Australian Government

Cyber Security Challenge Australia 2014
www.cyberchallenge.com.au

CySCA2014 Network Forensics Writeup

Background: RL Forensics Inc. has a number of network stream captures it needs to process for forensic investigations. However they have a large backlog of cases and have hired FortCerts to assist with analysis. Since both companies have a large backlog of work, FortCerts wants you to perform these forensics tasks for them.

Network Forensics 1 - Not Enough Magic

Question: You have been supplied with the following network capture with a note mentioning that the suspect has previously hidden information, although basically. Analyse the network capture to recover the flag hidden by the suspect.

Designed Solution: Players need to save all of the HTTP objects from the pcap file. They then need to run the file command across all of them to find the flag in the EXIF comment of the 9996296a1ea2320620b1e7188d4c44a2 file.

Write Up:
Lets start by downloading and opening the supplied pcap file in Wireshark and performing a quick analysis.

Looking at the Protocol Hierarchy Statistics (Statistics -> Protocol Hierarchy) we can see that all of the traffic in the pcap is HTTP over TCP.

To find out what the actual HTTP requests were, we will set the filter to ‘http’. This shows us only the re-assembled http requests and responses and not the individual packets that make them up.

Looking at the uncompressed body of the first GET request's response (Packet No 15) we can see that it contains a directory listing containing a few files. A number of these files listed in the directory listing are subsequently downloaded.

We extract the downloaded files from this pcap using Wiresharks export object feature (File -> Export Objects -> HTTP) and save all of the files.

Now that we have exported all of the HTTP objects from the pcap we will run the file command across them to get a quick idea of their contents.

```
#> cd export
#> file *
%2f:                                HTML document, ASCII text
43c20729bb03986ca09dc18974c994ec.gz: gzip compressed data, was
"43c20729bb03986ca09dc18974c994ec", from Unix, last modified: Mon Feb 24 11:26:52
2014
9996296a1ea2320620b1e7188d4c44a2: JPEG image data, JFIF standard 1.01, comment:
"CreamRainySpecify702"
a2a1132631b8a49a7dc19952af71e14f: JPEG image data, JFIF standard 1.01
bcbc19e73ac903e0159e988ce7ead809: PE32 executable (GUI) Intel 80386 (stripped to
external PDB), for MS Windows
c96f400e9130eadc212bfeb092102644: Standard MIDI data (format 1) using 21 tracks at
1/144
dbe1f3c8f3a0b2db52b7d59417891117: data
```

The output of the file command reveals the flag **CreamRainySpecify702** in the JPG comment of the file 9996296a1ea2320620b1e7188d4c44a2.

Network Forensics 2 - Notwork Forensics

Question: You have been given a network capture file of an exchange between two suspected criminals. Analyse the session and files transferred to recover the suspects flag.

Designed Solution: Players will need to extract the disk image transferred through IRC. This image is a NTFS disk that has many files, including a deleted password encrypted file. Additionally there is a password file that has been moved into Recycle Bin and overwritten with junk. The history of the file reveals that it was once in the MFT Resident Data and can be recovered.

Write Up:
We start by opening the provided pcap file in Wireshark.

Looking at the Protocol Hierarchy Statistics (Statistics -> Protocol Hierarchy) we can see the pcap contains both TCP data and IRC traffic.

Setting the filter to 'irc' and following the TCP stream reveals a conversation between two persons: badguy and otherbadguy.

```
:badguy!~root@10.0.0.103 PRIVMSG otherbadguy :The goose chased the fallen cloud.
PRIVMSG badguy :what wait? O_o
PRIVMSG badguy :did you follow my instructions?
:badguy!~root@10.0.0.103 PRIVMSG otherbadguy :i've encrypted the file
PRIVMSG badguy :what about the password?
:badguy!~root@10.0.0.103 PRIVMSG otherbadguy :i've overwritten the password file
like you said and deleted it
PRIVMSG badguy :just to be safe, send me a copy
:badguy!~root@10.0.0.103 PRIVMSG otherbadguy :one sec
:badguy!~root@10.0.0.103 PRIVMSG otherbadguy :.DCC SEND diskimage.gz 199 0 3230287
55.
PRIVMSG badguy :.DCC SEND diskimage.gz 167772264 37376 3230287 55.
PRIVMSG badguy :i'll let you know... same time tomorrow
```

Additionally, we can see that badguy used DCC to send a file called diskimage.gz to otherbadguy. The diskimage.gz file was 3230287 bytes long.

The question talked about analysing the session and files transferred so we will extract and analyse the diskimage.gz file.

We identify the stream containing diskimage.gz by using Wireshark's conversation statistics feature (Statistics -> Conversations). In the Conversations window we click the TCP tab, select the largest conversation and follow it.

Now in the follow TCP stream window we change the direction display to only display data from badguy to otherbadguy (ie. 10.0.0.103 to 10.0.0.104). The entry in the dropdown should show 3230287 bytes in the round brackets.

We ensure the raw radio button is selected and click Save As to save the diskimage.gz file.

We unzip diskimage.gz and run file to determine what type of file this is.

```
#> gzip -d diskimage.gz
#> file diskimage
diskimage: x86 boot sector; partition 1: ID=0x7, starthead 2, startsector 128,
59392 sectors, code offset 0xc0, OEM-ID "      M", Bytes/sector 190,
sectors/cluster 124, reserved sectors 191, FATs 6, root entries 185, sectors 64514
(volumes <=32 MB) , Media descriptor 0xf3, sectors/FAT 20644, heads 6, hidden
sectors 309755, sectors 2147991229 (volumes > 32 MB) , physical drive 0x7e, dos <
4.0 BootSector (0x0)
```

We can see that this is a disk image dumped to a file as you would expect from the name. Let's perform some further analysis of this disk image.

For our disk analysis we will use The Sleuth Kit.

Lets try to determine the partitions that are present on this disk image. We run mmls to list the partitions and their types.

```
#> mmls diskimage
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors

      Slot    Start        End          Length       Description
00:  Meta    0000000000   0000000000   0000000001   Primary Table (#0)
01:  -----   0000000000   0000000127   0000000128   Unallocated
02:  00:00   0000000128   0000059519   0000059392   NTFS (0x07)
03:  -----   0000059520   0000065535   0000006016   Unallocated
```

There is a NTFS partition at sector 128 i.e. 65536 bytes offset (this is typical for Windows). We calculate 65536 using the sector size in bytes times the start offset.

We next want to list the files in the NTFS partition. We use fls for this, remembering to provide the offset of the NTFS partition. (-r recursive, -p fullpath, -u undeleted files).

```
#> fls -o 128 diskimage -r -p -u
****SNIP****
r/r 0-128-1:  $MFT
****SNIP****
```

```
r/r 76-128-1:
$RECYCLE.BIN/S-1-5-21-2229788878-2747424913-2242611199-1000/$ISH2ZGB.txt
r/r 590-128-5:
$RECYCLE.BIN/S-1-5-21-2229788878-2747424913-2242611199-1000/$RSH2ZGB.txt
****SNIP****
```

We notice that the $MFT is at record 0, we also notice some "deleted" files that are in the recycle bin.

We use fls to list the deleted files on the partition and notice three files containing secret in their names.

```
#> fls -o 128 diskimage -r -p -d
****SNIP****
-/r * 589-128-1:    files/secret.7z
-/r * 591-128-1:    files/secret.db
-/r * 592-128-1:    files/secret.png
****SNIP****
```

Lets try dumping the content of the secret files and seeing what they contain. We use icat with the offset to the partition and the mft record to extract the file contents.

```
#> icat -o 128 diskimage 589 > secret.7z
#> icat -o 128 diskimage 591 > secret.db
#> icat -o 128 diskimage 592 > secret.png
```

Secret.png is a png file that contains a demotivator saying Nope. Secret.db seems to be a html fragment. They both appear to be decoys. Secret.7z requires a password which we don't have at the moment.

Without the password we can still list the files present in secret.7z and this reveals the presence of a file named secret.txt that could be interesting. Could this be the encrypted file badguy was talking about?

```
#> 7z l secret.7z
   Date      Time    Attr         Size   Compressed  Name
------------------- ----- ------------ ------------
2014-02-25 11:20:49 ....A           20           32  secret.txt
------------------- ----- ------------ ------------
                                    20           32  1 files, 0 folders
```

Not knowing the password for secret.7z we think back to the IRC conversation we read between badguy and otherbadguy. Badguy mentioned overwriting and deleting the password file, so let's refocus on files in the recycle bin.

When a file is to sent to the Recycle Bin, two files are created, $I<uid> and $R<uid>. The $I files contains metadata associated with the file, in the event that the user would like to "undelete" the file.

Lets extract the metadata file and use the rifiuti-vista tool to display the deleted files metadata.

```
#> icat -o 128 diskimage 76 > ISH2ZGB.txt
#> rifiuti-vista ISH2ZGB.txt
INDEX_FILE  DELETION_TIME   SIZE  FILE_PATH
ISH2ZGB.txt 2014-02-27 11:14:18 1008    F:\files\My Secret Passwords.txt
```

Okay, so the original filename for the deleted file was My Secret Passwords.txt. This is probably what we are looking for. Now lets dump this file, remembering that the file content is actually in RSH2ZGB.txt.

```
#> icat -o 128 diskimage 590 > RSH2ZGB.txt
#> cat RSH2ZGB.txt
Lorem ipsum dolor sit amet, consectetur
****SNIP****
```

Unfortunately, we just get a file full of Lorem Ipsum text. The badguy did say they overwrote the file before deleting it.

Fortunately, when badguy overwrote the file, they expanded the file size to over 750 bytes. If the original file was small, which a passwords text file probably is, this allows us to see a snapshot of the password file content in the MFT record's resident $DATA.

We will use get_file_info written by Willi Ballenthin will provide all the metadata associated with the MFT record for this file, including the $DATA slack. Make sure that you install all of the dependencies before using this script.

We dump the mft record to a file and display the details of mft entry with record 590.

```
#> icat -o 128 diskimage 0 > mft
#> python INDXParse/get_file_info.py mft 590
****SNIP****
Slack strings:
  ASCII strings:
       skype
       1~^bK6dA0>1rHX5j
       instagram
       M_8!58/;Wklng:h0
       phone pin
       902485
       7z file
       nRkmrtp2("u8~/ph
```

```
bank login
r0wvh1
```

In the slack strings we can see some old content of the file, this includes a password listed under the title 7z file. Now we have a password that we can use to try unzip the secret.txt file from the secret.7z file we recovered previously.

```
#> 7z x -p'nRkmrtp2("u8~/ph' secret.7z
Extracting  secret.txt

Everything is Ok
#> cat secret.txt
WhiteBelatedBlind439
```

Now that we have extracted secret.txt we cat the file and get the flag **WhiteBelatedBlind439.**

Network Forensics 3 - AYBABTU

Question: RL Forensics Inc. has supplied a network capture from one of their customers that was infected with trojan malware. The customer was able to capture a command and control session of the trojan communicating with the criminals server. They would like to know what data was stolen by criminals. Analyse the communications, determine the custom protocol and extract the stolen information to reveal the flag.

Designed Solution: Players decode the DNS tunnel data to extract the data stolen by the malicious attacker. The communications are using a custom protocol with a combination of base64 in TXT and base32 in query name. Players may notice a recurring pattern in this traffic including a zlib header. Decompressing the data will reveal some files transferred and shell commands, which eventually reveal a flag in secret_document.pdf.

Write Up:
For this challenge, we are provided with a 2MB pcap file. Opening the pcap in Wireshark reveals a series of DNS requests and responses between two IPs. The traffic contains DNS TXT requests and responses containing RR.rdata fields that look like Base64 ('=' padding). This is typical of a DNS tunnel.

Let's focus on the base64 TXT response answers initially. Using scapy to load the pcap, we can quickly extract the base64 data and do some protocol analysis.

```
>>> pkts = rdpcap('74db9d6b62579fea4525d40e6848433f-net03.pcap')
>>> from base64 import *
>>> for pkt in pkts:
>>>  if DNSRR in pkt:
>>>    data = pkt[DNSRR].rdata
>>>    data = b64decode(data[1:]) # first byte is a length byte (see txt record
rfc)
>>>    print ' '.join("{0:02x}".format(ord(c)) for c in data)
```

We run the above script in scapy and analyse the output.

```
00 00 00 02 00 00 00 00 00 00 00
00 00 00 03 00 00 00 00 00 00 00
00 00 00 04 00 00 00 00 00 00 00
00 00 00 05 00 00 00 00 00 00 00
[...]
00 00 00 31 00 00 00 1c 00 00 01 78 9c cb cd 4e c9 2c 52 48 b6 8a 89 29 cf ...
[...]
00 00 00 8b 00 00 00 be 00 00 02 78 9c 9d d2 31 0e c2 30 0c 05 d0 9d 53 78 ...
00 00 00 8b 00 00 00 be 00 01 02 2b 34 5e 02 b2 e1 17 bd 00 a6 87 42 15
[...]
00 00 01 e4 00 04 09 f1 00 00 02 78 9c ec bd 7f 60 13 55 b6 38 3e 93 4c db ...
00 00 01 e4 00 04 09 f1 00 01 02 bb ef 09 d8 7e 71 87 ad 46 bd db 76 e7 dd ...
```

```
00 00 01 e4 00 04 09 f1 00 02 02 bc cb 24 e9 c4 ac 44 29 ed 7f c0 75 eb b2 ...
00 00 01 e4 00 04 09 f1 00 03 02 2a c2 85 43 22 ac a3 93 e5 9b d8 2d 26 56 ...
00 00 01 e4 00 04 09 f1 00 04 02 5c 85 dd a4 f4 de 13 89 44 26 b4 57 6a c7 ...
00 00 01 e4 00 04 09 f1 00 05 02 e3 c6 d2 2c 11 ea 89 d5 1a 57 61 29 ff d7 ...
```

From the output of the scapy command we can see a few interesting bytes:

```
?? ?? CC CC ?? LL LL LL FF FF KK 78 9c ??+
```

CC is likely a counter or ID due to its incrementing nature
LL is the length of the data e.g. 0x1c = 28 bytes
FF is incremented when ID/counter is reoccurring i.e. fragment ID
KK is 0x01,0x02,0x03 for lengths > 0
78 9c is a header (Google says zlib compression). Each group of packets with fragment ids seem to start with these bytes.

The packet format appears to allow fragmentation, based on the reoccurring ID, some high data lengths (264689 bytes) and the second counter at the 10th byte.

Now that we have a reasonable idea of the data being transmitted in the response data, we can look at the data found in the TXT query names. We have to be aware that because queries are case insensitive, base64 will not work so it is more likely to be base32.

The format of the TXT query name seems to be {base32 data}-{16 characters}.badguy.com.

Back in scapy, we write a script to extract and base32 decode the data contained in the TXT query names.

```
>>> pkts = rdpcap('74db9d6b62579fea4525d40e6848433f-net03.pcap')
>>> from base64 import *
>>> for pkt in pkts:
>>>   if DNSRR in pkt:
>>>      data = pkt[DNS].qd.qname
>>>     data = data.split('-')[0].replace('.','')
>>>     data = b32decode(data, True) # decodes lowercase
>>>     print ' '.join("{0:02x}".format(ord(c)) for c in data)
```

The byte data format is pretty similar to the format we saw in the base64 encoding, except it contains NULL byte padding at the end of some packets. Since the DNS qname cannot contain '=' characters, the padding prevents this from occurring.

Given that the format of the client to server and server to client data is the same, we can start to unpack the data to see what's being sent. We write a python script to decompress the data (if possible) and print the result.

```
from scapy.all import *
from scapy.utils import *
from base64 import *
from struct import *
from zlib import *

def decode(data, dir):
  # unpack ?? ?? CC CC ?? LL LL LL FF FF KK 78 9c ??+
  (pid, dlen, fid, cmd) = unpack('>xxHIHB', data[:11])
  if dlen > 0 and fid == 0x00:
       try:
       data = decompress(data[11:])
       except Exception, e:
       print e
       print "%s pid=%s dlen=%s cmd=%s data=%s" % (dir, pid, dlen, cmd,
repr(data))

pkts = rdpcap('74db9d6b62579fea4525d40e6848433f-net03.pcap')
for pkt in pkts:
  if DNSRR in pkt:
       data = pkt[DNS].qd.qname
       data = data.split('-')[0].replace('.','')
       data = b32decode(data, True) # decodes lowercase
       decode(data, 'C>S')

       data = pkt[DNSRR].rdata
       data = b64decode(data[1:]) # first byte is a length byte (see txt record
rfc)
       decode(data, 'S>C')
```

A sample of the script output is below.

```
S>C pid=12 dlen=12 cmd=3 data='\x00\x00\x00d'
S>C pid=49 dlen=28 cmd=1 data='mkdir c:\\\\windows\\\\a'
C>S pid=175 dlen=8 cmd=6 data=''
S>C pid=88 dlen=20 cmd=5 data='c:\\windows\\a'
Error -5 while decompressing data: incomplete or truncated stream
S>C pid=139 dlen=190 cmd=2
data="\x00\x00\x00\x8b\x00\x00\x00\xbe\x00\x00\x02x\x9c\x9d\xd21\x0e\xc20\x0c\x05\
xd0 ..."
S>C pid=166 dlen=11 cmd=1 data='dir'
Error -5 while decompressing data: incomplete or truncated stream
C>S pid=293 dlen=198 cmd=6
data='\x00\x00\x01%\x00\x00\x00\xc6\x00\x00\x06x\x9c\x8d\xcfMk\xc2@\x10\xc6\xf1{
\xdf ...'
S>C pid=208 dlen=35 cmd=1 data='rename e76a523f1b.dat 1.bat'
****SNIP****
S>C pid=590 dlen=35 cmd=1 data='rename 31c9a36cdb.dat 2.bat'
C>S pid=2159 dlen=8 cmd=6 data=''
S>C pid=618 dlen=13 cmd=1 data='2.bat'
```

```
Error -5 while decompressing data: incomplete or truncated stream
C>S pid=2187 dlen=222 cmd=6
data='\x00\x00\x08\x8b\x00\x00\x00\xde\x00\x00\x06x\x9c\x95\x8e\xb1j\xc30\x14E\xd7
` ...'
S>C pid=643 dlen=13 cmd=4 data='o.dat'
Error -5 while decompressing data: incomplete or truncated stream
C>S pid=2210 dlen=187662 cmd=2
data='\x00\x00\x08\xa2\x00\x02\xdd\x0e\x00\x00\x02x\x9c\x00\x0e@\xf1\xbfRar!\x1a
...'
S>C pid=2712 dlen=12 cmd=3 data='\x00\x01\xd4\xc0'
```

From the script output we determine some of the command functions based on context and arguments. The commands are listed below.

```
KK = 0x01: run command
KK = 0x02: upload file
KK = 0x03: ???
KK = 0x04: download file
KK = 0x05: change path
KK = 0x06: return result
```

An interesting string worth noting in the output is **a.exe a -hpqazWSXedc567 o.dat *.pdf**. This command will create a rar file containing the specified files and encrypts the rar file with the password “qazWSXedc567”.

Now that we know we can decode data successfully we will write a script to decode data from requests and responses, combine the fragmented streams and save them to a file for later analysis.

```
from scapy.all import *
from scapy.utils import *
from base64 import *
from struct import *
from zlib import *
from hashlib import *

def decode(data, frags, dir):
  frags = frags[dir]
  # unpack ?? ?? CC CC ?? LL LL LL FF FF KK 78 9c ??+
  (pid, dlen, fid, cmd) = unpack('>xxHIHB', data[:11])
  if not pid in frags:
       frags[pid] = {}
  frags[pid][fid] = data[11:]
  data = ''.join(frags[pid].values())
  if dlen > 0 and len(data) >= dlen:
       data = decompress(data[:dlen])
       if cmd in [0x02, 0x04]:
       filename = "%s-%s.dat" % (dir, md5(data).hexdigest())
```

```
        print "%s pid=%s dlen=%s cmd=%s" % (dir, pid, dlen, cmd)
        print "Writing out file to %s..." % filename
        f = open(filename, 'wb')
        f.write(data)
        f.close()
        else:
        print "%s pid=%s dlen=%s cmd=%s data=%s" % (dir, pid, dlen, cmd,
repr(data))

pkts = rdpcap('74db9d6b62579fea4525d40e6848433f-net03.pcap')

frags = {'C2S': {}, 'S2C': {}}
for pkt in pkts:
  if DNSRR in pkt:
        data = pkt[DNS].qd.qname
        data = data.split('-')[0].replace('.','')
        data = b32decode(data, True) # decodes lowercase
        decode(data, frags, 'C2S')
        data = pkt[DNSRR].rdata
        data = b64decode(data[1:]) # first byte is a length byte (see txt record
rfc)
        decode(data, frags, 'S2C')
```

We run our script to dump the streams and get the following output.

```
#> python soln.py
S2C pid=12 dlen=12 cmd=3 data='\x00\x00\x00d'
S2C pid=49 dlen=28 cmd=1 data='mkdir c:\\\\windows\\\\a'
C2S pid=175 dlen=8 cmd=6 data=''
S2C pid=88 dlen=20 cmd=5 data='c:\\windows\\a'
S2C pid=139 dlen=190 cmd=2
Writing out file to S2C-e76a523f1bda6dd97a5a65666b34a177.dat...
S2C pid=166 dlen=11 cmd=1 data='dir'
****SNIP****
C2S pid=2187 dlen=222 cmd=6 data='      1 file(s) copied.\r\n      1 file(s)
copied.\r\n\r\nRAR 5.01   Copyright (c) 1993-2013 Alexander Roshal   1 Dec
2013\r\nTrial version            Type RAR -? for help\r\n\r\nEvaluation copy.
Please register.\r\n\r\nCreating archive o.dat\r\n\r\nAddingsecret document.pdf
       \x08\x08\x08\x08 46%\x08\x08\x08\x08\x08  OK \r\nAdding     sudo.pdf
       \x08\x08\x08\x08100%\x08\x08\x08\x08\x08  OK \r\nDone\r\n'
S2C pid=643 dlen=13 cmd=4
Writing out file to S2C-f1833826f4859ef5bf16cdd938a86a36.dat...
C2S pid=2210 dlen=187662 cmd=2
Writing out file to C2S-9452bed4ad10bfd78677061d16ca32bc.dat...
S2C pid=2712 dlen=12 cmd=3 data='\x00\x01\xd4\xc0'
```

We then run the file command over the extracted files to determine their types.

```
#>  file *.dat
```

```
C2S-9452bed4ad10bfd78677061d16ca32bc.dat: RAR archive data, v2,
C2S-ecf7d4b244845914798350fb4de2971d.dat: ASCII English text, with CRLF, LF line
terminators
S2C-070d15cd95c14784606ecaa88657551e.dat: PE32 executable (console) Intel 80386,
for MS Windows
S2C-31c9a36cdba366a7dd228161e9c8b0af.dat: DOS batch file, ASCII text
S2C-3d6af1d48c05feea9e48b66f999f4540.dat: ASCII text, with no line terminators
S2C-e76a523f1bda6dd97a5a65666b34a177.dat: DOS batch file, ASCII text
S2C-f1833826f4859ef5bf16cdd938a86a36.dat: ASCII text, with no line terminators
```

Remembering the rar creation string we noticed previously we are particularly interested in the the file C2S-9452bed4ad10bfd78677061d16ca32bc.dat. With the password used to create the rar in hand we extract two files from the rar file.

```
#> unrar -pqazWSXedc567 e C2S-9452bed4ad10bfd78677061d16ca32bc.dat
Extracting  secret document.pdf                                       OK
Extracting  sudo.pdf                                                  OK
```

Opening secret document.pdf reveals the flag for the final network forensics challenge.
HoardDirectCrumb136