LINUX VIRUSES – ELF FILE FORMAT

Marius Van Oers
AVERT-NAI Labs, Gatwickstraat 25, 1043 GL Amsterdam, The Netherlands, Europe
Tel: +31 20 586 6136 • Fax: +31 20 586 6101 • Email: mvanoers@nai.com

ABSTRACT

The use of Linux *as an operating system is increasing rapidly, thanks partly to popular distributions such as 'RedHat' and 'Suse'. So far, there are very few* Linux *file infectors and they do not pose a big threat yet. However, with more desktops running* Linux, *and probably more* Linux *viruses, the* Linux *virus situation could become a bigger problem.*

So far, Linux *viruses are either prependers or regular file infectors that change entry points and modify the actual host code etc.*

Nowadays, the most common Linux *file type in use is called 'ELF': short for Executable and Linkable Format. ELF supports 32- as well as 64-bit objects.*

This paper will take a look at the Linux *ELF file format layout and examine some file virus infectors.*

1 ELF FILE FORMAT LAYOUT

There are currently quite a few flavours/versions of *Linux* available. Popular distributions are for example *RedHat* and *Suse*. I used two versions of *RedHat*. The main reason for this is that *Linux* virus infection/replication might be dependent on the *Linux* version/kernel level. The test systems used for the purposes of this paper were *RedHat 5.2* (Apollo) – Kernel 2.0.36 – on an i586, and *RedHat 6.1* (Cartman) – Kernel 2.2.12-20 – on an i686.

Nowadays, the most common *Linux* file format type in use is called ELF. ELF is short for Executable and Linkable Format. Objects can be viewed from a Linking or Executable perspective. Linking view is important if you want to build/compile files and want to 'link-in' a specific file. Execution view is important for 'running' a specific file. The different viewing perspectives are shown in Figure 1.

Linking View	*Execution View*
ELF Header	ELF Header
Program Header Table (optional)	Program Header Table
Section1	Segment 1
Section2	Segment 2
..	….
Section Header Table	Section Header Table (optional)

Figure 1: ELF objects can be viewed from a Linking or Executable perspective.

Local Test Systems Used:	***RedHat 5.2*** (Apollo)	Kernel 2.0.36 on an i586
	RedHat 6.1 (Cartman)	Kernel 2.2.12-20 on an i686

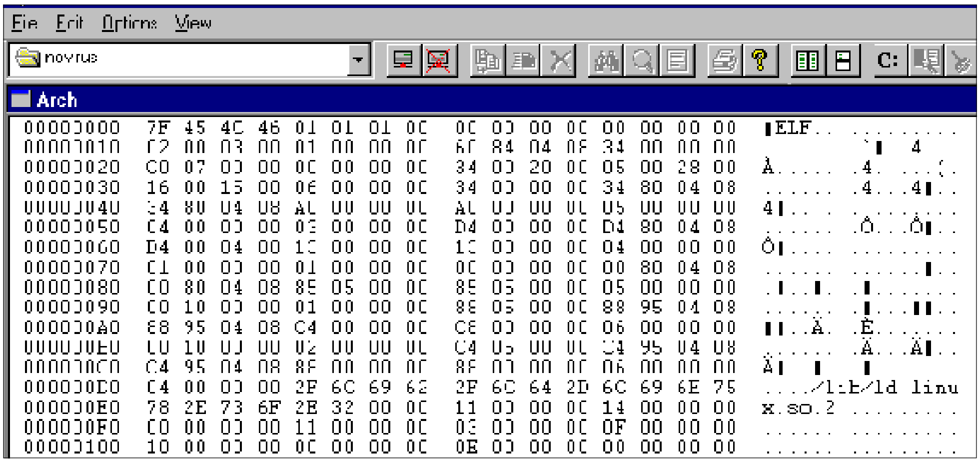
So, for the linking view, the Sections and Section Header Table are important, the Program Header Table is optional. On the other hand, for the execution view, the Segments and Program Header Table are important, and the Section Header Table is optional.

Usually, a file may contain both a Program Header Table and Segments, as well as a Section Header Table and Sections. However, according to the specific viewing perspective, certain areas are important and others are not. ELF supports 32- as well as 64-bit objects. Usually, a *Linux* installation on an *Intel*-based system is 32-bit. However, an installation on a *Dec Alpha*-based system might be 64-bit; this is not very common as not many people use *Dec Alpha* systems, but in the future we might see more 64-bit systems coming from *Intel*, *AMD* etc.

1.1 ELF Header

In the sample analysis below, an arbitrary file is selected – in this case 'arch', a clean *Linux Redhat v5.2* executable file, which was found in …/bin/arch. This file is marked with (*) in our analysis to indicate that the values found are specifically for this file. Figure 2 shows the beginning of the file viewed with a hex editor.

The ELF file format is well documented and available at various locations on-line (see references). Let's start with a line-by line inspection of what we would -encounter. Usually, just to get an indication, the ELF Header occupies the area from 0000-0033 (hex).



```
00000000  7F 45 4C 46 01 01 01 00  00 00 00 00 00 00 00 00  ▌ELF.. .........
00000010  02 00 03 00 01 00 00 00  60 84 04 08 34 00 00 00          `▌  4
00000020  C0 07 00 00 00 00 00 00  34 00 20 00 05 00 28 00  À..... .4.  ...(.
00000030  16 00 15 00 06 00 00 00  34 00 00 00 34 80 04 08  ...... .4...4▌..
00000040  34 80 04 08 A0 00 00 00  A0 00 00 00 05 00 00 00  4▌.. . .........
00000050  C4 00 00 00 03 00 00 00  D4 00 00 00 D4 80 04 08  ...... .Ô...Ô▌..
00000060  D4 00 04 00 13 00 00 00  13 00 00 00 04 00 00 00  Ô▌.... .........
00000070  C1 00 00 00 01 00 00 00  00 00 00 00 00 80 04 08  ...... ......▌..
00000080  C0 80 04 08 85 05 00 00  85 05 00 00 05 00 00 00  .▌..▌. .▌.......
00000090  C0 10 00 00 01 00 00 00  88 05 00 00 88 95 04 08  ...... .▌...▌▌..
000000A0  88 95 04 08 C4 00 00 00  C8 00 00 00 06 00 00 00  ▌▌..Ä. .È.......
000000B0  C0 10 00 00 02 00 00 00  C4 05 00 00 C4 95 04 08  ...... .Ä...Ä▌..
000000C0  C4 95 04 08 8F 00 00 00  8F 00 00 00 06 00 00 00  Ä▌   ▌   ▌
000000D0  C4 00 00 00 2F 6C 69 62  2F 6C 64 2D 6C 69 6E 75  ..../lib/ld linu
000000E0  78 2E 73 6F 2E 32 00 00  11 00 00 00 14 00 00 00  x.so.2 .........
000000F0  C0 00 00 00 11 00 00 00  03 00 00 00 0F 00 00 00  ...... .........
00000100  10 00 00 00 00 00 00 00  0E 00 00 00 00 00 00 00  ...... .........
```

Figure 2: A clean** Linux **RedHat v5.2 executable file /bin/arch (*)

```
00000000   7F 45 4C 46 01 01 01 00   00 00 00 00 00 00 00 00   ▌ELF............
```

0000–0003	EI_MAG0-EI-MAG3:	ELF identification
0004	EI_CLASS:	1: 32 bit object *
		2: 64 bit object
0005	EI_DATA Encoding:	1: LSB * (value reading from right to left)
		2: MSB (value reading from left to right)
0006	EI_VERSION:	ELF Header version number, currently it is the same as EV_CURRENT
0007-000F	EI-PAD:	Unused/Reserved

```
00000010   02 00 03 00 01 00 00 00   60 84 04 08 34 00 00 00   ........`▌..4...
```

0010-0011	E-TYPE:	1: Relocatable file
		2: Executable file *
		3: Shared object file
		4: Core file
0012-0013	E_MACHINE	3:Intel 80386 *
0014-0017	E_VERSION	Object File Version
		0: Invalid
		1:Current version *
0018-001B	E-ENTRY	Virtual Address Starting Process
001C-001F	E-PHOFF	Program Header Table, File Offset (*: 34)

```
00000020   C0 07 00 00 00 00 00 00   34 00 20 00 05 00 28 00   À.......4. ...(.
```

0020-0023	E_SHOFF	Section Header Table, File Offset (* : in this case it starts at byte 07C0)
0024-0027	E_FLAGS	Processor Specific Flags
0028-0029	E_EHSIZE	ELF Header Size (*:in this case: 34(h) bytes, and at byte 0034 the Program Header Table starts.)
002A-002B	E-PHENTSIZE	Each Program Header Table's Item Size (*:in this case each item occupies 20(h) bytes)
002C-002D	E_PHNUM	Number of Items in Program Header Table (*: in this case 5 items)
002E-002F	E_SHENTSIZE	Each Section Header Item's Size (*:in this case each item occupies 28(h)bytes)

```
00000030   16 00 15 00 06 00 00 00   34 00 00 00 34 80 04 08   ........4...4▌..
```

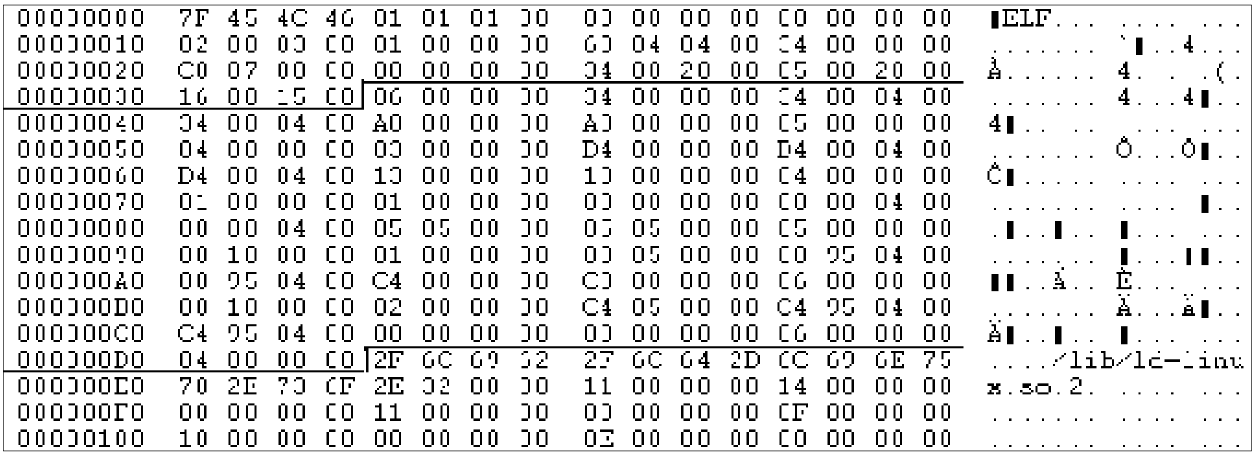
0030-0031	E_SHNUM	Number of Items in Section Header Table (*:in this case 16(h) entries, index 0-15(h)
0032-0033	E_SHSTRNDX	String Table Index in Section Header Table

In this case, therefore, for the clean *RedHat v5.2* file called arch (*):

- the Program Header Table **starts** at 0034, (e-phoff)

- there are five items (for Segments), with index 0-4, of 20(h) bytes, (e_phnum, e-phentsize).
  Item(0) of program header starts at 0034
  Item(1) follows after 20 (h) bytes at 0054
  Item(2) at 0074
  Item (3) at 0094 and
  Item(4) at 00B4.
- the **Program Header Table** occupies space from 34 to D3.

A global overview of the file header (for arch *) marking the ELF Header and the Program Header Table is displayed in Figure 3.



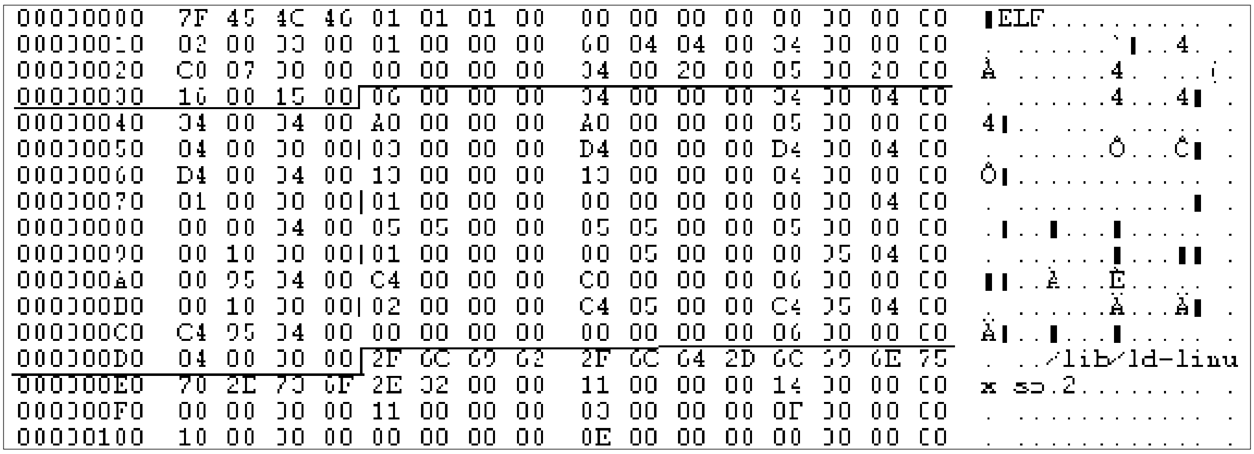
```
00000000   7F 45 4C 46 01 01 01 00   00 00 00 00 C0 00 00 00   ▮ELF............
00000010   02 00 03 C0 01 00 00 00   60 04 04 00 34 00 00 00   ........`▮..4...
00000020   C0 07 00 C0 00 00 00 00   34 00 20 00 C5 00 20 00   À.......4.....(.
00000030   16 00 15 C0|06 00 00 00   34 00 00 00 34 00 04 00   ........4...4▮..
00000040   34 00 04 C0 A0 00 00 00   A0 00 00 00 C5 00 00 00   4▮..............
00000050   04 00 00 C0 03 00 00 00   D4 00 00 00 D4 00 04 00   ........Ô...Ô▮..
00000060   D4 00 04 C0 13 00 00 00   13 00 00 00 C4 00 00 00   Ĉ▮..............
00000070   01 00 00 C0 01 00 00 00   00 00 00 00 C0 00 04 00   ............▮...
00000000   00 00 04 C0 05 05 00 00   05 05 00 00 C5 00 00 00   .▮..▮...▮.......
00000090   00 10 00 C0 01 00 00 00   00 05 00 00 C0 95 04 00   ........▮...▮▮..
000000A0   00 95 04 C0 C4 00 00 00   C0 00 00 00 C6 00 00 00   ▮▮..Ä...È.......
000000D0   00 10 00 C0 02 00 00 00   C4 05 00 00 C4 95 04 00   ........Ä...ä▮..
000000C0   C4 95 04 C0 00 00 00 00   00 00 00 00 C6 00 00 00   Ä▮..▮...▮.......
000000D0   04 00 00 C0|2F 6C 69 62   2F 6C 64 2D 6C 69 6E 75   ..../lib/ld-linu
000000E0   70 2E 73 CF 2E 32 00 00   11 00 00 00 14 00 00 00   x.so.2..........
000000F0   00 00 00 C0 11 00 00 00   00 00 00 00 CF 00 00 00   ................
00000100   10 00 00 C0 00 00 00 00   0E 00 00 00 C0 00 00 00   ................
```

Figure 3: File header overview. 0000-0033: ELF header, 0034-00D3: Program Header Table

1.2 Program Header Table

Now, let's examine the file from the Executable perspective, looking for Segments. We have seen before that in this case (*) the Program Header Table starts at 0034 with five items (index 0-4) of 20(h) bytes. Item(0) of program header starts at 0034, item(1) follows after 20(h) bytes at 0054, item(2) at 0074, item (3) at 0094 and item(4) at 00B4.

The Program Header Table determines the Segments – this information is needed for executable/ shared object files. A Segment may contain multiple Sections. The Program Header Table with the five (Segment) entries is shown in Figure 4.

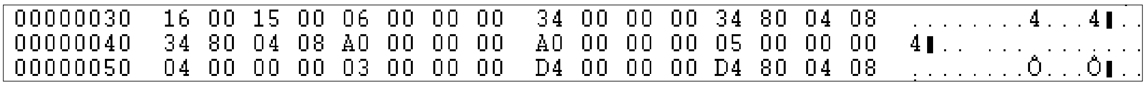


```
00000000   7F 45 4C 46 01 01 01 00   00 00 00 00 00 00 00 C0   ▮ELF............
00000010   02 00 03 00 01 00 00 00   60 04 04 00 34 00 00 C0   ........`▮..4...
00000020   C0 07 00 00 00 00 00 00   34 00 20 00 05 00 20 C0   À.......4.....(.
00000030   16 00 15 00|06 00 00 00   34 00 00 00 34 00 04 C0   ........4...4▮..
00000040   34 00 34 00 A0 00 00 00   A0 00 00 00 05 00 00 C0   4▮..............
00000050   04 00 00 00|03 00 00 00   D4 00 00 00 D4 00 04 C0   ........Ô...Ĉ▮..
00000060   D4 00 34 00 13 00 00 00   13 00 00 00 04 00 00 C0   Ô▮..............
00000070   01 00 00 00|01 00 00 00   00 00 00 00 00 00 04 C0   ............▮...
00000000   00 00 34 00 05 05 00 00   05 05 00 00 05 00 00 C0   .▮..▮...▮.......
00000090   00 10 00 00|01 00 00 00   00 05 00 00 00 95 04 C0   ........▮...▮▮..
000000A0   00 95 34 00 C4 00 00 00   C0 00 00 00 06 00 00 C0   ▮▮..À...È.......
000000D0   00 10 00 00|02 00 00 00   C4 05 00 00 C4 95 04 C0   ........Ä...Ä▮..
000000C0   C4 95 34 00 00 00 00 00   00 00 00 00 06 00 00 C0   Ä▮..▮...▮.......
000000D0   04 00 00 00|2F 6C 69 62   2F 6C 64 2D 6C 69 6E 75   ..../lib/ld-linu
000000E0   70 2E 73 6F 2E 32 00 00   11 00 00 00 14 00 00 C0   x.so.2..........
000000F0   00 00 00 00 11 00 00 00   03 00 00 00 0F 00 00 C0   ................
00000100   10 00 00 00 00 00 00 00   0E 00 00 00 00 00 00 C0   ................
```

Figure 4: Program Header Table (location 34-D3) with the five Segment entries (the start of each entry is graphically marked with the sign |)

VIRUS BULLETIN CONFERENCE **©2000 Virus Bulletin Ltd, The Pentagon, Abingdon, Oxfordshire, OX14 3YP, England. Tel +44 1235 555139.** No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form, without the prior written permission of the publishers.

So, the Program Header Table has five (Segment) entries. Let's start by looking at the first (Segment) entry, at location 0034-0053. To make it clearer, the specific area has been extracted from Figure 4, and is shown in Figure 5.



```
00000030  16 00 15 00 06 00 00 00  34 00 00 00 34 80 04 08  ........4...4▌..
00000040  34 80 04 08 A0 00 00 00  A0 00 00 00 05 00 00 00  4▌..............
00000050  04 00 00 00 03 00 00 00  D4 00 00 00 D4 80 04 08  ........Ô...Ô▌..
```

Figure 5: The first (Segment) entry in the Program Header Table is at location 34-53

0034-0037	:	P_TYPE	Segment type, in this case value 06.
0038-003B	:	P_OFFSET	Segment offset, value from beginning of file, in this case of value 34. This Segment starts at 34, which is the start of the Program Header Table.
003C-003F	:	P_VADDR	Segment Virtual Address, this case 08048034
0040-0043	:	P_PADDR	Segment Physical Address, this case 08048034
0044-0047	:	P_FILESZ	Size in bytes in file, in this case A0 bytes. So, with this Segment starting at 34, the next Segment will start at offset 34+A0=D4, which is the start 0034-0037:P_TYPE Segment type, in this case value 06.
0048-004B	:	P_MEMSZ	Size in bytes in Memory Image, this case A0 bytes.
004C-004F	:	P_FLAGS	Segment Flags.
0050-0053	:	P_ALIGN	Segment Alignment, in File and Memory Image.

After performing a similar check for all five (Segment) entries, the results presented in Figure 6 were obtained:

Segment	**Type**	**File Location**	**vaddr**	**filesz**	**memsz**	**flags**	**align**
'0'	6	0034-00D3	08048034	A0	A0	5	04
'1'	3	00D4-00E7	080480D4	13	13	4	01
'2'	1	0000-0585	08048000	0585	0585	5	1000
'3'	1	0588-064B	08049588	C4	C8	6	1000
'4'	2	05C4-064B	080495C4	88	88	6	04

Figure 6: Overview of the five segments as given by the Program Header Table

Note that the File Location area is given by: Offset (first value) + FilesSZ.

Comments on the Segment types:

- Segment '0' has the type value 6: PT_PHDR, the Program Header itself. The file location range 34–D3 is, indeed, the correct area.
  • Segment '1' has the type value 3: PT_INTERP, the location of a null-terminated path name to invoke as an interpreter. In this case: /lib/ld-lix.so.2.
  • Segment '2' has the type value 1: PT_LOAD, the loadable Segment.
  • Segment '3' has the type value 1: PT_LOAD, the loadable Segment.
  • Segment '4' has the type value 2: PT_DYNAMIC, dynamic linking information.

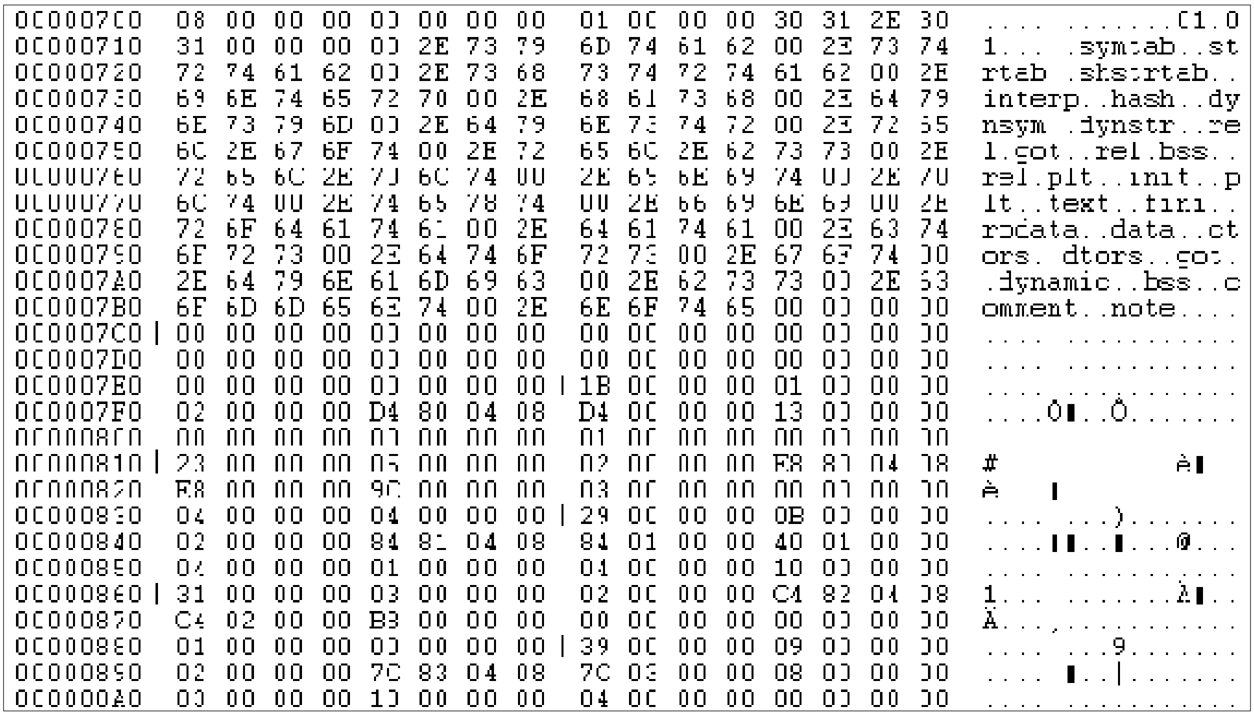
1.3 Section Header Table

Having examined the Program Header Table and the Segments, it is now time to look at the

Section Header Table and Sections.

The Sections Header Table and Sections contain important information when linking. The ELF Header shows that for the case of arch (*):

- the Section Header Table starts at 07C0, (e-shoff).
- in total 16 (h) (section) items (index 0-15(h)) of 28 (h) bytes, (e_shnum, e_shentsize).
- Item(0) of section header table starts at 07C0, item(1) follows after 28 (h) bytes at 07E8, item(2) at 0810, ... item(14) at 0AE0, item (15) from 0B08, until EOF (End Of File) 28 (h) bytes further at 0B2F.



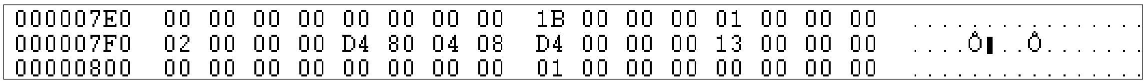
```
000007C0   08 00 00 00 00 00 00 00  01 00 00 00 30 31 2E 30   ............01.0
00000710   31 00 00 00 00 2E 73 79  6D 74 61 62 00 2E 73 74   1.....symtab..st
00000720   72 74 61 62 00 2E 73 68  73 74 72 74 61 62 00 2E   rtab..shstrtab..
00000730   69 6E 74 65 72 70 00 2E  68 61 73 68 00 2E 64 79   interp..hash..dy
00000740   6E 73 79 6D 00 2E 64 79  6E 73 74 72 00 2E 72 65   nsym..dynstr..re
00000750   6C 2E 67 6F 74 00 2E 72  65 6C 2E 62 73 73 00 2E   l.got..rel.bss..
00000760   72 65 6C 2E 70 6C 74 00  2E 69 6E 69 74 00 2E 70   rel.plt..init..p
00000770   6C 74 00 2E 74 65 78 74  00 2E 66 69 6E 69 00 2E   lt..text..fini..
00000780   72 6F 64 61 74 61 00 2E  64 61 74 61 00 2E 63 74   rodata..data..ct
00000790   6F 72 73 00 2E 64 74 6F  72 73 00 2E 67 6F 74 00   ors..dtors..got.
000007A0   2E 64 79 6E 61 6D 69 63  00 2E 62 73 73 00 2E 63   .dynamic..bss..c
000007B0   6F 6D 6D 65 6E 74 00 2E  6E 6F 74 65 00 00 00 00   omment..note....
000007C0 | 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
000007D0   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
000007E0   00 00 00 00 00 00 00 00 |1B 00 00 00 01 00 00 00   ................
000007F0   02 00 00 00 D4 80 04 08  D4 00 00 00 13 00 00 00   ....Ô▮..Ô.......
00000800   00 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00
00000810 | 23 00 00 00 05 00 00 00  02 00 00 00 E8 81 04 08   #           è▮
00000820   E8 00 00 00 9C 00 00 00  03 00 00 00 00 00 00 00   è    ▮
00000830   04 00 00 00 04 00 00 00 |29 00 00 00 0B 00 00 00   ........).......
00000840   02 00 00 00 84 81 04 08  84 01 00 00 40 01 00 00   ....▮▮..▮...@...
00000850   04 00 00 00 01 00 00 00  04 00 00 00 10 00 00 00   ................
00000860 | 31 00 00 00 03 00 00 00  02 00 00 00 C4 82 04 08   1...........Ä▮..
00000870   C4 02 00 00 B3 00 00 00  00 00 00 00 00 00 00 00   Ä...¸...........
00000880   01 00 00 00 00 00 00 00 |39 00 00 00 09 00 00 00   ........9.......
00000890   02 00 00 00 7C 83 04 08  7C 03 00 00 08 00 00 00   ....▮..|.......
000000A0   03 00 00 00 10 00 00 00  04 00 00 00 00 00 00 00   ................
```

Figure 7: Section Header Table with Section entries, location 07C0-0B2F

The Section Header Table with (section) entries is shown in Figure 7:

Sections

The Section Header Table has 16(h) Section entries: entry #0 starts at 07C0, #1 at 07E8, #2 at 080F. Let's start by looking at section entry #1. To make it clearer, the specific area has been extracted from Figure 7 and is shown in Figure 8:



```
000007E0  00 00 00 00 00 00 00 00  1B 00 00 00 01 00 00 00  ................
000007F0  02 00 00 00 D4 80 04 08  D4 00 00 00 13 00 00 00  ....Ô▮..Ô.......
00000800  00 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  ................
```

Figure 8: Section entry #1 in the Section Header Table, at location 07E8-080F

The first four bytes hold the name of the Section item, and so for entry #1:

07E8-07EB	:	SH_NAME	
07EC-07EF	:	SH_TYPE	1: SHT_PROGBITS
07F0-07F3	:	SH_FLAGS	2: SHF_ALLOC (4: SHF_EXECINSTR)
07F4-07F7	:	SH_ADDR	Starts address Memory Image : 0x080480D4

07F8-07FB	:	SH_OFFSET	Offset from beginning of file : D4 bytes. So, in this case, Section 1 starts at file location 00D4.
07FC-07FF	:	SH_SIZE	This Section size is 13 (h) bytes, so section #1starts at address 00D4 until 00D4+13=00E7. So, section #2 will probably start at 00E8. To make sure, if we look at Section Header Table #2, we see that the starting byte offset is 00E8, so it is correct.
0800-0803	:	SH_LINK	
0804-0807	:	SH_INFO	
0808-080B	:	SH_ADDRALIGN	Alignment constraints, 0-1: no constraints.
080C-080F	:	SH_ENTSIZE	Size of each sub-entry if multiple sub-entries exist, (*) 0: none.

Section Index	File Location	Image address	Type	Flags	
0	—	—	—	—	
1	00D4-00E7	080480D4	1	2	.interp
2	00E8-0183	080480E8	5	2	.hash
3	0184-02C3	08048184	B	2	.dynsym
4	02C4-037B	080482C4	3	2	.dynstr
5	037C-0383	0804837C	9	2	.rel.got
6	0384-038B	08048384	9	2	.rel.bss
7	038C-03BF	0804838C	9	2	.rel.plt
8	03C0-03EB	080483C0	1	6	.init
9	03EC-045F	080483EC	1	6	.plt
A	0460-055F	08048460	1	6	.text (E_ENTRY)
B	0560-057B	08048560	1	6	.fini
C	057C-0587	0804857C	1	2	.rodata
D	0588-058B	08049588	1	3	.data
E	058C-0593	0804958C	1	3	.ctors
F	0594-059B	08049594	1	3	.dtors
10	059C-05C3	0804959C	1	3	.got
11	05C4-064B	080495C4	6	3	.dynamic
12	064C-06AF	0804964C	8	3	.bbs
13	(.commnt), 14 (.note), 15..				

Figure 9: Overview of the 16(h) Sections as given by the Section Header Table

After performing a similar check for all 16(h) Section entries, the results shown in Figure 9 were obtained.

According to the ELF Header, the E_ENTRY (0018-001B) virtual address starting process starts at the value (*) 08048460. So this means that the section with index ‘A’ is the entry point – located at the file offset location 0460 from the beginning of the file.

So, so far for this sample (*), we have: **Linking View**

0000-0033 : ELF Header
0034-00D3 : Program Header Table
00D4-07BF : Sections
07C0-0B2F : Section Header Table

1.4 The GNU Debugger – gbd

The various Sections can also be obtained by debugging the file using **gdb**, the GNU debugger. (It can, for example, debug programs C/C++ etc.)

I put the file arch in the directory/danger.

```
[root@localhost /danger]# gdb arch <enter>
(gdb)  maintenance info sections <enter>
       q
[root@localhost /danger]#
```

This gives the various Sections. Or alternatively, you can use:

```
[root@localhost /danger]# gdb arch <enter>
(gdb)  info files <enter>
       q
[root@localhost /danger]#
```

This gives the various Sections and mentions the file type (ELF32-i386) and the (Image) entry point : 0x08048460.

1.5 Looking at other files

Now let's take a look at other 32-bit files, using *RedHat 5.2* on an *Intel* system. The Image entry [E_ENTRY] can be looked up in the ELF Header. One way to determine the file entry point is by searching for the specific Section entry which has exactly the same Image as that given by [E_ENTRY].

- Note that EI_CLASS, at offset 0004, has value 1: 32 bit object.
- Note also that EI_DATA, encoding, at offset 0005, has value 1: LSB (value reading 'from right to left').

So, look up the Image (E_ENTRY=SH_ADDR) under the Section Header Table – the SH_OFFSET is given by the next four bytes. For example, for ARCH, E_ENTRY = 0x08048460, and so one needs to search the Section Header Table for 60 84 04 08.

When found, the next 4 bytes are: 60 04 00 00, so SH_OFFSET is: 0x0460. Consider the following three files:

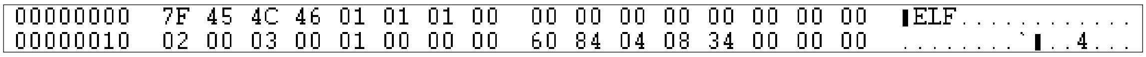
```
10/07/98 02:27a 19,116 UMOUNT  080492CC   12CC
10/16/98 12:11a 3,168 USLEEP  08048470   0470
09/10/98 08:49a 45,388 ZCAT  08048E40   0E40
```

For these files, the physical file entry point location = [E_ENTRY] – 0x08048000.

32-bit files

Let's try a similar check on 32-bit files, on a *Sun Solaris 2.6.*

File: Adb

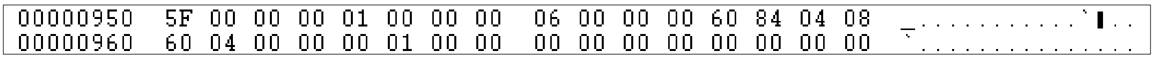


```
00000000  7F 45 4C 46 01 01 01 00  00 00 00 00 00 00 00 00  ▌ELF............
00000010  02 00 03 00 01 00 00 00  60 84 04 08 34 00 00 00  ........`▌..4...
```

- Note that EI_CLASS, at offset 0004, has value 1: 32 bit object.
- Note also that EI_DATA, encoding, at offset 0005, has value 2: MSB (value reading 'from left to right').

So, look up the Image (E_ENTRY = SH_ADDR) under the Section Header Table – the SH_OFFSET is given by the next four bytes. For example, for Adb, the E_ENTRY

= 0x00013BAC.



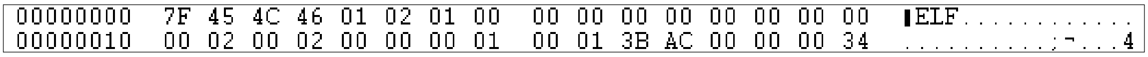
```
00000950  5F 00 00 00 01 00 00 00  06 00 00 00 60 84 04 08  _...........`▌..
00000960  60 04 00 00 00 01 00 00  00 00 00 00 00 00 00 00  `...............
```

When found, the next four bytes are 00 00 3B AC, and so SH_OFFSET is: 0x3BAC

```
02/15/00 03:21p 124,680 adb   00013BAC   3BAC
02/15/00 03:21p 345,728 admintool 00018BF0   8BF0
02/15/00 03:21p 15,784 aliasadm 000111E4   11E4
```

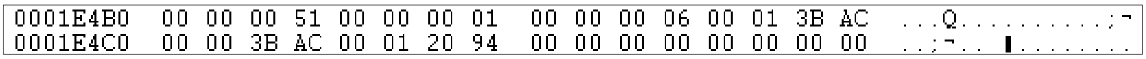
So, for these three files, the physical file entry point location = [E_ENTRY] – 0x00010000.

64-bit files
Again, let's a similar check on 64-bit files, *Red Hat 5.2* on a *Dec Alpha*.
File: arch



```
00000000  7F 45 4C 46 01 02 01 00  00 00 00 00 00 00 00 00  ▌ELF............
00000010  00 02 00 02 00 00 00 01  00 01 3B AC 00 00 00 34  ..........;¬...4
```

• Note that EI_CLASS, at offset 0004, has value 2: 64 bit object.
• Note also that EI_DATA, encoding, at offset 0005, has value 1: LSB (value reading 'from right to left').

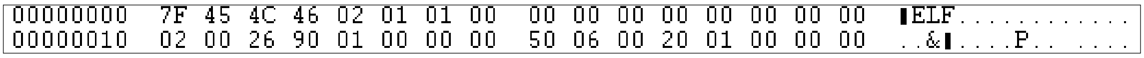


```
0001E4B0  00 00 00 51 00 00 00 01  00 00 00 06 00 01 3B AC  ...Q..........;¬
0001E4C0  00 00 3B AC 00 01 20 94  00 00 00 00 00 00 00 00  ..;¬.. ▌........
```

So, look up the Image (E_ENTRY=SH_ADDR) under the Section Header Table – the E_ENTRY = 0x20000650, so search the Section Header Table:

Now, instead of the next four bytes (32-bit), the offset is given after the next eight bytes (64-bit). In this case: 0x0650.

```
02/18/00 09:26a 4,392 arch  20000650   0650
02/18/00 09:26a 109,128 ash   200013C0         13C0
02/18/00 09:26a 244,896 ash.static 20000100   0100
02/18/00 09:26a 7,920 basename 20000980         0980
```

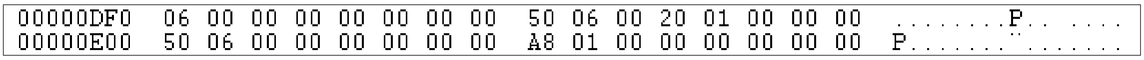


```
00000000  7F 45 4C 46 02 01 01 00  00 00 00 00 00 00 00 00  ▌ELF............
00000010  02 00 26 90 01 00 00 00  50 06 00 20 01 00 00 00  ..&▌....P.. ....
```

So, for these four files, the physical file entry point location = [E_ENTRY] – 0x20000000.

In the previous three cases we have seen:

physical file entry point location = [E_ENTRY] – 0x08048000
physical file entry point location = [E_ENTRY] – 0x00010000



```
00000DF0  06 00 00 00 00 00 00 00  50 06 00 20 01 00 00 00  ........P.. ....
00000E00  50 06 00 00 00 00 00 00  A8 01 00 00 00 00 00 00  P.......¨.......
```

physical file entry point location = [E_ENTRY] – 0x20000000

For these samples it seems like an **Image Base**. Sometimes it is the same as the lowest Segment's VADDR, although this is not the case for all samples.

According to documentation:

> ‘The base address of a file is calculated during execution from 3 values:
>
> - memory load address
> - maximum page size
> - lowest virtual address of a program’s loadable segment
>
> The virtual addresses in the program headers might not represent the actual virtual addresses of the program’s memory image.
>
> To compute the base address, one determines the memory address associated with the lowest p_vaddr value for a PT_LOAD segment. One then obtains the base address by truncating the memory address to the nearest multiple of the maximum page size. Depending on the kind of file being loaded into memory, the memory address might or might not match the p_vaddr values.’

2 ELF FILE VIRUSES

Unix/*Linux* is a very good security model. For example, without root (administration) rights it is very difficult to change ELF binary files. So, for a virus to be successful, it needs high rights. Another aspect to consider is that there are quite a lot of different ‘flavours’ of Unix around, and so a Unix virus will most likely not infect on all systems. Nevertheless, with the increase of popularity of *Linux* it is possible that we will see more *Linux* viruses in the future.

Generally, a file virus can either be a relatively simple prepender or of a more advanced nature – for example by changing internal section items. Recently, at the beginning of 2000, a number of *Linux* viruses were encountered – they were from virus collections, however, and not ‘real’ infections from in the wild.

2.1 Lin/Bliss

The first *Linux* binary virus, Lin/Bliss, was encountered in 1997 – it demonstrated that *Linux* could be vulnerable to binary viruses. Lin/Bliss is a relatively simple prepender, and so far there are a few variants (prepending either 17,892 or 18,604 bytes). The infected files have two ELF headers, the first from the virus, the second from the original (uninfected) file. For infected files:

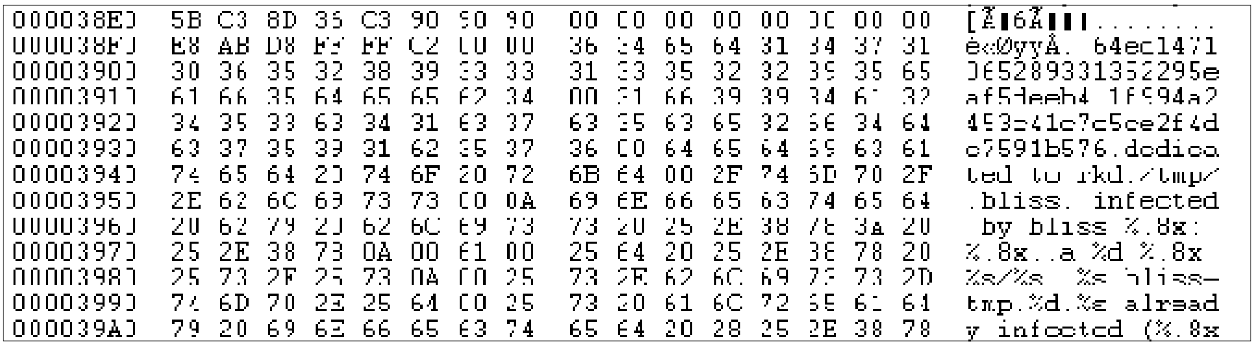
The second ELF header starts at offset 45E4 (hex) = 17,892 (dec), or
the second ELF header starts at offset 48AC (hex) = 18,604 (dec).

So, with prependers like Lin/Bliss, detection and repair is easy.

Technical Details
For a Lin/Bliss sample called BLI17892.LNX:

EI_Class:	1 – 32-bit.
EI_Data:	1 – LSB, value reading from right to left.
E_Entry:	08049120.
Section Header Table Offset:	429C (28 bytes Table Section items, 15 sections in all, which within viral range of 45E4 total virus code).
Program Header Table, Offset:	34 (20 hex entries in Table, five entries)

Part of a Lin/Bliss-infected file is shown in Figure 10:



```
000038E0  5B C3 8D 36 C3 90 90 90  00 C0 00 00 00 0C 00 00  [Ã.6Ã...........
000038F0  E8 AB D8 FF FF C2 C0 00  36 34 65 63 31 34 37 31  è«ØÿÿÂ..64ec1471
00003900  30 36 35 32 38 39 33 33  31 33 35 32 32 39 35 65  065289331352295e
00003910  61 66 35 64 65 65 62 34  00 31 66 39 39 34 61 32  af5deeb4.1f994a2
00003920  34 35 33 63 34 31 63 37  63 35 63 65 32 66 34 64  453c41c7c5ce2f4d
00003930  63 37 35 39 31 62 35 37  36 00 64 65 64 69 63 61  c7591b576.dedica
00003940  74 65 64 20 74 6F 20 72  6B 64 00 2F 74 6D 70 2F  ted to rkd./tmp/
00003950  2E 62 6C 69 73 73 00 0A  69 6E 66 65 63 74 65 64  .bliss..infected
00003960  20 62 79 20 62 6C 69 73  73 20 25 2E 38 78 3A 20   by bliss %.8x: 
00003970  25 2E 38 78 0A 00 61 00  25 64 20 25 2E 38 78 20  %.8x..a.%d %.8x 
00003980  25 73 2F 25 73 0A 00 25  73 2E 62 6C 69 73 73 2D  %s/%s..%s.bliss-
00003990  74 6D 70 2E 25 64 00 25  73 20 61 6C 72 65 61 64  tmp.%d.%s alread
000039A0  79 20 69 6E 66 65 63 74  65 64 20 28 25 2E 38 78  y infected (%.8x
```

Figure 10: Lin/Bliss-infected file

2.2 Lin/Glaurung.676/666 (alias Mandragore)

A so-called **appending** virus, Lin/Glaurung is encrypted. When running infected samples on a an *Intel* machine with *RedHat 5.2*, an error occurs reporting a ‘Segmentation fault’ (i.e. core dumped). This error was encountered with all samples on the specified machine and, as expected, no replication/further infection was seen.

So, the good thing is that a Unix virus will probably spread only on certain flavours and/or versions or kernel versions of Unix operating systems. This is a bad thing for the AV industry since it requires more test machines running the various Unix/*Linux* configurations in order to investigate samples fully.

When running infected samples on an *Intel* machine with *RedHat 6.1*, no error occurred. The direct infection mechanism simply infected a lot of ELF binary files in the /bin directory after running one infected file just once.

RedHat 6.1 file called DOEXEC, ‘clean’ file size is 3,028 bytes (dec), (0BD3 hex)
RedHat 6.1 file called DOEXEC, ‘infected’ file size is: 3,694 bytes (dec), (0E6E hex) – an increase of 666 bytes (dec), (29A hex).

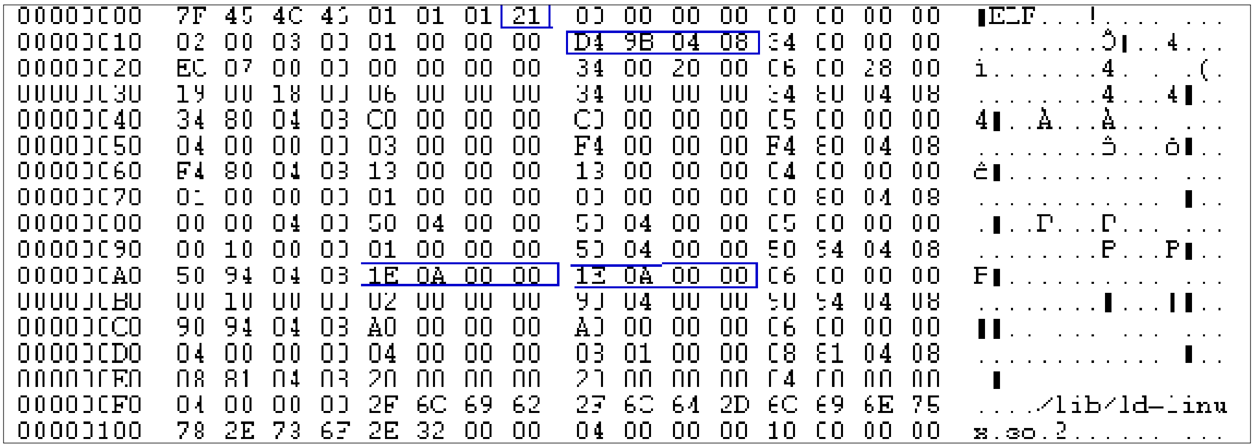
The infected file header is shown in Figure 11a, the infection mechanism in Figure 11b.

The entry EI_PAD, from offset 0007-000F, is normally unused/reserved (normally 00). In all Lin/Glaurung-infected files, the byte at offset 07 is used, with the value 21 (hex). This seems to be a quick marker to determine if the viral code is already present or not. For the file DOEXEC:

	E-entry value	File entry value
Clean	0x08048320	0x0320
Infected	0x08049BD4	0x0BD4

The infected file entry value for DOEXEC (0x0BD4) is exactly the start of the appending viral code (remember the EOF of the clean file was 0BD3 hex). The Program Header Table has six entries, numbered 0 to 5. Table entry #3 differs in its clean and infected states:

• Clean P_Filesz, size in bytes in file:	0x00E0, infected:	0x0A1E
• Clean P_Memsz, size in bytes in memory Image:	0x00F8, infected:	0x0A1E



```
00000000   7F 45 4C 46 01 01 01 21  00 00 00 00 00 00 00 00   ▮ELF...!........
00000010   02 00 03 00 01 00 00 00  D4 9B 04 08 34 00 00 00   ........Ô▮..4...
00000020   EC 07 00 00 00 00 00 00  34 00 20 00 06 00 28 00   ì.......4. ...(.
00000030   19 00 18 00 06 00 00 00  34 00 00 00 34 80 04 08   ........4...4▮..
00000040   34 80 04 08 C0 00 00 00  C0 00 00 00 05 00 00 00   4▮..À...À.......
00000050   04 00 00 00 03 00 00 00  F4 00 00 00 F4 80 04 08   ........ô...ô▮..
00000060   F4 80 04 08 13 00 00 00  13 00 00 00 04 00 00 00   ô▮..............
00000070   01 00 00 00 01 00 00 00  00 00 00 00 00 80 04 08   .............▮..
00000080   00 80 04 08 50 04 00 00  50 04 00 00 05 00 00 00   .▮..P...P.......
00000090   00 10 00 00 01 00 00 00  50 04 00 00 50 94 04 08   ........P...P▮..
000000A0   50 94 04 08 1E 0A 00 00  1E 0A 00 00 06 00 00 00   P▮..............
000000B0   00 10 00 00 02 00 00 00  90 04 00 00 90 94 04 08   ........▮...▮▮..
000000C0   90 94 04 08 A0 00 00 00  A0 00 00 00 06 00 00 00   ▮▮..............
000000D0   04 00 00 00 04 00 00 00  08 01 00 00 08 81 04 08   .............▮..
000000E0   08 81 04 08 20 00 00 00  20 00 00 00 04 00 00 00   .▮.. ... .......
000000F0   04 00 00 00 2F 6C 69 62  2F 6C 64 2D 6C 69 6E 75   ..../lib/ld-linu
00000100   78 2E 73 6F 2E 32 00 00  04 00 00 00 10 00 00 00   x.so.2..........
```

Figure 11a: Lin/Glaurung-infected file

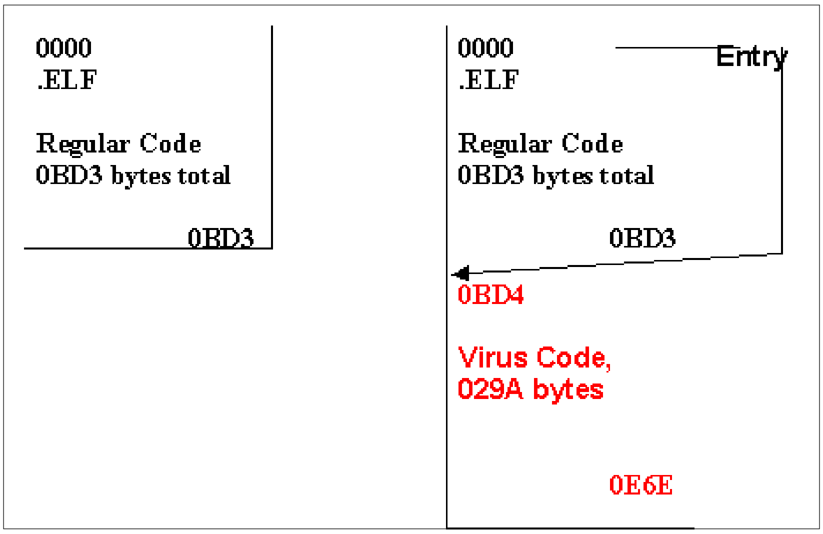


Figure 11b: The Lin/Glaurung infection scheme

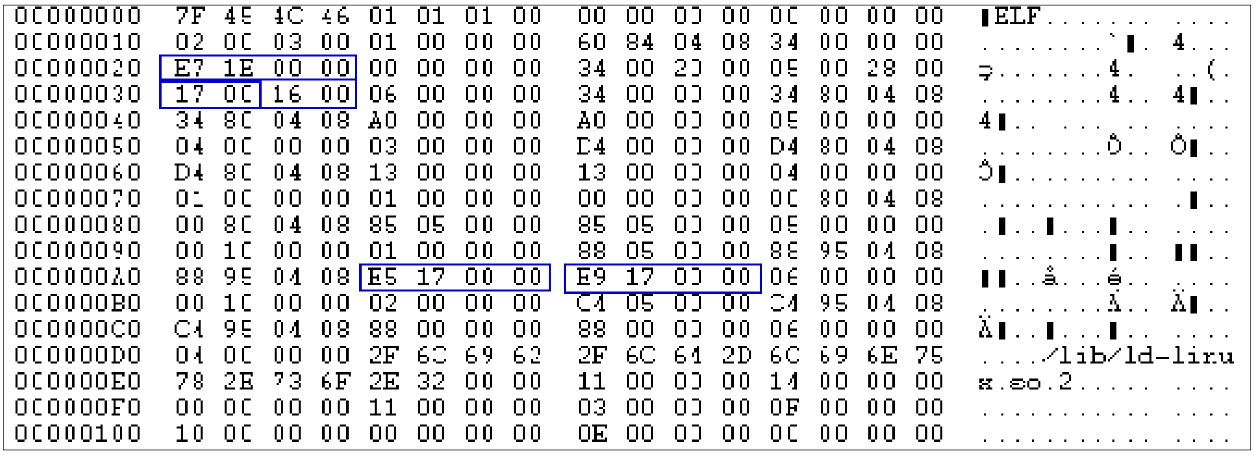
2.3 Lin/Silv.A

This infects without problems on an *Intel* machine running *RedHat 5.2*. The clean file 'arch' has a file size of 2,864 (dec) bytes, whereas the infected file is 8,831 bytes long, representing an increase of 5,967 bytes. This virus does not append/prepend but inserts its code into slack space. As a result, the file size increase with this 32-bit file infector is hardly constant. Figure 12a shows the Lin/Silv-infected 'arch' (*) file. Looking at the ELF header, we can see that E_SHOFF (offset from the beginning of the file to the Section Header Table) has been changed.

E_SHOFF (clean):	0x07C0
E_SHOFF (infected):	0x1EE7, so the Section header is further down in the file. An observation such as this could be the first sign that viral code has been inserted between regular code.

- The value for E_SHNUM (the number of items in the Section Header Table) changed as well, from 16(h) to 17(h). The virus seems to add one Section (Data1).
- Consequently, E_SHSTRNDX (the String table index in the Section Header Table) was changed from 15 to 16.

Bearing in mind the Segment information for the clean ‘arch’ (*) file (see Figure 6), the information for Segment 3 of the infected file is now:



```
0C000000  7F 45 4C 46 01 01 01 00  00 00 0J 00 0C 00 00 00  ▌ELF........ ....
0C000010  02 0C 03 00 01 00 00 00  60 84 04 08 34 00 00 00  ........`▌. 4...
0C000020  E7 1E 00 00 00 00 00 00  34 00 2J 00 05 00 28 00  ╤.......4.  ..(.
0C000030  17 0C 16 00 06 00 00 00  34 00 0J 00 34 80 04 08  ........4..  4▌..
0C000040  34 8C 04 08 A0 00 00 00  A0 00 0J 00 05 00 00 00  4▌.. ....... ....
0C000050  04 0C 00 00 03 00 00 00  D4 00 0J 00 D4 80 04 08  ........Ô..  Ô▌..
0C000060  D4 8C 04 08 13 00 00 00  13 00 0J 00 04 00 00 00  Ô▌......... ....
0C000070  01 0C 00 00 01 00 00 00  00 00 0J 00 0C 80 04 08  ........... .▌..
0C000080  00 8C 04 08 85 05 00 00  85 05 0J 00 05 00 00 00  .▌..▌...▌.. ....
0C000090  00 1C 00 00 01 00 00 00  88 05 0J 00 8E 95 04 08  ........▌..  ▌▌..
0C0000A0  88 95 04 08 E5 17 00 00  E9 17 0J 00 0E 00 00 00  ▌▌..å...é.. ....
0C0000B0  00 1C 00 00 02 00 00 00  C4 05 0J 00 D4 95 04 08  ........Ä..  Ä▌..
0C0000C0  C4 95 04 08 88 00 00 00  88 00 0J 00 0E 00 00 00  Ä▌..▌...▌.. ....
0C0000D0  04 0C 00 00 2F 6C 69 62  2F 6C 61 2D 6C 69 6E 75  ..../lib/ld-linu
0C0000E0  78 2E 73 6F 2E 32 00 00  11 00 0J 00 14 00 00 00  x.so.2..... ....
0C0000F0  00 0C 00 00 11 00 00 00  03 00 0J 00 0F 00 00 00  ........... ....
0C000100  10 0C 00 00 00 00 00 00  0E 00 0J 00 0C 00 00 00  ........... ....
```

Figure 12a: Lin/Silv-infected file

Segment	**Type**	**File Location**	**vaddr**	**filesz**	**memsz**	**flags**	**align**
‘3’	1	0588-1D6D	08049588	17E5	17E9	6	1000

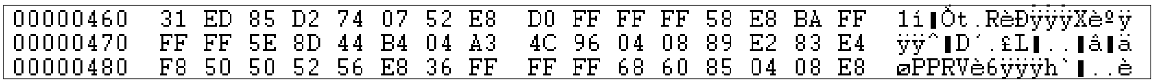
As can be seen, in the Infected file the value for P_FILESZ for segment ‘3’ has changed from C4 to 17E5, which accounts for a file size increase of 1,721(h) bytes (5921(d)). This is very close to the total file size increase of 5,967(d) bytes (for this specific sample only). So, the file location is 0588 to 0588+17E5 = 1D6D.

Also, the value for P_MEMSZ in the infected file has increased from C8 to 17E9. This also represents an increase of 1721(h) / 5921(d) bytes. The Section layout Image address is given by gdb arch files information, as shown in Figure 12b below:

```
Clean ‘arch’                                Lin/Silv.A-infected ‘arch’
File type ELF32-i386                              File type ELF32-i386
Entry point: 0x08048460                     Entry point: 0x08048460
      0x080480D4 – 0x080480E7         .interp                   same
            80E8          8184        .hash                     same
            8184          82C4        .dynsym                   same
            82C4          837C        .dynstr                   same
            837C          8384        .rel.got                  same
            8384          838C        .rel.bss                  same
            838C          83BC        .rel.plt                  same
            83C0          83EC        .init                     same
            83EC          845C        .plt                      same
            8460          8560        .text (e_entry)           same
            8560          857C        .fini                     same
            857C          8585        .rodata                   same
            9588          958C        .data                     same
            958C          9594        .ctors                    same
            9594          959C        .dtors                    same
            959C          95C4        .got                      same
            95C4          964C        .dynamic                  same
            964C          9650        .bbs     ßà    964C AD6D .data1
```

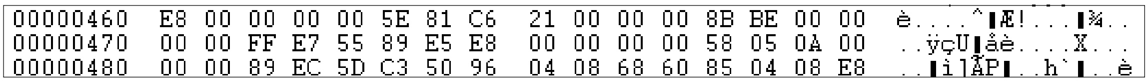
Figure 12b: Lin/Silv-infected file changes Section

Clean ‘arch’ file:



```
00000460  31 ED 85 D2 74 07 52 E8  D0 FF FF FF 58 E8 BA FF  1í▌Òt.RèÐÿÿÿXèºÿ
00000470  FF FF 5E 8D 44 B4 04 A3  4C 96 04 08 89 E2 83 E4  ÿÿ^▌D´.£L▌..▌â▌ä
00000480  F8 50 50 52 56 E8 36 FF  FF FF 68 60 85 04 08 E8  øPPRVè6ÿÿÿh`▌..è
```

Infected ‘arch’ file:



```
00000460  E8 00 00 00 00 5E 81 C6  21 00 00 00 8B BE 00 00  è....^▌Æ!...▌¾..
00000470  00 00 FF E7 55 89 E5 E8  00 00 00 00 58 05 0A 00  ..ÿçU▌åè....X...
00000480  00 00 89 EC 5D C3 50 96  04 08 68 60 85 04 08 E8  ..▌ì]ÃP▌..h`▌..è
```

Figure 12c: Lin/Silv modifies the actual code at the unchanged entry point

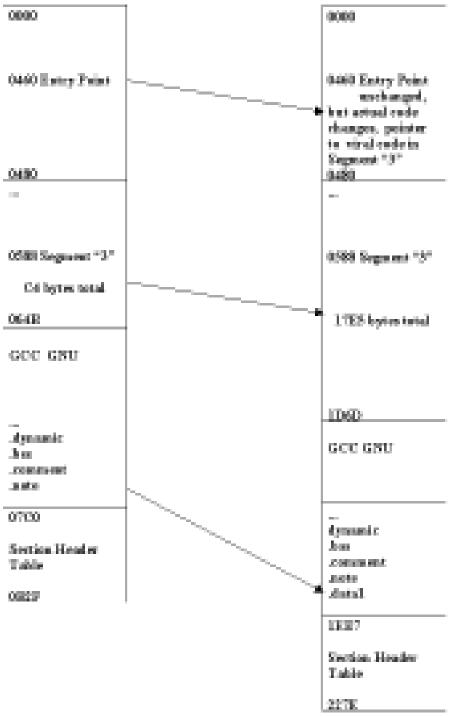


Figure 12d: Lin/Silv.A-infected ‘arch’ file

We can see that this virus places its viral code at the end of the host file. The virus does not seem to change the entry point (e_entry). So, how does the virus code become activated? Well, although the virus does not *seem* to change the entry point (e_entry) initially, it actually modifies the code at the entry point such that it takes control.

2.4 Lin/Obsidian.E

The viruses in the Lin/Obsidian family do not replicate correctly on all systems. The variants A through D did not replicate whatsoever under *RedHat 5.2*. The .E variant, however, replicated fine. Lin/Obsidian.E is a so-called prepender, inserting its viral code before the target file. So, in this case we end up with a file with two ELF headers: firstly, the viral one and secondly, the one from the regular work file. As an example, let us look at a sample file called DOEXEC:

Clean file DOEXEC:	2,652 bytes (dec), *RedHat 5.2*, ELF32-i386
Infected file DOEXEC:	10,652 bytes (dec)

E-Entry Image infected file doexec.inf: 08048970, this is pointing to a file location which is well within the viral body – nothing strange here (for this sample series: 0x08048970-0x08048000 = 0x0970). The infected files are really strange in the following respects:

- If we try to use gbd on the file, using ‘gdb doexec.inf’, an error message results (“/danger/doexec.inf” : not in executable format: File truncated).
- If we use the command ‘info files’ nothing happens, no information is provided.
- If we try to use the command ‘maintenance info section’ then nothing happens, again no information is provided.

If we look at the infected file called DOEXEC.INF manually, we see that the second .ELF header starts at offset 1F40(h), so the virus inserted 8,000 bytes. This is OK if we look at the file increase from 2,652 to 10,652. So, all the viral code seems to be inserted, with nothing in between or appended.

If we look at the Section Header Table Offset, for all samples it always has the value 25F0. This is strange for two reasons. Firstly, the value is always constant for all infected files, which would indicate that the Section Header Table is at a random, incorrect location. Secondly, the virus inserts 1F40(h) bytes in total, so the Section Header Table Offset as given in the viral code (in the first ELF header) is pointing to a random location in the ‘second part’ of the file (the code from the regular work file). But infected files still run. Why? The question is therefore:

Can the Section header table be ignored for executing files?
I took a clean *RedHat 6.1* file called ‘arch’, for which the Section Header Table offset was 0x0890. I replaced the complete Section Header Table with zeroes until the file ended at 0x0C77. I also took a clean *RedHat 6.1* file called ‘date’, for which the Section Header Table offset was 0x64EC. Again, I replaced the complete Section Header Table with zeroes until the file ended at 0x68FC. I tried to execute both of these files and they both ran fine!

From the ELF documentation we recall that we can look at binary files from different viewpoints. For a Linking viewpoint, a Section Header Table is required. At a minimum, the ELF header (the Program Header Table is optional), Section1, Section2, etc, and the Section Header Table are required. From an Execution viewpoint, a Section Header Table is optional, and the minimum requirements are the ELF header, Program Header Table, Segment1, Segment2, etc.

If we look at the infected file called DOEXEC.INF manually, we see that the following information can be retrieved from the Program Header Table concerning the various Segment items for the viral code:

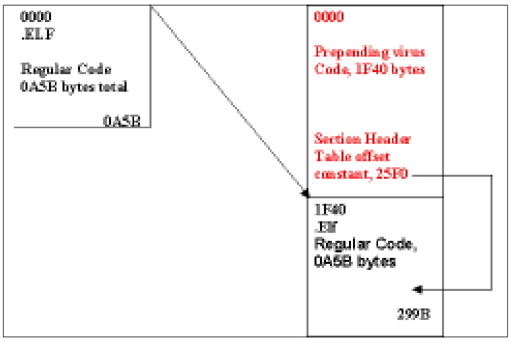


Figure 13: Linux/Obsidian.E infection scheme

Segment	Type	Offset	Vaddr	Filesz	Memsz	Flags	Alignment	File location
‘0’	06	34	08048034	A0	A0	5	4	34-D4
‘1’	03	D4	080480D4	13	13	4	1	D4-E7
‘2’	01	00	08048000	17A5	17A5	5	1000	00-17A5
‘3’	01	17A8	0804A7A8	0130	0268	6	1000	17A8-18D8
‘4’	02	1850	0804 A850	88	88	6	4	1850-18D8

The infection scheme employed by Lin/Obsidian.E is shown in Figure 13.

2.5 Lin/Vit.4096

Lin/Vit.4096 samples did infect on my test system running 32-bit *Intel i586*, *Redhat 5.2*.

Clean file (DOEXEC): 2,652 bytes, 0A5C(h)
Infected file (DOEXEC): 6,748 bytes, 1A5C(h)

On the sample file, the virus adds 4,096 bytes, 1000(h)

The clean E_Entry has the value: 0x080484700
The viral E-Entry has the value: 0x08048B3C

The virus changes the section called ‘.Fini’ (the maintenance information sections):

Clean file DOEXEC	Infected file DOEXEC
080484D0-080484EC .Fini	080484D0-08048DB6 .Fini

Clean file DOEXEC Segments:

Segment	Type	FileLocation	Vaddr	FileSz	MemSz	Flags	Align	[FileUsage]
‘0’	06	34	08048034	A0	A0	5	04	0034-00D4
‘1’	03	D4	080480D4	13	13	4	01	00D4-00E7
‘2’	01	00	08048000	04EC	04EC	5	1000	0000-04EC
‘3’	01	04EC	080494EC	BC	C0	6	1000	04EC-05A8
‘4’	02	0520	08049520	88	88	6	04	0520-05A8

Infected file DOEXEC Segments:

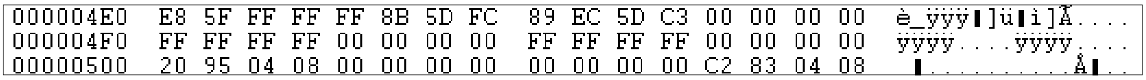
Segment	Type	FileLocation	Vaddr	FileSz	MemSz	Flags	Align	[FileUsage]
‘0’	06	34	08048034	A0	A0	5	04	0034-00D4
‘1’	03	D4	080480D4	13	13	4	01	00D4-00E7
‘2’	01	00	08048000	0DB6	0DB6	5	1000	0000-0DB6
‘3’	01	14EC	080494EC	BC	C0	6	1000	14EC-15A8
‘4’	02	1520	08049520	88	88	6	04	1520-05A8

Figure 14a: Lin/Vit.4096-infected file Segment differences

Clean file DOEXEC: Section Header Table starts at offset 0x0714 from beginning of file.
Infected file DOEXEC: Section Header Table starts at offset 0x1714 from beginning of file.

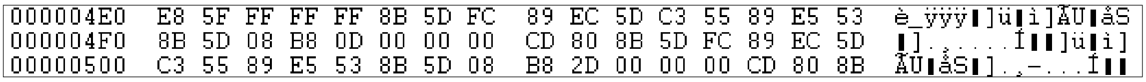
The various segment changes after the file was infected by Lin/Vit.4096 can be seen in Figure 14a. Figure 14b shows a section of the viral code inserted into the middle of the file, and the end of the viral code can be seen in Figure 14c.

Clean DOEXEC file:



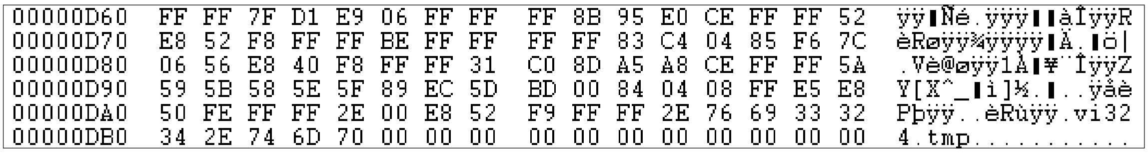
```
000004E0  E8 5F FF FF FF 8B 5D FC  89 EC 5D C3 00 00 00 00  è_ÿÿÿ▮]ü▮ì]Ã....
000004F0  FF FF FF FF 00 00 00 00  FF FF FF FF 00 00 00 00  ÿÿÿÿ....ÿÿÿÿ....
00000500  20 95 04 08 00 00 00 00  00 00 00 00 C2 83 04 08   ▮..........Â▮..
```

Infected DOEXEC file:



```
000004E0  E8 5F FF FF FF 8B 5D FC  89 EC 5D C3 55 89 E5 53  è_ÿÿÿ▮]ü▮ì]ÃU▮åS
000004F0  8B 5D 08 B8 0D 00 00 00  CD 80 8B 5D FC 89 EC 5D  ▮].¸....Í▮▮]ü▮ì]
00000500  C3 55 89 E5 53 8B 5D 08  B8 2D 00 00 00 CD 80 8B  ÃU▮åS▮].¸-...Í▮▮
```

Figure 14b: Lin/Vit inserts code to the ‘middle’ of the file, Section 3 in this case, at 04EC



```
00000D60  FF FF 7F D1 E9 06 FF FF  FF 8B 95 E0 CE FF FF 52  ÿÿ▮Ñé.ÿÿÿ▮▮àÎÿÿR
00000D70  E8 52 F8 FF FF BE FF FF  FF FF 83 C4 04 85 F6 7C  èRøÿÿ¾ÿÿÿÿ▮Ä.▮ö|
00000D80  06 56 E8 40 F8 FF FF 31  C0 8D A5 A8 CE FF FF 5A  .Vè@øÿÿ1À▮¥¨ÎÿÿZ
00000D90  59 5B 58 5E 5F 89 EC 5D  BD 00 84 04 08 FF E5 E8  Y[X^_▮ì]½.▮..ÿåè
00000DA0  50 FE FF FF 2E 00 E8 52  F9 FF FF 2E 76 69 33 32  Pþÿÿ..èRùÿÿ.vi32
00000DB0  34 2E 74 6D 70 00 00 00  00 00 00 00 00 00 00 00  4.tmp...........
```

Figure 14c: The end of the Lin/Vit viral code, followed by filling up/alignment zeroes

So we see that for the DOEXEC sample file, the Lin/Vit.4096 virus inserts its viral code at the start of segment ‘3’. The original segment ‘3’ is moved down by 1,000(h)/4,096(d) bytes. A similar situation exists for the gnu/gcc/symtab and Section Header Table (Figure 14d). The original segment ‘3’ started at offset 04EC from the beginning of the file, yet in the infected file: it starts at offset 14EC. However, the virus does not take up the full 1,000(h) bytes. In the case of our test file DOEXEC, the actual viral bytes end (with vi324.tmp) at offset 0DB6, which is the end of Segment ‘2’ in the infected file, leaving the area from 0DB6 to 14EC for zeroes/empty space.

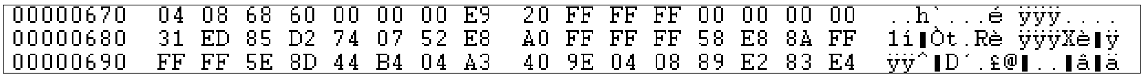
2.6 Lin/Diesel

Under a *32-bit Intel i586* with *Redhat 5.2*, samples were readily infected with **Lin/**Diesel.969:

Clean file base name:	4,892 bytes (dec)
Infected file base name:	5,909 bytes (dec)

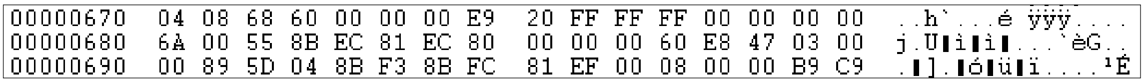
The clean E_Entry has the value 0x08048680, the entry at the file is at offset 0680 from the beginning. The virus does not change the value for E-Entry, but instead changes the actual bytes at the entry point, as shown in Figure 15a.

Clean base name:



```
00000670  04 08 68 60 00 00 00 E9  20 FF FF FF 00 00 00 00  ..h`...é ÿÿÿ....
00000680  31 ED 85 D2 74 07 52 E8  A0 FF FF FF 58 E8 8A FF  1í▮Òt.Rè ÿÿÿXè▮ÿ
00000690  FF FF 5E 8D 44 B4 04 A3  40 9E 04 08 89 E2 83 E4  ÿÿ^▮D´.£@▮..▮â▮ä
```

Infected base name:



```
00000670  04 08 68 60 00 00 00 E9  20 FF FF FF 00 00 00 00  ..h`...é ÿÿÿ....
00000680  6A 00 55 8B EC 81 EC 80  00 00 00 60 E8 47 03 00  j.U▮ì▮ì▮...`èG..
00000690  00 89 5D 04 8B F3 8B FC  81 EF 00 08 00 00 B9 C9  .▮].▮ó▮ü▮ï....¹É
```

Figure 15a: Lin/Diesel changes bytes at the entry point, not the entry point itself

The virus puts/overwrites its viral at location 0680 (file entry) to 0A49, which is 3C9(h) bytes (969dec). The end of the viral code can be seen in Figure 15b:

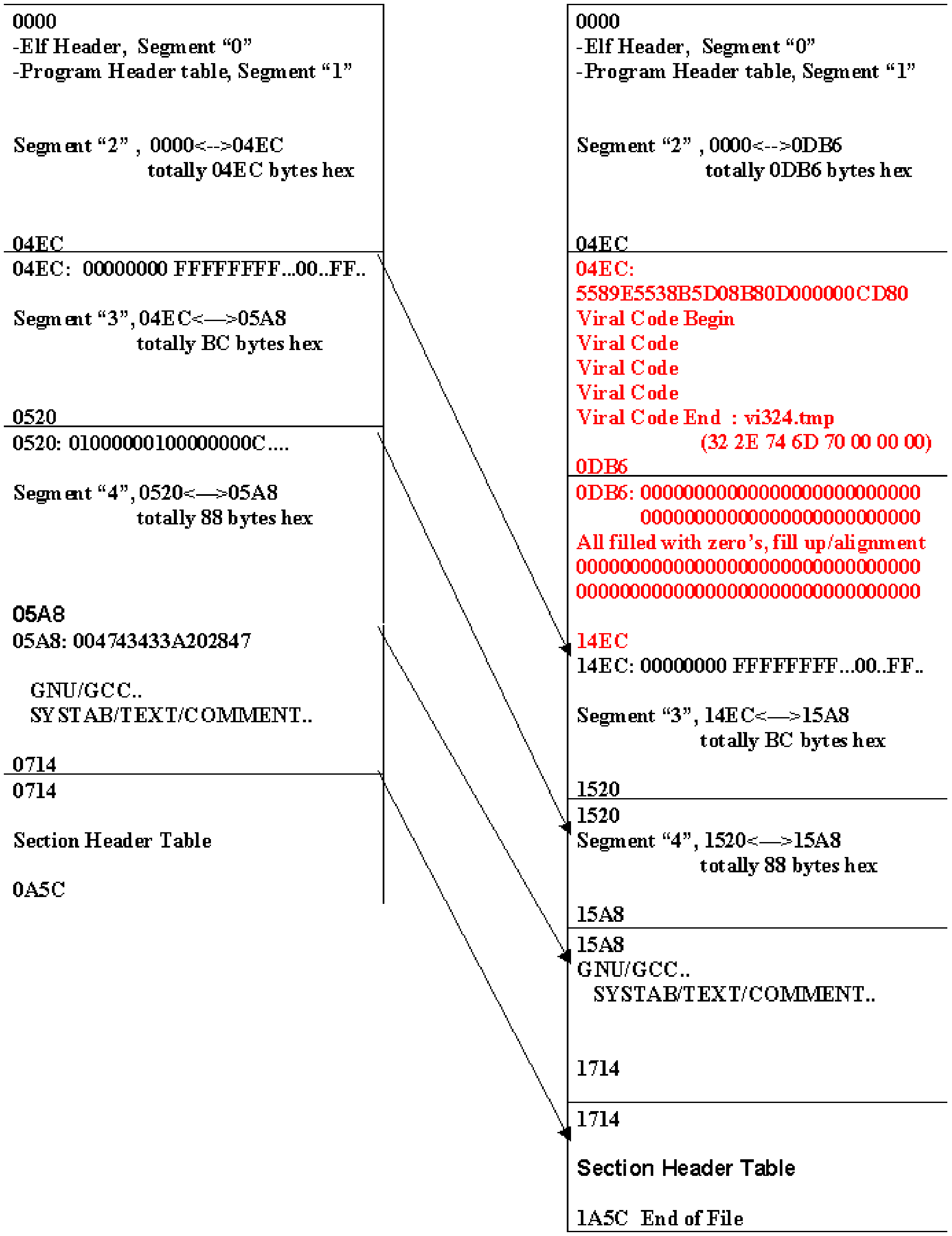
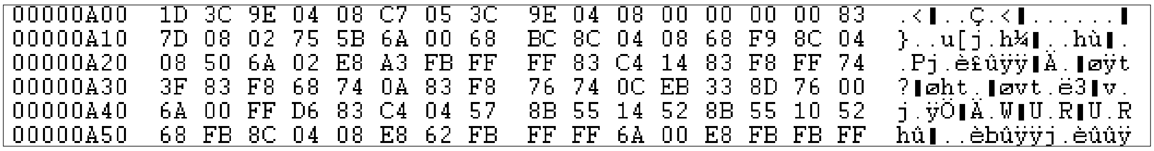


Figure 14d: Lin/Vit infection scheme

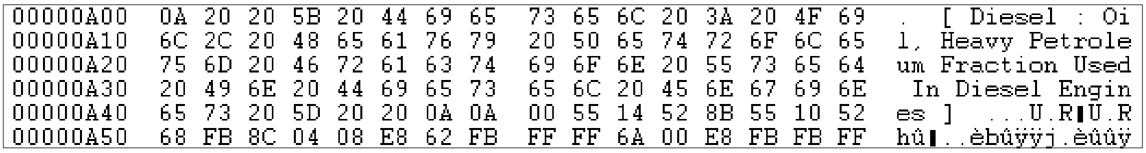
The original bytes in the host file that got replaced/overwritten are appended at the end of the file (after the original file end, therefore following the Section Header Table). A summary of the infection scheme adopted by Lin/Diesel is shown in Figure 15c overleaf.

Clean base name:



```
00000A00  1D 3C 9E 04 08 C7 05 3C  9E 04 08 00 00 00 00 83  .<▌..Ç.<▌......▌
00000A10  7D 08 02 75 5B 6A 00 68  BC 8C 04 08 68 F9 8C 04  }..u[j.h¼▌..hù▌.
00000A20  08 50 6A 02 E8 A3 FB FF  FF 83 C4 14 83 F8 FF 74  .Pj.è£ûÿÿ▌Ä.▌øÿt
00000A30  3F 83 F8 68 74 0A 83 F8  76 74 0C EB 33 8D 76 00  ?▌øht.▌øvt.ë3▌v.
00000A40  6A 00 FF D6 83 C4 04 57  8B 55 14 52 8B 55 10 52  j.ÿÖ▌Ä.W▌U.R▌U.R
00000A50  68 FB 8C 04 08 E8 62 FB  FF FF 6A 00 E8 FB FB FF  hû▌..èbûÿÿj.èûûÿ
```

Infected base name:



```
00000A00  0A 20 20 5B 20 44 69 65  73 65 6C 20 3A 20 4F 69  .  [ Diesel : Oi
00000A10  6C 2C 20 48 65 61 76 79  20 50 65 74 72 6F 6C 65  l, Heavy Petrole
00000A20  75 6D 20 46 72 61 63 74  69 6F 6E 20 55 73 65 64  um Fraction Used
00000A30  20 49 6E 20 44 69 65 73  65 6C 20 45 6E 67 69 6E   In Diesel Engin
00000A40  65 73 20 5D 20 20 0A 0A  00 55 14 52 8B 55 10 52  es ]  ...U.R▌U.R
00000A50  68 FB 8C 04 08 E8 62 FB  FF FF 6A 00 E8 FB FB FF  hû▌..èbûÿÿj.èûûÿ
```

Figure 15b: Lin/Diesel – end of viral code

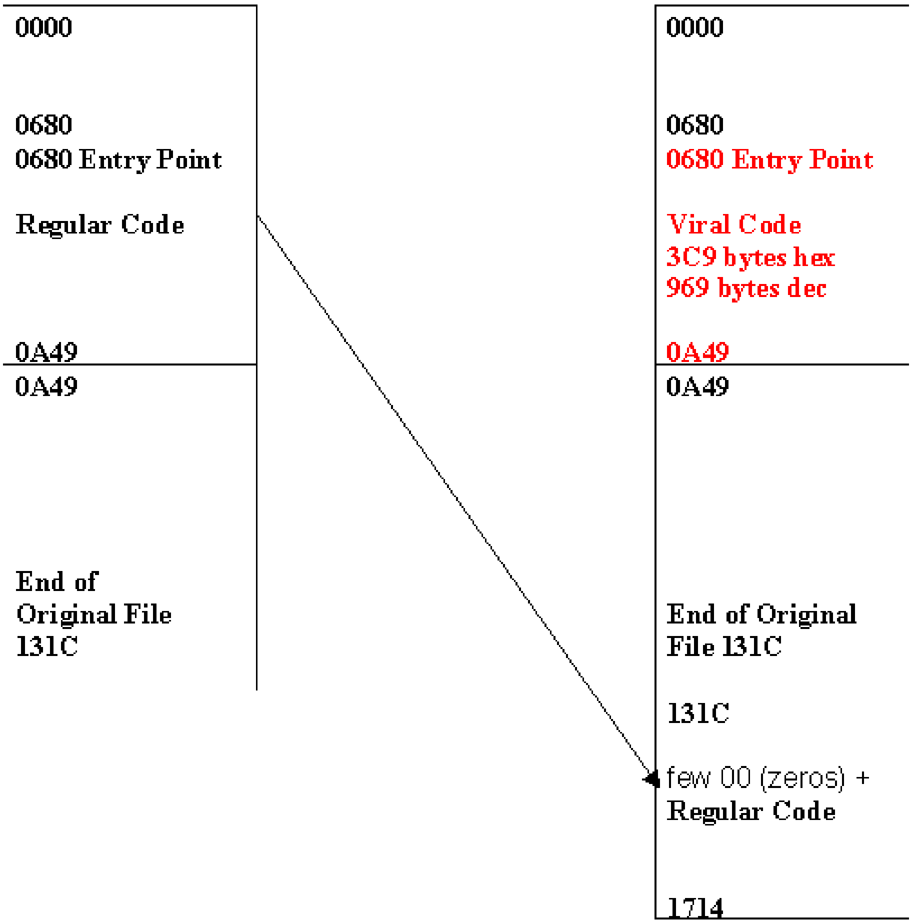


Figure 15c: The Lin/Diesel infection scheme

3 SUMMARY AND CONCLUSIONS

Linux virus techniques:

- Prepending viral code
- Appending viral code
- Adding a section
- Increasing an existing section

Things to consider:

- Replication might be OS (*RedHat* in this case) version/kernel-dependent
- Searching E_Entry in the Section Header Table then determining its file offset does not always work. Remember, the Section Header Table is not needed for Execution viewing.

Conclusions:

- Documented ELF file format might increase virus risk
- Native ELF *Linux* viruses are technically possible
- *Linux* viruses could become an issue of increased importance, as the popularity of the *Linux* OSes increases.

5 REFERENCES

• Full documentation on the ELF layout is available at various locations on-line. For example, http://suncite.unc.edu/pub/*Linux*/GCC/ELF.doc.tar.gz

• A lot of good information on gdb is available in the following book:
'Using GDB: A guide to the GNU Source-Level Debugger', Richard M. Stallman and Roland H. Pesch. The book is also available on-line.