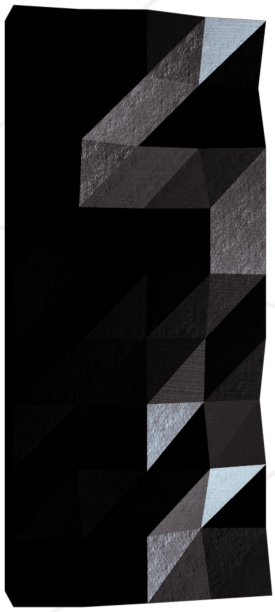

Project UNIX

Death

Summary: In this project, you will code your last "metamorphic" virus.

Version: 3

Contents

Chapter I

Preamble

Chapter II

Introduction

If you take this project, on behalf of the whole staff, we want to congratulate you. You have achieved "great things", but you still have to take the ultimate challenge of your virologist training.

After the oligomorphy with `Pestilence` and the polymorphy with `War`, you will start a journey along the metamorphic path. A metamorphic virus' main characteristic lies in its capacity to modify its internal structure as well as the instructions that compose it.

Metamorphism is a defense mechanism used by different programs to go unnoticed by control and protection programs such as antivirus software.

A metamorphic virus contains neither decoder nor constant viral body, but it can create new generations of itself for each replication while preventing one generation from resembling the previous one.

Chapter III

Objectives

Now, let the fun part begin! Thanks to the virus you've already developed, you have discovered obfuscation methods with polymorphic programming... But you have some funny things left to discover. For this project, you're going to have to think out of the box. The methods you've used so far to find solutions to a given problematic are about to be seriously challenged.

Indeed, adding to the functions you've implemented to the previous projects, this project will require that you create a code which internal structure will partially develop. In order to do so, you will have to use the machines language. You will just have to develop one last program with several proprieties you've already learnt. Mais we're gonna make our program far more interesting.

In regards of its difficulty, this project is not made for anyone. If you're willing to take it, you will need motivation et should never give up. The result is very gratifying :)

Chapter IV

Mandatory part

`Death` is a binary of your own design that will have to:

- like `Famine`, infect binaries present in 2 different specific folders and apply its signature without altering the way said binary works.
- like `Pestilence`, not trigger the infection process if a targeted process is being executed, if the program is launched from any debugger and present a part of the infection routine in an obfuscated manner.
- like `War`, carry a FINGERPRINT developing with each new infection routine execution.

The signature will have to look remotely like this:

```
D34TH version 1.0 (c)oded by <first-login> - <second-login> - [FINGERPRINT]
```

Now, on with the crazy part of this subject. Take all the time you need to read this part again and again: you must make sure your program will never be `structurally` the same, once its execution is done.

In order to do that, you will just have learn to redevelop your virus so that the infectious part can develop. You should be able to infect 2 identical files that will show a real difference in their machine code. Of course, the infected files should be able to run flawlessly.

General instructions:

- The exe will be named `Death`.
- This exe will be coded as an assembler, in C or C++. Nothing else. Java will be tolerated for this project and this is it.
- Your program will not display anything on the standard out or error port.
- You **WILL HAVE TO** work in a VM.

- As usual, you're free to choose the targeted OS. Nevertheless, you will have to set an appropriate VM during the evaluation.
- Your program will have to act on folders /tmp/test and /tmp/test2 - and these folders only - or the equivalent, depending on your targeted OS.You're responsible for the propagation of your program.
- **WARNING!** One and only one infection will be possible on the chosen binary.
- Infections will start on binaries of your OS type built on a 64 bits architecture.

Chapter V

Examples of use

Here is an example of use:

Let's lay the foundation:

```
# ls -al ~/Death
total 736
drwxr-xr-x 3 root root 4096 May 24 08:03 .
drwxr-xr-x 5 root root 4096 May 24 07:32 ..
-rwxr-xr-x 1 root root 744284 May 24 08:03 Death
```

We creat a .c sample for our tests:

```
# nl sample.c
1 #include <stdio.h>
2 int
3 main(void) {
4       printf("This is hard...\n");
5       return 0;
6 }
# gcc -m64 ~/Virus/sample/sample.c
#
```

We copy our binaries (tests + ls) for our tests:

```
# cp ~/Virus/sample/sample /tmp/test2/.
# ls -al /tmp/test
total 16
drwxr-xr-x 2 root root 4096 May 24 08:07 .
drwxrwxrwt 13 root root 4096 May 24 08:08 ..
-rwxr-xr-x 1 root root 6712 May 24 08:11 sample
# /tmp/test/sample
Hello, World!
# file /tmp/test/sample
/tmp/test/sample: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
      /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=938[...]10b, not stripped
# strings /tmp/test/sample | grep "wandre"
# cp /bin/ls /tmp/test2/
# ls -al /tmp/test2
total 132
drwxr-xr-x 2 root root  4096 May 24 08:11 .
drwxrwxrwt 14 root root 4096 May 24 08:11 ..
-rwxr-xr-x 1 root root 126480 May 24 08:12 ls
# file /tmp/test2/ls
/ tmp/test2/ls: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /
     lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=67e[...]281, stripped
#
```

We launch `Death` without the "test" process and watch the result:

```
# pgrep "test"
# ./Death
# strings /tmp/test/sample | grep "wandre"
virus.custom version 1.3 (c)oded may-2017 by wandre - 42424242
# /tmp/test/sample
This is hard..
# strings /tmp/test2/ls | grep "wandre"
virus.custom version 1.3 (c)oded may-2017 by wandre - 43434343
# /tmp/test2/ls -la /tmp/test2/
total 132
drwxr-xr-x 2 root root  4096 May 4 12:13 .
drwxrwxrwt 14 root root 4096 May 4 12:11 ..
-rwxr-xr-x 1 root root xxxxxx May 4 12:19 ls
# gcc -m64 ~/Virus/sample/sample.c -o /tmp/test/sample
# ls -al /tmp/test
total 16
drwxr-xr-x 2 root root 4096 May 4 12:13 .
drwxrwxrwt 13 root root 4096 May 4 12:11 ..
-rwxr-xr-x 1 root root xxxx May 4 12:19 sample
# /tmp/test/sample
This is hard..
# file /tmp/test/sample
/tmp/test/sample: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
      /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=
     xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, not stripped
# strings /tmp/test/sample | grep "wandre"
# /tmp/test2/ls -la /tmp/test2/
total 132
drwxr-xr-x 2 root root  4096 May 3 12:03 .
drwxrwxrwt 14 root root 4096 May 3 12:01 ..
-rwxr-xr-x 1 root root xxxxxx May 3 12:20 ls
# strings /tmp/test/sample | grep "wandre"
virus.custom version 1.3 (c)oded may-2017 by wandre - 444444
#
```

We see very clearly the development of the "signature". Of course, you will have to make sure that the modification is clearly visible. The signature part that is supposed to develop must be controlled. If it is random, so will be your grading. Remember: a double infection should not occur. For this part, I suggest you develop your logic as an assembler. You might get some results with some C/C++ methods, but I believe it will seriously impair this project achievement.

We launch `Death` with the "test" process along with the initial environment and watch the result:

```
# pgrep "test"
57452
# ./Death
# strings /tmp/test/sample | grep "wandre"
# /tmp/test/sample
This is hard..
# strings /tmp/test2/ls | grep "wandre"
# /tmp/test2/ls -la /tmp/test2/
total 132
drwxr-xr-x 2 root root  4096 May 4 12:03 .
drwxrwxrwt 14 root root 4096 May 4 12:01 ..
-rwxr-xr-x 1 root root xxxxxx May 4 12:21 ls
# gcc -m64 ~/Virus/sample/sample.c -o /tmp/test/sample
# ls -al /tmp/test
total 16
drwxr-xr-x 2 root root 4096 May 3 12:03 .
drwxrwxrwt 13 root root 4096 May 3 12:01 ..
-rwxr-xr-x 1 root root xxxx May 3 12:21 sample
# /tmp/test/sample
This is hard..
# file /tmp/test/sample
/tmp/test/sample: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
      /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=
     xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, not stripped
# strings /tmp/test/sample | grep "wandre"
# /tmp/test2/ls -la /tmp/test2/
total 132
drwxr-xr-x 2 root root  4096 May 3 12:03 .
drwxrwxrwt 14 root root 4096 May 3 12:01 ..
-rwxr-xr-x 1 root root xxxxxx May 3 12:23 ls
# strings /tmp/test/sample | grep "wandre"
#
```

We're gonna try to launch `Death` in the debugger. I'm gonna use gdb. I wrote a little note to make it clearer:

```
# gdb -q ./Death
(gdb) run
Starting program: /root/Death
DEBUGGING..
[Inferior 1 (process 2683) exited with code 01]
# strings /tmp/test/sample | grep "wandre"
# /tmp/test/sample
This is hard..
# strings /tmp/test2/ls | grep "wandre"
#
```

And now, on with the metamorphic part. We're just gonna take 2 identical samples we will naturally name sample1 and sample2 and make a diff to counter the number of differences with a simple diff. For the sake of clarity, I'm gonna make it as easy to read as possible.

```
# nl sample.c
1 #include <stdio.h>
2 int
3 main(void) {
4       printf("This is hard..\n");
5       return 0;
6 }
# gcc -m64 ~/Virus/sample/sample.c -o /tmp/test/sample1
# cp /tmp/test/sample1 /tmp/test/sample2
# /tmp/test/sample1
This is hard..
# /tmp/test/sample2
This is hard..
# objdump -D /tmp/test/sample1 > samp ; objdump -D /tmp/test/sample2 > samp2 ; diff -y --suppress-
    common-lines samp samp2 | grep '^' | wc -l
1
```

You'll notice no difference here, except for the program's name. The diff will only show you the different number of lines between each binary on the machine code level. We're gonna launch the virus and watch the results.

```
# ./Death
# /tmp/test/sample1
This is hard..
# /tmp/test/sample2
This is hard..
# strings /tmp/test/sample1 | grep "wandre"
virus.custom version 1.3 (c)oded may-2017 by wandre - 41414141
# strings /tmp/test/sample2 | grep "wandre"
virus.custom version 1.3 (c)oded may-2017 by wandre - 42424242
# objdump -D /tmp/test/sample1 > samp ; objdump -D /tmp/test/sample2 > samp2 ; diff -y --suppress-
    common-lines samp samp2 | grep '^' | wc -l
165
#
```

No need to go any deeper. You get the drill. You've got some work ahead of you.

Chapter VI

Bonus part

Bonus will be taken into account only if the mandatory part is PERFECT. PERFECT meaning it is completed, that its behavior cannot be faulted, even because of the slightest mistake, improper use, etc... Practically, it means that if the mandatory part is not validated, none of the bonus will be taken in consideration.

Bonus ideas:

- Being able to infect 32 bits binaries.
- Being able to infect all the files starting from your OS root in a recursive way.

You will optimize this part executing infected binaries.

- Allowing infection on non-binary files.
- Using packing type methods directly on the virus to make binary as light as possible.
- Adding a total metamorphism to you program (judging from the difficulty of the task, this will validate the whole bonus section).

Chapter VII

Turn-in and peer-evaluation

- This project will not only be reviewed by humans. You're free to organize and name your files as you will. You will just have to respect the following instructions.
- The routine making your program "polymorphic" will be controlled. This is very important.
- You will have to explain your metamorphic part.
- You must manage errors a reasonable way. Your program should never quit unexpectedly (segmentation fault, etc).
- As usual, turn in your work on your repo `GiT`. Only the work included on your repo will be reviewed during the evaluation.
- During evaluation, you must be in a VM. For your information, the grading scale was built with a stable 64 bits Debian 7.0.
- You can use anything you will need except libraries that will do the dirty work for you. This would be considered cheating.
- You can post your questions on the forum, Jabber, IRC, Slack...