going through JavaScript.

"Access-Control-Allow-Origin: *" means that JavaScript from any origin can access the contents, hence one must not use this when there are confidential contents involved. This is because if an attacker issues a XHR request from a malicious page, JavaScript from the malicious page may access the confidential contents. "Access-Control-Allow-Origin: *" should only be used for contents that are designed to be publicly available, and for any other contents, origins should be specified like "Access-Control-Allow-Origin: http://example.jp".

By default, Cookies are not sent when XHR cross-origin requests are issued. In order to send Cookies in cross-origin, the withCredentials property must be set to 'true' as in the example below. When using Cookies, "Access-Control-Allow-Origin: *" is prohibited by its specification, and XHR cannot read this information.

```
var xhr = new XMLhttpRequest();
xhr.open( "GET", "http://other.example.jp", true );
xhr.withCredentials = true; // Cookie becomes able to be sent
xhr.onreadystatechange = function(){
    if( xhr.readyState == 4 && xhr.status == 200 ){
        div.appendChild( document.createTextNode( xhr.responseText ) );
    }
};
xhr.send( null );
```

```
GET http://other.example.jp/ HTTP/1.1
Host: other.example.jp
Origin: http://example.jp
User-Agent: Mozilla/5.0(Windows NT 6.0; rv:18.0)
Cookie: session=12AFE9BD34E5A202
Connection: keep-alive

HTTP/1.1 200 OK
Date: Tue, 1 Jan 2013 09:00:00 GMT
```

```
Content-Length: 1512
Content-Type: text/plain; charset=utf-8
Access-Control-Allow-Origin: http://example.jp

...
```

Also, it may be possible to perform cross-origin CSRF attacks through XHR. Conventional CSRF countermeasures such as using tokens must be put in place. For more information about CSRF, please refer to section 3.2.

4.3.3. XSS caused by Ajax data

When data exchanged through XHR (Ajax data) contains strings that can be interpreted as HTML, there is a possibility that XSS may occur when opening this data directly with a browser.

```
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Length: 86
Content-Type: application/json; charset=utf-8

{
    "name" : "foo",
    "value" : "<html><script>alert(1);</script></html>"
}
```

When JSON data shown above is directly opened by Internet Explorer, the string at the beginning of the data is checked and after determining that this data is HTML content, JavaScript may be executed. XSS attacks may occur with various types of Ajax data, such as text/plain or text/csv, in addition to the JSON data.

An attacker may view Ajax data that contains confidential information without conducting a XSS attack. This can be accomplished when an attacker prepares a malicious page and reading in the targeted Ajax data as a source for the <script> element. This method is typically used to target JSON or CSV data that can be interpreted as JavaScript.

```
In the attacker’s trap site Ajax data including sensitive information is loaded
as JavaScript
<script src="http://example.jp/target.json"></script>
<script src="http://example.jp/target.csv"></script>
```

To address this issue, it is recommended to perform both of the countermeasures described in 4.3.3.1 and 4.3.3.2. This is due to Internet Explorer 6 and 7 not being able to handle X-Content-Type-Options response header described in 4.3.3.1.

4.3.3.1. Specify X-Content-Type-Options response header

In Internet Explorer 8 and later, by setting "X-Content-Type-Options: nosniff" in the response header, it will disable the browser from determining the Content-Type by analyzing the responses that it received. This also prevents XSS attacks as a result of non-HTML contents being processed as HTML contents. For more details, refer to section 5.2.

4.3.3.2. Respond to request only from XHR

In order to return Ajax data only for XHR requests, the client includes special character string in the request header for the XHR request, and the server will respond with an error when receiving a request without that special character string. This enables Ajax data not to be referenced directly from the browser.
During the request, add a custom header in the setRequestHeader as shown below.

```
var XHR = new XMLHttpRequest();
XHR.open( "GET", "http://example.jp/foo.json", true );
XHR.onreadystatechange = function(){ ... };
XHR.setRequestHeader( "X-Request-With", "XMLHttpRequest" );
XHR.send( null );
```

```
GET http://example.jp/foo.json HTTP/1.1
Host: example.jp
X-Request-With: XMLHttpRequest
User-Agent: Mozilla/5.0(Windows NT 6.0; rv:18.0)
Cookie: session=12AFE9BD34E5A202
Connection: keep-alive
```

The server will check if a custom header such as X-Request-With request header exists, and if it does, check the value of the header. This determines whether the browser directly opened the contents or if the request was issued by XHR. It will return a normal response if it is from XHR and return status code 403 or other appropriate responses if it is not. Note that libraries such as jQuery and prototype.js will automatically add request headers like above.

When using the setRequestHeader method for Cross-Origin Communication, an OPTIONS method request (a “preflight” request) is issued prior to the original request in methods such as GET and POST. When this occurs, the method used in the original request is sent in the Access-Control-Request-Method request header and custom header set by the setRequestHeader method is sent in the Access-Control-Request-Headers request header. The preflight request processing is automatically done by the web browser; thus the client-side does not need to explicitly implement any routine to dispatch preflight request in JavaScript, but the server-side may need to prepare some routine to handle preflight requests.

```
// Example of use of setRequestHeader during cross-origin communication
var XHR = new XMLHttpRequest();
XHR.open( "GET", "http://other.example.jp/foo.json", true );
XHR.withCredentials = true; // Send the Cookie
XHR.onreadystatechange = function(){ ... };
XHR.setRequestHeader( "X-Request-With", "XMLHttpRequest" );
XHR.send( null );
```

In the example below, foo.json contains a string that can be interpreted as HTML, but restriction to respond to only requests from XHR prevents the browser from directly referencing Ajax data.

```
OPTIONS http://other.example.jp/foo.json HTTP/1.1
Host: other.example.jp
Origin: http://example.jp
User-Agent: Mozilla/5.0(Windows NT 6.0; rv:18.0)
Cookie: session=12AFE9BD34E5A202
Connection: keep-alive
Access-Control-Request-Method: GET
Access-Control-Request-Headers: X-Request-With

HTTP/1.1 200 OK
Date: Tue, 1 Jan 2013 09:00:00 GMT
Access-Control-Allow-Origin: http://example.jp
Access-Control-Allow-Methods: GET, OPTIONS
Access-Control-Allow-Headers: X-Request-With
Access-Control-Max-Age: 1728000
Content-Length: 0
Connection: Keep-Alive
Content-Type: application/json

GET http://other.example.jp/foo.json HTTP/1.1
Host: other.example.jp
```

```
X-Request-With: XMLHttpRequest
Origin: http://example.jp
User-Agent: Mozilla/5.0(Windows NT 6.0; rv:18.0)
Cookie: session=12AFE9BD34E5A202
Connection: keep-alive

HTTP/1.1 200 OK

Date: Tue, 1 Jan 2013 09:00:00 GMT
Access-Control-Allow-Origin: http://example.jp
Connection: Keep-Alive
Content-Length: 86
Content-Type: application/json; charset=utf-8
{
    "name" : "foo",
    "value" : "<html><script>alert(1);</script></html>"
}
```

5. Security features in HTML5

This section will provide an overview of the following security features that have been implemented on some of the major web browsers to protect users and points to keep in mind in order to utilize these features effectively. These features can be enabled by specifying them in HTTP response header. Note that most of these features have not been standardized at this time, and depending on the browser, the functions may not exist or there may be differences in the way the features may function.

- X- XSS-Protection
- X-Content-Type-Options
- X-Frame-Options
- Content-Security-Policy
- Content-Disposition
- Strict-Transport-Security

5.1. X-XSS-Protection

Some major web browsers contain features to protect users from XSS attacks. The names of these features vary depending on the browser.

・Internet Explorer after version 8 : XSS Filter
・Google Chrome : XSS Auditor
・Safari : XSS Auditor

There are some differences between browsers, but they share a common rationale where it is considered a XSS attack if elements such as <script> are contained in both the request and the response. Blocking or disabling this prevents reflected XSS attacks. This rationale may result in XSS false positives if the request and response contain similar strings.
The XSS filter in Internet Explorer issues a notification when XSS is detected, as shown in Figure 5-1. However, Google Chrome and Safari do not display any notifications.

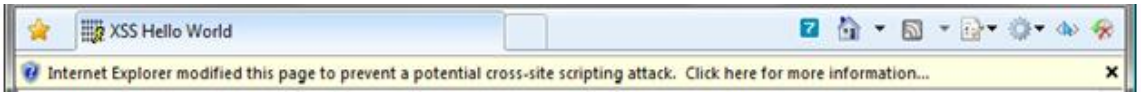

Figure 5-1: XSS Protection Filter in Internet Explorer Issues Notification

This function can be controlled using the X-XSS-Protection response header as shown in table 5-1.

Table 5-1 Settings for X-XSS-Protection

X-XSS-Protection: 0	Disable XSS protection filter feature
X-XSS-Protection: 1	Enable XSS protection filter feature
X-XSS-Protection: 1;mode=block	Enable XSS protection filter feature. Display blank page if XSS attack is detected

If “0” is specified, the web browser XSS protection feature is temporarily disabled. If “1”is specified, the web browser XSS protection feature is enabled. If “1;mode=block” is specified, the web browser will not only delete the element that caused the XSS attack to be detected but will then display a blank document. (Internet Explorer displays “#” only, and Google Chrome and Safari display an “about:blank” page) If X-XSS-Protection is not specified, the web browser XSS protection feature is enabled as if “1” is being specified.

The X-XSS-Protection header is only enabled if it is specified in the response header. The filter will not work if it is specified within the HTML using <meta http-equiv>

Disabling the XSS protection filter using "X-XSS-Protection: 0" should only be used for pages that may have a lot of false positives or any other special reason.

5.2. X-Content-Type-Options

If “X-Content-Type-Options: nosniff” is specified in the response header, Internet Explorer after version 8 will handle the content according to the value in the Content-Type response header.

```
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Length: 42
X-Content-Type-Options: nosniff
Content-Type: text/plain; charset=utf-8

<html>
<script>alert(1)</script>
</html>
```

If Internet Explorer 6 or 7 opens such contents, it will determine that it is HTML through MIME sniffing even though the Content-Type is text/plain which will result in JavaScript execution. But in Internet Explorer 8 or later, if X-Content-Type-Options: nosniff is specified, the contents are handled as text/plain as specified in the Content-Type.

By using X-Content-Type-Options, XSS attacks where non-HTML contents are interpreted as HTML contents as described in section 4.3.3 can be prevented in Internet Explorer 8 or later. On the other hand, Internet Explorer 6 and 7 cannot use the X-Content-Type-Options response header, so in order to handle XHR data, countermeasures described in section 4.3.3.2 need to be in place. Also, for contents that may be accessed by data other than XHR,

- Escape so that there are no issues even if the content is interpreted as HTML
- Add the Content-Disposition: attachment response header so that it is not opened directly in the browser
- Place contents in another domain

In Internet Explorer 9 and 10, X-Content-Type-Options: nosniff applies when contents are directly opened and when external scripts are read in through stylesheet or <script> element. Only the following Content-Type can be used:

・Stylesheet

text/css

・Script

application/ecmascript	text/ecmascript	text/x-javascript
application/javascript	text/javascript	text/vbs
application/x-javascript	text/jscript	text/vbscript

X-Content-Type-Options is only enabled if it is specified in the response header. It will not be enabled if it is specified within the HTML using <meta http-equiv>

By specifying X-Content-Type-Options: nosniff in the HTTP response header, you are able to restrict the Content-Type of the stylesheet and script source that can be read in. Therefore, it is recommended to add X-Content-Type-Options: nosniff to all dynamically generated contents.

5.3. X-Frame-Options

Major browsers except Internet Explorer 6 and 7 can prevent clickjacking by specifying X-Frame-Options in the response header. This prevents the embedding of contents in the iframe or frame.

By adding a HTTP response header like the example below, you can prevent contents from being displayed in another page's iframe.

```
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Length: 28400
X-Frame-Options: DENY
Content-Type: text/html; charset=utf-8

<!doctype>
<html>....</html>
```

The 3 values can be specified in X-Frame-Options as shown below.

Table5-2: Values allowed in X-Frame-Options

X-Frame-Options: DENY	Prevent the web browser from displaying the content in iframes
X-Frame-Options: SAMEORIGIN	Allow the web browser to display the content only if the content in iframe belongs to the same origin as the original web page. If the origin does not match, prevent displaying.
X-Frame-Options: ALLOW-FROM uri	Allow the web browser to display only if the origin matches the origin specified in uri. If the origin does not match, prevent displaying

"Allow-FROM uri" is only supported in Internet Explorer 8 or later and Firefox (except ESR) for PC and Android. The protocol scheme must be included in the uri. The following example allows display when contents are read in by an iframe from a page in http://example.jp

```
X-Frame-Options: ALLOW-FROM http://example.jp/
```

The X-Frame-Options only functions in some browsers when specified in the response header and will not function when specified in <meta http-equiv>.

In addition to the X-Frame-Options response header, code called "frame busting" that prevents the embedding of contents in an iframe is known to prevent clickjacking, but this method contains a number of issues, and thus is not recommended.

```
// Example of frame busting code
if( top != self ){
    top.location = self.location;
}
```

5.4. Content-Security-Policy

Content Security Policy (CSP) is a feature to mitigate XSS attacks by limiting web browser features by specifying the origin to which readable resources belong. Currently, Firefox, Google Chrome and Safari support CSP.

When CSP is enabled, the following web browser features are limited

- Reading resources through elements such as <script>, <img>, and <iframe>
- Inline JavaScript execution such as and <script>alert(1)</script>
- Code generation from strings through the “eval” function or Function constructor in JavaScript
- Javascript scheme and data scheme
-

CSP is specified by X-Content-Security-Policy response header in Firefox, Content-Security-Policy response header or X-WebKit-CSP response header in Google Chrome and X-WebKit-CSP response header in Safari. In the future, it will probably be unified to the Content-Security-Policy response header. In order to handle all of them, the 3 types of response headers should be output as shown below.

```
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Type: text/html; charset=utf-8
X-Content-Security-Policy: default-src 'self'; img-src img.example.jp
X-WebKit-CSP: default-src 'self'; img-src img.example.jp
Content-Security-Policy: default-src 'self'; img-src img.example.jp
```

In this example, image files are allowed to be read only from img.example.jp, and all other resources are allowed to be read only from the same origin where documents belong to. In the above CSP

example, default-src and img-src are referred to as directives, and self and img.example.jp are referred to as directive sources. Directives and directive sources have many definitions provided by the W3C, and implementation status differs between browsers. Some well-known directives are listed below.

Table 5-3 Example of directives

default-src	Specify valid sources allowed as default or not specified in the other directives
script-src	Specify valid sources of JavaScript
style-src	Specify valid sources of stylesheets
img-src	Specify valid sources of images
frame-src	Specify valid sources of loading frames

Directive sources can be specified by host name such as img.example.jp shown above and by using a wild card like *.example.jp or scheme name such as https: or reserved names listed below.

Table 5-4: Example of directive sources

'self'	Allow only if sources belong to the same origin as the document's
'none'	Do not allow any origins
'unsafe-inline'	Allow inline script and style description in script-src and style-src
'unsafe-eval'	Allow JavaScript features to dynamically generate code from strings such as eval, Function, setTimeout and setInterval

In CSP, if a policy violation is detected, the browser can send a report automatically to the server. The destination of the report is specified by the report-uri directive (Response headers other than Content-Security-Policy are omitted).

```
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Type: text/html; charset=utf-8
Content-Security-Policy: default-src 'self'; img-src img.example.jp;
    report-uri http://example.jp/report.cg
```

By receiving policy violation reports, the site administrator can detect XSS attacks earlier.

If Content-Security-Policy-Report-Only (or X-WebKit-CSP-Report-Only) or Also specifying Content -Security-Policy-Report-Only (or X-WebKit-CSP-Report-Only or X-Content –Security –Policy

–Report-Only) in place of Content-Security-Policy, the resource is not blocked and read in even if a policy is violated, and only the violation report is sent.

```
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Type: text/html; charset=utf-8
Content-Security-Policy-Report-Only: default-src 'self';
     img-src img.example.jp; report-uri http://example.jp/report.cgi
```

CSP is continuing to undergo specification changes and implementation. The status of implementation differs among not only browsers but their versions.

5.5. Content-Disposition

By specifying Content-Disposition: attachment in the response header, the contents are not displayed in the browser but can be saved as a file. This function is widely used in Webmail or message boards to provide a function for file attachments. In Internet Explorer, by pressing "Open" in the dialog box that is displayed, a XSS attack may occur.

```
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Length: 42
Content-Disposition: attachment; filename="index.html"
Content-Type: text/html; charset=utf-8

<html>
<script>alert(1)</script>
</html>
```

For example, when accessing a server that responds with the above response using Internet Explorer 8, a dialog like below is displayed.

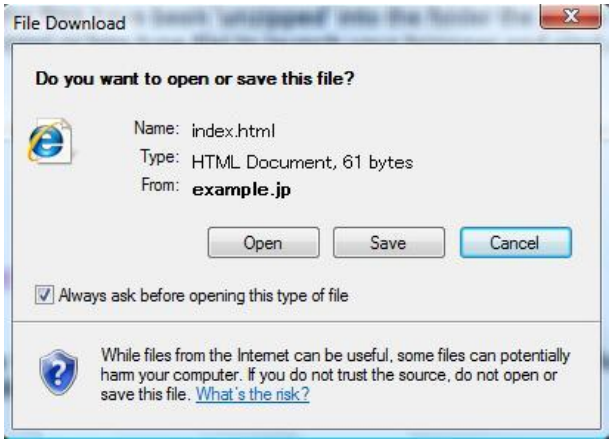

Figure 5-2: Confirmation Dialog for Downloading

If a user selects "Open" in this dialog, the contents are displayed in the browser with the server as the origin, which may result in a stored XSS attack. In Internet Explorer 8 or later, you can prevent displaying the "Open" button by specifying the X-Download-Options: noopen response header.

```
Content-Type: text/html; charset=utf-8
Date: Tue, 1 Jan 2013 09:00:00 GMT
Content-Length: 42
Content-Disposition: attachment; filename="index.html"
X-Download-Options: noopen

<html>...
<script>alert(1)</script>
</html>
```

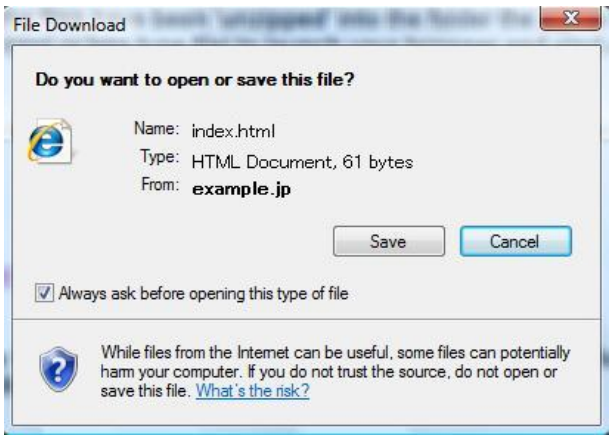

Figure 5-3: Confirmation Dialog for Downloading

5.6. Strict-Transport-Security

By using the HTTP Strict Transport Security (herein “HSTS”) feature, it will force the web browser to access all sites that return the HSTS response header via HTTPS for a period of time.

This feature can be enabled by adding Strict-Transport-Security response header to the HTTPS response header as shown below. This response header is ignored if it is added to the HTTP response header.

```
Strict-Transport-Security:max-age=1800
```

In the above example, the browser will access the site via HTTPS from which the Strict-Transport-Security response header was received from for 30 minutes.
If the web browser receives Strict-Transport-Security response header again within the period specified in max-age, the validity period of HSTS will be updated to the new value.
If the includeSubdomains parameter is specified as option, HSTS will be enabled including the subdomains. If includeSubdomains is not specified, the subdomains are not included.

```
Strict-Transport-Security:max-age=1800; includeSubdomains
```

Currently, Firefox, Google Chrome, and Opera support HSTS.

Reference: Do not Track

This section has described server-side features that can be configured or implemented by developers and what kinds of processes are expected from the client-side browser. But the user can configure the browser and add a request header to expect a server-side process.

Do Not Track (herein ”DNT”) is a request header used to express a user intention “Do not track my browsing history” to web sites. In Firefox, Google Chrome (for PC only), Safari, Opera, and Internet Explorer after version 9, users can enable this feature by explicitly changing the setting (In Internet Explorer 10, this feature is enabled as default if it is installed with “Easy Install”), and “DNT: 1” will be added to each request as the request header.

The specific response for web applications that receive a request header including “DNT: 1” is left up to the service providers’ discretion, but user privacy should be considered carefully during the implementation depending on the service providers or the nature of service. “Do not Track Field Guide” published by Mozilla Japan describes some case studies as the policy that the service providers may adopt.

6. Conclusion

This report examined functions that must be considered in terms of security prior to developing HTML5 web applications and consolidated security information that is scattered across the globe. We would be glad if you would use this report as a reference when developing HTML5 web applications.

This report will be revised as changes to the HTML5 specification occur.

References

- HTML5
  http://www.w3.org/TR/html5/
  (Last access: Oct 30th, 2013)
- HTML Standard
  http://www.whatwg.org/specs/web-apps/current-work/multipage/
  (Last access: Oct 30th, 2013)
- “How to Secure Your Web Site” 6th edition
  https://www.ipa.go.jp/security/vuln/websecurity.html
  (Last access: Oct 30th, 2013)
- Information-technology Promotion Agency, Japan :IPA Technical Watch:” Report regarding DOM Based XSS”
  https://www.ipa.go.jp/about/technicalwatch/20130129.html
  (Last access: Oct 30th, 2013)
- Cross-Origin Resource Sharing
  http://www.w3.org/TR/cors/
  (Last access: Oct 30th, 2013)
- RFC 6454 - The Web Origin Concept
  https://tools.ietf.org/html/rfc6454
  (Last access: Oct 30th, 2013)
- XMLHttpRequest
  http://www.w3.org/TR/XMLHttpRequest/
  (Last access: Oct 30th, 2013)
- The WebSocket API
  http://www.w3.org/TR/websockets/
  (Last access: Oct 30th, 2013)
- Web Workers
  http://www.w3.org/TR/workers/
  (Last access: Oct 30th, 2013)
- HTTP Header X-Frame-Options
  http://tools.ietf.org/html/draft-ietf-websec-x-frame-options
  (Last access: Oct 30th, 2013)
- MIME-Handling Change: X-Content-Type-Options: nosniff (Windows)
  http://msdn.microsoft.com/en-us/library/ie/gg622941(v=vs.85).aspx
  (Last access: Oct 30th, 2013)
- Controlling the XSS Filter - IEInternals - Site Home - MSDN Blogs
  http://blogs.msdn.com/b/ieinternals/archive/2011/01/31/controlling-the-internet-explorer-xss-filter-with-the-x-xss-protection-http-header.aspx

(Last access: Oct 30th, 2013)

- Tracking Preference Expression (DNT)
  http://www.w3.org/TR/tracking-dnt/
  (Last access: Oct 30th, 2013)
- HTTP Strict Transport Security (HSTS)
  https://tools.ietf.org/html/draft-ietf-websec-strict-transport-sec
  (Last access: Oct 30th, 2013)
- Content Security Policy 1.0
  http://www.w3.org/TR/CSP/
  (Last access: Oct 30th, 2013)
- HTML5 Security Cheatsheet
  http://html5sec.org/
  (Last access: Oct 30th, 2013)
- HTTP access control (CORS) - HTTP | MDN
  https://developer.mozilla.org/en-US/docs/HTTP/Access_control_CORS
  (Last access: Oct 30th, 2013)
- Do Not Track Field Guide
  http://www.mozilla.jp/static/docs/firefox/dnt-guide.pdf
  (Last access: Oct 30th, 2013)
- HTML5 Implementation Issues in IE8 (and IE9) - IEInternals - Site Home - MSDN Blogs
  http://blogs.msdn.com/b/ieinternals/archive/2009/09/16/bugs-in-ie8-support-for-html5-postmessage-sessionstorage-and-localstorage.aspx
  (Last access: Oct 30th, 2013)

Appendix

Browsers Used in this Report

The browsers and their versions that were used for the investigation are listed below. If the version is not listed here, they have been made clear throughout the report. All patches released by February, 2013 were applied to the web browsers.

Browsers Used (For PC)

- Internet Explorer 6 through10
- Google Chrome version 25.0.1364.97 m (Released version)
- Mozilla Firefox 18.0.2 (release channel)
- Opera 12.14
- Safari 6.0.2 (For Mac)

Browsers Used (For Smartphones)

- Mozilla Firefox for Android 19.0
- Android 2.3 Default Web Browser
- Opera Mobile 12.10.ADR
- Safari 5.1 for iOS 5.1.1