



Published on Linux Journal (<http://www.linuxjournal.com>)

How a Corrupted USB Drive Was Saved by GNU/Linux

By Collin Park

Created 2005-06-14 01:00

My friend's brother had a 512MB Lexar Media Jumpdrive Pro USB drive that became corrupted after using it with Windows 2000. His IT department was able to get back some but not all of the file contents, but without any file names. On his own, he tried some recovery utilities, but all failed. Using a typical Linux distro--in this case SuSE 8.0--however, it wasn't hard to recover almost all of the data from the drive along with the filenames and to burn a CD-ROM of the contents.

USB Drive Ruined by Windows

Here's what I heard about the data loss:

```
Date: Sun,  1 Aug 2004 17:06:03 -0700
Subject: USB
```

```
... My USB drive is a
Lexar Media USB Jumpdrive Pro 2.0 (512 MB). I was working
on it in a computer with Windows 2000 and logged off before
ejecting the drive. Next time when I tried to use it,
it showed up as a Removable drive rather than the usual
Lexar Media drive and when I tried to open it, it said the
drive was not formatted; and under Properties, 0 bytes free
and used space and file system "RAW"
```

```
According to Lexar tech support, there is a bug with
Windows 2000 (that MS never bothered to fix) and can corrupt
the drive when it is removed without proper eject. They
recommend EasyRecovery Pro for data recovery which did
allow me to recover some files (> 500) using their RAW data
recovery program (all other tool failed because usually
said "no recognizable file on disc"). Unfortunately,
all the file names are lost and some files are gone.
```

The big questions was "can Linux read the drive?" A Web search of "linux usb jumpdrive pro" gave me hope that my kernel, 2.4.18 on SuSE 8.0, would recognize the drive in question. So, as root, I typed:

```
# tail -f /var/log/messages
```

and plugged the drive into a USB socket. Here's what appeared; I removed "Aug 5 01:32:15 linux kernel:" from each line below):

```

usb.c: registered new driver usb-storage
scsi0 : SCSI emulation for USB Mass Storage devices
usb-uhci.c: interrupt, status 3, frame# 1313
  Vendor: LEXAR      Model: JUMPDRIVE PRO      Rev: 0
  Type:   Direct-Access          ANSI SCSI revision: 02
Attached scsi removable disk sda at scsi0, channel 0, id 0, lun 0
SCSI device sda: 1001952 512-byte hdwr sectors (513 MB)
sda: Write Protect is off
  sda: sda1
WARNING: USB Mass Storage data integrity not assured
USB Mass Storage device found at 4
USB Mass Storage support registered.

```

Encouraged by that report, I tried this:

```
# dd if=/dev/sda of=/tmp/r1 bs=512
```

which reported that 1,001,952 blocks had been transferred. I then unplugged the drive and did the rest of my work using the image stored in /dev/sda.

Condition of the Boot Sector

The master boot record, which is the boot sector for the entire drive and its first sector, has a partition table, as well as other interesting things:

```

# od -Ax -tx1 /tmp/r1 | less
...
*
0001b0 00 00 00 00 00 00 00 00 48 04 07 c9 00 00 80 01
0001c0 01 00 06 0f ff e0 3f 00 00 00 b1 45 0f 00 00 00
0001d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
0001f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa

```

The boot sector has a reasonable-looking partition table with one entry. It began at offset 0x1be, the two bytes 80 01. Your favorite search engine can give you other information about the partition table, but I note two things here. First, the entry has an LBA32 format--starting logical sector 0x3f, length 0xf45b1. Now, 0xf45b1 is 1000881 decimal. That plus 63 (0x3f) is 1000944. The difference between the 1001952 and this 1000944 is 1008, that is, 63*16. I guess this has something to do with cylinder boundaries. The second thing of note is the byte at 0x1c2, with value 06; this is the partition type. What does 06 mean?

Typing `fdisk /dev/hda` as root and giving the command `l` to list, shows that type 6 is:

| | | | | | | | |
|---|------------|----|--------------|-------|-----------------|----|-----------------|
| 0 | Empty | 1c | Hidden Win95 | FA 65 | Novell Netware | bb | Boot Wizard hid |
| 1 | FAT12 | 1e | Hidden Win95 | FA 70 | DiskSecure Mult | c1 | DRDOS/sec (FAT- |
| 2 | XENIX root | 24 | NEC DOS | 75 | PC/IX | c4 | DRDOS/sec (FAT- |
| 3 | XENIX usr | 39 | Plan 9 | 80 | Old Minix | c6 | DRDOS/sec (FAT- |

```

4 FAT16 <32M      3c PartitionMagic  81 Minix / old Lin c7 Syrinx
5 Extended        40 Venix 80286     82 Linux swap      da Non-FS data
6 FAT16           41 PPC PReP Boot  83 Linux           db CP/M / CTOS / .
...

```

So, it's FAT16.

Now, if I had been watching carefully, I would have known from the line `sda: sda1` in `/var/log/messages` that the partition table was okay and contained only one entry.

Finding the FATs

When I actually started looking, however, I wasn't really sure if this was a FAT16 vs FAT12. The drive's capacity of 512MB suggested it could be either FAT16 or FAT32. I also somehow had the impression that the partition could have contained a FAT32 filesystem in the same partition type. As I continued to look through the filesystem, I noticed this:

```

# od -Ax -w8 -tx1 -tc /tmp/r1 | less

045400 4c 45 58 41 52 20 4d 45 L   E   X   A   R           M   E
045408 44 49 41 28 00 00 00 00 D   I   A   (   \0   \0   \0   \0
045410 00 00 00 00 00 00 4b 5a \0   \0   \0   \0   \0   \0   K   Z
045418 33 2b 00 00 00 00 00 00 3   +   \0   \0   \0   \0   \0   \0
045420 41 52 00 53 00 54 00 55 A   R   \0   S   \0   T   \0
045428 00 4c 00 0f 00 9a 6f 00 \0   L   \0 017   \0 232   o   \0
045430 67 00 2e 00 78 00 6c 00 g   \0   .   \0   x   \0   1   \0
045438 73 00 00 00 00 00 ff ff s   \0   \0   \0   \0   \0
045440 52 4e 41 4c 4f 47 7e 31 R   S   T   L   O   G   ~   1
045448 58 4c 53 20 00 b8 03 61 X   L   S           \0   003   a
045450 50 30 e4 30 00 00 ca 74 P   0           0   \0   \0           t
045458 4b 30 f2 6a 00 3e 00 00 K   0           j   \0   >   \0   \0
...

```

On a side note, I recently discovered the hard way that `CMD | less` doesn't do what you want it to if the output of `CMD` is too long. In this case it was okay to use, but it isn't always; this probably is system-dependent. If you have enough space on your hard drive, it may pay to do something like this:

```
# od -Ax -w8 -tx1 -tc /tmp/r1 > /tmp/r2; less r2
```

or

```
# hexdump -C /tmp/r1 > /tmp/r2; less r2
```

So this looks like the start of a directory. Immediately above that area, though, I saw this:

```
042420 00 00 00 00 00 00 14 dd 15 dd 16 dd 17 dd 18 dd
```

```

042430 19 dd 1a dd 1b dd 1c dd 1d dd 1e dd 1f dd 20 dd
042440 21 dd 22 dd 23 dd 24 dd 25 dd 26 dd 27 dd 28 dd
042450 29 dd 2a dd 2b dd 2c dd 2d dd 2e dd 2f dd 30 dd
042460 31 dd 32 dd 33 dd 34 dd 35 dd 36 dd ff ff 00 00
042470 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
```

That looked like an allocation chain with 16-bit entries. If these had taken the form 31 dd 00 00 32 dd 00 00 rather than 31 dd 32 dd, I might have thought I was looking at FAT32.

I had heard somewhere that typically two FATs can be found together, one right after the other. I told less(1) to find another line resembling the line at 0x42460, by typing ?31 dd 32 dd 33 dd. In response, less(1) showed me this:

```

023a20 00 00 00 00 00 00 14 dd 15 dd 16 dd 17 dd 18 dd
023a30 19 dd 1a dd 1b dd 1c dd 1d dd 1e dd 1f dd 20 dd
023a40 21 dd 22 dd 23 dd 24 dd 25 dd 26 dd 27 dd 28 dd
023a50 29 dd 2a dd 2b dd 2c dd 2d dd 2e dd 2f dd 30 dd
023a60 31 dd 32 dd 33 dd 34 dd 35 dd 36 dd ff ff 00 00
023a70 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
026a00 f8 ff ff ff 03 00 e6 02 21 03 a0 03 15 03 91 03
026a10 ff ff 0a 00 0b 00 ff ff 0d 00 0e 00 0f 00 10 00
```

The data at 0x42460 and at 0x23a60 are the same; this told me that the offset between tables was:

$$0x42460 - 0x23a60 = 0x1ea00$$

because 0x26a00 is the start of FAT#2. Therefore, the start of FAT#1 should be at

$$0x26a00 - 0x1ea00 = 0x08000$$

But when I looked there, I saw this instead:

```

007c00 ff ff ff ff ff ff ff ff ff ff ff ff ff ff
*
012400 01 52 02 52 03 52 04 52 05 52 06 52 07 52 08 52
012410 09 52 0a 52 0b 52 0c 52 0d 52 0e 52 0f 52 10 52
012420 11 52 12 52 13 52 14 52 15 52 16 52 17 52 18 52
```

Somebody had written a whole mess of 0xff bytes. I guess this was part of the corruption.

At this point, 0x12400 looked okay, but was it? What's in the corresponding place in FAT#2?

$$0x12400 + 0x1ea00 = 0x30e00$$

```

030e00 01 52 02 52 03 52 04 52 05 52 06 52 07 52 08 52
030e10 09 52 0a 52 0b 52 0c 52 0d 52 0e 52 0f 52 10 52
030e20 11 52 12 52 13 52 14 52 15 52 16 52 17 52 18 52

```

Luckily, this looked okay too. In fact, FAT#2 might be completely okay even though the first 40KB or so of FAT#1 had been corrupted.

Repair Attempt #1

All of this has been interesting, but the point of this exercise was to repair the filesystem and read the data. So I now turned to my friend `fsck` for the repair work, in particular `fsck.msdos`, `err` and `dosfsck(8)`. I took the filesystem image and did what needed to be done with a spare loop device:

```

# losetup /dev/loop2 /tmp/r1
# fsck.msdos /dev/loop2

```

But according to `fsck.msdos(8)`, the "disk" claimed to have something near 165 FATs, whereas `fsck.msdos` only supports two. Apparently, some filesystem parameters were messed up severely.

Shortcut to Filesystem Repair

I started looking at the source code for `mkfs.msdos`, also known as `mkdosfs(8)`, but then came up with a better idea. What if I could create a filesystem with the FAT parameters arranged so that the FATs and the directory in this new filesystem were in the same place where the FATs and directory were in the disk image I already had? The bytes that read LEXAR MEDIA probably were the volume name. Maybe, by giving the right parameters to `mkfs.msdos(8)`, I could create a filesystem image wherein `0x08000` would point to the first FAT, `0x26a00` would point to the second FAT and `0x45400` would point to the volume label.

On the `mkdosfs(8)` manpage, I found:

SYNOPSIS

```

mkdosfs [ -A ] [ -b sector-of-backup ] [ -c ] [ -l file
name ] [ -C ] [ -f number-of-FATs ] [ -F FAT-size ] [ -i
volume-id ] [ -I ] [ -m message-file ] [ -n volume-name ]
[ -r root-dir-entries ] [ -R number-of-reserved-sectors ]
[ -s sectors-per-cluster ] [ -S logical-sector-size ] [ -v
] device [ block-count ]

```

Therefore, I specified `-f 2` for two FATs and `-n mkfs__msdos`--that is, a string I could find easily--for the volume name. This way I could tell where the vol-name landed.

How about the other parameters? I saw above that the FATs were `0x1ea00` bytes apart; if they landed the wrong distance from each other, I could tweak `-F` and maybe `-s`. I found on-line that for a filesystem of this size, the clusters would be 8192 bytes; in other words, there would be 16 512-byte sectors per cluster. The cluster is the file allocation unit described by the FAT. Hence, it would be `-s 16`.

As for where to create the filesystem, it wouldn't do to put it on the USB drive. Instead, I created a file the same size as the drive image but filled with zeroes:

```
# dd if=/dev/zero of=/tmp/r2x bs=512 count=1001952
```

After creating the filesystem, I figured I'd mount it and create a file. The file would have enough data in it that we could see a reasonable allocation chain. To accomplish this, I wrote a script and prepared to call it with parameters until I happened to find everything where I wanted it. I called it b.sh:

```
#!/bin/bash
# parameters added to mkfs.msdos....
ARGS="$*"
if mount | grep /tmp/r2d; then umount /tmp/r2d; fi
losetup -d /dev/loop2
losetup /dev/loop2 /tmp/r2x
mkfs.msdos -n mkfs__msdos -s 16 $ARGS /dev/loop2
mount -t vfat /dev/loop2 /tmp/r2d
yes hello | dd bs=8192 count=3 of=/tmp/r2d/foo.txt
umount /tmp/r2d
```

My plan was to try running this script with different parameters until I got it right. 0x8000 is 32KB. In 512-byte sectors, that's 64. Because the first FAT started at 0x8000, I decided to try `-R 64`, like this:

```
# sh b.sh -R 64
mkfs.msdos 2.8 (28 Feb 2001)
Loop device does not match a floppy size, using default hd params
2+1 records in
2+1 records out
#
```

The surprising thing was my first guess turned out to be right, at least as far as the FAT placement:

```
# hexdump -C /tmp/r2x | less
...
00008000 f8 ff ff ff 03 00 04 00 f8 ff 00 00 00 00 00 |.....|
00008010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00026a00 f8 ff ff ff 03 00 04 00 f8 ff 00 00 00 00 00 |.....|
00026a10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00045400 6d 6b 66 73 5f 5f 6d 73 64 6f 73 08 00 00 71 89 |mkfs__msdos...q.|
00045410 0f 31 0f 31 00 00 71 89 0f 31 00 00 00 00 00 00 |.1.1..q..1.....|
00045420 41 66 00 6f 00 6f 00 2e 00 74 00 0f 00 65 78 00 |Af.o.o...t...ex.|
00045430 74 00 00 00 ff ff ff ff ff ff 00 00 ff ff ff ff |t.....|
00045440 46 4f 4f 20 20 20 20 20 54 58 54 20 00 00 71 89 |FOO    TXT ..q.|
00045450 0f 31 0f 31 00 00 71 89 0f 31 02 00 00 50 00 00 |.1.1..q..1...P..|
00045460 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
```

```
00049400  68 65 6c 6c 6f 0a 68 65 6c 6c 6f 0a 68 65 6c 6c |hello.hello.hell|
00049410  6f 0a 68 65 6c 6c 6f 0a 68 65 6c 6c 6f 0a 68 65 |o.hello.hello.he|
...
```

I didn't check the directory size, but it apparently it was okay as well--more on that below.

Grafting Filesystems

I now had a boot sector that would tell `fsck.msdos` to expect the FATs and the root directory at all the right places. So what if I created a filesystem image where the first sector was that one, but all the rest of the sectors contained data from the USB drive? Then, `fsck.msdos` would read the boot sector; I'd tell it to use FAT#2 to repair everything; and we'd see how it turned out.

Repair Attempt #2

To summarize exactly what fixed the USB device:

- Step 1: create a filesystem image of the right size, with FATs and the directory in the right places:

```
# dd if=/dev/zero of=/tmp/r2x bs=512 count=1001952
# losetup /dev/loop2 /tmp/r2x
# mkfs.msdos -n mkfs__msdos -s 16 -R 64 /dev/loop2
```

- Step 2: copy bytes from the corrupt image, except the boot sector, onto the filesystem image created in step 1:

```
# dd if=r1 of=r2x bs=512 skip=1 seek=1
```

- Step 3: execute filesystem repair on that image:

```
# fsck.msdos -f -r /dev/loop2
```

Because I knew that FAT1 was bogus, I told it to use FAT2, and it reported success. It asked me whether to write the changes, and I said yes.

The filesystem images in `/tmp/r2x` and `/dev/loop2` now were consistent. The acid test was to try to mount the filesystem:

```
# mkdir /tmp/r2d
# mount -t vfat /dev/loop2 /tmp/r2d
# ls -lRA /tmp/r2d
```

After which all kinds of good stuff appeared.

Note: A good result to `ls -lR` showed that I was lucky in one other way: I didn't know if the boot sector had a

good value for the size of the root directory, the `-r` parameter to `mkfs.msdos`. I simply used the default and it turned out fine.

Burning CDs

At this point, I decided I had better burn a CD. I burn and read CDs all the time on Linux, but I rarely burn CDs to be read by Windows. Again I did a Web search, and a page from IBM's DeveloperWorks site turned up. I had searched "linux burn CD windows" or something like that. So I tried this:

```
# mkisofs -J -r -v /tmp/r2d | \
    cdrecord -v -pad -eject fs=4m speed=4 dev=0,0,0 -
```

I wasn't 100% sure that Windows would like this CD, but fortunately I have Windows95 under Win4Lin. Its sole purpose for me is to run Quicken and TurboTax, but I fired it up and pointed Windows Explorer at the just-burned CD-ROM. Explorer loved it. I used `gimp(1)` to capture a screenshot and e-mailed the image to my friend's brother--he was ecstatic.

APPENDIX: The Bash Script Explained

Shell jockeys need not read this.

```
1 #!/bin/bash
2 # parameters added to mkfs.msdos....
3 ARGS="$*"
4 if mount | grep /tmp/r2d; then umount /tmp/r2d; fi
5 losetup -d /dev/loop2
6 losetup /dev/loop2 /tmp/r2x
7 mkfs.msdos -n mkfs__msdos -s 16 $ARGS /dev/loop2
8 mount -t vfat /dev/loop2 /tmp/r2d
9 yes hello | dd bs=8192 count=3 of=/tmp/r2d/foo.txt
10 umount /tmp/r2d
```

Line 1 identifies to `exec(2)` that this is supposed to be run by the shell. I've become accustomed to `bash`, the Bourne again shell.

Line 2 simply explains line 3, that the parameters you type after `b.sh` are parameters to add to the `mkfs.msdos` command line.

Lines 4-6 establish `/dev/loop2` as the block device whose contents are in the filesystem image kept in `/dev/r2x`. Line 4 unmounts the artificial filesystem if it was mounted; this is done because we're about to make some changes to it. Lines 5-6 make sure that `/dev/loop2` is connected to `/tmp/r2x` and only to `/tmp/r2x`.

Line 7 creates an artificial filesystem image with whatever additional parameters the user gave--remember `$ARGS` from line 3?.

Line 8 mounts the filesystem onto `/tmp/r2d`. Line 9 creates a file of about 24KB (three clusters), so I have a filename to look for at the beginning of the directory.

Line 10 then unmounts the artificial filesystem image, so the kernel does not think there are inconsistencies if I play with /tmp/r2x.

Source URL: <http://www.linuxjournal.com/article/8366>