binamuse
security research

Adobe Reader X
BMP/RLE heap corruption

Adobe Reader X is a powerful software solution developed by Adobe Systems to view, create, manipulate, print and manage files in Portable Document Format (PDF). Since version 10 it includes the Protected Mode, a sandbox technology similar to the one in Google Chrome which improves the overall security of the product.
Adobe Reader X fails to validate the input when parsing an embedded BMP RLE encoded image. Arbitrary code execution in the context of the sandboxed process is proved possible after a malicious bmp image triggers a heap overflow.

Dec 2012

Felipe Andres Manzano
feliam@binamuse.com

Contents

1 Target summary

Title: Adobe Reader X BMP/RLE heap corruption

Product: Adobe Reader X

Version: 10.x

Product Homepage: adobe.com

Binary affected: AcroForm.api

Binary Version: 10.1.4.38

Binary MD5: 8e0fc0c6f206b84e265cc3076c4b9841

2 Vulnerability brief information

Vulnerability Class	Memory Corruption
Affected Versions	10.1.6/11.0.2 and below
Affected Platforms	Microsoft Windows OSX Linux
Reliability Rating	Complete (100%)
Configuration Requirements	Default configuration
Attack Vector	Client-Side File Format
Exploitation Impact	Code Execution
Exploitation Context	Sandbox
Patch	`ftp://ftp.adobe.com/pub/adobe/reader/`
CVE	CVE-2013-2729
Reference	`http://blog.binamuse.com/2013/05/readerbmprle.html`

3 Common Vulnerability Scoring System

Base Metrics		
Access **V**ector	**L**ocal	The vulnerability is exploitable with only local access requires the attacker to have either physical access to the vulnerable system or a local (shell) account
Access **C**omplexity	**L**ow	Specialized access conditions or extenuating circumstances do not exist
Authentication	**N**one	Authentication is not required to exploit the vulnerability.
Confidentiality Impact	**C**omplete	There is total information disclosure, resulting in all system files being revealed
Integrity Impact	**P**artial	Modification of some system files or information is possible, but the attacker does not have control over what can be modified, or the scope of what the attacker can affect is limited
Availability Impact	Partial (P)	There is reduced performance or interruptions in resource availability

Temporal Metrics		
Exploitability	**F**unctional	Functional exploit code is available. The code works in most situations where the vulnerability exists
Remediation **L**evel	**U**navailable	There is either no solution available or it is impossible to apply
Report **C**onfidence	**N**ot **D**efined (ND)	Assigning this value to the metric will not influence the score. It is a signal to the equation to skip this metric

Environmental Metrics		
Collateral **D**amage **P**otential	**M**edium-**H**igh	A successful exploit of this vulnerability may result in significant loss of revenue or productivity
Target **D**istribution	**H**igh	Between 76% - 100% of the total environment is considered at risk

4 Vulnerability Details

The issue presented here is related to the parsing of a BMP file compressed with RLE8. The bug is triggered when Adobe Reader parses a BMP RLE encoded file embedded in an interactive PDF form. The dll responsible of handling the embedded XFA interactive forms(and the BMP) is the `AcroForm.api` plugin. First we need to reach the XFA code.

4.1 PDF Forms

A PDF file can contain interactive Forms in two flavors:

- The legacy, Forms Data Format (FDF or AcroForms)
- The XML based, XML Forms Architecture (XFA)

There is support for different XFA Specifications since Acrobat 8.0 (ref. `http://blogs.adobe.com/livecycle/2011/09/compatibility-matrix-for-xfa.html`).

XFA Version	Acrobat Version
2.6	Acrobat 8.1/Acrobat 8.11
2.7	Acrobat 8.1
2.8	Acrobat 9.0, Acrobat 9 ALang features
3.0	Acrobat 9.1
3.3	Acrobat 10.0

Table 1: XFA Support

We will focus on last XFA specification available.

4.2 XFA, The XML Forms Architecture

The XML Forms Architecture (XFA) provides a template-based grammar and a set of processing rules that allow business to build interactive forms. At its simplest, a template-based grammar defines fields in which a user provides data. Among others it defines buttons, textfields, choicelists, images and a scripting API to validate the data and interact. It supports Javascript, XSLT an FormCalc as scripting language. A small XFA containing an image looks like this:

```
<template   xmlns:xfa="http://www.xfa.org/schema/xfa-template/3.1/">
   <subform name="form1" layout="tb" locale="en_US" restoreState="auto">
      <pageSet>
         <pageArea name="Page1" id="Page1">
            <contentArea x="0.25in" y="0.25in" w="576pt" h="756pt"/>
            <medium stock="default" short="612pt" long="792pt"/>
            </pageArea>
         </pageSet>
      <subform w="576pt" h="756pt">
      <field name="ImageField" >
        <ui>
            <imageEdit data="embed"/>
        </ui>
        <value>
            <image> AAAAA.. AAAAAA</image>
        </value>
      </field>
      </subform>
   </subform>
</template>
```

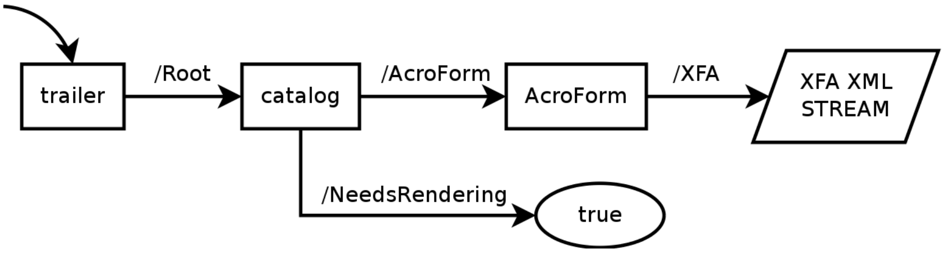
An XFA Form can be embedded in a common pdf stream and be rendered by all modern versions of Adobe Reader. The PDF catalog must contain the /NeedsRendering, /Extensions and /AcroForm fields. /AcroForm field must point to the form dictionary. Something like this..

```
3 0 obj
<< /Length 12345 >>
stream
 XFA....
endsream

2 0 obj
<< /XFA 3 0 R >>
endobj

1 0 obj
<<  /Type /Catalog
    /NeedsRendering true
    /AcroForm 2 0 R
    /Extensions <<   /ADBE <<     /BaseVersion /1.7
                                  /ExtensionLevel 3
                            >>
                >>
    ...
>>
```

Graphically a PDF containing an XFA form has this structure:



So at this point we can build a PDF containing a XFA Form containing an image. Let's see the BMP bug.

4.3 BMP - Run length encoding

The BMP can be compressed in two modes, absolute mode and RLE mode. Both modes can occur anywhere in a single bitmap. Ref. `http://www.fileformat.info/format/bmp/corion-rle8.htm`

The RLE mode is a simple RLE mechanism, the first byte contains the count, the second byte the pixel to be replicated. If the count byte is 0, the second byte is a special, like EOL or delta.

In absolute mode, the second byte contains the number of bytes to be copied literally. Each absolute run must be word-aligned that means you might have to add an additional padding byte which is not included in the count. After an absolute run, RLE compression continues.

Second byte	Meaning
0	End of line
1	End of bitmap
2	Delta. The next two bytes are the horizontal and vertical offsets from the current position to the next pixel.
3-255	Switch to absolute mode

Table 2: RLE Modes of operation

4.4 Bug pseudocode

Consider listing 1. This pseudo code is derived from the function responsible of expanding an RLE encoded BMP, found in `AcroForm.api`. The functions `feof()`, `fread()` and `malloc()` are the usual ones. The `stream` is a file from where it has already read the complete BMP header, including the height and the width. The main purpose of function is to expand the RLE encoded data. First it allocates enough memory to hold the complete image. Then it reads one byte to decide between one of the two modes: RLE or Absolute. In the RLE mode it repeats the next byte a number of times. In the Absolute mode there are more options implemented as a switch:

0. End of line, fix the xpos/ypos indexes to point to the start of the next line.

1. End of file, finish processing.

2. Delta, moves the write pointer (e.g. to skip blank regions).

d. Literal data, copies data literally from the file

Try to find the bug here:

```
char* rle(FILE* stream, unsigned height, unsigned width){
  assert(height < 4096 && height < 4096);
  char * line;
  char aux;
  unsigned count;
  struct {
    unsigned char reps;
    unsigned char value;
  }cmd;
  unsigned char xdelta, ydelta;
  unsigned xpos = 0;
  unsigned ypos = height - 1;
  char * texture = malloc(height*width); //Safe mult!
  assert(texture);
  while ( !feof(stream)) {
    fread(&cmd, 1, 2, stream);
    if ( cmd.reps ) {
      assert ( ypos < height && cmd.reps + xpos <= width );
      for(count = 0; count<cmd.reps; count++) {   //RLE Mode, repeat the
         value
          line = texture+(ypos*width);
          line[xpos++] = cmd.value;
        }
    }
    else {  // if rep is zero then value is a command
        switch(cmd.value){
            case 0:                           //End of line
                ypos -= 1;
                xpos = 0;
                break;
            case 1:                           //End of bitmap. Done!
                return texture;
            case 2:                           //Delta case, move bmp
               pointer
                read(&xdelta, 1, 1, stream); // read one byte
                read(&ydelta, 1, 1, stream); // read one byte
                xpos += xdelta;
                ypos -= ydelta;
                break;
            default:                          // literal case
                assert ( ypos < height && cmd.value + xpos <= width );
                for(count = 0;count < cmd.value; count++){
                    fread(&aux, 1, 1, stream);
                    line = texture+(width*ypos);
                    line[xpos++] = aux;
                  }
                if ( cmd.value & 1 )              // padding
                  fread(&aux, 1, 1, stream);
        }//switch(cmd.value)
    }//if (cmd.reps)
  }//while(!feof(stream))
  return texture;
}
```

Listing 1: The RLE bug

As you probably've found out, there are no asserts at the "delta" case. So we could move the destination pointers arbitrarily, even outside the limits of the texture buffer. However, there are boundary checks when you try to actually write something to the texture buffer as in the line

```
    assert ( ypos < height && cmd.reps + xpos <= width );
```

Note that this leaves a corner case in which a heap overflow condition can be triggered. Suppose we repeatedly pass delta commands advancing the xpos index. And we continue to do so without trying to write anything until xpos gets really big, for example 0xffffff00. To accomplish this, the BMP should contain $0xffffff00/0xff$ delta commands each one incrementing the xpos in $0xff$ like this:

```
    bmp += '\x00\x02\xff\x00' * ((0xffffffff-0xff) / 0xff)
```

Then after padding, we pass a literal command to actually write up to $0xff$ bytes of data directly from the file to the pointed address. But as xpos+len(payload) overflows the 32bits integer representation, the boundary assertion holds and the overflow is possible.

```
bmp += '\x00\x02'+chr(0x100-len(payload))+'\x00'
bmp += '\x00'+chr(len(payload))+payload
```

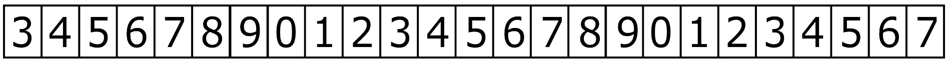
Summing up, using this bug we can **overwrite** the `with` bytes immediately before the texture buffer.

5 Exploitation detail

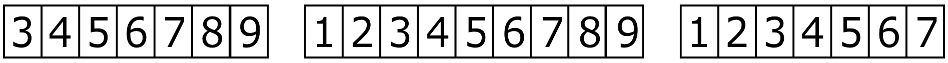
The texture is allocated in the heap using the width and height found in the BMP header. So we control the size of the overflow-able allocation and we need to choose it wisely to overwrite something useful.

But first to increase reliability it is better to prepare the heap with a sequence of allocations. We use the well known javascript method for allocating and freeing heap chunks. The exploitation script would be like this:

- allocate 1000 0x12C chunks of controlled data. Very likely triggering a LFH of size 0x12C *0x12 (18)consecutive allocations will guarantee LFH enabled for a given SIZE.



- free one every 10 chunks of the previously allocated chunks, generating several holes separated 10 chunks from each other

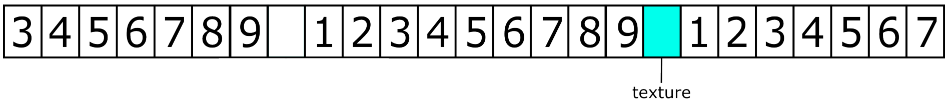


It has been found that a structure of size 0x12C bytes is used after the decoding of all images. It contains pointers to the specific vtables and functions. The goal is to read and **write** this structure from javascript.

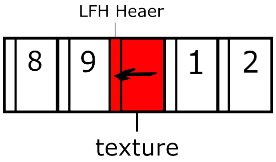
5.1 Read the struct

The first sub-goal is to be able to read the structure from javascript in order to learn the address of some dlls and bypass ASLR. To get this we'll load a broken BMP image corrupting an LFH chunk header thus trick the allocator into believing that an alive javascript string memory is free.

- Load a broken BMP with dimensions {1 , 0x12C}, its pixel texture (of size 0x12C) will be allocated in one of the prepared holes. The allocator will most likely assign one of the previously prepared holes to it.

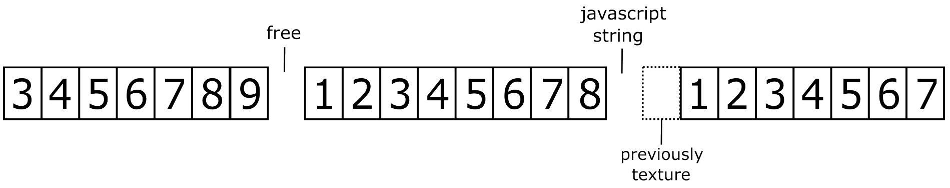


- Using the bug in the RLE parsing, overwrite and corrupt the header of the image texture chunk.

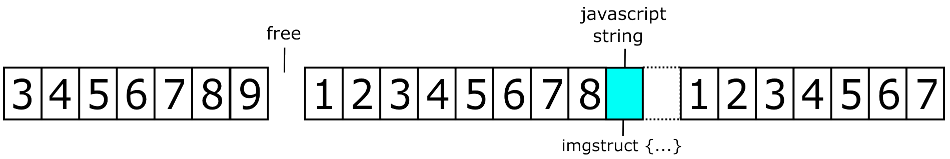


- An exception in the RLE decoder will delete all the used structures. In particular, the image texture chunk is freed. As its header is corrupted, this deletion will in fact delete the previous chunk and will leave the texture chunk alone. This wrongly deleted chunk is still used by the javascript interpreter. One of the string object leaving in the javascript interpreter still holds a pointer to the recently freed chunk.

> *If you can overflow into a chunk that will be freed, the SegmentOffset in the heap chunk header can be used to point to another valid _HEAP_ENTRY. This could lead to controlling data that was previously allocated. See `https://www.lateralsecurity.com/downloads/hawkes_ruxcon-nov-2008.pdf`*



At this point we have a javascript string using memory that is known to be free. An allocation of 0x12C will probably be assigned to the same memory overlapping the javascript string. We aim for a javascript string to share the same memory with an object containing vtables so we can learn the location of some dll from the js interpreter. As we have chosen the chunk size carefully this happens automatically and an interesting object gets allocated in the memory actually pointed by one of the javascript strings



- Now lets' iterate over all javascript strings searching for the one that has changed

```
for (i=0; i &lt; spray.size; i+=1)
   if ( spray.x[i] != null   &&
        spray.x[i][0] != "\u5858"){
   ...
   }
```

- If found, parse its contents and discover the address of AcroRd32.dll

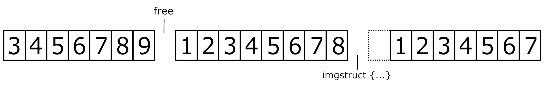
```
found = i;
acro = (( util.unpackAt(spray.x[i], 14) >> 16) - 0xa4) << 16;
break;
```

At this point we have pinpointed the exact string index that shares the memory with an imgstruct and leaked the address of AcroRd.dll to the javascript interpreter.

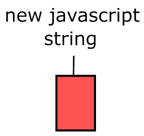
5.2 Write the struct

In javascript, strings are simply not writable. You need to free the old string and make a new copy of the string with the modifications you like. Usually, if the new string is the same size as the old one it will be allocated in the same spot. So to change the object contents we need to free the selected javascript string and realloc another in the same memory with different content.

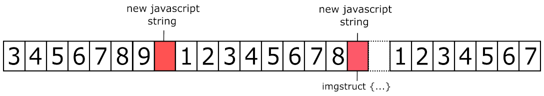
- Free the selected javascript string (which shares memory with the object)



- Build a new 0x12C length string with the desired content using the leaked addresses, and spray it a bit so it is eventually allocated over the desired object



- Allocate several new strings with the new content.



At this point the object is most likely replaced by a new one pointing to a ROP sequence.

5.3 Controlling the flow

Calling the doc.close() function from the js interpreter will trigger the unload of all loaded xfa images and the use of the overwritten vtable Thus the replaced pointers in the object are used once more in the destructors and the control flow is captured.

One last step involves to heap spray a pointer bed at a known address. A more specific technique(provided upon request) in which other heap addresses are leaked to the interpreter doesn't need this step.