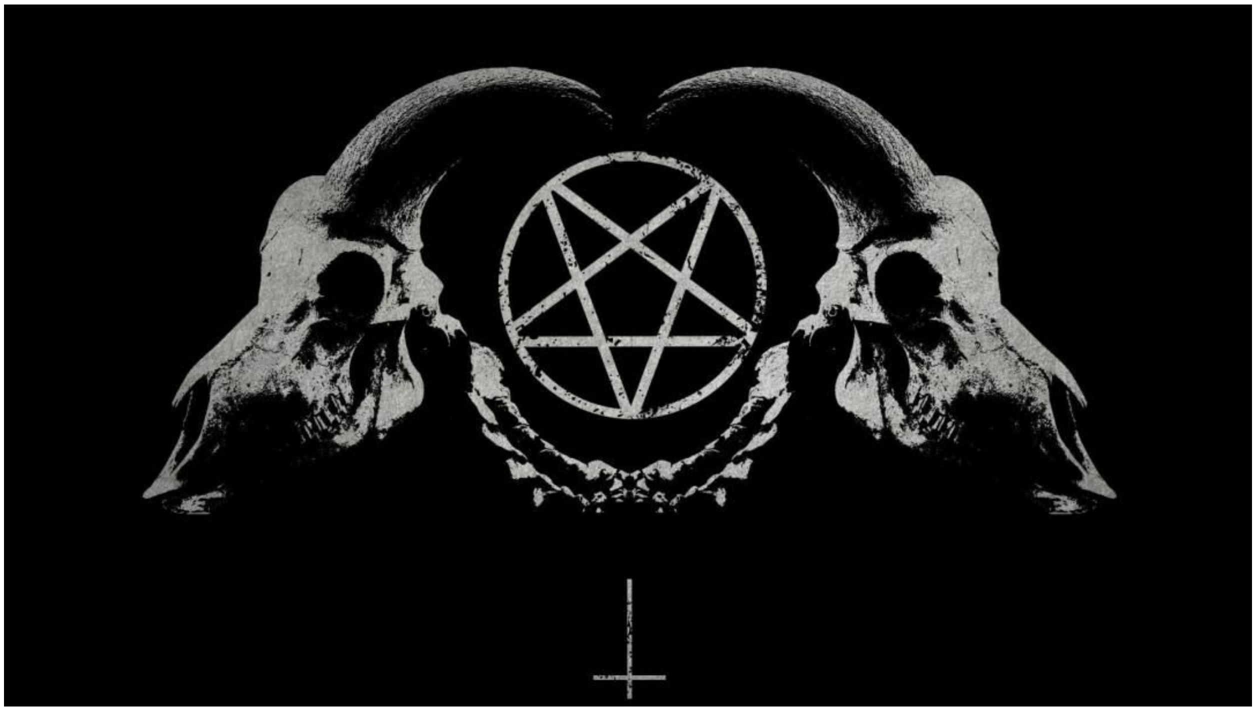
DISASSEMBLERS WITHIN VIRUSES



(x) 2001
xlated in 2002

The more different features (possibilities) our virus has, and the more there are relations and reactions between these features and external environments, the more our virus is alive, and the more complex it is. Here comes to mind such technologies as modularity, portable viruses written in own emulated scripts, worms, distributed networks, and other complex stuff.

In this connexion lots of useful tools, toolkits, include files, articles and libraries, oriented to help others in coding common stuff, are required today.

One of such things is disassembler. It can be used everywhere, and wherever it is used, especially in viruses, it gives good effect -- mostly all good infection- and morphing- related technologies are based on disassembling.

Code analysis and parsing it into single instructions can be done by means of sequential calls to length disassembler. Such disassembler is used in permutation and code integration.

The trivial usage of length-disassembler is hooking functions in memory or, while infecting a file, inserting virus call some commands after the entrypoint.

The code I'm considering is 32-bit x86 binary code. But, the length-disassembler shown here can be easily changed to work with 16-bit.

Now, what will we disassemble, i.e. which kind of information we want to extract from instruction flow. Is only length of instructions enough, or we also want to know something about prefixes, operation code, arguments etc.

Some years ago, in 1997, i didn't knew that universal disassembler is very useful thing, and in ZCME there where the following commands:

```
      inc   cx                  ; 5
      cmp   al, 0EAh
      je    @@exit
      cmp   ax, 3E80h           ; cmp [xxxx], yy
      je    @@exit
      inc   cx                  ; 6
      ...
@@exit:            ; return cx
```

As you can see, this is disassembler for only instructions, which were present in the ZCME virus. Year after i changed the virus, and so, i had to change disassembler too. And then once again, and again, until universal length disassembler (LDE) were written, and then there appeared different LDE modifications for different tasks.

In the example above i only need to call the length-disassembler, and then check current instruction opcode for being EB,E8,E9,7x,0F 8x, and etc., while in more advanced tasks i need to know much more information about instructions -- for example, knowing register usage, it is possible to insert own instructions into the middle of the program's code:

```
mov   eax, [ebx+4]
```

…

```
mov   eax, vir_1
add   vir_2, eax
mov   eax, [ebx+4]
```

Also, while permutation, knowing registers and stack usage is necessary to mix instructions between each other.

In other words, the more advanced our disassembler is, the more our knowledge about instructions is, and as such our imagination is less limited by our possibilities, and the more good things can be done.

In the end of this article there is DISASM.CPP, the source of instruction parser.

It is called as following;

```
int disasm_ok = disasm( &buf[ip] );
```

As a result, 1 is returned if instruction is known, and 0 if some error has occurred.

In case of successful disassembly, disasm function will parse given instruction into the following parts:

```
DWORD disasm_len;         -- total instruction length in bytes, 0 if error

DWORD disasm_flag;        -- bitmask, flags, see C_xxx
   C_66      -- there is 66-prefix
   C_67      -- there is 67-prefix
   C_LOCK    -- there is LOCK-prefix (F0)
   C_REP     -- there is REPZ- or REPNZ-prefix, exact value in disasm_rep
   C_SEG     -- there is seg-prefix, exact value in disasm_seg
   C_OPCODE2 -- there is 2nd opcode (1st one is 0x0F), value in
disasm_opcode2
   C_MODRM   -- there is modrm, value in disasm_modrm
   C_SIB     -- there is sib, value in disasm_sib

DWORD disasm_memsize;     -- length of the memory address,
                             if used in instruction, value in disasm_mem
BYTE  disasm_mem[8];      -- memory address (length in disasm_memsize)

DWORD disasm_datasize;    -- length of data, used in instructions (in
bytes),
                             value in disasm_data
BYTE  disasm_data[8];     -- data (length in disasm_datasize)

BYTE  disasm_seg;         -- C_SEG: seg-prefix (CS DS ES SS FS GS)
BYTE  disasm_rep;         -- C_REP: rep-prefix REPZ/REPNZ
BYTE  disasm_opcode;      -- opcode itself, not depending on flags
BYTE  disasm_opcode2;     -- C_OPCODE2: 2nd opcode (if 1st one is 0x0F)
BYTE  disasm_modrm;       -- C_MODRM: value of modxxxrm
BYTE  disasm_sib;         -- C_SIB: value of sib (scale-index-base)
```

So, assembling instruction from all the stuff listed above, looks as following:

```
if (disasm_flag & C_66)       *outptr++ = 0x66;
if (disasm_flag & C_67)       *outptr++ = 0x67;
if (disasm_flag & C_LOCK)     *outptr++ = 0xF0;
if (disasm_flag & C_REP)      *outptr++ = disasm_rep;
if (disasm_flag & C_SEG)      *outptr++ = disasm_seg;
*outptr++ = disasm_opcode;
if (disasm_flag & C_OPCODE2) *outptr++ = disasm_opcode2;
if (disasm_flag & C_MODRM)   *outptr++ = disasm_modrm;
if (disasm_flag & C_SIB)      *outptr++ = disasm_sib;
for (DWORD i=0; i<disasm_memsize;  i++)  *outptr++ = disasm_mem[i];
for (DWORD i=0; i<disasm_datasize; i++)  *outptr++ = disasm_data[i];
```

Source code can be found here