Write-up of HoneyNet Scan of the Month 24 solution

Florian Buchholz

October 10, 2005

The object of the HoneyNet Scan of the Month #24 challenge is to analyze a recovered floppy from a fictitious drug case. Details can be found at the HoneyNet website (`www.honeynet.org/scans/scan24`). This write-up explains how to solve the challenge with basic Linux tools and a good understanding of the FAT12 file system. This document is similar to the one I submitted as part of the CERIAS forensics group for the actual challenge but has additional information about the tools and the file system.

On the challenge website, the image file of the floppy disk is provided. In a real investigation, the image would have to be generated first from the floppy. This can be done with the `dd` (disk dump) tool. `dd` takes an input file or device and an output file or device as parameters and copies chunks of bytes from the input to the output. There are options to specify the block size, number of blocks to copy and whether to skip bytes in the input or output devices. A floppy disk mounted under `/dev/floppy` can be imaged to the file `image` as follows:

```
dd if=/dev/floppy of=image
```

This creates a true copy of all the sectors of the floppy in the output file.

1 Initial examination

After downloading the image file, the first step is to verify the file was received correctly. The website lists an MD5 fingerprint for the zipped image:

```
image.zip MD5 = b676147f63923e1f428131d59b1d6a72 ( image.zip )
```

We can compute the MD5 hash sum of the downloaded file with the `md5sum` command, which computes the MD5 cryptographic hash value:

```
> md5sum image.zip
b676147f63923e1f428131d59b1d6a72  image.zip
```

Having verified that the downloaded file has the same fingerprint we can now unzip the image file. It is very important to always verify that the image files one works on has not been modified. Computing and comparing hash values is an accepted way of doing so. If we had imaged the floppy disk ourselves as described above, we also would have computed the MD5 hash value of the floppy disk (`md5sum /dev/floppy`)and compared that to that of the output file after the imaging.

An image file is a true copy of the floppy disk's data, but as a file it is simply a collection of bytes. Furthermore, in case we modify the file during the investigation (accidental or intentionally) we need to make a copy of the image file so as not having to generate a new image from the floppy again.

As a first step we can try looking at the file system's view of the floppy image. We could write the image back to an empty floppy with `dd` again, but fortunately Linux supports the mounting of image files directly into the directory tree. This is done with the `mount` command and a special device, the *loopback device*. We also want to specify the read-only option so that we do not modify the image file.

```
# mount -o loop,ro image mnt/
```

Note that we need to have superuser rights to perform the mount (indicated by the # in the prompt). This mounts the file `image` under the directory `mnt/` and we can access it like any other directory:

```
> ll mnt
total 17
-rwxr-xr-x  1 root root 15585 Sep 11  2002 cover page.jpgc
-rwxr-xr-x  1 root root  1000 May 24  2002 schedu~1.exe
```

It appears that there are two files contained in the image, one might be a JPG file and the other appears to be a DOS or Windows executable. To find out more about the two files, we can use the `file` command. `File` is a tool that tries to determine the type of a file by comparing it to a database of well-known headers and keywords. We can even use the tool to analyze the image file itself:

```
> file image
image: x86 boot sector, code offset 0x3c, OEM-ID "MSDOS5.0",
root entries 224, sectors 2880 (volumes <=32 MB) , sectors/FAT 9,
serial number 0xc4b1cdcf, unlabeled, FAT (12 bit)
```

This verifies that the image file is indeed a FAT12 file system and already shows us a few parameters as we will see below.

```
> file mnt/cover\ page.jpgc\\ \ \ \ \ \ \ \ \ \
mnt/cover page.jpgc           : ERROR: cannot read ‘mnt/cover page.jpgc  ’
  (Input/output error)
```

The input/output error indicates that there is something wrong with the file. Also note the spaces at the end of the "jpgc" suffix (they are escaped by the "\" character by the shell).

```
> file mnt/schedu~1.exe
mnt/schedu~1.exe: Zip archive data, at least v2.0 to extract
```

This file seems to be a Zip file, and we can try to uncompress it:

```
> unzip mnt/schedu~1.exe
Archive:  mnt/schedu~1.exe
  End-of-central-directory signature not found.  Either this file is not
  a zipfile, or it constitutes one disk of a multi-part archive.  In the
  latter case the central directory and zipfile comment will be found on
  the last disk(s) of this archive.
note:  mnt/schedu~1.exe may be a plain executable, not an archive
unzip:  cannot find zipfile directory in one of mnt/schedu~1.exe or
        mnt/schedu~1.exe.zip, and cannot find mnt/schedu~1.exe.ZIP, period.
```

Again, there is something wrong with the file. There is nothing further we can do with the mounted file system, so we need to use other means to continue the investigation.

The `strings` command prints the printable strings of a certain length of a file. The default string length is 4. Here is the `strings` output for the image file for a length of 8 (I added line-breaks for readability, indicated by "\"):

```
> strings -n 8 image
MSDOS5.0
NO NAME    FAT12   3
NTLDR
Remove disks or other media.
Disk error
Press any key to restart
IMMYJ~1DOC
COVERP~1JPG
SCHEDU~1EXE
Jimmy Jungle
626 Jungle Ave Apt 2
Jungle, NY 11111
Dude, your pot must be the best
 it made the cover of High Times Magazine! Thanks for sending me the Cover \
Page. What do you put in your soil when you plant the marijuana seeds? At \
least I know your growing it and not some guy in Columbia.
These kids, they tell me marijuana isn
t addictive, but they don
t stop buying from me. Man, I
m sure glad you told me about targeting the high school students. You must \
havesome experience. It
s like a guaranteed paycheck. Their parents give them money for lunch and \
they spend it on my stuff. I
m an entrepreneur. Am I only one you sell to? Maybe I can become distributor \
ofthe year!
I emailed you the schedule that I am using. I think it helps me cover myself \
and not be predictive.  Tell me what you think. To open it, use the same \
password that you sent me before with that file. Talk to you later.
urn:schemas-microsoft-com:office:smarttags
```

```
urn:schemas-microsoft-com:office:smarttags
urn:schemas-microsoft-com:office:smarttags
urn:schemas-microsoft-com:office:smarttags
urn:schemas-microsoft-com:office:smarttags
urn:schemas-microsoft-com:office:smarttags
PostalCode
Jimmy Jungle
Microsoft Word 10.0
Jimmy Jungle
Microsoft Word Document
MSWordDoc
Word.Document.8
 $.' ",#
(7),01444
'9=82<.342
!2222222222222222222222222222222222222222222222222
%&'()*456789:CDEFGHIJSTUVWXYZcdefghijstuvwxyz
&'()*56789:CDEFGHIJSTUVWXYZcdefghijstuvwxyz
O4p(i$TR
v:I5}61k
pw=goodtimes
Scheduled Visits.xls
\05"s}U7
Scheduled Visits.xlsPK
```

Simply running `strings` on the image file already delivered a good amount of information that we could not retrieve by mounting the image. There seems to be a Microsoft Word document present in the image, of which we can see much or even all of the text content already. Other interesting strings are `pw=goodtimes`, hinting at a password of some sort, and `Scheduled Visits.xls` and `Scheduled Visits.xlsPK`, which hint at the existence of an Excel file, possibly contained in a PK Zip archive.

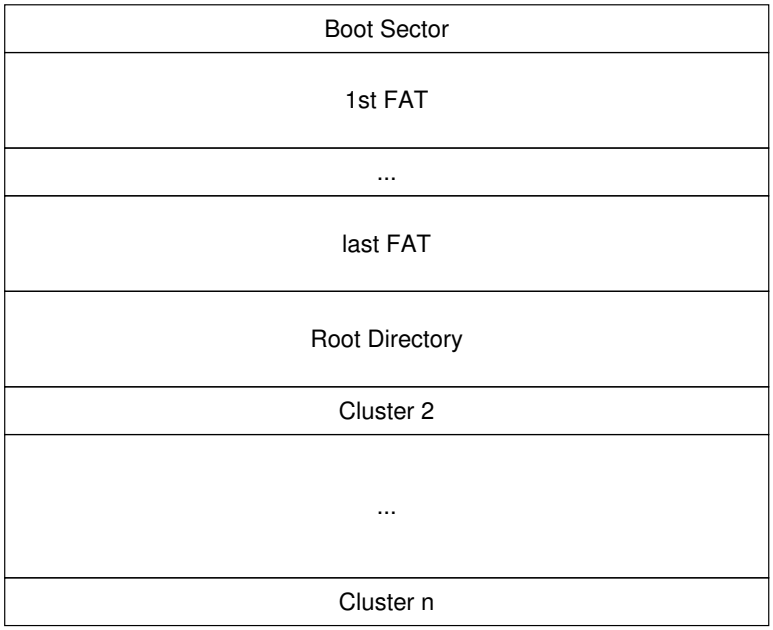
2 The FAT12 file system

Before we start a low-level analysis of the image, we need some background information on FAT12. A detailed description of the FAT on-disk structure can be found at Microsoft: `www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx`

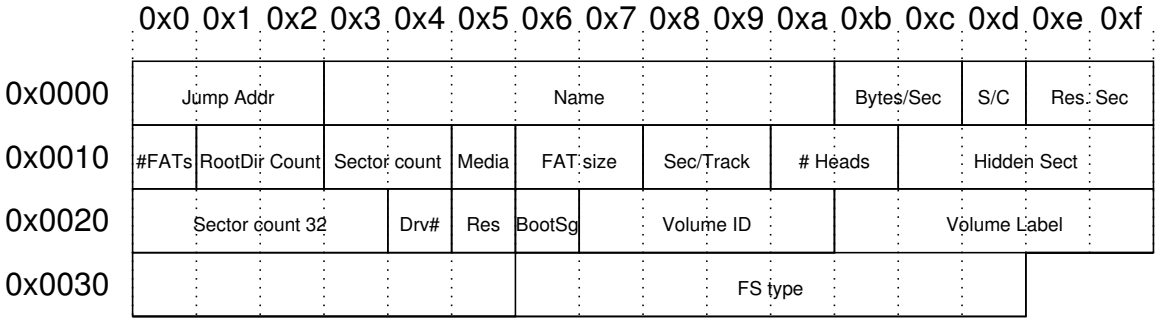
The image file can be viewed like any other file with any hex viewer/editor, such as `hexdump` or `khexedit`. The first few bytes look like this:

```
00000000  eb 3c 90 4d 53 44 4f 53 35 2e 30 00 02 01 01 00   .<.MSDOS5.0.....
00000010  02 e0 00 40 0b f0 09 00 12 00 02 00 00 00 00 00   ...@............
00000020  00 00 00 00 00 00 29 cf cd b1 c4 4e 4f 20 4e 41   ......)....NO NA
00000030  4d 45 20 20 20 20 46 41 54 31 32 20 20 20 33 c9   ME    FAT12   3.
00000040  8e d1 bc f0 7b 8e d9 b8 00 20 8e c0 fc bd 00 7c   ....{.... .....|
00000050  38 4e 24 7d 24 8b c1 99 e8 3c 01 72 1c 83 eb 3a   8N$}$....<.r...:
```

To interpret the data correctly, we need to know about the FAT12 structure. The addressing space in FAT12 is divided into four major areas: Boot sector and reserved space, FAT area, Root directory, and data area.



To make a low-level examination of the file system, we need to find out where the boundaries of the areas are as a byte offset from the start of the data. This is information that can be calculated using parameters contained in the boot sector as well as constant values that hold true for all FAT file systems. The structure of the FAT12 boot sector is as follows:




```
00000000  eb 3c 90 4d 53 44 4f 53 35 2e 30 00 02 01 01 00   .<.MSDOS5.0.....
00000010  02 e0 00 40 0b f0 09 00 12 00 02 00 00 00 00 00   ...@............
00000020  00 00 00 00 00 00 29 cf cd b1 c4 4e 4f 20 4e 41   ......)....NO NA
00000030  4d 45 20 20 20 20 46 41 54 31 32 20 20 20 33 c9   ME    FAT12   3.
00000040  8e d1 bc f0 7b 8e d9 b8 00 20 8e c0 fc bd 00 7c   ....{.... .....|
00000050  38 4e 24 7d 24 8b c1 99 e8 3c 01 72 1c 83 eb 3a   8N$}$....<.r...:
```

When interpreting the fields from the boot sector, we need to consider the byte order of the system that was used to write the data. The byte order can be *little endian* or *big*

endian, and depend on the processor that is used in the system. Suppose we have one word (4 bytes) of data that we want to store. Let's say the number is 287454020, or 0x11223344. On a little-endian system (e.g. Intel), the lowest byte (0x44) is stored first in memory, then the next lowest, and so on. Thus the on-disk storage of the word would be `44 33 22 11`. Big-endian systems (e.g. PowerPC), however store the data as humans read it, starting with the highest byte (0x11), so here the data is stored as `11 22 33 44`. FAT12 is always little-endian.

The important parameters from the file system are thus as follows:

```
Bytes per sector:      512
Sectors per cluster:     1
Reserved sectors:        1
Number of FATs:          2
Root directory count:  224 (with 32 bytes per entry)
FAT size:                9 sectors
```

For example, the count of the root directory entries is located in bytes 17 (0xe0) and 18 (0x00). Given that it is a little-endian system we need to reverse the byte order, and the count is 0x00e0 or 224.

A sector is the smallest unit that can be read from the physical disk (the floppy in this case). A cluster is the smallest unit that FAT can address. In this case, there is only one sector per cluster, but for disks with larger capacity the number will be bigger. This is because FAT12 only has a 12-bit addressing space, which means that it can only number $2^{12} = 4096$ clusters, from 0 to 4095. If a cluster were always the same as a sector, then the maximum space that can be managed would only be $4096 \times 512 = 2097152$ bytes (2MB). With a larger cluster size larger disks can be addressed. Theoretically this goes up to 510MB, because the sectors per cluster is a one byte field with a value of up to 255. However, under FAT12 the cluster size is limited to 4 sectors, yielding a limit of 8MB for a FAT12 partition (the exact number is slightly less, as the first sectors of the file system are not clusters and we start with Cluster 2).

With the above parameters, we can now determine the boundaries of the file system: there is only one reserved sector (there can be additional reserved sectors following the boot sector). Thus the FAT area starts at byte 512 (0x200) of the file system. There are two FATs (file allocation tables), and each FAT takes up 9 sectors. The root directory therefore starts at sector 19, or byte address 9728 (0x2600). There are 224 entries of 32 bytes in the root directory, which means that the directory takes up 7168 bytes (14 sectors). The data area starting with Cluster 2 starts at sector 33, or byte address 16896 (0x4200). This means that a Cluster x has sector number $31 + x$ and a byte address of $(31 + x) \times 512$.

The file allocation table (FAT) is a lookup table that tells the operating system how the clusters of one file are chained together. If a file is larger than one cluster in size (512 bytes in this case), additional clusters are allocated to contain the extra data. The directory entry (see below) for a file only contains the first cluster number. The entries in the FAT are used to look up subsequent clusters. The FAT is like a large array that contains cluster numbers. At the n-th FAT entry we can look up the cluster number that follows cluster

n in the file. For example, if we have a file of 1200 bytes we need 3 clusters to store the data. The first two contain the full 512 bytes of data, and the last one only 176 bytes (the remaining space in the cluster is unused slack space). If the clusters for the file are 2, 8, and 6 (in that order), then the directory entry for that file would point to Cluster 2, the FAT entry for Cluster 2 would point to Cluster 8, the FAT entry for Cluster 8 would point to Cluster 6, and the FAT entry for Cluster 6 would contain an end of file marker (0xfff or -1 for FAT12). A FAT entry of 0 indicates that the cluster is free for allocation.

In FAT12 each FAT entry has an addressing space of 12 bits. As this is not a power of 2 and not to waste space, two FAT entries share three bytes of data in the table. To make matters worse, the three bytes are not split in the middle but the half bytes (a single hexadecimal number) are assigned as follows: let's say the first entry of the shared duo points to Cluster 291 (0x123) and the second to Cluster 1110 (0x456). The three bytes would then contain the following:

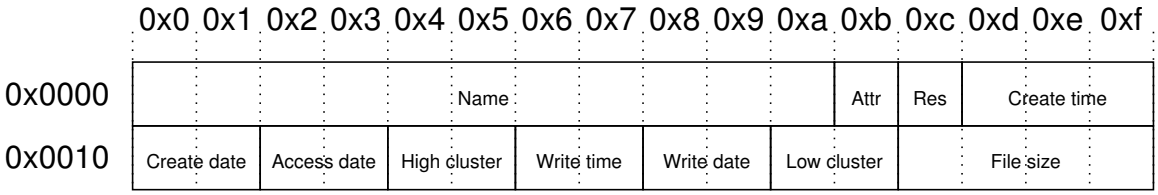
```
Entry1      Entry2        Byte1 Byte2 Byte3
 123         456    =>     23    61    45
```

Conversely, if we have three bytes 0x12, 0x34, and 0x56, then the entries are:

```
Byte1 Byte2 Byte3        Entry1      Entry2
 12    34    56    =>     412         563
```

Note that these are hexadecimal numbers, where a half-byte is exactly one digit and we can simply append high and low (half-)bytes.

The root directory area simply contains directory entries, in this case 224 of them. Each directory entry has a size of 32 bytes and has the following format:



3 Analysis

The root directory entries start at sector 19 (byte address 0x2600). When looking at the first three entries, we see the following:

```
00002600  e5 64 00 6f 00 63 00 00 00 ff ff 0f 00 bc ff ff   .d.o.c..........
00002610  ff ff ff ff ff ff ff ff ff ff 00 00 ff ff ff ff   ................
00002620  e5 4a 00 69 00 6d 00 6d 00 79 00 0f 00 bc 20 00   .J.i.m.m.y.... .
00002630  4a 00 75 00 6e 00 67 00 6c 00 00 00 65 00 2e 00   J.u.n.g.l...e...
00002640  e5 49 4d 4d 59 4a 7e 31 44 4f 43 20 00 68 38 46   .IMMYJ~1DOC .h8F
00002650  2b 2d 2b 2d 00 00 4f 75 8f 2c 02 00 00 50 00 00   +-+-..Ou.,...P..
```

Note that the first two entries do not yield any useful information when interpreting them as directory entries. This is because they are the long file name entries for the third entry. Long file name entries are a hack from Microsoft to address the need for file names that are longer than the standard DOS 8.3 notation. For this purpose, those entries are marked with the read-only, hidden, system, and volume label flags to "hide" them when listing the directory with an operating system that cannot interpret long file names. For our analysis, long file names are not relevant.

3.1 The Word file

The entry starting at byte address 0x2640 contains the directory entry for a file called ".IMMYJ .DOC". Note that the first character is a non-printable character with the byte value of 0xe5. This is a special character in FAT when appearing as the first character of a file name and it indicates that this file has been deleted. The other important fields for the reconstruction of the file are the cluster number (in FAT12 only the low cluster number, which is 16 bits in size, is used) and the file size:

```
00002640  e5 49 4d 4d 59 4a 7e 31 44 4f 43 20 00 68 38 46   .IMMYJ~1DOC .h8F
00002650  2b 2d 2b 2d 00 00 4f 75 8f 2c 02 00 00 50 00 00   +-+-..Ou.,...P..
```

The cluster number, bytes `02 00` in little endian, is 2, and the file size, bytes `00 50 00 00`, is 0x5000 (20480) bytes. When looking at the FAT entry for Cluster 2 (starting at byte 515), however, we discover, that it is 0. This is due to the fact that the file has been deleted:

```
00000200  f0 ff ff 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000210  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000220  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000230  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 2b   ...............+
```

If the file data is located in consecutive clusters, it is still possible to recover the file by copying out those clusters into a file. The FAT indicates that the next allocated cluster is Cluster 42 (FAT byte address 0x23f). The file length of 20480 bytes takes up exactly 40 clusters, so the entire file should be recoverable if the file was not fragmented (i.e. the file data lies in consecutive clusters). We can use `dd` to copy out the sectors:

```
> dd bs=512 if=image skip=33 count=40 of=jimmy.doc
40+0 records in
40+0 records out
```

The `bs` parameter specifies the block size in bytes, in this case we set one block to the size of a sector. With `skip` we skip ahead to sector 33, where Cluster 2 is located, and with `count` we copy 40 sectors/clusters.

```
> file jimmy.doc
jimmy.doc: Microsoft Office Document
```

The `file` command recognizes the file as a valid Word document, and we can view the file with an office program.

3.2 The JPG file

The next entry in the root directory looks as follows (ignoring long file name entries again):

```
000026a0  43 4f 56 45 52 50 7e 31 4a 50 47 20 00 6d 4d 46   COVERP~1JPG .mMF
000026b0  2b 2d 2b 2d 00 00 da 43 2b 2d a4 01 e1 3c 00 00   +-+-...C+-...<..
```

The file name is complete this time, indicating that the file has not been deleted. We have a starting cluster number of 0x01a4 (420) and a file size of 0x3ce1 (15585) bytes. Looking at the FAT entry for Cluster 420 (starting at byte 0x318), we again see that it is zero:

```
00000300  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000310  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000320  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000330  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000340  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000350  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

The file was not marked as deleted in the directory entry, so now we should look at Cluster 420:

```
00038600  f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6   ................
00038610  f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6   ................
00038620  f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6   ................
00038630  f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6   ................
00038640  f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6   ................
00038650  f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6 f6   ................
```

There is certainly no usable data here, and as a matter of fact, the entire remainder of the disk image from Sector 109 on is filled with 0xf6 characters. The file name extension of "JPG" indicates that the file is a JPG file. JPG files all start with a header that contains (among other things) the string "JFIF." Searching the image for this string with `khexedit` revealed that the string was found starting at byte 0x9206. This is in Cluster 42, the cluster right after the Word document. The FAT entry for Cluster 42 starts at byte 0x23f and is the following:

```
00000230  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 2b   ...............+
00000240  c0 02 2d e0 02 2f 00 03 31 20 03 33 40 03 35 60   ..-../..1 .3@.5`
00000250  03 37 80 03 39 a0 03 3b c0 03 3d e0 03 3f 00 04   .7..9..;..=..?..
00000260  41 20 04 43 40 04 45 60 04 47 80 04 ff af 04 4b   A .C@.E`.G.....K
```

The value is 43 (0x2b), and the FAT entry for Cluster 43 is 44. We can follow the FAT entry chain of consecutive clusters all the way to Cluster 72 (starting at byte 0x26c), where we find an entry of 0xfff (-1) indicating the end of the file. The file thus takes up 31 clusters (15872 bytes), which is consistent with the file length of 15585. This means that 287 bytes of Cluster 72 are slack space. Looking at the end of Cluster 72, we find:

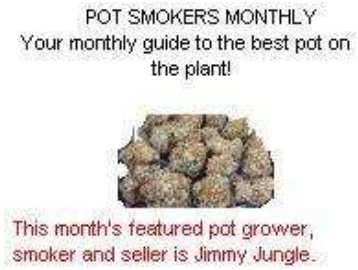
```
0000ced0  a2 8a 00 28 a2 8a 00 28 a2 8a 00 28 a2 8a 00 ff   ...(...(...(....
0000cee0  d9 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0000cef0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0000cf00  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0000cf10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0000cf20  70 77 3d 67 6f 6f 64 74 69 6d 65 73 00 00 00 00   pw=goodtimes....
0000cf30  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

The slack space was used to store data that is not part of the file, probably some sort of password. To extract the image, we can use `dd` again:

```
> dd bs=512 if=image skip=73 count=31 of=cover.jpg
31+0 records in
31+0 records out

> file cover.jpg
cover.jpg: JPEG image data, JFIF standard 1.01
```

The extracted image looks like this:



3.3 The Zip file

The last entry in the root directory is the following:

```
00002700  53 43 48 45 44 55 7e 31 45 58 45 20 00 53 53 46   SCHEDU~1EXE .SSF
00002710  2b 2d 2b 2d 00 00 90 42 b8 2c 49 00 e8 03 00 00   +-+-...B.,I.....
```

The file is not deleted, the start cluster number is 73 (0x49), and the file size is 1000 (0x03e8) bytes. Looking at the FAT entries for the file, we have:

```
00000260  41 20 04 43 40 04 45 60 04 47 80 04 ff af 04 4b   A .C@.E`.G.....K
00000270  c0 04 4d f0 ff 00 00 00 00 00 00 00 00 00 00 00   ..M.............
```

This is a consecutive chain of cluster from Cluster 73 to Cluster 77 (0x4d). The reported length of the file is 1000 bytes, but the FAT entries show that it takes up 5 clusters (up to 2560 bytes). When examining the data clusters, it also becomes clear that the data continues beyond Cluster 74. Again, we copy out the data:

```
> dd bs=512 if=image skip=104 count=5 of=schedule.exe
5+0 records in
5+0 records out

> file schedule.exe
schedule.exe: Zip archive data, at least v2.0 to extract
```

When unzipping the file with `unzip`, we are prompted for a password. When entering "goodtimes" the file is extracted:

```
> unzip schedule.exe
Archive:  schedule.exe
[schedule.exe] Scheduled Visits.xls password:
  inflating: Scheduled Visits.xls

> file Scheduled\ Visits.xls
Scheduled Visits.xls: Microsoft Office Document
```

The unzipped file "Scheduled Visits.xls" is an Excel file that contains information that can be used to answer one of the HoneyNet challenge questions.

4 Repairing the file system

The modifications that were done to each file to prevent access were effective yet simple: the Word document was deleted, the JPG file had the wrong start cluster in its directory entry (420 instead of 42), and the Zip file had an incorrect file length in its directory entry (1000 instead of 2560).

We can "repair" the file system by reconstructing the FAT entries for the Word file and putting back the proper values into the directory entries:

```
00000200  f0 ff ff 03 40 00 05 60 00 07 80 00 09 a0 00 0b   ...@..`........
00000210  c0 00 0d e0 00 0f 00 01 11 20 01 13 40 01 15 60   ......... ..@..`
00000220  01 17 80 01 19 a0 01 1b c0 01 1d e0 01 1f 00 02   ................
00000230  21 20 02 23 40 02 25 60 02 27 80 02 29 f0 ff 2b   ! .#@.%`.'..)..+

00002640  4a 49 4d 4d 59 4a 7e 31 44 4f 43 20 00 68 38 46   JIMMYJ~1DOC .h8F
00002650  2b 2d 2b 2d 00 00 4f 75 8f 2c 02 00 00 50 00 00   +-+-..Ou.,...P..

000026a0  43 4f 56 45 52 50 7e 31 4a 50 47 20 00 6d 4d 46   COVERP~1JPG .mMF
000026b0  2b 2d 2b 2d 00 00 da 43 2b 2d 2a 00 e1 3c 00 00   +-+-...C+-*./=..

00002700  53 43 48 45 44 55 7e 31 5a 49 50 20 00 53 53 46   SCHEDU~1ZIP .SSF
00002710  2b 2d 2b 2d 00 00 90 42 b8 2c 49 00 70 09 00 00   +-+-...B.,I.p...
```

The changes can be made in a hex editor such as `khexedit` and then saved. We can now mount the file system again and observe the `file` output:

```
# mount -o loop,ro repaired_image mnt/

> ll mnt/
total 38
-rwxr-xr-x  1 root root 15585 Sep 11  2002 cover page.jpgc
-rwxr-xr-x  1 root root 20480 Apr 15  2002 jimmyj~1.doc
-rwxr-xr-x  1 root root  2416 May 24  2002 schedu~1.exe

> file mnt/*
mnt/cover page.jpgc           : JPEG image data, JFIF standard 1.01
mnt/jimmyj~1.doc:               Microsoft Office Document
mnt/schedu~1.exe:               Zip archive data, at least v2.0 to extract
```