Examination of overwritten files with The Sleuth Kit

March 3, 2010
rationallyparanoid.com

Most computer users today are familiar with the concept of overwriting their files instead of deleting them in order to prevent the unwanted recovery of those files. However these same users might not be familiar with what occurs behind the scenes when a file is overwritten. In this article we will save and then overwrite a file on a USB drive, use open source forensic tools to examine the data on the drive and perform some simple data carving.

The basic process will be as follows: In order to start clean we'll wipe all data from a USB drive and format it to FAT32. We'll then save a single text file on the drive, take an image of this and perform some forensic examination. Then using Eraser we'll wipe the file on the USB drive, take another image and examine it once more. The Sleuth Kit will be part of the forensic tools used.

Part I: Setup & Examination

We begin by zeroing all data on the USB drive in Linux with the dd command. In our example the USB drive is located on /dev/sdc since /dev/sda and /dev/sdb are used by the hard drives. A block size of 4096 bytes (bs=4096) is specified simply for performance reasons to speed up this operation:

```
root@Linux:/home/user# dd if=/dev/zero bs=4096 of=/dev/sdc
dd: writing `/dev/sdc': No space left on device
978433+0 records in
978432+0 records out
4007657472 bytes (4.0 GB) copied, 246.643 s, 16.2 MB/s
```

In the operation above, if we would not have specified a block size of 4096, the operation would have taken significantly longer as shown here:

```
root@Linux:/home/user# dd if=/dev/zero of=/dev/sdc
7827457+0 records in
7827456+0 records out
4007657472 bytes (4.0 GB) copied, 830.552 s, 4.8 MB/s
```

(830 seconds versus 246 seconds when specifying a block size of 4096 bytes).

Next we use a use a partitioning utility and format the USB drive and create a FAT32 partition.

Now in Windows XP we use Notepad to create a single text file on the USB drive called FILE1.txt consisting of the following 2 lines:

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ

After saving the file, we cleanly unmount the USB drive and insert it back into a Linux computer. We create an image of the USB drive using dd and call it usb4gb.dd:

```
root@Linux:/home/user# dd if=/dev/sdc bs=4096 of=usb4gb.dd
```

```
978432+0 records in
978432+0 records out
4007657472 bytes (4.0 GB) copied, 204.737 s, 19.6 MB/s
```

(Note: for those who prefer it, the parameters **conv=notrunc,noerror,sync** could also have been added to the command above.)

We'll use mmls to see the layout of the USB image:

```
user@Linux:~$ mmls usb4gb.dd
DOS Partition Table
Offset Sector: 0
Units are in 512-byte sectors

     Slot    Start        End          Length       Description
00:  -----   0000000000   0000000000   0000000001   Primary Table (#0)
01:  -----   0000000001   0000000062   0000000062   Unallocated
02:  00:00   0000000063   0007823654   0007823592   Win95 FAT32 (0x0B)
03:  -----   0007823655   0007827455   0000003801   Unallocated
```

We see above that the FAT32 partition begins at sector 63 (to see this look at the last column in the row where it says Win95 FAT32. This is what we are interested in. Now look to the left in the start column of the row FAT32 and you'll see the value 0000000063). So for all of the following commands we will need to specify an offset of 63.

We use fls to see the file listing of the USB drive:

```
user@Linux:~$ fls -o 63 usb4gb.dd
r/r * 3:    _ILE1.txt
r/r 4:      FILE1.txt
```

Notice that there are two files, one called _ILE1.txt which is a deleted file, and the other FILE1.txt. We had not deleted any files when saving the file in Notepad.

Lets see if we can retrieve both files from the image. First we try to recover the deleted file and save it to a new file called inode3.txt:

```
user@Linux:~$ icat -o 63 -r usb4gb.dd 3 > inode3.txt
Error recovering deleted file (Starting cluster of deleted file is allocated)
```

That does not work. We now try to recover the second file which hasn't been deleted and therefore should be straightforward to recover:

```
user@Linux:~$ icat -o 63 -r usb4gb.dd 4 > inode4.txt
```

This worked. Let's take a look at the contents of the recovered file:

```
user@Linux:~$ cat inode4.txt
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

This matches the file that we created earlier in Windows. So what is the deal with the deleted file that we cannot

recover? We can look at the meta-data with istat:

```
user@Linux:~$ istat -o 63 usb4gb.dd 3
Directory Entry: 3
Not Allocated
File Attributes: File, Archive
Size: 0
Name: _ILE1.txt

Directory Entry Times:
Written:    Thu Feb 25 20:33:44 2010
Accessed:   Thu Feb 25 00:00:00 2010
Created:    Thu Feb 25 20:33:42 2010

Sectors:

Recovery:
File recovery not possible
```

This shows that no sectors are allocated for this file, and istat reports that file recovery won't be possible. Contrast this with the file that we were able to recover:

```
user@Linux:~$ istat -o 63 usb4gb.dd 4
Directory Entry: 4
Allocated
File Attributes: File, Archive
Size: 54
Name: FILE1.txt

Directory Entry Times:
Written:    Thu Feb 25 20:33:44 2010
Accessed:   Thu Feb 25 00:00:00 2010
Created:    Thu Feb 25 20:33:42 2010

Sectors:
15292 15293 15294 15295 15296 15297 15298 15299
```

Here we can see that the file size is 54 bytes, and that the file is allocated across 8 sectors (from 15292 to 15299). Why is it using 8 sectors when the file is only 54 bytes? Because for this FAT32 partition the sector size is 512 bytes, and the cluster size is 4096 bytes. So all files take up file space in 4096 byte "chunks". Why 8 sectors? Because 4096 divided by 512 is 8.

For kicks we'll run the strings command against the image to see if there is anything interesting in there:

```
user@Linux:~$ strings -tx usb4gb.dd
   7e03 mkdosfs
   7e46 K           FAT32
   7e77 This is not a bootable disk.  Please insert a bootable floppy and
   7eba press any key to try again ...
   8000 RRaA
   81e4 rrAa
   8a03 mkdosfs
   8a46 K           FAT32
   8a77 This is not a bootable disk.  Please insert a bootable floppy and
   8aba press any key to try again ...
 77e601 ILE1   TXT
```

```
 77e610 Y<Y<
 77e620 FILE1   TXT
 77e630 Y<Y<
 77f600 abcdefghijklmnopqrstuvwxyz
 77f61c ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

We can see data relevant to a file system, the file names, and the contents of our file.

Now we do the same with hexdump. Lines that are a repeat of the one above it are replaced with an asterisk, however I've only included the sections relevant to our files:

```
user@Linux:~$ hexdump -C usb4gb.dd
<snip>
0077e600  e5 49 4c 45 31 20 20 20  54 58 54 20 10 40 35 a4  |.ILE1   TXT .@5.|
0077e610  59 3c 59 3c 00 00 36 a4  59 3c 00 00 00 00 00 00  |Y<Y<..6.Y<......|
0077e620  46 49 4c 45 31 20 20 20  54 58 54 20 10 40 35 a4  |FILE1   TXT .@5.|
0077e630  59 3c 59 3c 00 00 36 a4  59 3c 03 00 36 00 00 00  |Y<Y<..6.Y<..6...|
0077e640  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
0077f600  61 62 63 64 65 66 67 68  69 6a 6b 6c 6d 6e 6f 70  |abcdefghijklmnop|
0077f610  71 72 73 74 75 76 77 78  79 7a 0d 0a 41 42 43 44  |qrstuvwxyz..ABCD|
0077f620  45 46 47 48 49 4a 4b 4c  4d 4e 4f 50 51 52 53 54  |EFGHIJKLMNOPQRST|
0077f630  55 56 57 58 59 5a 00 00  00 00 00 00 00 00 00 00  |UVWXYZ..........|
0077f640  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
eee00000
```

Now back in Windows, we use Eraser to run a 7 pass DoD overwrite of FILE1.txt, cleanly unmount the USB drive once done, and plug it back into Linux.

After this is completed, we once again acquire an image of the drive with dd and call it usb4gb_erased.dd:

```
root@Linux:/home/user/# dd if=/dev/sdc bs=4096 of=usb4gb_erased.dd
978432+0 records in
978432+0 records out
4007657472 bytes (4.0 GB) copied, 205.624 s, 19.5 MB/s
```

Lets see what files exist:

```
user@Linux:~$ fls -o 63 usb4gb_erased.dd
r/r * 3:    _ILE1.txt
r/r * 4:    _5DM5.FV1
```

_ILE1.txt is still there, but notice that FILE1.txt now has random characters replacing the original file name, and is deleted? When FILE1.txt was overwritten, Eraser also randomized the file name for confidentiality reasons.

We use istat to see the meta-data again:

```
user@Linux:~$ istat -o 63 usb4gb_erased.dd 3
Directory Entry: 3
Not Allocated
File Attributes: File, Archive
Size: 0
Name: _ILE1.txt
```

```
Directory Entry Times:
Written:    Thu Feb 25 20:33:44 2010
Accessed:   Thu Feb 25 00:00:00 2010
Created:    Thu Feb 25 20:33:42 2010

Sectors:

Recovery:
File recovery not possible
```

```
user@Linux:~$ istat -o 63 usb4gb_erased.dd 4
Directory Entry: 4
Not Allocated
File Attributes: File, Archive
Size: 0
Name: _5DM5.FV1

Directory Entry Times:
Written:    Tue Jan  1 00:00:00 1980
Accessed:   Tue Jan  1 00:00:00 1980
Created:    Tue Jan  1 00:00:00 1980

Sectors:

Recovery:
File recovery not possible
```

Nothing was changed for the original deleted file, but for our original FILE1.txt (now called _5DM5.FV1) istat reports that file recovery won't be possible. Also notice that istat is reporting the timestamps for the overwritten file as January 1 1980?

Can we confirm that the contents of the file are all gone? Lets use strings again:

```
user@Linux:~$ strings -tx usb4gb_erased.dd
   7e03 mkdosfs
   7e46 K           FAT32
   7e77 This is not a bootable disk.  Please insert a bootable floppy and
   7eba press any key to try again ...
   8000 RRaA
   81e4 rrAa
   8a03 mkdosfs
   8a46 K           FAT32
   8a77 This is not a bootable disk.  Please insert a bootable floppy and
   8aba press any key to try again ...
 77e601 ILE1   TXT
 77e610 Y<Y<
 77e621 5DM5   FV1
 77f690 cd]g
 77f6e4 gKjI _
 77f79d GP|KD
 77f8b5 D_x47
 77f9ba CgMbN
 77f9c7 {K2}
 77fa5e \5kg
```

```
 77fa8d L>"`
 77fb05 V=Zs9
 77fb18 n:!0
 77fb4f .vf@YG
 77fbac     )m1
 77fbcb H+U0,
 77fbfe F'VF
 77fc10 >wLQ
 77fc19 E/nk%
 77fce2 9L_?
 77fcfe %hPf^fJ   lL
 77fd3c G+H\
 77fdb5 ;K$o7~
 77fe33 Um?A
 77fe72 _;9I
 77fe90 \P<G{
 77fef4 14(D
 77ff53 4##Z
 77ffc6 gIj|H
 780012 Lf-R
 78002b n1Lv
 780079 _02N
 780098 i N)>     ku
 78014e *HQg
 7801a9 YE{D
 7801ef c&IX
 78020b k+M0
 780232 hu2&
 78023e 6C+h4rCo
 780264 >u9'
 780285 0OWl
 780309 j/b|
 780332  =E*
 780348 H<u5
 78037e eg'B
 7803fc ZO\S9
 780437 )Dbl@{
 780450 &U(.
 780499 Oijs/
 7804ce RIC6
 7804ea ^z<J
 780508 {UBm
 78050d ^gcS7
 7805a7 V%T
user@Linux:~$
```

This time we see what appears to be random data. This is due to the specifics of the US DoD 5220.22-M(ECE) 7 pass overwrite method which uses random data for the last pass. In the table below (taken from Eraser version 5.8.8) X, Y, Z are values from 0 to 255:

Pass	**Matrix**	**Pattern**
1	E [1]	Random character X
2	E [2]	Bit-wise complement of X
3	E [3]	Random data
4	C	Random character Y
5	E [1]	Random character Z
6	E [2]	Bit-wise complement of Z
7	E [3]	Random data

To illustrate how the data is overwritten:

```
user@Linux:~$ hexdump -C usb4gb_erased.dd
<snip>
0077e600  e5 49 4c 45 31 20 20 20  54 58 54 20 10 40 35 a4  |.ILE1   TXT .@5.|
0077e610  59 3c 59 3c 00 00 36 a4  59 3c 00 00 00 00 00 00  |Y<Y<..6.Y<......|
0077e620  e5 35 44 4d 35 20 20 20  46 56 31 20 00 00 00 00  |.5DM5   FV1 ....|
0077e630  21 00 21 00 00 00 00 00  21 00 00 00 00 00 00 00  |!.!.....!.......|
0077e640  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
0077f600  18 cd b3 57 63 04 67 7f  3b 56 17 4d 2f d3 e5 54  |...Wc.g.;V.M/..T|
0077f610  70 0a d3 eb 53 77 ac 39  9f 9c 97 8f 91 b6 ea a4  |p...Sw.9........|
0077f620  c3 6b 7a c8 6b 60 e0 67  da 35 d1 0c 86 61 e1 51  |.kz.k`.g.5...a.Q|
0077f630  af 75 bd 1d f3 4d ab 5c  8c 7e 22 32 7f 50 4f b3  |.u...M.\.~"2.PO.|
0077f640  57 47 61 d1 ae eb e8 15  f2 95 7e 1b ee ff c2 3d  |WGa.......~....=|
0077f650  8d 2d 83 0a 2f d3 0f 89  f2 b9 fc 75 bc 6e a7 37  |.-../......u.n.7|
0077f660  af 95 c7 27 f0 b1 1b 7e  31 cc 21 fe cf a0 47 83  |...'...~1.!...G.|
0077f670  c3 60 67 bd 01 40 e8 1a  03 f5 c0 80 a4 bd 46 c9  |.`g..@........F.|
0077f680  de 7c db c1 01 88 2f ae  99 b9 6c 95 26 fa a7 07  |.|..../...l.&...|
<snip>
00780570  fc f6 55 a4 82 30 86 5d  93 4a 2f b2 5b a9 ad ba  |..U..0.].J/.[...|
00780580  c0 93 86 28 01 72 3b 38  e3 8e 06 3c 16 f4 d0 97  |...(.r;8...<....|
00780590  33 c7 75 38 f2 8a ff 68  12 f0 00 fe 02 b5 24 bf  |3.u8...h......$.|
007805a0  48 b5 a3 15 33 53 e3 56  25 54 09 b9 04 2c 22 b7  |H...3S.V%T...,".|
007805b0  39 85 d0 47 f0 7c b1 09  39 04 6a f6 7e d9 4b d9  |9..G.|..9.j.~.K.|
007805c0  70 12 fa 6d 0d 45 99 bf  7c 35 56 a0 47 b9 6b 8d  |p..m.E..|5V.G.k.|
007805d0  fc 91 9a 8c 04 74 6c c7  94 50 e1 88 12 17 8c 46  |.....tl..P.....F|
007805e0  15 cd 08 e9 74 c0 d1 85  c0 e1 02 6b a1 31 f3 ed  |....t......k.1..|
007805f0  84 52 25 13 57 38 6e b7  36 ba e6 99 16 fd e0 57  |.R%.W8n.6......W|
00780600  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
eee00000
```

If we do the math using a hex calculator to calculate the difference between the address where the random data ends (780600) to where it begins (77F600) we see that 780600-77F600 = 1000 (in hex), which in decimal is equal to 4096. In other words, not only has the 54 bytes of data in the file (i.e. the abcd...WXYZ) been overwritten and replaced with random data, but all the file slack for FILE1.txt as well. If we had instructed Eraser not to erase the file slack (which it calls “Cluster Tips”), only the first 512 bytes (the sector size) would have been overwritten (from 77F600 to 77F800) as shown below:

```
0077e600  e5 49 4c 45 31 20 20 20  54 58 54 20 10 40 35 a4  |.ILE1   TXT .@5.|
0077e610  59 3c 59 3c 00 00 36 a4  59 3c 00 00 00 00 00 00  |Y<Y<..6.Y<......|
0077e620  e5 35 44 4d 35 20 20 20  46 56 31 20 00 00 00 00  |.5DM5   FV1 ....|
0077e630  21 00 21 00 00 00 00 00  21 00 00 00 00 00 00 00  |!.!.....!.......|
0077e640  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
0077f600  aa 22 04 7c 9b ea a1 7d  00 c8 89 58 a1 1d f6 6b  |."|...}...X...k|
0077f610  73 fe c0 a4 fe ee fc 7a  91 4a d0 13 84 34 16 e0  |s......z.J...4..|
0077f620  68 c1 bd 2e ce 6a be 1e  bc 2a 70 d9 9a 9a 8b 0c  |h....j...*p.....|
0077f630  ce d0 9d 6e dc 2b 93 27  11 a0 1e 57 38 c1 3a 25  |...n.+.'...W8.:%|
0077f640  90 84 fa a9 76 32 4c e6  c5 e3 5b 38 94 1d d6 b5  |....v2L...[8....|
0077f650  ab db 16 8b eb 7e b6 dd  55 b6 d4 9b 58 6b 4e 4a  |.....~..U...XkNJ|
0077f660  2a 4b 59 82 19 42 9d 58  59 5e 45 8b e4 89 fd a8  |*KY..B.XY^E.....|
0077f670  7e 71 4b 5d 48 18 b1 51  a4 18 0c 40 2c 4f a3 82  |~qK]H..Q...@,O..|
0077f680  d0 80 36 0e 50 7d c0 93  dd 87 57 05 b6 eb 54 1a  |..6.P}....W...T.|
<snip>
```

```
0077f780  84 c4 22 c4 4d 4c 5f bb  c8 7f fd 39 3b c5 1b d3  |.."".ML_....9;...|
0077f790  75 5b b7 81 56 87 4b 2f  11 13 a8 b3 27 dc 3f 72  |u[..V.K/....'.?r|
0077f7a0  ac ef 5e 8a 17 69 09 ea  33 15 6e 75 52 70 77 40  |..^..i..3.nuRpw@|
0077f7b0  0a 72 92 8b 2b c3 e7 84  ef 7b 76 48 b9 d0 e3 e0  |.r..+....{vH....|
0077f7c0  1c 72 8d 58 fd 9a c5 95  35 63 eb aa 06 99 90 6d  |.r.X....5c.....m|
0077f7d0  38 9a a5 d0 1e 39 d4 77  df dc b6 7b 12 7e 6c 8a  |8....9.w...{.~l.|
0077f7e0  67 0b 8e 14 80 d2 12 a8  67 93 57 6c e0 8a 9e 57  |g.......g.Wl...W|
0077f7f0  4b 3d ca 36 0b 27 18 12  39 80 b3 d3 3d 67 30 17  |K=.6.'..9...=g0.|
0077f800  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
eee00000
```

If there is one lesson that I would like to stress is that wiping individual files should always be seen as a significantly weaker method for sanitizing your data. Given the peculiarities of each file system, operating system, software application, and the media in question, your best option is always to wipe the entire media clean than to try to guess what may or may not still reside on a media when you wipe only a single file. For example have you ever noticed how Microsoft Word creates multiple temporary versions of a document that you are editing? In their research paper, Marcel Breeuwsma et al. noted that *flash file systems often contain different versions of the same data objects because flash memory can't be erased in small quantities. Especially for small objects (much smaller than one flash block) with a high update frequency, a lot of old versions might exist outside of the normal high level file system.*[1]

As the final part of the experiment we will perform some data carving against the first image.

Part II: Data Carving

We begin by using fsstat to confirm the sector and cluster size (512 and 4096 bytes respectively) of the image:

```
user@Linux:~$ fsstat -o 63 usb4gb.dd
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: FAT32

OEM Name: mkdosfs
Volume ID: 0x4b871ef2
Volume Label (Boot Sector):
Volume Label (Root Directory):
File System Type Label: FAT32
Next Free Sector (FS Info): 15300
Free Sector Count (FS Info): 7808288

Sectors before file system: 0

File System Layout (in sectors)
Total Range: 0 - 7823591
* Reserved: 0 - 31
** Boot Sector: 0
** FS Info Sector: 1
** Backup Boot Sector: 6
* FAT 0: 32 - 7657
* FAT 1: 7658 - 15283
* Data Area: 15284 - 7823591
** Cluster Area: 15284 - 7823587
*** Root Directory: 15284 - 15291
** Non-clustered: 7823588 - 7823591
```

1 http://www.ssddfj.org/papers/SSDDFJ_V1_1_Breeuwsma_et_al.pdf

```
METADATA INFORMATION
------------------------------------------
Range: 2 - 124932930
Root Directory: 2

CONTENT INFORMATION
------------------------------------------
Sector Size: 512
Cluster Size: 4096
Total Cluster Range: 2 - 976039

FAT CONTENTS (in sectors)
------------------------------------------
15284-15291 (8) -> EOF
15292-15299 (8) -> EOF
```

We then use fls to learn what files exist:

```
user@Linux:~$ fls -o 63 usb4gb.dd
r/r * 3:    _ILE1.txt
r/r 4:      FILE1.txt
```

Next we use istat to see what sectors are allocated for FILE1.txt:

```
user@Linux:~$ istat -o 63 usb4gb.dd 4
Directory Entry: 4
Allocated
File Attributes: File, Archive
Size: 54
Name: FILE1.txt

Directory Entry Times:
Written:    Thu Feb 25 20:33:44 2010
Accessed:   Thu Feb 25 00:00:00 2010
Created:    Thu Feb 25 20:33:42 2010

Sectors:
15292 15293 15294 15295 15296 15297 15298 15299
```

Here is where we need to do some math. The file FILE1.txt begins at sector 15292, however if we simply try to carve out sectors 15292 to 15299 we won't get the file that we wanted, because we aren't factoring in the fact that the Win95 FAT32 partition begins at sector 63. So we need to add 63 to 15292 to pinpoint the first sector in which the file resides within the whole image.

This calculation can be done at the command line as follows:

```
user@Linux:~$ echo "15292+63" | bc
15355
```

Now we will use dd to extract the first sector of this file, and output the display to xxd as a quick way to confirm that we have the right file. In the command below the parameters bs=512 specifies the block size (512 bytes, which we confirmed with fsstat) and skip=15355 tells dd to skip to sector 15355 as its starting point. count=1 specifies dd to extract only one block (512 bytes) of data:

```
user@Linux:~$ dd if=usb4gb.dd bs=512 skip=15355 count=1 | xxd
1+0 records in
1+0 records out
512 bytes (512 B) copied, 2.8346e-05 s, 18.1 MB/s
0000000: 6162 6364 6566 6768 696a 6b6c 6d6e 6f70  abcdefghijklmnop
0000010: 7172 7374 7576 7778 797a 0d0a 4142 4344  qrstuvwxyz..ABCD
0000020: 4546 4748 494a 4b4c 4d4e 4f50 5152 5354  EFGHIJKLMNOPQRST
0000030: 5556 5758 595a 0000 0000 0000 0000 0000  UVWXYZ..........
0000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000090: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000100: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000110: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000120: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000130: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000140: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000150: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000160: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000170: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000180: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000190: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00001a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00001b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00001c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00001d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00001e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00001f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

We can confirm that we have the file. However in the example above we actually have slightly more than we want, as we have managed to acquire not only the text (abcd ... WXYZ) but also 458 bytes worth of "zeroes" at the end, which is part of the file slack. If we wanted to cleanly extract this file, we would need to instruct dd to grab only 54 bytes out of this sector. How can we do this? A quick and dirty way would be to take the output of the original dd command and pipe it to another dd command that you instruct to extract only 54 bytes:

```
user@Linux:~$ dd if=usb4gb.dd bs=512 skip=15355 count=1 | dd bs=1 count=54 | xxd
1+0 records in
1+0 records out
512 bytes (512 B) copied, 2.5068e-05 s, 20.4 MB/s
54+0 records in
54+0 records out
54 bytes (54 B) copied, 0.000880428 s, 61.3 kB/s
0000000: 6162 6364 6566 6768 696a 6b6c 6d6e 6f70  abcdefghijklmnop
0000010: 7172 7374 7576 7778 797a 0d0a 4142 4344  qrstuvwxyz..ABCD
0000020: 4546 4748 494a 4b4c 4d4e 4f50 5152 5354  EFGHIJKLMNOPQRST
0000030: 5556 5758 595a                           UVWXYZ
```

However the more "educational" way (in order to understand what is taking place) would be the following: Begin by calculating at what byte within the image the contents of FILE1.txt begins. So we multiply 512 bytes by the sector number (15355) where FILE1.txt begins:

```
user@Linux:~$ echo "512*15355" | bc
7861760
```

Now we use dd, specify a block size of 1 (bs=1) tell it to skip 7861760 bytes (skip=7861760), and to extract 54 blocks (count=54). Remember, in this case we specified a block size of 1, so dd will extract exactly 54 bytes:

```
user@Linux:~$ dd if=usb4gb.dd bs=1 skip=7861760 count=54 | xxd
54+0 records in
54+0 records out
54 bytes (54 B) copied, 0.000108779 s, 496 kB/s
0000000: 6162 6364 6566 6768 696a 6b6c 6d6e 6f70  abcdefghijklmnop
0000010: 7172 7374 7576 7778 797a 0d0a 4142 4344  qrstuvwxyz..ABCD
0000020: 4546 4748 494a 4b4c 4d4e 4f50 5152 5354  EFGHIJKLMNOPQRST
0000030: 5556 5758 595a                           UVWXYZ
```

To output the results into a new file instead of on-screen, you would simply specify the parameter **of=<filename>** instead of piping to xxd:

```
user@Linux:~$ dd if=usb4gb.dd bs=1 skip=7861760 count=54 of=FILE1.txt
54+0 records in
54+0 records out
54 bytes (54 B) copied, 0.0044641 s, 12.1 kB/s

user@Linux:~$ cat FILE1.txt
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

To see a list of various Sleuth Kit commands along with examples of how they are used, see the guide at http://rationallyparanoid.com/articles/sleuth-kit.html