Examples - Tutorial

Each example you can see here is using one feature of the disassemble library. At first, you can see how to use the library in the simplest manner (to retrieve a simple instructions list). Then, you can see how to use specific options to make special tasks like data-flow analysis or control-flow analysis.

1. How to decode 100 lines of code ?
2. How to decode a limited block of bytes ?
3. How to decode bytes in an allocated buffer while keeping original virtual addresses ?
4. How to use nasm syntax with prefixed numbers ?
5. How to retrieve only instructions that modify the register eax ?
6. How to decode instructions and 'follow' unconditional branch instructions ?
7. How to use BeaEngine with masm32, nasm, fasm or GoAsm ?
8. How to use BeaEngine with masm64 ou GoAsm64 ?
9. How to use BeaEngine with WinDev (by Vincent Roy) ?

1. How to decode

BeaEngine does not need special initialization. The Disasm function do it for you. The only task you have to perform is setting to zero the **_Disasm** structure and filling the field **infos.EIP** (offset where you want to disassemble).

Doing it with Python is the simplest way because of its specific wrapper used to hide ctypes complexity :

```
# Python example

from BeaEnginePython import *
buffer = bytes.fromhex('6202054000443322')
target = Disasm(buffer)
target.read()
print(target.repr())
```

Output is :

```
vpshufb zmm24, zmm31, zmmword ptr [r11+r14+0880h]
```

You can even do a disasm loop on a binary file :

```
# Python example

with open("target.bin", 'rb') as f:
  buffer = f.read()
  instr = Disasm(buffer)
  while instr.read() > 0:
    print(instr.repr())
```

Note : In Python, *infos* structure is reachable with *disasm.infos*.

Let's see how to do it in C :

```
#include <stdio.h>
#include <string.h>
#include "BeaEngine.h"

int main(void)
{
  DISASM infos;
  int len, i = 0;

  (void) memset (&infos, 0, sizeof(DISASM));
  infos.EIP = (UInt64) main;

  while ((infos.Error == 0) && (i < 100)){
    len = Disasm(&infos);
    if (infos.Error != UNKNOWN_OPCODE) {
```

```
      (void) puts(infos.CompleteInstr);
      infos.EIP += len;
      i++;
    }
  }
  return 0;
}
```

2. How to decode a limited block of bytes

It is possible to ask to BeaEngine to decode a limited block of bytes. This small program decodes instructions of its own code located between 2 virtual addresses. That means BeaEngine won't read any bytes outside these limits even if it tries to decode an instruction starting just before the upper bound. To realize this restriction, BeaEngine uses the field **infos.SecurityBlock** which defines the number of bytes we want to read. By default, an intel instruction never exceeds 15 bytes. Thus, only SecurityBlock values below this limit are used. In all cases, BeaEngine stops decoding an instruction if it exceeds 15 bytes.

```
#!/usr/bin/python3

# Python wrapper already handles this feature without any
# specific option

from BeaEnginePython import *

buffer = bytes.fromhex('4831c04889fbffc04989c49031ed66586a005f80c40c')
instr = Disasm(buffer)
while instr.read() > 0:
  print(instr.repr())
```

Let's see how to do it in C :

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "BeaEngine.h"

void DisassembleCode(char *start_offset, int size)
{
  DISASM infos;
  int len;
  char *end_offset = (char*) start_offset + size;

  (void) memset (&infos, 0, sizeof(DISASM));
  infos.EIP = (UInt64) start_offset;

  while (!infos.Error){
    infos.SecurityBlock = (int) end_offset - infos.EIP;
    if (infos.SecurityBlock <= 0 ) break;
    len = Disasm(&infos);
    switch(infos.Error)
      {
        case OUT_OF_BLOCK:
          (void) printf("disasm engine is not allowed to read more memory \n");
          break;
        case UNKNOWN_OPCODE:
          (void) printf("%s\n", &infos.CompleteInstr);
          infos.EIP += 1;
          infos.Error = 0;
          break;
        default:
          (void) printf("%s\n", &infos.CompleteInstr);
          infos.EIP += len;
```

```
      }
  };
  return;
}

int main(void)
{
  /* 1 byte is missing at the end of this buffer */
  char *buffer = "\x90\x90\x90\x90\x90\x0f\x2b";
  DisassembleCode (buffer, strlen(buffer));
  return 0;
}
```

3. How to decode bytes in an allocated buffer while keeping original virtual addresses

This time, we are in a real and usual situation. We want to decode bytes copied in an allocated buffer. However, you want to see original virtual addresses. The following program allocates a buffer with the function malloc , copies in it 200 bytes from the address &main and decodes the buffer :

```
# Python example

from BeaEnginePython import *
buffer = bytes.fromhex('6202054000443322')
instr = Disasm(buffer)
instr.infos.VirtualAddr = 0x401000
while instr.read() > 0:
  print(instr.repr())
```

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "BeaEngine.h"

#define BUFFER_SIZE 200

/*
 * display instructions with correct VA
 */

void DisplayInstr(char *start_offset, char *end_offset, int (*virtual_address)(void))
{
  int len;
  DISASM infos;
  (void) memset (&infos, 0, sizeof(DISASM));
  infos.EIP = (UInt64) start_offset;
  infos.VirtualAddr = (UInt64) virtual_address;

  while (infos.Error == 0){
    infos.SecurityBlock = (int) end_offset - infos.EIP;
    if (infos.SecurityBlock <= 0 ) break;
    len = Disasm(&infos);
    switch(infos.Error)
      {
        case OUT_OF_BLOCK:
          (void) printf("disasm engine is not allowed to read more memory \n");
          break;
        case UNKNOWN_OPCODE:
          (void) printf("unknown opcode");
          infos.EIP += 1;
```

```
          infos.VirtualAddr += 1;
          break;
        default:
          (void) printf("%.16llx %s\n",infos.VirtualAddr, &infos.CompleteInstr);
          infos.EIP += len;
          infos.VirtualAddr += len;
      }
  }
  return;
}

/*
 * main
 */

int main(void)
{
  void *pBuffer;
  pBuffer = malloc(BUFFER_SIZE);
  (void) memcpy (pBuffer, main, BUFFER_SIZE);
  DisplayInstr(pBuffer, (char*) pBuffer + BUFFER_SIZE, main);
  return 0;
}
```

4. How to use nasm syntax with prefixed numbers

BeaEngine is able to use a set of syntaxes : masm, nasm, GoAsm. You can display numbers under two formats : suffixed numbers and prefixed numbers. You can display or not the segment registers used in memory addressing. You can even use a tabulation between mnemonic and first operand.

```
# Python example

from BeaEnginePython import *
buffer = bytes.fromhex('6202054000443322')
instr = Disasm(buffer)
instr.infos.Options = NasmSyntax + PrefixedNumeral
while instr.read() > 0:
  print(instr.repr())
```

In C:

```
void DisplayInstr(char *start_offset, char *end_offset, int (*virtual_address)(void))
{
  int len;
  DISASM infos;
  (void) memset (&infos, 0, sizeof(DISASM));
  infos.EIP = (UInt64) start_offset;
  infos.VirtualAddr = (UInt64) virtual_address;
  infos.Options = Tabulation + NasmSyntax + PrefixedNumeral + ShowSegmentRegs;
  [...]
}
```

5. How to retrieve only instructions that modify the register eax

This is the first example of how to realize a data-flow analysis with BeaEngine. By using infos.Operand1.AccessMode and infos.Operand1.Registers , you can determine for example if the register rax is modified or not by the analyzed instruction. AccessMode allows us to know if the argument is written or only read. Registers let us know if the register is rax. We don't forget that some instructions can modify registers implicitly. We can control that by looking at the field infos.Instruction.ImplicitModifiedRegs .

```python
# Python example

from BeaEnginePython import *
buffer = bytes.fromhex('6202054000443322')
instr = Disasm(buffer)
while instr.read() > 0:
  if instr.modifies("rax"):
    print(instr.repr())
```

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "BeaEngine.h"

/*
 * disasm function to analyze instructions
 */

void DisassembleCode(char *start_offset, char *end_offset, int (*virtual_address)(void))
{
  DISASM infos;
  int len;

  (void) memset (&infos, 0, sizeof(DISASM));
  infos.EIP = (UInt64) start_offset;
  infos.VirtualAddr = (UInt64) virtual_address;

  while (!infos.Error){
    infos.SecurityBlock = (int) end_offset - infos.EIP;
    if (infos.SecurityBlock <= 0 ){
      (void) printf("buffer end reached \n");
      break;
    }
    len = Disasm(&infos);
    switch(infos.Error)
      {
        case OUT_OF_BLOCK:
          (void) printf("disasm engine is not allowed to read more memory \n");
          break;
        case UNKNOWN_OPCODE:
          (void) printf("unknown opcode\n");
          infos.EIP += 1;
          infos.VirtualAddr += 1;
          infos.Error = 0;
          break;
        default:
          /*
            gpr means General Purpose Register
            xxxx.gpr & REG0 means RAX is used
          */
          if (
            ((infos.Operand1.AccessMode == WRITE) && (infos.Operand1.Registers.gpr & REG0)) ||
            ((infos.Operand2.AccessMode == WRITE) && (infos.Operand2.Registers.gpr & REG0)) ||
            (infos.Instruction.ImplicitModifiedRegs.gpr & REG0)
            ) {
            (void) printf("%.16llx %s\n", infos.VirtualAddr, &infos.CompleteInstr);
          }
          infos.EIP += len;
          infos.VirtualAddr += len;
      }
  };
```

```
  return;
}

/*
 *  main
 */

int main(void)
{
  void *pBuffer;
  pBuffer = malloc(300);
  (void) memcpy (pBuffer, main, 300);
  (void) printf("Display only Instructions modifying RAX. \n");
  DisassembleCode (pBuffer, (char*) pBuffer + 300, main);
  return 0;
}
```

6. How to decode instructions and 'follow' unconditional branch instructions

In some cases, unconditional jumps are used in obfuscation mechanisms. This program shows how to eliminate these naugthy jumps by "following" them. To realize that task, we have to use the fields infos.Instruction.BranchType and infos.Instruction.AddrValue. In the next program, I have coded the function RVA2OFFSET just to convert the virtual address pointed by the unconditional jump in a "real" address that can be used by infos.EIP.

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include "BeaEngine.h"

DISASM infos;
int len,i,FileSize;
unsigned char *pBuffer;
int  (*pSourceCode) (void);   /* function pointer */
FILE *FileHandle;

int RVA2OFFSET(int Address, unsigned char *pBuff);

/* ============================================================================*/
/*                                                                             */
/*            eliminate JUMPS  and reorder instructions                        */
/*                                                                             */
/*=============================================================================*/

void DisassembleCodeFilter(unsigned char *StartCodeSection, unsigned char *EndCodeSection, int (Virtual_Address)
{
  (void) memset (&infos, 0, sizeof(DISASM));
  infos.EIP = (UInt64) StartCodeSection;
  infos.VirtualAddr = (UInt64) Virtual_Address;

  /* ============================== Loop for Disasm */
  while (!infos.Error){
    infos.SecurityBlock = (int) EndCodeSection - infos.EIP;
    len = Disasm(&infos);
    if (infos.Error >= 0) {
      if (
        (infos.Instruction.BranchType == JmpType) &&
        (infos.Instruction.AddrValue != 0))
      {
        infos.EIP = RVA2OFFSET((int) infos.Instruction.AddrValue - 0x400000,pBuffer);
        infos.VirtualAddr = infos.Instruction.AddrValue;
      }
```

```
      else {
        (void) printf("%.8X %s\n",(int) infos.VirtualAddr, &infos.CompleteInstr);
        infos.EIP += len;
        infos.VirtualAddr += len;
      }
    }
  }
  return;
}
/* ======================================================================*/
/*                                                                       */
/*          Convert Relative Virtual Address to offset in the file       */
/*          works fine even in naughty binaries                          */
/*      BeatriX manufacture :)                                           */
/*                                                                       */
/*=======================================================================*/

int RVA2OFFSET(int RVA, unsigned char *pBuff)
{
int RawSize, VirtualBorneInf, RawBorneInf, SectionHeader;
int OffsetNtHeaders,OffsetSectionHeaders, NumberOfSections, SizeOfOptionalHeaders, VirtualAddress;

OffsetNtHeaders = (int) *((int*) (pBuff + 0x3c));
NumberOfSections = (int) *((unsigned short*) (pBuff + OffsetNtHeaders + 6));
SizeOfOptionalHeaders = (int) *((unsigned short*) (pBuff + OffsetNtHeaders + 0x14));
OffsetSectionHeaders = OffsetNtHeaders + SizeOfOptionalHeaders + 0x18;
VirtualBorneInf = 0;
RawBorneInf = 0;
VirtualAddress = 0;
SectionHeader = 0;
while (VirtualAddress <= RVA) {
  if (VirtualAddress != 0) {
    VirtualBorneInf = VirtualAddress;
    RawSize = (int) *((unsigned int*) (pBuff + OffsetSectionHeaders + 0x10));
    RawBorneInf = (int) *((unsigned int*) (pBuff + OffsetSectionHeaders + 0x14));
  }
  VirtualAddress = (int) *((unsigned int*) (pBuff + OffsetSectionHeaders + SectionHeader*0x28 + 0x0C));
  SectionHeader ++;
}
if ((RVA-VirtualBorneInf)>RawSize) return -1;
RawBorneInf = RawBorneInf >> 8;
if (RawBorneInf & 1) RawBorneInf--;
RawBorneInf = RawBorneInf << 8;
return RVA - VirtualBorneInf + RawBorneInf + (int) pBuff;
}
/* ======================================================================*/
/*                                                                       */
/*                              MAIN                                     */
/*                                                                       */
/*=======================================================================*/
int main(void)
{

  FileHandle = fopen("msgbox.exe", "rb");
  (void)fseek(FileHandle,0,SEEK_END);
  FileSize = ftell(FileHandle);
  (void)rewind(FileHandle);
  pBuffer = malloc(FileSize);

  (void)fread(pBuffer,1,FileSize, FileHandle);
  (void)fclose(FileHandle);
```

```
  (void) printf("Disassemble code by following jumps\n");

  DisassembleCodeFilter ((unsigned char*) pBuffer + 0x400, (unsigned char*) pBuffer + 0x430, 0x401000);
  return 0;
}
```

7. How to use BeaEngine with masm32, nasm, fasm or GoAsm

BeaEngine is distributed with headers for nasm, GoAsm, fasm , masm.

Using BeaEngine with masm32

```
.386
.MODEL flat,stdcall
option casemap:none
.mmx

include \masm32\include\kernel32.inc
include \masm32\include\windows.inc
includelib \masm32\lib\kernel32.lib
include BeaEngineMasm.inc

puts PROTO:DWORD

.data

    infos           _Disasm         <>
    szoutofblock    BYTE            "Security alert. Disasm tries to read unreadable memory",0
    i               DWORD           100

.code

start:

  ; ********************** Init EIP
  mov eax, start
  mov infos.EIP, eax

  ; ********************** Just for fun : init VirtualAddr with funky value :)
  mov eax, 0bea2008h
  movd mm0, eax
  movq infos.VirtualAddr, mm0

  ; ********************** loop for disasm
  MakeDisasm:
    push offset infos
    call Disasm
    .if (infos.Error == OUT_OF_BLOCK)
      push offset szoutofblock
      call puts
      add esp, 4
      push 0
      call ExitProcess
    .elseif (infos.Error == UNKNOWN_OPCODE)
      push offset infos.CompleteInstr
      call puts
      add esp, 4
      push 0
      call ExitProcess
    .endif
    add infos.EIP, eax
```

```
    push offset infos.CompleteInstr
    call puts
    add esp, 4
    dec i
    jne MakeDisasm

  push 0
  call ExitProcess
End start
```

Using BeaEngine with nasm

```
extern _puts@4        ; define external symbols
extern _ExitProcess@4
extern _Disasm@4
global start
%include "BeaEngineNasm.inc"

section .data use32

    i       db 100

    infos:
      istruc _Disasm
      iend

section .text use32

start:

  ; ***************************** Init EIP
  mov eax, start
  mov [infos+EIP], eax

  ; ***************************** just for fun : init VirtualAddr with weird address :)
  mov eax, 0xbea2008
  movd mm0, eax
  movq [infos+VirtualAddr], mm0

    ; ***************************** loop for disasm
MakeDisasm:
  push infos
  call _Disasm@4
  cmp eax, UNKNOWN_OPCODE
  je IncreaseEIP
    add [infos+EIP], eax
    jmp DisplayInstruction

  IncreaseEIP:
    inc dword [infos+EIP]

  DisplayInstruction:
    push infos+CompleteInstr
    call _puts@4
    add esp, 4
    dec byte [i]
    jne MakeDisasm
  push 0
  call _ExitProcess@4
```

Using BeaEngine with fasm

```
format MS COFF

; ************************************ Define "prototype"

extrn '_puts@4' as puts:dword
extrn '_Disasm@4' as Disasm:dword
extrn '_ExitProcess@4' as ExitProcess:dword

; ************************************ includes
include '\fasm\INCLUDE\win32ax.inc'     ; <--- extended headers to enable macroinstruction .if .elseif .end
include 'BeaEngineFasm.inc'


section '.data' data readable writeable

  infos       _Disasm       <>
  i               db            100
  szoutofblock    db            "Security alert. Disasm tries to read unreadable memory",0


section '.text' code readable executable

 public start

 start:

  ; ********************** Init EIP
  mov eax, start
  mov [infos.EIP], eax

  ; ********************** loop for disasm
  MakeDisasm:
    push infos
    call Disasm
    .if eax = OUT_OF_BLOCK
      push szoutofblock
      call puts
      add esp, 4
      push 0
      call ExitProcess
    .elseif eax = UNKNOWN_OPCODE
      inc [infos.EIP]
    .else
      add [infos.EIP], eax
    .endif
    push infos.CompleteInstr
    call puts
    add esp, 4
    dec byte [i]
    jne MakeDisasm
  push 0
  call ExitProcess
```

Using BeaEngine with GoAsm

```
#include BeaEngineGoAsm.inc
Disasm = BeaEngine.lib:Disasm

.data

  infos       _Disasm      <>
  szoutofblock    db           "Security alert. Disasm tries to read unreadable memory",0
```

```
  i               db            100
.code

start:

  ; ********************** Init EIP
  mov eax, offset start
  mov [infos.EIP], eax

  ; ********************** loop for disasm
MakeDisasm:
  push offset infos
  call Disasm
  cmp eax, OUT_OF_BLOCK
  jne >
    push offset szoutofblock
    call puts
    add esp, 4
    push 0
    call ExitProcess
  :
  cmp eax, UNKNOWN_OPCODE
  jne >
    inc d[infos.EIP]
    jmp Display
  :
    add [infos.EIP], eax
Display:
  push offset infos.CompleteInstr
  call puts
  add esp, 4
  dec b[i]
  jne MakeDisasm
  push 0
  call ExitProcess
```

8. How to use BeaEngine with masm64 or GoAsm64

Using BeaEngine with masm64

```
include ..\..\HEADERS\BeaEngineMasm.inc

extrn puts:PROC
extrn ExitProcess: PROC


.data

  infos        _Disasm         <>
  szoutofblock    BYTE            "Security alert. Disasm tries to read unreadable memory",0
  i               DWORD           100


.code

main proc

  ; ********************** Init EIP
  mov rax, main
  mov infos.EIP, rax
```

```
  ; ********************** Init Architecture
  mov infos.Archi, 64

  ; ********************** loop for disasm
  MakeDisasm:
    mov rcx, offset infos
    call Disasm
    cmp eax, OUT_OF_BLOCK
    jne @F
        mov rcx, offset szoutofblock
        sub rsp, 20h
        call puts
        add rsp, 20h
        mov rcx, 0
        call ExitProcess
    @@:
    cmp eax, UNKNOWN_OPCODE
    jne @F
        inc infos.EIP
        jmp Display
    @@:
        add infos.EIP, rax
  Display:
    mov rcx, offset infos.CompleteInstr
    sub rsp, 20h
    call puts
    add rsp, 20h
    dec i
    jne MakeDisasm

  mov rcx, 0
  call ExitProcess
main endp

end
```

Using BeaEngine with GoAsm64

```
#include BeaEngineGoAsm.inc
Disasm = BeaEngine64.lib:Disasm

.data

    infos       _Disasm      <>
    szoutofblock    db           "Security alert. Disasm tries to read unreadable memory",0
    i               db            100
.code

start:

    ; ********************** Init EIP
    mov rax, offset start
    mov q [infos.EIP], rax

    ; ********************** Init Architecture
    mov d [infos.Archi], 64

    ; ********************** loop for disasm
MakeDisasm:
    mov rcx, offset infos
    call Disasm
    cmp rax, OUT_OF_BLOCK
```

```
    jne >
        mov rcx, offset szoutofblock
        sub rsp, 20h
        call puts
        add rsp, 20h
        mov rcx, 0
        call ExitProcess
    :
    cmp rax, UNKNOWN_OPCODE
    jne >
        inc q[infos.EIP]
        jmp Display
    :
        add q[infos.EIP], rax
Display:
    mov rcx, offset infos.CompleteInstr
    sub rsp, 20h
    call puts
    add rsp, 20h
    dec b[i]
    jne MakeDisasm
    mov rcx, 0
    call ExitProcess
```

9. How to use BeaEngine with WinDev ?

Here is an example coded by a friend, Vincent Roy, specialized in WinDev language.

```
// Creation du Header beaEngine pour Windev

// Creation des constantes
CONSTANT
  NoTabulation   = 0x0
  Tabulation   = 0x1
  MasmSyntax   = 0x000
  GoAsmSyntax   = 0x100
  NasmSyntax   = 0x200
      ATSyntax = 0x400
  PrefixedNumeral = 0x10000
  SuffixedNumeral = 0x00000
      ShowSegmentRegs = 0x1000000
  UNKNOWN_OPCODE  = -1
FIN

// Creation des structures
// Rajout Code Vince pour la nouvelle DLL de beatrix2004
REX_Struct est une structure
  W_ est un entier sur 1 octets
  R_ est un entier sur 1 octets
  X_ est un entier sur 1 octets
  B_ est un entier sur 1 octets
  state est un entier sur 1 octets
FIN

PREFIXINFO est une structure
  Number est un entier
  NbUndefined est un entier
  LockPrefix est un entier sur 1 octets
  OperandSize est un entier sur 1 octets
  AddressSize est un entier sur 1 octets
  RepnePrefix est un entier sur 1 octets
```

```
  RepPrefix est un entier sur 1 octets
  FSPrefix est un entier sur 1 octets
  SSPrefix est un entier sur 1 octets
  GSPrefix est un entier sur 1 octets
  ESPrefix est un entier sur 1 octets
  CSPrefix est un entier sur 1 octets
  DSPrefix est un entier sur 1 octets
  REX est un REX_Struct
FIN


EFLStruct est une structure
  nOF_ est un entier sur 1 octet
  nSF_ est un entier sur 1 octet
  nZF_ est un entier sur 1 octet
  nAF_ est un entier sur 1 octet
  nPF_ est un entier sur 1 octet
  nCF_ est un entier sur 1 octet
  nTF_ est un entier sur 1 octet
  nIF_ est un entier sur 1 octet
  nDF_ est un entier sur 1 octet
  nNT_ est un entier sur 1 octet
  nRF_ est un entier sur 1 octet
  nAlignment est un entier sur 1 octet
FIN

MEMORYTYPE est une structure
  nBaseRegister est un entier sur 4 octets
  nIndexRegister est un entier sur 4 octets
  nScale est un entier sur 4 octets
  nDisplacement est un entier sur 8 octets
FIN

INSTRTYPE est une structure
  nCategory est un entier sur 4 octets
  nOpcode est un entier sur 4 octets
  Mnemonic est une chaîne fixe sur 16
  nBranchType est un entier sur 4 octets
  stFlags est un EFLStruct
  nAddrValue est un entier sur 8 octets
  nImmediat est un entier sur 8 octets
  nImplicitModifiedRegs est un entier sur 4 octets
FIN

OPTYPE est une structure
  OpMnemonic est une chaîne fixe sur 32
  nOpType est un entier sur 4 octets
  nOpSize est un entier sur 4 octets
  nAccessMode est un entier sur 4 octets
  stMemory est un MEMORYTYPE
      nSegmentReg est un entier sur 4 octets
FIN

_Disasm est une structure
  EIP est un entier sans signe sur 8 octets
  VirtualAddr est un entier sans signe sur 8 octets
  SecurityBlock est un entier sur 4 octets
  CompleteInstr est une chaîne fixe sur 64
  Archi est un entier sur 4 octets
  nOptions est un entier sur 4 octets
  stInstruction est un INSTRTYPE
```

```
  stOperand1 est un OPTYPE
  stOperand2 est un OPTYPE
  stOperand3 est un OPTYPE
  Prefix est un PREFIXINFO
FIN

// Creation d un objet Disasm (equivalent à struct _Disasm MonDisasm; en C)
Disasm est un _Disasm

// Mise à jour des Options (optionnel)
Disasm:nOptions = Tabulation+NasmSyntax+PrefixedNumeral

// Chargement de la dll BeaEngine (ChargeDLL est une fonction windev)
HandleDLL est un entier = ChargeDLL ("C:\BeaEngine.dll")
si HandleDLL = -1 ALORS
  Erreur ("chargement impossible de la DLL.")
  RETOUR
FIN

// Initialisation des Datas
Disasm:EIP    = 0x401000
Disasm:VirtualAddr   = 0x0
Disasm:Archi     = 0

len   est un entier   = 0
myError est un entier   = 0
i   est un entier   = 1


TANTQUE (i<=100 ET myError=0)

  // Appel de la fonction exportée Disam (AppelDLL32 est une fonction windev)
  len = AppelDLL32("C:\BeaEngine.dll","Disasm",&Disasm)
  SI (len=UNKNOWN_OPCODE) ALORS
    myError = 1
  SINON
    // Liste les instructions (Trace est une fonction windev. Elle affiche une fenetre de "Trace" de couleur jau
    Trace (Disasm:CompleteInstr)
    Disasm:EIP += len
    i++
  FIN
FIN

dechargeDLL (HandleDLL)
```