APPLIED SECURITY PROJECT

Davide Martintoni

January 22, 2015

Abstract

This report explains my work on the exploitation of a known vulnerability in an historical version of Firefox. Here I show the result of my analysis on the vulnerability and the creation of a malicious web page that uses this issues in library included in some version of Firefox, in order to inject shellcode into the client.

1 Introduction

For this research project I would have had to choose a known vulnerability of an old version of Firefox and write a working exploit for it. I've scaned the web in order to find a well documented vulnerability that seems to be exploitable. After a quick research I've noticed that it's really hard to find precise information on most of them. However I decided to work on a vulnerability (CVE 2009-3373[1]) of the GIF management in the library "libpr0n".
This vulnerability was reported by iDefense to the security@mozilla.org. Credit should go to regenrecht[2]. iDefense confirmed the existence of this vulnerability using the Mozilla Firefox versions 3.0.13 and 3.5.2 on 32-bit Windows XP SP3. Other versions, and potentially other applications using libpr0n, are also suspected to be vulnerable.

2 Vulnerability CVE 2009-3373

This weakness is located in the GIF management system, on closer analysis in the GIF color map parsing library that can lead to a remote exploitation of a buffer overflow in the Mozilla Foundation's libpr0n image processing library allowing attackers to execute arbitrary code[2].

The libpr0n GIF parser was designed using a state machine which is represented as a series of switch/case statements. A particularly interesting state, 'gif_image_header', is responsible for interpreting a single image/frame description record. One of the fields, 'depth', specifies the number of bits per pixel in the image. A single GIF file may contain many images, each one of them with

```
mGIFStruct.local_colormap_size = 1 << depth;
PRUint32 paletteSize;
if (mGIFStruct.images_decoded) {
  // Copy directly into the palette of current frame,
  // by pointing mColormap to that palette.
  mImageFrame->GetPaletteData(&mColormap, &paletteSize);
} else {
  // First frame has local colormap, allocate space for it
  // as the image frame doesn't have its own palette
  paletteSize = sizeof(PRUint32) << realDepth;
  if (!mGIFStruct.local_colormap) {
    mGIFStruct.local_colormap = (PRUint32*)PR_MALLOC(paletteSize);
    if (!mGIFStruct.local_colormap) {
      mGIFStruct.state = gif_oom;
      break;
    }
  }
  mColormap = mGIFStruct.local_colormap;
}
const PRUint32 size = 3 << depth;
if (paletteSize > size) {
  // Clear the notfilled part of the colormap
  memset(((PRUint8*)mColormap) + size, 0, paletteSize - size);
}
if (len < size) {
  // Use 'hold' pattern to get the image colormap
  GETN(size, gif_image_colormap);
  break;
}
// Copy everything, go to colormap state to do CMS correction
memcpy(mColormap, buf, size);
```

Figure 1: Vulnerable code of library "libpr0n"

a different color map associated.

Consider the source code from "mozilla\modules\libpr0n\decoders\gif2.cpp"[3] in figure 1

The problem here is that it is possible to enter the 'gif_image_header' state more than once while still having 'mGIFStruct.images_decoded' equal to zero. Normally 'images_decoded' is incremented after every whole image has been decoded. However, by providing invalid LZW data, one can skip this step and enter the 'gif_image_header' state again.

On the second iteration, I have the first image already decoded but for this bug I hit the "else" part of the code in figure 1. Since the local color map was already allocated in the previous iteration, the branch with 'PR_MALLOC()' will not be taken. As a result, the memory allocated for the color map remains the same size regardless of the 'depth' field of a second image. So after a few other operation in the last line of the code in figure 1 the second image's data is copied into the buffer without respecting the buffer's size. This results in an

exploitable heap overflow condition.

2.1 How to exploit CVE 2009-3373

This vulnerability allow me to overflow the heap right after the malformed GIF that the browser load in the memory. Once I can overwrite the memory I should use those bytes of memory to store a return address that point a memory location in which I can store my shellcode. To load my shellcode in memory I can use a Javascript based heap spray. All the detail of this process are presented in the followings sections.

Exploitation of this vulnerability results in the execution of arbitrary code with the privileges of the user running the vulnerable application. To exploit this vulnerability, a targeted user must load a malicious Web page created by an attacker that contains a malformed GIF that trigger the vulnerability and a few line of javascript that store my shellcode in the heap at the desired address. An attacker typically accomplishes the infection via social engineering or injecting contents into compromised, trusted sites.

3 Environment Setup

Once gathered all the necessary information I've setup my working environment. I've loaded a virtual machine with Winodows XP sp3, installed Firefox 3.5.2 and I've also found some useful tools in order to deeply understand this vulnerability.

I've used "OllyDbg[4]", a debugger that allowed me to attach to the Firefox process, check memory, registers, and disassmbled the memory. Furthermore I could set conditional breakpoint on some calls (e.g I've used a breakpoint in the "memcpy" instruction only when the "EAX" register had a precise value). Another brilliant feature of this software is the opportunity of setting breakpoints on memory location to access or write.

An additional help was "VMMap[5]", a useful software that allowed me to check the memory of the process graphically, splitting it in heap, personal data, stack and others. Thanks to this tool I've learned how and where Firefox allocates data stored with javascript code.

4 Analysis

After the initial setup difficulty I've started to analyze the vulnerability by myself. First step I've written a C++ file that writes a "malicious" GIF with two images. The first one has associated a color map of 12 8-bit char of letter "B". With this known character I can search inside the memory and dump it in order to find where this header is stored. The second image contained a bigger (21 char) color map, but, without the last byte. Chopping that last byte is quite important. Otherwise execution would go to the function "ConvertColormap",

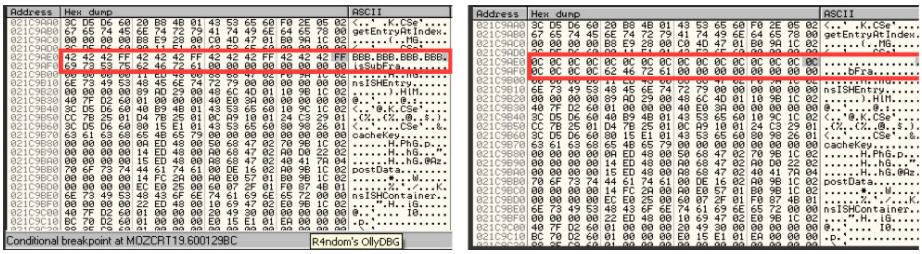

Figure 2: Overwrite of memory locations that contains "isSubFra"

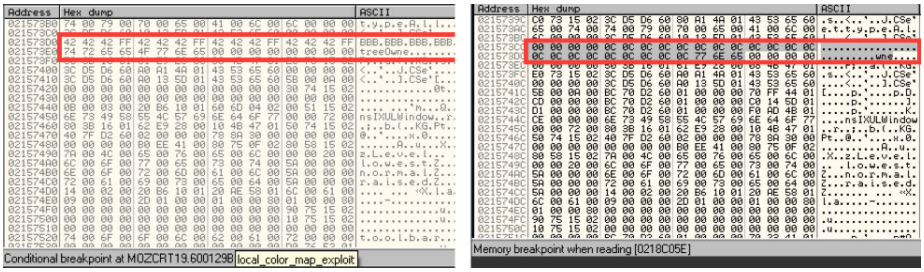

Figure 3: Overwrite of memory locations that contains "treeOwne"

where our overflown buffer full of RGBRGBRGB would turn into RGB + alpha where alpha is set to 0xFF. Lead any important structure or function pointer to 0xFFbbggrr would not be useful, as it would point outside the user space memory. The second header, in the color map field, contains the address where I want to redirect the programm flow. I have chosen the adress 0x0c0c0c0c for heuristic reasons and so my header contains those characters to overflow important things with my address. Javascript in a frash page will start to store string near the 0x06000000 address and then increase the address until it ends to load every string. Indeed 0x0c0c0c0c is certainly located after the start point of the Javascript string location. Moreover 0x0c0c0c0c can be read from the computer as an instruction and not only as an address but this will be explained in the attack section.

Once prepared the GIF I've written a simple web page with those images and I've attached my debugger to see what really causes my malicious GIF. I've found out that the GIF management loads the header in a private heap, in the middle of other information that this browser use to provide its service. Unfortunately the heap management is not predictible and every time that I load my GIF, the first header is stored in a different place, and my second header full of "0c0c0c0c", overflowing and overwriteing random information.

As we see in figure 2 and figure 3 my second GIF overflow after the header of the first one (in figure 424242FF..) but this leads always to overwrite memory

cells that contains different information. This is caused by the heap management of the browser that always store the GIF in a different place. So we can't predict what I'm going to overwrite with my header and this is a big threat to my attack reliability. Indeed my analysis shows that sometimes those code that I've overwrote isn't used by the browser, so nothing happened. On the contrary, when the browser hits my overflow this code arrives into the program execution but always in a different way.

5 Heap Spray

Now that I have a way to redirect the execution of the program I need to inject the shell code that I want to run in the client machine. So I have written a few lines of JavaScript that allocate strings in the heap of my browser.

The heap management is not predictible so I can't choose where my shellcode are going to be stored. The only thing that I can do is to store a huge amount of data in order to make it more likely, that at the desired address I have my code. The first thing to do is to prepare the string: it's almost impossible that at my desired address I find the beginning of my shellcode. So the heap spray technique suggests to append the malicious code after a huge amount of NOP instructions[6][7].

So after a few attempts I discovered that Firefox uses the low memory addresses for its own memory and uses the bigger addreses to store data that arrives from JavaScript code. As we see in figure 5 the low range of address is really fragmented and so the probability of hitting wrong data is higher. Indeed if we look in the highlighted range we can see that we have only private data and so my predictible address should be there.

After a deep analysis with the VMMap tool and having read of a lot of tutorials online, I've decided that my target should be 0x0c0c0c0c that every time is above the limit where the data of the browser ends and where we can find only our private data. Once I've taken this decision I noticed that is not so simple to build an heap spray that always comprises my desired address. Often between my memory chunks of NOP + shellcode I've empty space which is not used by the browser.

Then I've understood that my browser doesn't allocate the space for my string in a precise way but it always store chunks of memory multiples of 512K that can contains the desired string. So, for example, if my string's length is 600K, my browser allocates a chunk of 1028K with the first 600K containing my code and the remaining 400K empty but not usable by other strings. So when I've tried to spray the heap there is always a range of address after every chunk that I allocate, which are empty.

As soon as the moment I've noticed this problem, it was easy for me to balance my NOP instruction in the blocks, in order to fill up almost exactly

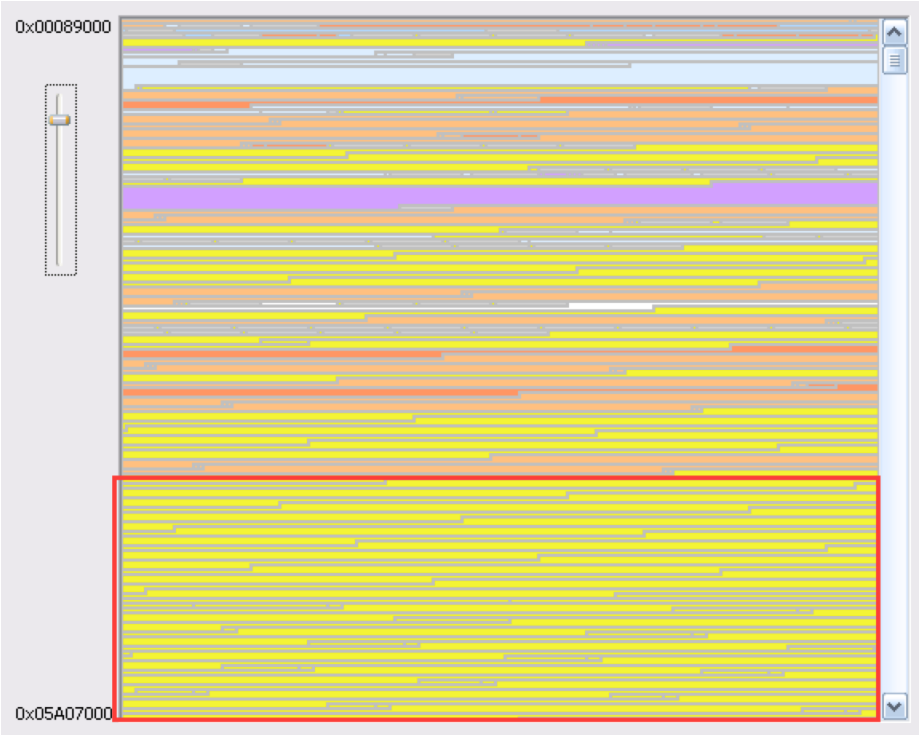

Figure 4: Yellow blocks are my chunks of data sprayed with JS. The other segments are other informations stored by the browser

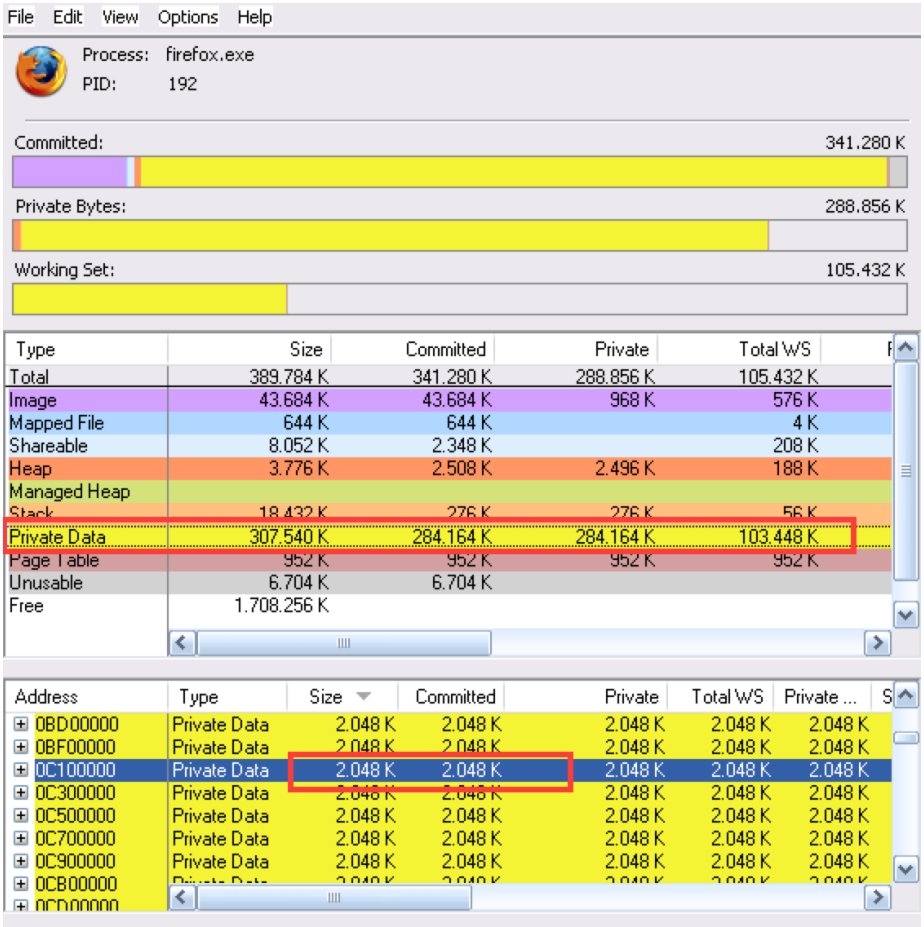

Figure 5: We can see with VMMap that I've stored 307K of private data in blocks of length 2.048K with everyone exactly 2.048K of commited data.

the size of the allocated chunk. Once I've prepared my string I decided with an heuristic metrics the number of chunks to store. After many attempt I've decided to store 58 blocks that, for my experiment results, is the number that ensures me that I always manage to fill up a range of memory that exceed my desired address, but keep the loading time of my page acceptably low.

Unfortunately, even with all this adjustments, the heap spray is a probabilistic attack and so we have no security that my desired address is filled up with my NOP instructions.

6 Attack

Now I've a GIF that overflows in the heap of firefox without any predictability of what I've overwritten. With a lot of disassembly sessions I've discovered that sometimes my value arrives in the execution flow but I've never seen that the EIP register assumes directly the value of my overflown. Everytime that the execution hits my code, this value arrives in one of the other registers, usually EAX, ECX or ESI. Then this value can be stored in the stack or used as a pointer to get the value that it contains.

For this reason I've sprayed my heap with "0c0c0c0c" that can be a pointer to my sprayed region or a kind of NOP instruction. If my programm use this string as an instructions it becomes "OR al,0xc", that for my purpose is a NOP because AL is an 8-bit register that is usually used to access to I\O port or to perform arithmetical operations. So change this register value doesn't alter the program flow but is only a void operation that evaluate an OR between two values. So with this value I have got an increased probability that this value arrives in my EIP register and when it arrives those values are used as NOP instructions to slide the program to my shellcode.

For example, in the screenshot in figure 6 the value of my overflow after a push in the stack is loaded in the ECX register that, at that point, is supposed to be the address of a memory cell. This value is the base and with an offset of "0xc" is used to store the address of a function. So my code tries to look the value in the cell 0x0c0c0c18 (0x0c0c0c0c + 0xc) where I have my NOP instruction. But my NOP instruction conicide with the address where I want to find my spray. So the function called is stored at 0x0c0c0c0c where the execution slides down the hill and gets to the shellcode that I've placed at the end of my NOP instructions.

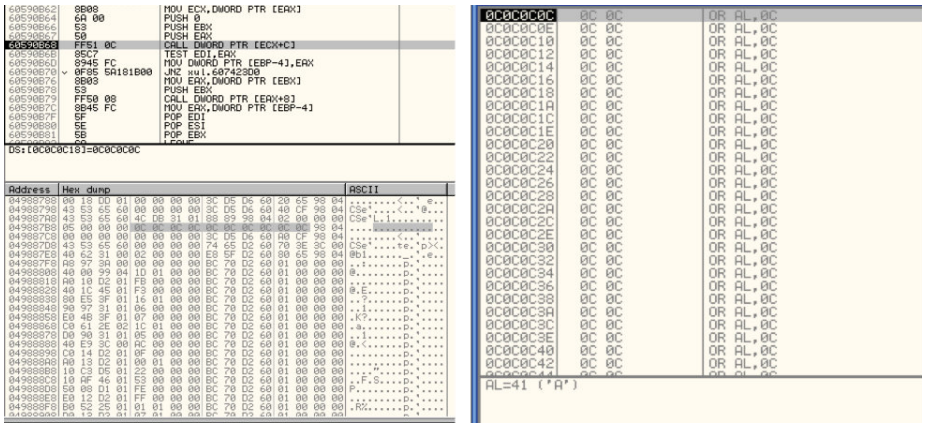

Figure 6: A call that leads to the execution of my injected shellcode

7 Reliability

At the end of my work I've found a lot of problems for the design of a working attack that exploits this vulnerability. The main one is that I can't know what I'm going to overwrite with my GIF header. Most of the time the execution doesn't arrive in my overflown region and for this reason I've found a solution: I placed in the header of my malicious webpage a metatag that reloaded my page every second, so my GIF is loaded lots of times until something appened. The problem is that this behavior is suspicious and I think that most of the users will kill my page if they see a lot of reloads.

But even if my page triggers this exploit, I don't have any security that the execution leads to my code. Often there is only a crash of the browser or a loop, because the code that I've overwritten doesn't use my address as a pointer, but for other purposes. Another reason that made me fail is that my heap spray is not predictible as I said, so there is the possibility that I have an access fail when I try to read the address 0x0c0c0c0c that may be not allocated.

As we can see in the table 7 I've made some tests and I've calculated that my exploit have a reliability of approximately 20% on my windows XP sp3 environment. As we can see we need an avarage of '8.7' reloads of the page to trigger the vulnerability and this make my execution really slow. Another important that we can read in table 7 is that 30% of the fail are caused by access violation of the memory, sign that my heap spray has failed to store my payload of NOP and shell code in the address 0x0c0c0c0c. The others 60% of fails are Firefox's crash that are caused by various problem.

At the end this is not a vulnerability that can be exploited to be used for a real working attack because this statistic is not enough to be useful. Moreover for the test I used a Windows XP sp3 environment that with default settings

Table 1: Test on Windows XP sp3 of the exploit

RunID	Success	Nr of Reload	Why it failed
#1	No	4	Access violation on reading 0x0c0c0c0c
#2	Yes	13	-
#3	No	2	My overwrite leads to a loop
#4	No	18	Access violation on reading 0x0c0c0c0c
#5	No	8	-
#6	No	11	Access violation on overwriting
#7	Yes	5	-
#8	No	9	My overwrite leads to a loop
#9	No	4	Access violation on reading 0x0c0c0c0c
#10	No	6	-
#11	No	14	Firefox freeze and crash
#12	No	16	Firefox freeze and crash
#13	Yes	7	-
#14	No	4	Access violation on reading 0x0c0c0c0c
#15	No	6	My overwrite leads to a loop
#16	No	2	Access violation on reading 0x0c0c0c0c
#17	No	8	My overwrite leads to a loop
#18	No	16	Firefox freeze and crash
#19	Yes	9	-
#20	No	12	Access violation on overwriting

has DEP (Data Execution Prevention) enabled only for software and services essential for Windows. Once I've tested my exploit I tried to activate this control on all the programs and after that the reliability of my program his fall to 0%. With active DEP the heap in whitch I have stored my shellcode is flagged as non-executable and so my exploit can't work. Another interesting case would be the test of my exploit in a system with ASLR (Address Space Layout Randomization) but this feature is implemented only from Windows Vista so in my Windows XP is not available and I can't test with this feature activated.

References

[1] NVD. Nvd report on vulnerability cve 2009-3373. http://nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-3373, 2009. [Online; last read 06-Dec-2014].

[2] Brandon Sterne. Bugzilla report on vulnerability cve 2009-3373. https://bugzilla.mozilla.org/show_bug.cgi?id=511689, 2009. [Online; last read 06-Dec-2014].

[3] Mozilla. Mozilla 1.9.2 modules. http://hg.mozilla.org/releases/mozilla-1.9.2/file/3bc177bd871f/modules/libpr0n/decoders/gif/nsGIFDecoder2.cpp, 2001. [Online; last read 06-Dec-2014].

[4] OllyDbg. Ollydbg official website. http://www.ollydbg.de, 2000. [Online; last read 06-Dec-2014].

[5] Mark Russinovich and Bryce Cogswell. Vmmap web page. http://technet.microsoft.com/en-us/sysinternals/dd535533.aspx, 2014. [Online; last read 06-Dec-2014].

[6] Corelan. Corelan exploit tutorial part 11: Heap spray. https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/, 2011. [Online; last read 06-Dec-2014].

[7] Jon Erickson. *Hacking: The Art of Exploitation, 2Nd Edition*. No Starch Press, San Francisco, CA, USA, second edition, 2008.