security-assessment.com

Security-Assessment.com White Paper

Cross Context Scripting with Firefox

Prepared by:

Roberto Suggi Liverani
Senior Security Consultant
Security-Assessment.com

Date: 21 April 2010

Contents

Abstract

Cross Context Scripting (XCS) is a term coined for a browser based content injection in the Firefox chrome zone. This term was originally used by researcher Petro D. Petkov (pdp), when David Kierznowski found a vulnerability in the Sage RSS Reader Firefox extension[1].
XCS injection occurs between different security zones, an untrusted and a trusted zone. The untrusted zone is not trusted by the browser - this can be an Internet page located on a remote server, for example. Firefox also has a trusted zone, named Chrome. Chrome allows extensions to access and interface with core components of Firefox, such as XPCOM. In this manner, extensions can provide extra functionality to the user and extend the web browsers capability.

Same origin policy (SOP) restrictions do not allow untrusted content to interact or access resources within the Chrome zone (chrome://).
However, the Chrome zone can access untrusted content - and that's when "Cross Context Scripting" attacks are possible. If untrusted content is executed or rendered in the Chrome privileged zone, a malicious user has a means to inject code into a privileged browser zone.

This paper details several XCS cases. XCS attacks may be possible due to a lack of input filtering controls for example. However, other components may be vulnerable as well, including wrappers, XPCOM components, XUL overlays, the browser sandbox and DOM events.

This paper can be seen as complimentary to the presentations given at EUSecWest 2009[2], DEFCON 17[3] and "SecurityByte & OWASP AppSec Asia 2009"[4] security conferences. Additionally, an addendum to this whitepaper has been produced – Exploiting Cross Context Scripting Vulnerabilities in Firefox[5]. The addendum includes a number of exploits tailored for Cross Context Scripting vulnerabilities.

[1] Cross Context Scripting with Sage - http://www.gnucitizen.org/blog/cross-context-scripting-with-sage
[2] EUSecWest 2009 (London, UK) - Exploiting Firefox Extensions
http://dragos.com/esw09/exploiting_firefox_extensions-liverani-freeman_eusecwest09.pptx
[3] DEFCON 17 (Las Vegas, US) - Abusing Firefox Extensions
http://www.defcon.org/images/defcon-17/dc-17-presentations/defcon-17-roberto_liverani-nick_freeman-abusing_firefox.pdf
[4] OWASP AppSec Asia & SecurityByte 2009 (Gurgaon, IN) - Exploiting Firefox Extension
http://www.securitybyte.org/Slides/Day2_Orchid/Exploiting_Firefox_Extensions.pdf
[5] Exploiting Cross Context Scripting Vulnerabilities in Firefox – http://www.security-assessment.com/files/whitepapers/Exploiting_Cross_Context_Scripting_vulnerabilities_in_Firefox.pdf

1. Introduction

This section provides a brief introduction on Firefox extension technology and the Mozilla platform architecture. It does not intend to be a comprehensive overview of the Firefox extension architecture, but rather a reference to the concepts and technologies which are discussed in the rest of this paper.

The Mozilla platform is made of multiple components. The most important components in regards of Firefox extensions are XPCOM[6] (Cross Platform Component Object Model), JavaScript, XUL[7] (XML User Interface Language), Chrome[8], XBL[9] (XML Binding Language), XUL overlay[10], themes, skins and locales. These components can be found in the structure of an extracted extension file .xpi, a zip archive container.

1.1 XPCOM Component Model

XPCOM is the lowest layer of the Mozilla platform architecture. XPCOM provide core functionality to Firefox and extensions. XPCOM components are built out of objects and interfaces which can be queried by the browser and by the extensions. For example, when Firefox loads a web page, XPCOM components from the Necko (Mozilla network library) are employed to initiate a network connection. Gecko (Mozilla layout engine) XPCOM interfaces are then used for displaying and presenting the web page content.

Extensions can introduce new XPCOM components once installed. XPCOM components support multiple programming languages such as C++, Java, Python and JavaScript. JavaScript interfaces with XCOM objects via the XPConnect[11] layer. XPCOM objects can be found in the *components* folder of a Firefox extension.

1.2 XUL

XUL is type of XML dialect which provides GUI functions such as graphical gadgets, buttons and forms. Firefox extensions make extensive use of XUL to display pages and configuration windows. XUL pages can be found in the *content* folder of a Firefox extension.

[6] XPCOM – MDC - https://developer.mozilla.org/en/XPCOM
[7] XUL – MDC - https://developer.mozilla.org/en/XUL
[8] Chrome – MDC - https://developer.mozilla.org/en/Chrome
[9] XBL – MDC - https://developer.mozilla.org/en/XBL
[10] XUL Overlays – MDC - https://developer.mozilla.org/en/XUL_Overlays
[11] XPConnect – MDC - https://developer.mozilla.org/en/XPConnect

1.3 Chrome

Chrome is a term used in multiple contexts by Mozilla. Chrome generally represents a set of resources defined by a special URL scheme: chrome://

Since the installation location of Firefox can change from system to system, the chrome: scheme provides the browser a shortcut for mapping URIs to XUL content. An example of a chrome URL can be: *chrome://extensionname/content/page.xul*

Chrome is also used to indicate a special trusted zone within Firefox. Firefox extensions run from this zone and are completely trusted by the browser. This is a critical aspect of the Mozilla architecture security and this paper refers to Chrome mostly in reference to this definition.

Chrome can also be referred to as a package which contains XUL documents, JavaScript and XBL binding files. This is defined as the *content* of the extension. The Chrome package also includes *locale* (DTD files) and *skins* (CSS and images).

1.4 XBL - Custom tags

XBL (XML Binding Language) allows the definition of new XML nodes/elements or custom tags. Custom tags can inherit processing logic. The connection between the new tag and the processing logic is called a *binding*. The object logic can be written to take advantage of any of the services available to the platform, including all the XPCOM objects and other custom tags that possess their own bindings. XML content and XPCOM can be tied together via an XBL binding. The “tabbed navigation” in Firefox is an example of XBL.

1.5 XUL Overlay

XUL Overlays are a way of attaching other UI widgets to a XUL document at run time. Overlays can be added or merged to existing XUL elements. A new extension button on the Firefox status bar is an example of XUL overlay.

1.6 Themes, Skins and Locales

Chrome URLs are modified by the current browser language (the locale) as well as by the current theme. This means that supporting both an English and an Italian “Back” button is just a matter of having Chrome files defined in both languages.

Skins are composed of CSS files and images, which can be combined with JavaScript, XBL, XML and XUL content to provide a richer user experience.

2. XCS Cases

In this paper, we will focus on XCS injections which originates from untrusted content zones (such as the Internet) and are executed in the trusted Firefox chrome:// zone.

Multiple cases are covered in this paper and were studied for the purpose of analysing XCS attacks against vulnerable Firefox extensions. However, other cases can be encountered as well, depending on the nature of the extension and its functionality. The cases analysed in this paper are based on components and functionality that are typically encountered in Firefox extensions.

2.1 Case I: XCS via Event Handlers – Drag and Drop

In Firefox, a Drag and Drop[12] action is managed by a collection of event handlers, including dragstart, dragenter, dragover, dragleave, drag, drop and dragend. A Drag and Drop operation can involve text, links, images and DOM nodes. When the Drag and Drop action is performed on a DOM node, all node properties, attributes and methods are also included in the dragged object.

In this case, we will analyse an example of a vulnerable extension which trusts dragged DOM nodes from an untrusted web page. A malicious user may exploit this trust by creating a web page, in order to exploit unaware users who 'Drag and Drop' a malicious DOM node, such as a picture into a privileged zone.

In this scenario, an unaware user drags the malicious picture into the extension HTML editor - a Chrome privileged window. Even if filters exist to prevent <script> or other HTML tag injection, any JavaScript payload passed through DOM event handlers may be rendered as part of the node attribute (onLoad, onError, etc). In our case, the vulnerable extension appends the image to a DOM element *status-bar*, which is part of the browser.xul interface and belongs to the trusted Chrome zone. This is seen below in the example below:

Vulnerable Extension Code

```
[…]
<script>
// HTML Editor preview function
function preview(image) {
statusb = document.getElementById("status-bar");
statusb.appendChild(image);
}
</script>
<textbox type="txt" ondragover="preview
(event.dataTransfer.mozGetDataAt('application/x-moz-node',0));"/> […]
```

[12] MDC - Drag and Drop
https://developer.mozilla.org/en/DragDrop/Drag_and_Drop

The vulnerability in this case is located in the *preview()* function which implicitly trusts the *image* element passed via the Drag and Drop action.

An example of malicious XUL page is shown in the following table:

Malicious XUL (Untrusted Content)

```
[…]
<div draggable="true"
ondragstart="event.dataTransfer.mozSetDataAt('application/x-moz-node',
document.getElementById('b'), 0)">
<html:img id="b" src="mypicture.jpg"
onmouseover="maliciouspayload();"></html:img>
</div>
[…]
```

When the picture is dragged from the element into the extension HTML editor window, the extension will receive input from the untrusted zone via the Drag and Drop event handler (in this case onDragOver). If the extension previews and renders the dragged image, *maliciouspayload()* will be executed with Chrome privileges when the onmouseover event is fired.

2.2 Case II: Attacking Custom DOM event handlers

A custom DOM event can be created via the *createEvent()*[13] function. Custom DOM events can be used to exchange data between the Chrome zone and other untrusted zones. In such case, an event listener is also created via the *addEventListener()*[14] function to listen for a determined custom DOM event. The exchanged data is then processed by other functions of the extension.

In this case, we will analyse an example of a vulnerable extension which expects a specific flow before triggering a determined custom event. A malicious user may circumvent the intended extension flow and setup a web page which automatically invokes the custom event. This can bypass the intended order of events and exploit an insecure function called by the custom event handler.

In this scenario, the extension may wait for a certain sequence of actions from the user before triggering a custom event. Some examples include:

- Selecting items;
- Dragging and dropping items;
- Adding a tab;
- Filling a text area.

Once the custom event is triggered, a XUL overlay is loaded. As mentioned in the introduction, XUL overlays are a method of attaching other UI widgets to a XUL document at specific merge points. In this case, a new overlay may be loaded following the result of a user action.

The following table describes an example of a vulnerable extension:

Vulnerable Extension Code

```
[…]<statusbar id="status-bar">
  <statusbarpanel id="merge_point" label="Default Overlay"  />
 </statusbar>
[…]
<script>
var customExtension = {
  customListener: function(evt) {
    document.loadOverlay(evt.target.getAttribute("url"), null);
  }
}
document.addEventListener("CustomEvent", function(e) {
customExtension.customListener(e); }, false, true);
[…]</script> […]
```

[13] document.createEvent - https://developer.mozilla.org/en/DOM/document.createEvent
[14] element.addEventListener - https://developer.mozilla.org/en/DOM/element.addEventListener

The extension is waiting for an event named *CustomEvent*. In this case, *CustomEvent* can be generated by any untrusted page, as there is no validation to ensure the event originated from a determined domain. Location checks can be performed by the extension by comparing the *evt.target.owner.location* property to a trusted domain. The *evt.target.owner.location* will return the location from which the event originated.

In case the extension is validating the location, a malicious user can still bypass this by chaining bugs in the domain trusted by the extension. An XSS attack into the trusted domain can still trigger the event handler and consequently the event location would originate from the expected domain.

By looking again at the source code in the table in the previous page, we understand that a malicious user needs to trigger the *CustomEvent* event handler to also invoke the *customListener()* function. This function receives an argument, which is an event handler itself. The *customListener()* function retrieves the value of the attribute URL from the event and then passes it to the *document.loadOverlay()* function. This function is responsible for loading an overlay.

In Firefox, there is no SOP restriction from the Chrome zone, therefore an external XUL overlay may be loaded. XUL overlays can also contain JavaScript, and if the overlay is merged within a XUL page in the Chrome zone, then the JavaScript will be executed with Chrome privileges.

In this scenario, the malicious user needs to convince the victim to visit a page which triggers the *CustomEvent* event handler and passes a URL which points to a malicious XUL overlay. The table below illustrates an example of this exploit:

Malicious HTML (Untrusted Content)

```
<script>
var element = document.createElement("CustomExtensionDataElement");
element.setAttribute("url", "http://path/to/malicious_overlay.xul");
document.documentElement.appendChild(element);
var evt = document.createEvent("Events");
evt.initEvent("CustomEvent", true, false);
element.dispatchEvent(evt);
</script>
```

The malicious code triggers the *CustomEvent*. The event is dispatched to a DOM element *CustomExtensionDataElement*. As soon as the event is dispatched, this is identified by the event listener running in the Chrome extension page. The extension will then load the malicious_overlay.xul page, which contains the embedded JavaScript payload. An example of malicious_overlay.xul can be found on the following page.

Malicious Overlay-XUL (Untrusted Content)

```
[…]
<?xml version="1.0"?>
<overlay id="sample2"
xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
 <statusbar id="status-bar">
  <statusbarpanel id="merge_point" label="Overlay Merged"  />
 </statusbar><script src="http://malicious/javascript.js"></script>
[…]</overlay>
```

This exploit case is composed of two XCS attack phases. First, the event is created and dispatched. Then the payload exploits the function which is triggered when the event is dispatched – and it is this function that is responsible for loading a XUL overlay.

2.3 Case III: Cross Domain Content/Script Include

Extensions may include external content in several ways, such as by opening new windows, initiating network streams/sockets, performing redirections or permitting access to external resources. The following table summarises some of the methods that can be used by an extension to request external resources and the relative security implications.

Methods	URI supported	Comment
window.open()	javascript: data:	Privileged access can be gained if opened from the Chrome zone.
window.opendialog()	javascript: data:	Privileged access can be gained if opened from the Chrome zone.
nsIWindowWatcher - openWindow	javascript: data:	Privileged access can be gained if opened from the Chrome zone.
XMLHTTPRequest (GET,POST)	data:	Although the status of the response is zero, the data: URI can still be passed and it is part of the response. This does not open a new window, but still performs a GET request. However, response content may be rendered within a Chrome privileged context.

The following is an example of a vulnerable extension which renders untrusted content in a privileged Chrome window. A malicious user may have the ability to control directly or indirectly the untrusted content.

In this scenario, the extension allows users to save a list of favorite sites to a sidebar. The sidebar is located in a Chrome privileged window. The user needs to right-click on an <a href> element to add the site to the sidebar. This action can be performed from a web page on the Internet. The following example demonstrates a common method used by developers.

Vulnerable Extension Code

```
var request = new XMLHttpRequest();
request.open("GET", favoriteURL, false);
request.send(null);

var Sidebar = {
  addtoSidebar: function(url) {
            var h_frame = document.createElement("iframe");
            h_frame.setAttribute("src",url);
      h_frame.setAttribute("style","width:800px;height:800px");
            h_frame.setAttribute("id","iframe1");
            document.getElementById("status-bar").appendChild(h_frame);
  }
}
setTimeout('Sidebar.addtoSidebar("data:text/html,"+encodeURIComponent(req
uest.responseText));',5000);
```

In the above example, the *favoriteURL* variable is requested via an XHR request. It is then passed to the *addtoSidebar()* function which creates an iframe element. The iframe element is appended into the DOM element *status-bar*, which is part of a Chrome privileged window. Exploitation of this vulnerability requires the malicious user to convince the victim to choose a particular link such as the following:

Malicious Link (Untrusted Content)

```
[…]
<a href="data:text/html;base64,base64encodedmaliciouspayload">Malicious
Link</a> […]
```

Once the site is added, the malicious base64 encoded payload will be appended to the *status-bar* DOM element, and consequently the XCS injection will be executed with Chrome privileges.

2.4 Case IV: Injection via XBL

XBL bindings allow extensions to create new logic by associating elements with script, event handlers and Cascading Style Sheets (CSS). This provides more functionality when defining new content, event handlers or even methods and properties of a XUL widget. When an extension makes use of bindings, elements within the bindings are attached to the invoking page.

In this case, we will analyse an example of a vulnerable extension which performs an XBL binding *include*[15]. A malicious user may control data included within the XBL binding.

The extension code below allows a user to select an image and a color theme (e.g. blue) for the user profile page, a Chrome privileged window. Image and color theme can be chosen from a web page on the Internet. The XBL binding is used to include selected content into the user profile page.

The vulnerable XUL user profile page includes a CSS resource and a XBL binding as shown below:

Vulnerable XBL Binding

XUL (extension page):

```
[…]<div class="hidden"></div>[…]
function setProfileImage(image_node,theme)
    {
        var b = document.getElementsByTagName('div')[0];
        b.appendChild(image_node);
         if (theme == 'blue') {
            b.class="blue-profile";
            b.style.display="inline";
  }
    }
[…]
SetProfileImage(content.document.getElementsByTagName('image')[0],'blue')
;
```

CSS:

```
div.hidden {
    display: none;
}
div.blueprofile {
    -moz-binding: url('binding.xml#default');
}
```

[15] XUL Tutorial – Anonymous content - https://developer.mozilla.org/en/XUL_Tutorial:Anonymous_Content#includes_Attribute

XBL:

```
[…]
<binding id="default">
     <content> <xul:box flex="1"/>
        <children includes="image">
        </children>
  </content>
  </binding>
[…]
```

The function *SetProfileImage()* is responsible for retrieving the user image and the theme selected. This function appends the image to a DOM element within the extension XUL user profile page. At this stage, the appended DOM element is not visible, as the element has its class set to *hidden*, where display is set to none (as per the CSS file). Therefore the picture selected by the user is not loaded. Even if a malicious user managed to convince a victim to select a picture with a malicious JavaScript payload attached to an onLoad event handler, this will not be exploitable as the picture is not yet loaded.

This vulnerability is related to how the XBL binds the elements within the XUL extension page. By looking at the XBL source code, the <children includes='image'> tag is an indication that the <image> tag will be included as a child element of the element, within the XUL page. The *includes* attribute also imports any attribute associated to the <image> DOM node. Therefore, only when the binding is performed does the malicious user have a chance to exploit this vulnerability – and only if the user selected an <image> with a malicious onLoad event handler.
Additionally, the 'blue' theme allows the vulnerability to be exploited as the class *blue-profile* is associated to the XBL binding. The exploit may look similar to the following if the malicious user is using a XUL page:

Malicious Payload Through XUL
`<image src="http://path/to/mymaliciouspicture/" onload="eviljspayload"/>;`

2.5 Case V: Attacking Wrappers

Multiple wrappers[16] exist in Firefox and are used to protect privileged interfaces, functions and objects. Firefox version 3 automatically protects objects against unsafe content.

In this case, we will analyse an example of a vulnerable extension employing a method of bypassing a Firefox wrapper. A malicious user may exploit this extension flaw and modify properties and methods of privileged objects.

The extension code below creates an object in Chrome which is then passed to a function on an untrusted page:

Vulnerable Extension Code (Chrome)

```
[…] var myObject = new Object();
  myObject.substract = function(a, b) {
    return (a - b)
  }; //defined in Chrome
[…]

var ObjCon = content.wrappedJSObject.contentfunction(myObject);
[…]

object.substract(5,3); // use of the subtract method following content
interaction
```

By default in Firefox 3.x, an implicit deep XPCNativeWrapper[17] is created for the *myObject* object when it is exposed to the untrusted content. This wrapper limits access to the properties and methods of the object protected. However, in this case the developer has striped the wrapper by using the wrappedJSObject[18] property on the content element. The wrappedJSObject property allows access to the underlying JavaScript object, bypassing the wrapper functionality.
If the *contentfunction()* method changes the Chrome object by overriding an existing method (*subtract()* in this case), then there is a chance of exploitation.

Malicious Content Function (untrusted page)

```
[…]function contentfunction(a) {
    a.subtract = function(a, b) {
        evilpayloadhere; return (a-b);   };
[…]
```

[16] XPConnect wrappers - https://developer.mozilla.org/en/XPConnect_wrappers
[17] MDC – XPCNativeWrapper - https://developer.mozilla.org/en/XPCNativeWrapper
[18] MDC – wrappedJSObject - https://developer.mozilla.org/en/wrappedJSObject

Any further use of the *subtract()* method is potentially unsafe, as a malicious user may have added an arbitrary JavaScript payload which is executed each time the *subtract()* method is used within the extension code.

2.6 Case VI: Attacking XPCOM Components

As mentioned in the introduction, XPCOM components provide core functionality to Firefox and Firefox extensions. Thousands of XPCOM components are shipped with Firefox, but extensions may include their own XPCOM components.

In this case, we will analyse an example of a vulnerable extension which includes an XPCOM component that trusts a locally-defined object, which inherits properties and methods from an untrusted page. A malicious user may exploit this trust to override properties of a privileged object.

This XPCOM component is used on an untrusted page to retrieve a specific JavaScript object. This object is used to preview content, so a window will be opened after retrieving all the object properties. The following is the extension code which makes use of the vulnerable XPCOM component, named *PreviewWindow*.

Example of Extension code

```
[…]
var myComponent =
Components.classes['@test.test.com/previewwindow;1'].createInstance(Comp
onents.interfaces.nsIPreviewWindow);
  var myComponent = Components.classes['@
test.test.com/previewwindow;1'].getService().wrappedJSObject;

[…]

[pobject, option] =
myComponent.preview(content.wrappedJSObject.renderobject);

[…]

var ww = Components.classes['@mozilla.org/embedcomp/window-
watcher;1'].getService(Components.interfaces.nsIWindowWatcher);

var win = ww.openWindow(null, my.url, 'title', option, null);
```

The variables *pobject* and *option* are invoking the *myComponent.preview* interface. This takes an object called *renderobject* as an argument, which originates from the untrusted web page. The wrappedJSObject is required to retrieve all the *renderobject* properties and methods.
The *PreviewWindow* XPCOM component is defined in a different file, normally placed in the *components* folder of the extension.
The vulnerable XPCOM component in this case is defined as the following:

Example of Vulnerable XPCOM Component

```
[…]
PreviewWindow.prototype = {
  // define the function we want to expose in our interface
  preview: function(a) {
var option = null;
     function checkprotection(protection,scheme)
    {
       if(protection=='enabled') {
        if(scheme=='data:') { alert('this URI scheme is not supported');
// this is not allowed
        scheme = scheme.replace("data:",""); // strips data: for security
[…]
        }
            option = "Chrome,centerscreen";
       }
           if(protection=='disabled') {

            option = "Chrome,centerscreen";
[…]
           }
    }
[…]
    var pobject = new Object;
    pobject = {protection:'enabled'};
[…]
pobject = a;
    checkprotection(pobject.protection,pobject.scheme);
return [pobject, option];
```

The XPCOM *PreviewWindow* preview interface takes an argument *a*, which in this case is the object *renderobject* from the untrusted page. The object *pobject* is initialised in the XPCOM component. *pobject* will inherit the properties from *renderobject*, including the URL as well as other properties. But a protection property is also initialised and *enabled* locally by the XPCOM interface. This protection is then checked by the *checkprotection()* function. This is a custom security measure implemented by the

extension developer. If protection is *enabled*, then the *checkprotection()* function performs a further check on the scheme property of the object passed. If a data URI scheme is found, an error is thrown by the extension. On the other hand, if the protection is *disabled*, any URI scheme is accepted as no further checks exist to validate the URI scheme.
The vulnerability here is in the order that the variable *pobject* inherits the properties of *renderobject*. This becomes clearer by looking at an untrusted malicious page:

Example of Malicious Untrusted Page

```
[…]
renderobject = {url: 'data:text/html,base64encodedevilpayload',
level:'1', protection:'disabled', scheme: 'data:', […] };
[…]
```

The object *renderobject*, expected from the XPCOM interface *PreviewWindow*, comes with several properties. The mistake made by the developer is to assume that a malicious page wouldn't override a property defined locally – *protection*.
Looking at the table above, the malicious payload sets the *protection* property to *disabled*.
If we go back to the XPCOM component code, we see that *pobject* inherits all the properties of *renderobject* in the statement *pobject = a*;.
Following this statement, the *checkprotection()* function is called to perform the security checks. At this stage, the protection property has already been overridden from the untrusted page and has been *disabled*. This disables the *checkprotection()* function and no validation will be performed of the URI scheme used.
When the window is created via the *win* declaration in the extension code to preview the content, the protection mechanism in the extension will be disabled. Because of this, any payload passed via the data: URI scheme property of *renderobject* will be executed with Chrome privileges.

2.7 Case VII: Sandbox Chrome Leakage

Sandboxes[19] are created by extensions to allow restricted execution of JavaScript content within a specified context, which may be a DOM window or a URI. A context is specified in order to apply the SOP. A sandbox is often used with JSON, as JSON format is typically employed to handle external untrusted data source. JSON content is often executed in a sandbox and then processed by the extension.

In this case, we will analyse an example of a vulnerable extension which handles sandbox objects in an insecure manner. A malicious user may exploit this extension flaw to invoke functions with Chrome privileges.

The code below shows the *untrusted_code* executed in a sandbox tied to the about:blank zone. This zone is special and is not privileged. *untrusted_code* cannot run with Chrome privileges in the case below:

Example of Sandbox

```
<script>
   String.prototype.toUpperCase = function() {
[…]
 // code here runs with Chrome privileges […]
foo = Components;
var g = foo.classes;
return this;
   }
   var s = Components.utils.Sandbox("about:blank");
    var result = Components.utils.evalInSandbox(untrusted_code, s);
if(typeof result == "string") {
   var j = result.toUpperCase();
}
    </script>
```

However, if the *result* variable ends up being a String data type, then the string *j* is declared and its content is defined by the code evaluated via the *evalInSandBox()* function. The *toUpperCase()* String function is invoked when *j* is declared, and runs with Chrome privileges. In this example, *foo.classes* will be declared even though *result* is a String object which belongs to about:blank.
Exploitation of this bug may be complicated depending on the function which is invoked from the chrome:// zone and how it handles untrusted data. A malicious user may be able to exploit this condition by making *untrusted_code* return a String object. This would make the extension declare *foo.classes* and potentially exploit a bug located in a different part of the extension.

[19] MDC – Sandbox - https://developer.mozilla.org/en/Components.utils.evalInSandbox

2.8 Case VIII: Bypassing nsIScriptableUnescapeHTML.parseFragment()

The *nsIScriptableUnescapeHTML* XPCOM interface is often used to filter HTML content rendered in privileged contexts such as the chrome:// zone. According to MDC (Mozilla Developer Center)[20], this interface provides one of the "ways" to unescape untrusted content. However this interface cannot be trusted completely, and in some cases it provides a false sense of security for developers. This can be confusing, as found in the WizzRSS case[21].

In this case, we will analyse an example of a vulnerable extension which trusts the *nsIScriptableUnescapeHTML* parsing function to filter untrusted content which is rendered in a Chrome privileged window. A malicious user may bypass input filtering performed by the *nsIScriptableUnescapeHTML* parsing function and perform a XCS injection attack.

This extension code is retrieving HTML content which is then appended to a DOM element. That element is part of the browser.xul page (*status-bar*), as can be seen in the code below:

Example of nsIScriptableUnescapeHTML

```
<script>
var target = document.getElementById("status-bar");
[…]
var fragment = Components.classes["@mozilla.org/feed-unescapehtml;1"]
.getService(Components.interfaces.nsIScriptableUnescapeHTML).parseFragment
(payload, false, null, target);
target.appendChild(fragment);
[…]
</script>
```

The *parseFragment()* function performs filtering on both HTML and XML payloads passed via the *payload* variable. The table on the following page summarises some of the test cases attempted.

[20] MDC - Web content in an extensions without security issues
https://developer.mozilla.org/En/Displaying_web_content_in_an_extension_without_security_issues
[21] nsIScriptableUnescapehtml.parseFragment () issues
http://wizzrss.blat.co.za/2009/11/17/so-much-for-nsiscriptableunescapehtmlparsefragment

Payload	Processed by parseFragment()	Comment
<textarea title="a" id="1" onfocus="alert(1)">jjjj</textarea>	<textarea title="a" id="1">jjjj</textarea>	Event attributes such as onLoad, onmouseover are not appended to the DOM.
<iframe src=data:text/html;base64,YWFhYWFh width=10 height=3 />	Null	<iframe> is not appended.
a<script>dump(1)</script>	a	<script> is not appended.
a	a	The href attribute points to a JavaScript URI scheme and it is appended to the DOM.
<form action="javascript:alert(window)"><input type=submit></form>	<form action="javascript:alert(window)"><input type=submit></form>	This goes unfiltered as in the case above.

The last two test cases, using the javascript: URI produced interesting injection vectors for a malicious user. For example, in the <a href> element case, when a user clicks on the link the code will be executed with Chrome privileges.

Even if filtering is performed by the *parseFragment()* function, other avenues of attack exist by injecting and/or storing potential code that can be triggered by a specific sequence of steps (e.g. click a link, select an option, submit a form).

3. Conclusion

This paper has demonstrated different ways of attacking Firefox extensions via Cross Context Scripting (XCS) vulnerabilities. Several XCS cases have been discussed, including vulnerable extension code and exploit.

XCS are made possible through the way Firefox browser and extensions work together. Firefox trusts extensions and execute them in the Chrome privileged security zone. A vulnerable extension can be exploited to execute malicious code in the Chrome zone.

This is particularly dangerous if the vulnerable extension handles untrusted content. In such case, a XCS attack will be launched from a web page on a remote server. The malicious page will contain an exploit for the vulnerable extension which would compromise a user's system.

As long as extensions are trusted and interact with the browser and the OS without restrictions, XCS remains a serious attack vector against Firefox users.

It is recommended that extension developers follow secure code practices when developing Firefox extension. Best practices can be taken from several resources, such as the OWASP Development guide[22] and the Mozilla Developer Center[23] (MDC).

In terms of testing security vulnerabilities in Firefox extensions, the OWASP Testing Guide[24] can be taken in consideration along with our EUSecWest 2009 presentation[25] , which includes a checklist to follow when auditing security extensions.

Finally, end-users may consider examining change logs of security issues before installing an extension. Extension should be updated whenever a patch is available and Safe Mode[26] setting can be enabled to disallow Firefox Extensions. This setting may be taken in consideration by high risk organizations as well.

[22] OWASP Development Guide - http://www.owasp.org/index.php/Category:OWASP_Guide_Project
[23] MDC - https://developer.mozilla.org/En
[24] OWASP Testing Guide - http://www.owasp.org/index.php/Category:OWASP_Testing_Project
[25] EUSecWest 2009 (London, UK) - Exploiting Firefox Extensions
http://dragos.com/esw09/exploiting_firefox_extensions-liverani-freeman_eusecwest09.pptx
[26] Safe Mode - http://support.mozilla.com/en-US/kb/Safe+Mode

4. References

- Cross Context Scripting with Sage
  http://www.gnucitizen.org/blog/cross-context-scripting-with-sage
- OWASP AppSec Asia & SecurityByte 2009 (Gurgaon, IN) - Exploiting Firefox Extension
  http://www.securitybyte.org/Slides/Day2_Orchid/Exploiting_Firefox_Extensions.pdf
- DEFCON 17 (Las Vegas, US) - Abusing Firefox Extensions
  http://www.defcon.org/images/defcon-17/dc-17-presentations/defcon-17-roberto_liverani-nick_freeman-abusing_firefox.pdf
- EUSecWest 2009 (London, UK) - Exploiting Firefox Extensions
  http://dragos.com/esw09/exploiting_firefox_extensions-liverani-freeman_eusecwest09.pptx
- Update Scanner - Chrome Privilege Code Execution
  http://www.security-assessment.com/files/advisories/Update_Scanner_Firefox_Extension_Security_Advisory.pdf
- Coolpreviews - Chrome Privileged Code Execution
  http://www.security-assessment.com/files/advisories/CoolPreviews_Firefox_Extension_Security_Advisory.pdf
- Yoono Firefox Extension Privileged Code Injection
  http://www.security-assessment.com/files/advisories/Yoono_Firefox_Extension_Privileged_Code_Injection.pdf
- ScribeFire Firex Extension Privileged Code Injection
  http://www.security-assessment.com/files/advisories/ScribeFire_Firefox_Extension_Privileged_Code_Injection.pdf
- Feed Sidebar Firefox Extension Privileged Code Injection
  http://www.security-assessment.com/files/advisories/Feed_Sidebar_Firefox_Extension_Privileged_Code_Injection.pdf
- WizzRSS Firefox Extension Privileged Code Injection
  http://www.security-assessment.com/files/advisories/WizzRSS_Firefox_Extension_Privileged_Code_Injection.pdf
- OWASP NZ Day (Auckland, NZ) - Exploiting Firefox Extensions
  http://www.owasp.org/images/6/6e/Owasp_nz_day_09_roberto_suggi_liverani_nick_freeman_exploiting_ff_extensions.pptx
- RSnake – XSS CheatSheet
  http://ha.ckers.org/xss.html
- WASC – Script Mapping
  http://projects.webappsec.org/Script-Mapping
- MDC - Drag and Drop
  https://developer.mozilla.org/en/DragDrop/Drag_and_Drop
- MDC - Interaction between privileged and non privileged pages
  https://developer.mozilla.org/en/Code_snippets/Interaction_between_privileged_and_non-privileged_pages#Sending_data_from_unprivileged_document_to_Chrome

- MDC - Web content in an extensions without security issues
  https://developer.mozilla.org/En/Displaying_web_content_in_an_extension_without_security_issues
- MDC - Working with Windows in Chrome code
  https://developer.mozilla.org/en/Working_with_windows_in_Chrome_code
- URI Scheme
  http://en.wikipedia.org/wiki/URI_scheme
- MDC - Introduction to XBL
  https://developer.mozilla.org/en/XUL_Tutorial:Introduction_to_XBL
- MDC – wrappedJSObject
  https://developer.mozilla.org/en/wrappedJSObject
- MDC - XPConnect wrappers
  https://developer.mozilla.org/en/XPConnect_wrappers
- MDC – XPCNativeWrapper
  https://developer.mozilla.org/en/XPCNativeWrapper
- MDC - Safely accessing content DOM from Chrome
  https://developer.mozilla.org/en/Safely_accessing_content_DOM_from_Chrome
- MDC - How to build an XPCOM in JavaScript
  https://developer.mozilla.org/en/How_to_Build_an_XPCOM_Component_in_Javascript
- MDC – Sandbox
  https://developer.mozilla.org/en/Components.utils.evalInSandbox
- Sandbox - http://hublog.hubmed.org/archives/001570.html
- nsIScriptableUnescapehtml.parseFragment () issues
  http://wizzrss.blat.co.za/2009/11/17/so-much-for-nsiscriptableunescapehtmlparsefragment
- MDC – nsIScriptableUnescapeHTML.parseFragment() -
  http://developer.mozilla.org/en/nsIScriptableUnescapeHTML
- Stefano Di Paola, kuza55 - Attacking Rich Internet Applications
  http://www.ruxcon.org.au/files/2008/Attacking_Rich_Internet_Applications.pdf
- Rapid Application Development with Mozilla!
  http://mb.eschew.org/

About Security-Assessment.com

Security-Assessment.com is an established team of Information Security consultants specialising in providing high quality Information Security Assurance services to clients throughout Australasia. We provide independent advice, in-depth knowledge and high level technical expertise to clients who range from small businesses to some of the world's largest companies

Security-Assessment.com provides security solutions that enable developers, government and enterprises to add strong security to their businesses, devices, networks and applications. We lead the market in on-line security compliance applications with the SA-ISO Security Compliance Management system, which enables companies to ensure that they are effective and in line with accepted best practice for Information Security Management.

Copyright Information