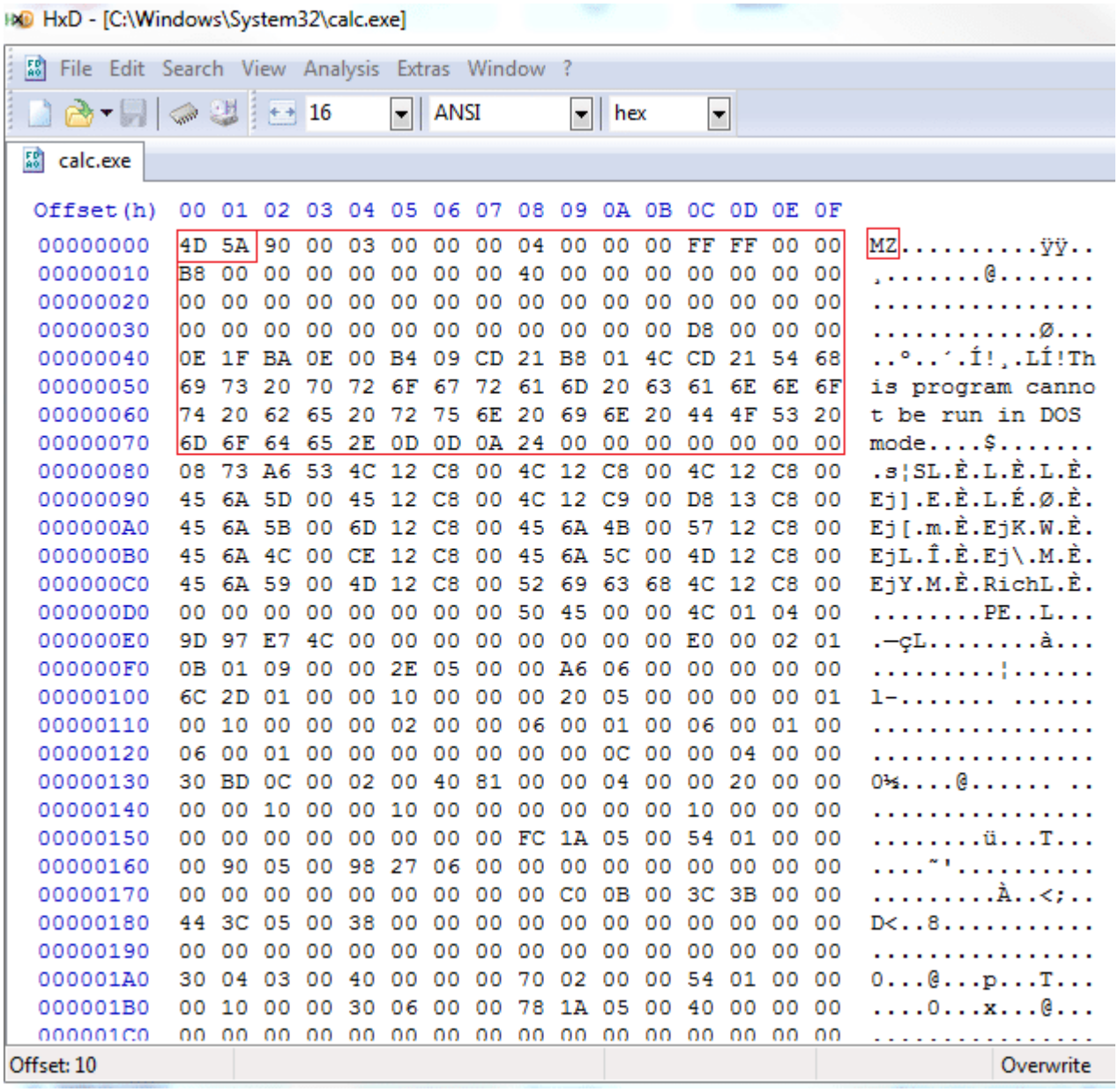
Magic Bytes – Identifying Common File Formats at a Glance

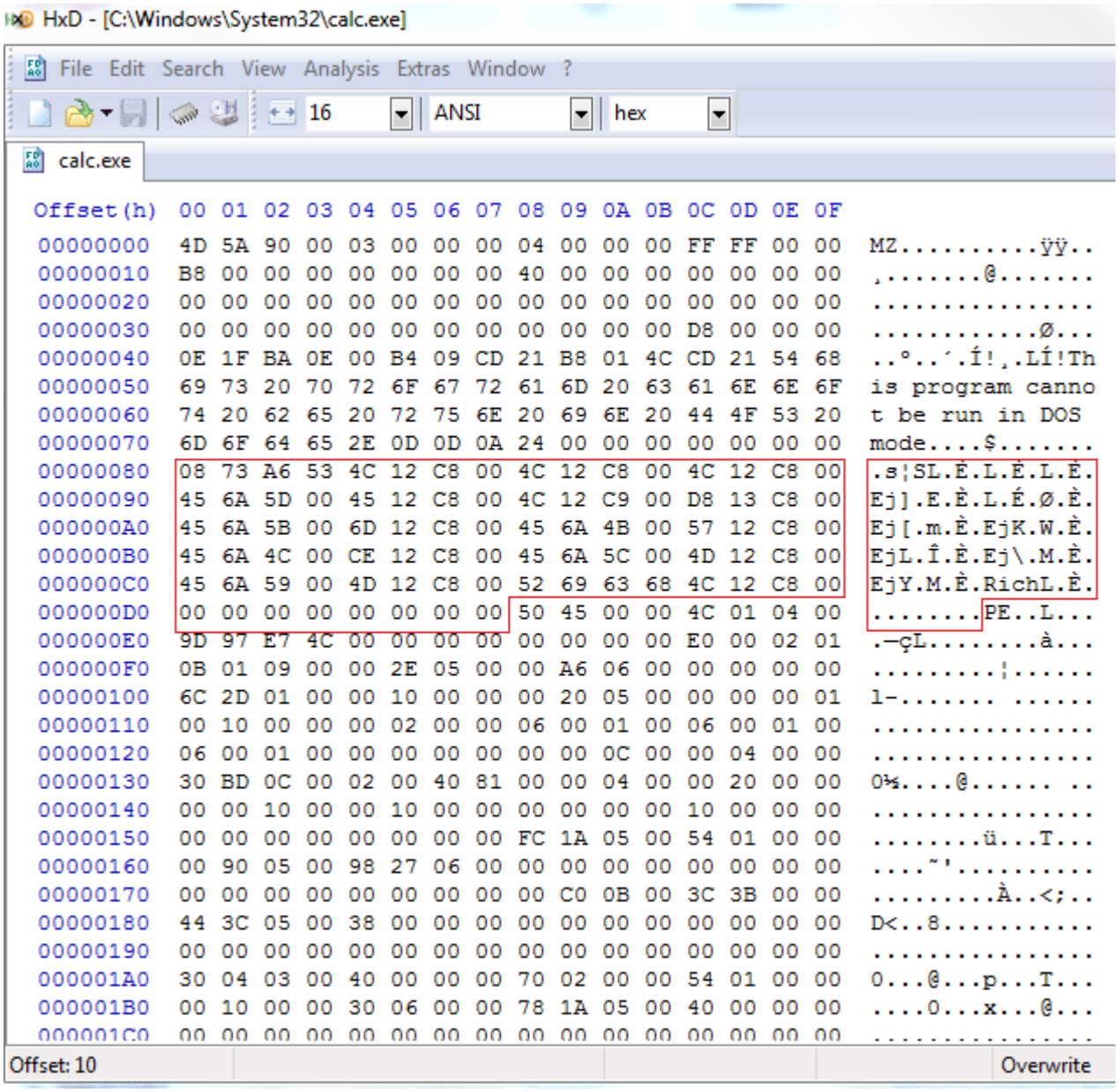
When assessing an application, one may run into files that have strange or unknown extensions or files not readily consumed by applications associated with those extensions. In these cases it can be helpful to look for tell-tale file format signatures and inferring how the application is using them based on these signatures, as well as how these formats may be abused to provoke undefined behavior within the application. To identify these common file format signatures one typically only need look as far as the first few bytes of the file in question. This is what's often called "magic bytes", a term referring to a block of arcane byte values used to designate a filetype in order for applications to be able to detect whether or not the file they plan to parse and consume is of the proper format. The easiest way to inspect the file in question will be to examine it with a hex editor. Personally for this task I prefer HxD for windows or hexdump under Linux, but really any hex editor should do just fine. With a few exceptions file format signatures are located at offset zero and generally occupy the first two to four bytes starting from the offset. Another notable detail is that these initial sequences of bytes are generally not chosen at random; that is most developers of a given format will choose a file signature whose ASCII representation will be fairly recognizable at a glance as well as unique to the format. This allows us to use the known ASCII representations of these signatures as a sort of mnemonic device to quickly identify a given file's format. Here's a few examples of common file signatures and their accompanying mnemonics:

Executable Binaries	**Mnemonic**	**Signature**
DOS Executable	"MZ"	0x4D 0x5A
PE32 Executable	"MZ"...."PE.."	0x4D 0x5A ... 0x50 0x45 0x00 0x00
Mach-O Executable (32 bit)	"FEEDFACE"	0xFE 0xED 0xFA 0xCE
Mach-O Executable (64 bit)	"FEEDFACF"	0xFE 0xED 0xFA 0xCF
ELF Executable	".ELF"	0x7F 0x45 0x4C 0x46
Compressed Archives	**Mnemonic**	**Signature**
Zip Archive	"PK.."	0x50 0x4B 0x03 0x04
Rar Archive	"Rar!...."	0x52 0x61 0x72 0x21 0x1A 0x07 0x01 0x00
Ogg Container	"OggS"	0x4F 0x67 0x67 0x53
Matroska/EBML Container	N/A	0x45 0x1A 0xA3 0xDF
Image File Formats	**Mnemonic**	**Signature**
PNG Image	".PNG...."	0x89 0x50 0x4E 0x47 0x0D 0x0A 0x1A 0x0A
BMP Image	"BM"	0x42 0x4D
GIF Image	"GIF87a"	0x47 0x49 0x46 0x38 0x37 0x61
	"GIF89a"	0x47 0x49 0x46 0x38 0x39 0x61

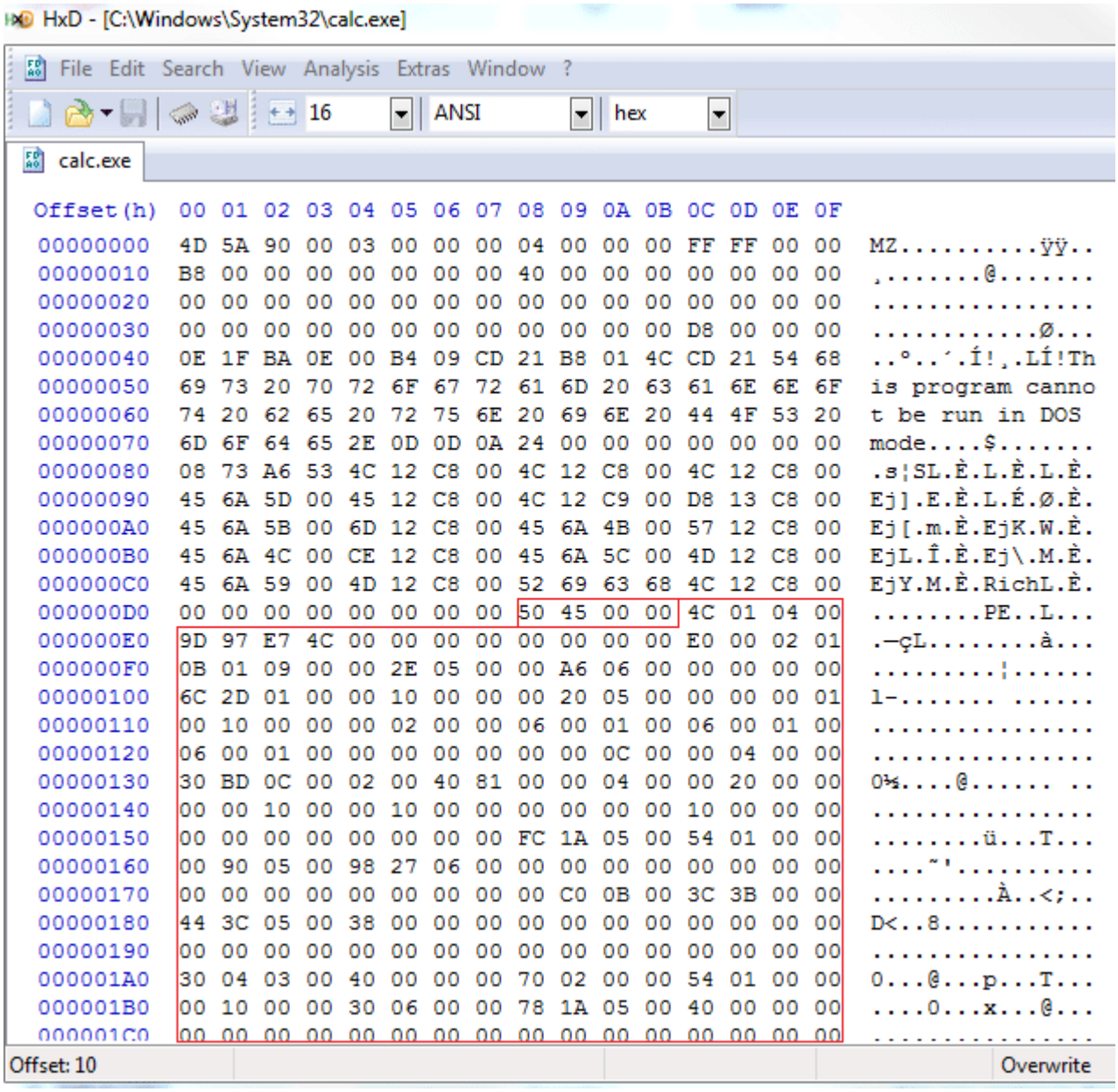
Let's take what we've learned so far and apply it toward an "unknown" file, calc.exe.



To avoid confusion it's worth noting that the PE32 executable format actually contains at minimum two sets of magic bytes: one set for the DOS executable header for DOS system compatibility and the other set to mark the beginning of the PE32 executable header. In this screenshot I've highlighted the DOS header, where we can see that the beginning of said header is marked with "MZ". Another characteristic of the DOS header that's an immediate give-away is the text "This program cannot be run in DOS mode.", which some may recognize as the error text displayed when one attempts to run a windows application in DOS mode.



Following the DOS header and preceding the PE header is what's known as the rich header and is represented in our mnemonic list as the "…" between the DOS and PE magic bytes. This header remains largely undocumented, however, so examining it at length is unlikely to yield any insightful information.



Finally, following the DOS and rich headers comes the PE header marked by “PE..”, or the byte sequence x50x45x00x00 which indicates that this file is a PE32 executable. Identifying other formats will follow the same principle, only one will generally only need the first step of the above process to identify the file format.