CodeBreakers Magazine
Security & Anti-Security - Attack & Defense

Volume 1, Issue 2, 2006

Everything is Byte

+Mala
May 2006

Abstract

Everything is byte. Of course, this won't sound SO strange to most of you. After all, everything which resides on a computer's HD, whether it is a sound, a movie or this plain text file, must be first converted to binary format. This takes us to some less obvious considerations: if everything shares the same format, why do I run some files while I play others? Can I read an executable? Can I listen to an image? The answers are, respectively, because there is something which tells the system what to do; of course you can!

This disclaimer is not meant to sidestep the responsibility for the material we will share with you, but rather is designed to emphasize the purpose of this CodeBreakers Magazine feature, which is to provide information for your own purposes. The subjects presented have been chosen for their educational value. The information contained herein consists of Secure Software Engineering, Software Security Engineering, Software Management, Security Analysis, Algorithms, Virus-Research, Software-Protection and Reverse Code Engineering, Cryptanalysis, White Hat and Black Hat Content, and is derived from authors of academically institutions, commercials, organizations, as well as private persons. The information should not be considered to be completely error-free or to include all relevant information; nor should it be used as an exclusive basis for decision-making. The user understands and accepts that if CodeBreakers Magazine were to accept the risk of harm to the user from use of this information, it would not be able to make the information available because the cost to cover the risk of harms to all users would be too great. Thus, use of the information is strictly voluntary and at the users sole risk.

The information contained herein is not a license, either expressly or impliedly, to any intellectual property owned or controlled by any of the authors or developers of CodeBreakers Magazine. The information contained herein is provided on an "AS IS" basis and to the maximum extent permitted by applicable law, this information is provided AS IS AND WITH ALL FAULTS, and the authors and developers of CodeBreakers Magazine hereby disclaim all other warranties and conditions, either express, implied or statutory, including, but not limited to, any (if any) implied warranties, duties or conditions of merchantability, of fitness for a particular purpose, of accuracy or completeness of responses, of results, of workmanlike effort, of lack of viruses, and of lack of negligence, all with regard to the contribution.

ALSO, THERE IS NO WARRANTY OR CONDITION OF TITLE, QUIET ENJOYMENT, QUIET POSSESSION, CORRESPONDENCE TO DESCRIPTION OR NON-INFRINGEMENT WITH REGARD TO CODEBREAKERS MAGAZINE.

IN NO EVENT WILL ANY AUTHOR OR DEVELOPER OF CodeBreakers Magazine BE LIABLE TO ANY OTHER PARTY FOR THE COST OF PROCURING SUBSTITUTE GOODS OR SERVICES, LOST PROFITS, LOSS OF USE, LOSS OF DATA, OR ANY INCIDENTAL, CONSEQUENTIAL, DIRECT, INDIRECT, OR PUNITIVE OR SPECIAL DAMAGES WHETHER UNDER CONTRACT, TORT, WARRANTY, OR OTHERWISE, ARISING IN ANY WAY OUT OF THIS OR ANY OTHER AGREEMENT RELATING TO CODEBREAKERS MAGAZINE, WHETHER OR NOT SUCH PARTY HAD ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGE.

1 Foreword

Some of you have waited much, much time to read this file; many, on the other side, just didn't mind. For the first ones, I must apologize and say that I have no excuses - I just lost my will to finish this file and spent months while trying to find it again. For the others, I hope that this text will teach you something you didn't know and that maybe you will be with the first ones next time.

One more thing: the idea of patching an exe with an image editor is NOT mine. I just found it on the web some months ago and I really don't remember the address... but I wanted you to know that someone else had the original idea. If you happen to find that page, let me know and I will add that address to this tutorial.

2 Everything Is Byte

Everything is byte. Of course, this won't sound SO strange to most of you. After all, everything which resides on a computer's HD, whether it is a sound, a movie or this plain text file, must be first converted to binary format. This takes us to some less obvious considerations: if everything shares the same format, why do I run some files while I play others? Can I read an executable? Can I listen to an image? The answers are, respectively, because there is something which tells the system what to do; of course you can; of course you can... have you ever tried to write, at the Linux prompt:

```
cat /usr/bin/netscape >/dev/dsp
```

3 Structure And Chaos: Headers Vs. Extensions

Now, what makes the OS understand what kind of file it is dealing with? Well, there are different ways: the buggiest one, for instance, is just looking at the file extension; a better one, instead, is giving a look at the file header or at a particular sequence of bytes which (almost always) exactly identifies the file type. Guessing which one is used by Windows is left as an exercise to the reader ;)

Just look at this example: on your windows disk (if you have one), find all the files that have ".jar" extension; then, copy one of them in another place and rename it as ".zip". Now double click on it and - voila' – it opens correctly as a ZIP file! Then, copy a file called c:\windows\system\shdoclc.dll in another place and rename it as shdoclc.html. If you double click it (I cannot assure it won't damage your system, some strange things will happen!

4 Why? What happened?

In the first case, JAR files are nothing more than ZIP files with another extension: so, since Windows recognizes files according to their extensions, it's not able to open it unless you change its filename in whatever.zip. In the second case, shdoclc.dll contains some html code to generate different pages, but is NOT an HTML document: it's an executable, so if you open it as a .html you will see some strange codes... and a strange browser behaviour, since it's parsing the content of different html pages all pasted together.

As you may understand, this method is quite buggy, because it doesn't let you really understand what you're working with. The worst case happens when some viruses copy themselves by mail as an attachment with a double extension (such as .txt.com or .mp3.pif): if you've left the "hide extension for know filetypes" option active, you might not notice they're executables and run them with a double click. In other cases, instead, this limitation in extension recognition may be useful for us, as you'll see later.

What can you do to be sure you're correctly identifying a file? Even if in some cases you CAN'T be sure, you can have better chances by using some tools called "file analyzers", running both under Windows and Linux. While under Linux you have the great FILE command, which will be described in detail in the next section.

5 Findig Structure in a File

5.1 The FILE Command

"file" (yes, the right command name is all lowercase!) is a great Unix file analyzer which, instead of just looking at name extensions, does various tests on filesystem, file data and (if data is text) language. The "data" test is the most interesting for us: during this test, FILE searches for particular data sequences (called "magic numbers") inside the files to understand their type. Even if it isn't perfect, it's still a very good tool and because of its structure it's the best one if you want to learn how file recognition works. If you type "man file", but still better "man magic", you will easily learn to look inside the configuration file (called "magic") and understand file types even without using any program!

Magic file format (under my Debian it's located in /usr/share/misc/magic, but you can even find it online searching Google for "/usr/share/misc/magic" AND "177ELF") is quite easy: every line is made up of the following fields

OFFSET
This is a number specifying the offset, in bytes, of the data which is to be tested inside the file. It can be preceded by one or more ">", which indicate the level of the test: if there are no ">" the test is level 0 and only if it succeeds tests of level 1 (one ">") are evaluated, followed by level 2 tests (">>") and so on. In a test whose level is higher than 1 you can also find the character "&" before the offset: this means you don't have to consider the offset as absolute, but as relative to the offset of the preceding higher level test.

Here's a little example: if you give a look at ELF section inside magic file, you'll be presented with the following

```
  0      string      177ELF     ELF
  >4     byte        0          invalid class
  >4     byte        1          32-bit
  ...
```

NOTE: 177 is the value of a byte IN OCTAL (0x7F, 127 dec).

This means: if the file starts with byte 0x7F, followed by the string "ELF", then it's an ELF file; then, if at offset 4 it has a byte whose value is 1, then it's a 32-bit ELF, while if it's 0 it's an invalid class ELF file.

TYPE
From the previous example you have seen how the TYPE field is used yet: it just contains the type of the data to be tested. The possible values are

```
byte:
A one-byte value

string:
A string of bytes

short, beshort, leshort:
A two-byte value (on most systems) in this
machine's native byte order, in big-endian order
(be-) ora in little-endian order

(le-)long, belong, lelong:
A four-byte value (on most systems) in this
machine's native byte order, in big-endian
order(be-) ora in little-endian order (le-)

date, bedate, ledate:
A four-byte value (on most systems) in this
machine's native byte order, in big-endian order
(be-) ora in little-endian order (le-), which is
interpreted as a UNIX date
```

TEST
This is the value to be compared with the value from the file. If the type is numeric, the value is specified in C form; if it's a string, it is specified as a C string with the usual escapes permitted (such as n for new line). On the test value, depending on its type, you can apply some operators, such as =, (which work for numbers and strings), & and ^ (AND and NOT, which work only with numbers and require some bits to be set or not). Give a look at the man page for a more detailed explanation.

MESSAGE
This is the message to be printed if the test succeeds. If the string contains a printf format specification (such as "%s"), the value from the file is printed using the message as the format string.

Here are just some of the things that you could notice after reading the magic file:

First of all, there is in fact something that lets your computer know what file it's dealing with: data themselves, with particular values and in particular positions inside the file, can identify the file type and let you discover many other info (just see all that ">>" stuff).

In most of the cases, the identifying bytes are at the beginning of the file, but sometimes important information is NOT necessarily in the header. And that would not be so interesting if it wasn't real for ZIP files too...

6 About Zip Files

The most interesting detail about ZIP files is that they keep information about their packed files in the LAST bytes of the zipped archive: this means that you can add whatever you want or make slight changes at their beginning and many programs, such as Winzip under Windows or unzip under Linux, will open them without any problems. Note that FILE utility, instead, will not recognize them anymore: the line

```
0  string   PK 03 04      Zip archive data
```

inside magic means that it just checks for the first four bytes, which means that if you change them with, for instance, "ZZ", FILE won't recognize the zip anymore, while other programs will be able to open it anyway.

After all, this is not a great limit: FILE also gives you the chance to use indirect offsets, which could help in tasks of this kind... and it's always possible to change the sources, so that you can make it recognize offsets starting from wherever you wish and not just from the beginning of the file. This, of course, is left as an exercise to the reader.

6.1 About Image Formats

As you've seen, ZIP utilities don't mind if you append anything at the beginning of their files. On the other side, there are some image formats which don't mind if you append something at their end: this is because, inside their header, image width and height are specified, so all the exceeding bytes are ignored. This works, for instance, for .gif and .jpg images... and joined with ZIP's property this means that you can append a JPG and a ZIP together (the image first, the archive last) and, under Windows, open the first or the second one just by changing the file extension!

6.2 Dumping

If you want to study a file, you should have a tool which lets you open it and dump its contents on the screen in a raw format. A hex editor is a good tool for this purpose, better if it lets you change the visualization mode from hex to ASCII, best if -like Hiew and Biew- it also lets you disassemble the files you open. Another great tool is list.com by Vernon Buerg, which lets you open BIG files of any kind, dump them in ASCII or hex, search very fast for strings and so on, all working in a DOS box in less than 30KB (don't search for the latest, "bloated", win9x versions: I've recently upgraded to v9.6d but v9.0h is still ok for my purposes).

I usually copy list.com in c:\windows\command directory, then run regedit and create the following register key:

```
    HKEY_CLASSES_ROOT*shellOpen with Listcommand
```

with the value:

```
    "c:windowscommandlist.com %1"
```

This allows me to open any unknown-type file with List just by double clicking it, and any other file clicking with the right mouse button and choosing "Open with List".

Once you've found a file dumper which suits your needs, learn to use it and USE it... a lot! After a while you will notice that many patterns occur in files of the same kind and you will be able to easily recognize them. Also, you will learn a lot of interesting, useful things. For instance many viruses put their name or some particular bytes at the beginning of the file they infect, so opening executables with a dumper will let you not only avoid being infected by viruses, but also understand which one is trying to hit you. And, since many Trojan viruses are around in these days, dumping your attachments before opening them is often a good idea.

Did you know that CuteFTP saves your password in clear inside its macros? Well, I know that you can open CuteFTP macro files with ANY text viewer, but this example is just to show you that you should try to open ANY file you find on your hard disk :) So, if you have just forgotten a password you've saved in CuteFTP "FTP site manager" you just have to start recording a macro, connect to the site you wish and then save the macro... you'll end up with a text file like this:

```
Host 123.123.123.123
RemoteDir /home/httpd/mywebsite
LocalDir D:mywebsite
Retry 20
Login Normal
User myusername
Pass mypassword
Connect
```

if you're so unlucky you REALLY have to use Microsoft Word (in most of the cases you don't REALLY have to do it, and if you do you're just stupid, not unlucky) start opening your .doc files: it's really amazing how much useless stuff is stored inside them. If you happened to enable the "quick save my documents" option, you probably have some of them pasted inside the files of some others, or your errors together with your corrections. An example? Here it is!

- Open Microsoft Word (this experiment has been made with Word97)
- Be sure that "quick saving" is activated (it's the Save tab inside the Options window).
- Create a new document and write: "Dear boss, you really stink."
- Now save your document with whichever name you like most (I've used example.doc).
- Now change the text so that it reads:
- "Dear boss, you're really a great man." (don't worry, if you don't feel you're able to write this you can change the text as you wish)
- Save again the document and close Word.
- Open the file with your dumper and think what could happen if your boss receives it.

Also, the file has grown drastically... but I don't think I will spend more time on this subject, I'll rather let you discover all the details alone, leaving you just one suggestion: when your Microsoft Word crashes (I'm sure it will do), making you lose all your last changes, try to open the ~backup files it has left on your hard disk with a dumper and recover most of the text with a cut & paste.

7 Zeroes

As you may have learned while looking at the files which reside on your hard disk, every format has some values which occur in particular places, or more frequently than others. The reason why I've called this section Zeroes is because they're often zeroes... but not always!

For example, text files may have a CR (or CR+LF) about every 80 characters: the line size isn't always the same, but you can suppose you will find some regularity - and in some cases this may let you understand that a file contains text even if it's encrypted (just give a look at a .box Calypso mailbox file and you will understand what I mean). For this kind of tasks, a good knowledge of the ASCII table may be helpful too... but you'll probably read something about it later.

If you happen to study executables or other binary files, instead, you will find that zeroes are widely used: not just as string terminators, but also as padding at the end of PE sections. If you don't understand this, just think that if you have long sequences of identical values they may probably be zeroes. Open, for instance, `c:\windows\system\systray.exe` and see how many padding zeroes are there: I wonder if any virus writer has ever thought of infecting it... it has so much space to use and it's always loaded at start-up.

Well, I'm sorry I cannot find a similar example under Linux... try to biew /usr/bin/vim and see what happens, but I'm afraid you won't find a PE file.

8 PLAYING WITH CHAOS

8.1 Some Basics

Now you know that files are just a bunch of bytes (wow, what a good piece of news!) and the software just understands them the way it wants. Some systems use extensions to recognize file types, others use particular sequences of bytes, but the most interesting things happen when you try to open a file with software which is not designed to handle it. Just one last thing and I'll stop boring you. If you use applications able to handle files in RAW format (which, in fact, is a non-format), you can read them as text files (as you have seen with list), images, sounds, whatever you want them to be.

8.2 Patching With Psp

Ever cracked a program? Well, don't be shy... it happens sometimes. Even if you are one of the lamest ones, and all you've done was running a crack patch, I hope you've AT LEAST understood what you were doing: you know, the program is a sequence of bytes (just to change, I'd say) and by modifying it for even only one byte you can make it do completely different things, such as telling you are a registered user even if you've inserted the wrong registration code.

Of course, the operation of patching a program can be done for many other purposes, like correcting errors or adding new functions to close software which comes compiled and without sources. Usually, to accomplish this task you can use Hex Editors like Hiew and Biew or tools like my old hexpatcher. This time, we will do that using Paint Shop Pro: anyway, PsP is only ONE of the programs you can use – you just need an image editing tool that lets you open images in RAW format.

In this example, we will try to patch a little program, Cruehead's CrackMe v1.0, available at http://3564020356/tutes/crackme.zip. This little Windows application does exactly nothing: it just stays there waiting to be cracked... and since it has a really easy protection, I won't spend much time on it. Just know that there's a regcode check and then a jump that either sends you to the "good guy" piece of code or shows you the "bad guy" message box. For those who wish to experiment with SoftICE, just bpx on messageboxa and when the debugger pops up return from the call you're in: the check and the jumps are just a few lines above the place where you are, at address 401243.

Now, we know from SoftICE that at 401243 in memory there's a jz (0x74) that we want to change with a jmp (0xEB): where can we find it inside the file which is saved on your disk? There are different ways to do it, depending on which tools you have (of course, we suppose you don't have a hex editor):

If you have a disassembler or any other program which shows you data about sections, you can read RVA and Offset and calculate the right offset inside the file:

```
       Offset in file = (Address) - (Imagebase)
       - (RVA) + (Offset of section)
```

For instance, in this case we have these data about CODE section:

```
  Object01: CODE
  RVA:      00001000
  Offset:   00000600
  Size:     00000600
  Flags:    6000020
```

And the Imagebase is 00400000. So, the right offset of instruction at 401243 is

```
  401243-401000+600 = 843 (HEX, of course)
```

If you have neither a disassembler nor a PE viewer, you might try with LIST (I did tell you it was useful!). Just open the executable with it, press ALT+H to view the data as HEX + dump, then hit backslash to search within the dump: just enter "c3 74

07" and BANG! You hit it at the first shot! As you can see, the "74" byte is at 0x843.

If you don't have either a disassembler, a PE viewer, or LIST... well, you might try to use PSP itself. WARNING: this is not easy and it might even become not funny too, if you have to search much data or a very common sequence of bytes. But in this case, fortunately, it's not so hard and you'll also have the chance to find the place where you have to patch inside the image... so I'll show you that technique in just a few lines.

If now you're asking yourselves HOW, in practice, you can use an image editing tool to work on data instead of images, let me explain...

If you want to open a file, preserving its binary information, you have to open it as RAW: i don't know how it's called inside other applications, but all you have to see is a greyscale image, where every pixel matches one byte inside the file you've just opened. Also, since the size of this image is not specified inside the file (_ALL_ the bytes are pixels), you will have to choose a size yourself. A good size to choose is 100 (or 1000, if the file is huge) for width and, for height, anything which multiplied by width gives you the size of the file (or something bigger, we don't mind adding zeroes at the end). In our example, since the size is 12288 bytes, we'll choose width=100 and height=130.

If you want to read what value is one byte (that is, one pixel), just put your "dropper" or "colour picker" tool above that pixel and click. If, instead, you want to change one byte, just choose a size 1 brush (or pencil or whatever, I think everything should be identical at size 1), the colour you want and click on the pixel/byte you want to change.

If you want to search for a sequence of bytes... well, I still don't know if it's possible (probably with GIMP, but definitely not with PSP). The technique I've learned is the following:

- Open the file as RAW and increase its colour depth to 16 million colours
- Use colour replacer to replace one byte (let's say "74") with a bright colour (let's say a red or a green: since the original image was greyscale, anyway, almost every other colour will seem bright). If you're comfortable with image editing tools you might also choose a size higher than 1 for your brush, while if you're comfortable with exe manipulations you can choose to replace colours only in code section, which is quite easy to individuate at a glance
- Now do the same thing with the byte value preceding the one you have chosen before (using another colour, of course). From now on, you can consider only the sequences of two coloured pixels in a row: if you have more than one, repeat this step.
- At the end, remember to close this version of the file: you don't want to deal with a 16 million colours image, but with your good old .exe raw file

NOTE: all the numbers we're dealing with now are hexadecimal, while you'll probably have to deal with decimal values inside your application, so learn to convert them quickly or make SoftICE do that for you with "?" command.

Doing this with crackme.exe will show you the right place in just two steps: you'll be able to see only two coloured pixels in the image, of which the rightmost one is the one you have to change. Its coordinates, provided you opened the image with 100x123 resolution, are (15, 21). If you already had the offset of the byte inside the file, you could have found the right coordinates just by dividing the offset for the width and keeping the result of the integer division as y, the remainder as x. In our case:

```
0x843 = 2115 dec
2115 / 100 = 21 = y
2115 % 100 = 15 = x
```

Now you have the coordinates, you just have to pick up the colour you wish (0xEB = 235) and "paint" the right pixel to patch your program. Save the file as raw and you'll have a cracked crackme.exe!

9 Executable Images

Gosh, when I first started to write this tutorial I had just experimented with this topic, but now so much time passed and I'm afraid I won't remember all the things I had found! Anyway, I'll try to explain them by steps and redo them while I'm writing. I hope it will sound understandable.

The first thing that led me to think about an image which was a running app at the same time was that Psp cracking I was telling you about a few lines above: we have seen that it's possible to view an executable as an image, but of course this image is senseless... is it possible to make a "good" image which is also an executable?

Of course it is, but we have to notice some details before:

- Windows executables have a fixed format (the one I called PE some times inside this text: to have more info about it, just search for a tutorial, there are plenty of them around the net) which requires them to have a header full of data you won't be able to move: this means that the high part of the image will always be full of junk, unless you use something different from PE. This is the reason why I decided to work with .com files, written in ASM and modified by hand
- Even if you create a .com file, it will have to start with some code so at least a few bytes will have to be present at the beginning of the image. It's not a big problem, anyway, since they'll just become small dots in the upper left corner (we'll see how to make them less noticeable later)
- The more flexibility we want for the data to be hidden, the lowest level programming language we have to use. That is, we might be able to write a C program and then fine tune it to work as an image too, but it will be easier for us if the program is written from scratch in ASM.
- Needless to say (you should have learned it from the Psp patching section yet, if you've experimented with it) there's a matching between byte values and greyscale colours: that is, 0x00 will be black while 0xff will become white; an unconditional jump (0xEB) will be quite bright, while a conditional one (0x74,0x75) will be darker; the same way, the longer are your jumps the brighter their offsets will become, but remember that if the jump offset is signed then a near negative jump will be even brighter than a far positive one; and so on... Don't start pulling your hair away from your head, I've just begun with the funny part,

So, we want to write a program, AND to create an image with some sense. The method I'm going to tell you is not formal nor official, it's just the way I created some examples: if you find a better one tell me and I'll be glad to publish it here. Now, let's start.

The first thing I did was to create an ASM program which had a lot of empty space inside it: I thought that few pixels distributed in a huge space would have been less noticeable, especially after inserting a real image between them. So, given a VERY easy ASM code to print a string on the screen:

```
   mov     ah, 09h
    mov     dx, offset id_msg
    int     21h
    ret
    id_msg  db      "Hello world",13,10,"$"
```

Let’s change it this way:

```
    mov     ah, 09h
    jmp     lbl00
    db 1323 dup 0ffh
lbl00:
    mov     dx, offset id_msg
    jmp     lbl01
    db 1323 dup 0ffh
lbl01:
    int     21h
    jmp     lbl02
    db 1323 dup 0ffh
lbl02:
    ret
    id_msg  db      "Hello world",13,10,"$"
```

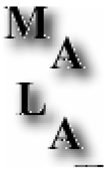
What did I do? Well, I've just put some empty spaces between a command and the following one. Why 1323? It's the number of spaces that will give me a file which is exactly 4000 byte long, and since we'll have to open it inside an image editor, size matters ;) Why 0xFFs and not zeroes? "Marty, you are not thinking quadrimensionally": what we consider zeroes here are ugly black dots inside our beautiful image, while FFs are empty white spaces inside the picture! :)

Now, let's open the file inside Psp: 4000 is 50x80, so let's open the .com as a raw image with this size, using a single colour channel (greyscale): in this way, every byte will be read as a single pixel value. Now we can choose different approaches to hide our data:

- Draw some random noise around the code pixels so that they won't be easily detectable: as an example, see http://3564020356.org/tutes/step01.gif. Of course, this is just a quick and dirty trick you might use just if you run out of fantasy: this will NOT generate an image, but just a bunch of pixels.

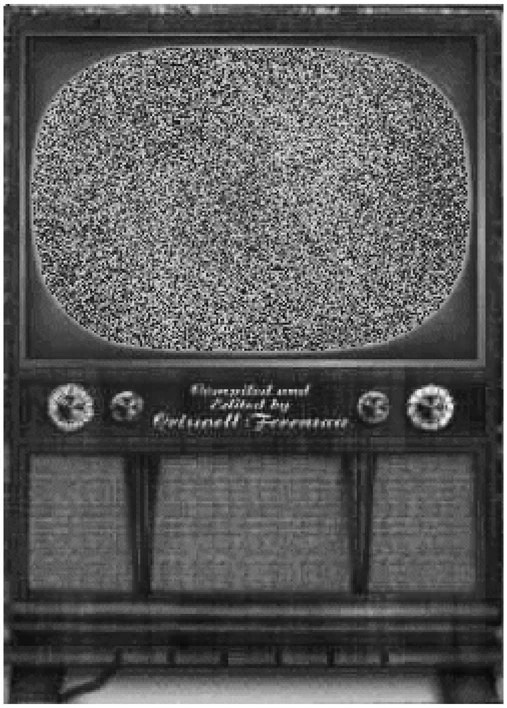
You'll find nothing here!

- try to draw something with sense, but which will not interfere much with the data: http://3564020356.org/images/exegif.gif is a writing I've used to hide the Hello World example you've seen before and, even if not completely (see the line of pixels at the bottom of the image, it's the "Hello World" string), it dissimulates the presence of an executable file quite well.



- Fine tune: this is the harder option you can take, but might be the one with the greatest impact. You should start from a ready-made image and tune the first jump so that you'll land in a place where your following code (and the subsequent jump) will have colours which are similar to the colours of the image. Then, modify the second jump so that the code after it and the third jump will be hidden well inside the image, and so on. Hard, but satisfying: I've used it to create a part of Riddle #4 on my website and a good part of the code is invisible (while I left some other around the image, more or less intentionally)

If you want to experiment a bit with this technique, I suggest you to use the good old "television trick", which is a bit harder than random noise and a bit easier than fine tuning. Its main idea is to put all the "noise content" (that is the code) in a rectangular area and then use it on a TV screen or something similar inside the image: of course, this is not a real way of hiding data and it's not even original (thanx ovid), but at least you have something with sense and having the chance to make a TV say something intelligent is always nice. Even if this is still quite easy, you will have some work to do before you can hide data this way: you'll have to choose a picture before you start, put noise inside an area and then arrange jumps in your program so that the noise generated by the code will stay in the chosen area. A funny example (I've called it "TELEVISION, THE DRUG OF A NATION") can be found here:



http://3564020356.org/pix/misc/silly.gif

Here's how I did it:

- first, I found an image with an old TV and I modified it so that the video contained white noise
- then, I decided a place to put the code which was easy to remember too: I chose to start pasting it at (50,30)
- since I had to jump immediately from the beginning of the code to 50,30 and the image resolution was 239x349, I knew that the offset from the beginning of the file where I had to jump was 249*30+50 = 7520 (0x1d60) and, since the unconditioned jump itself needed 3 bytes (one for the Opcode, two for the offset), I knew that the offset for the jump just had to be 249*30+50-3 = 7517 (0x1d5d)
- the source file for the executable, then, was the following (yes, it's SOEASY! Create a harder one if you can.

```
    jmp     lbl00
    db 7517 dup 0ffh
lbl00:
    mov     ah, 09h
    mov     dx, offset id_msg
    int     21h
    ret
    id_msg  db      "Watch me, you silly
                     slave!",13,10,"$"
```

- Once I assembled the .com file, I just had to open it as a .raw image, with 249*whatever (try a big one, I put 100) resolution, and paste the two pieces of code: respectively, the three pixels at the beginning, the other ones starting at 50x30. I suppose there should be a filter to paste the images in the right way as if they were simple layers, but I chose the secure way

That's all. Nah, I was pulling your leg, there's more: the image is so big that, when saved as a raw .com file, it WILL NOT WORK! You have to keep only a part of the image (for instance, only the screen: ALWAYS starting from the upper left corner, of course!) and then save it to have a working program.

Well, we can say we have some added security, but I think the truth is that there's a bug in everything I've explained until now ;)

10 Ceci n'est pas win.com

At http://3564020356.org/tutes/malawin_en.htm (malawin.htm is the Italian version of the page) you can find a tutorial I wrote in March, 2002. Wow, that's old! I am pointing you there because I think it might be a good example of how the "everything is byte paradigm" can be used in practice: working on good (?) old Win98SE win.com, I created a file which is both a working Windows executable and an image with some sense, then I started to think about how copyright could work in this case.

How can you do the same? Easy: open your win.com file (the version I have is 25175 byte long), as a raw file with height=101 pixels and width=250 pixels and write what you like inside the padding space. You can recognize it quite easily: some black (0x00) stripes where I decided to put the "Ceci n'est pas win.com" sentence, together with a "+malagritte" signature (for the most curious ones, the signature itself is a modification of the original Magritte's one).

This example, in particular, raises some interesting questions you could think about, if you have spare time:

- Is this win.com or not? If you run it, it is... even if it's modified: it should be win.com as much as your cracked version of your favourite application is still your favourite application. But if you save your raw file as a .gif you're using it as a graphical representation of win.com, which carries much more information than the program itself, 'cause you've added it. And what happens if you print it?
- Can you get some useful information from the graphical representation of the file? Sure: first of all, you can easily see where padding space is; also, you can find strings at a glance, because they're just grey stripes inside the image (inside win.com, you can see some at

the beginning and MANY at the end of the file)

- Can you do the same with other files? Sure: as I've written before, PE files have a padding made up of zeroes (or, sometimes, custom strings: try to find all the *.exe files which contain "PADDING" on your C: disk, for instance) at the end of their sections, so you can hide all the data you wish there. Anyway, make sure you've made a backup copy before you change a file, because some of that space might be checked or used by the program and the effects of your experiments are unknown.

11 Acknowledgements

I have to say a big, BIG thank you to all the +friends at 3564020356.org for the fun and the satisfaction they give me every day: you never stop amazing me!

A hug goes to all the friends of the groups I've been (and I'll always be, unless they kick me out ;)) part of: +HCU, RingZer0, RET (people, this one is new, go and give a look at their stuff ^__^). Another big hug goes to all the LOAnians: I haven't put LOA between groups because it's not a group, it's a forma mentis and it's the best (physical or not) place I've been in the last years.

A "bravo" to anyone who had the strength to reach this point. I promise I'll stop exactly NOW.