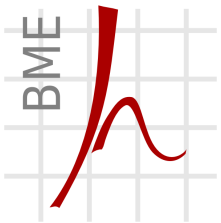
IDA

Kódvisszafejtés.



Híradástechnikai Tanszék

Izsó Tamás

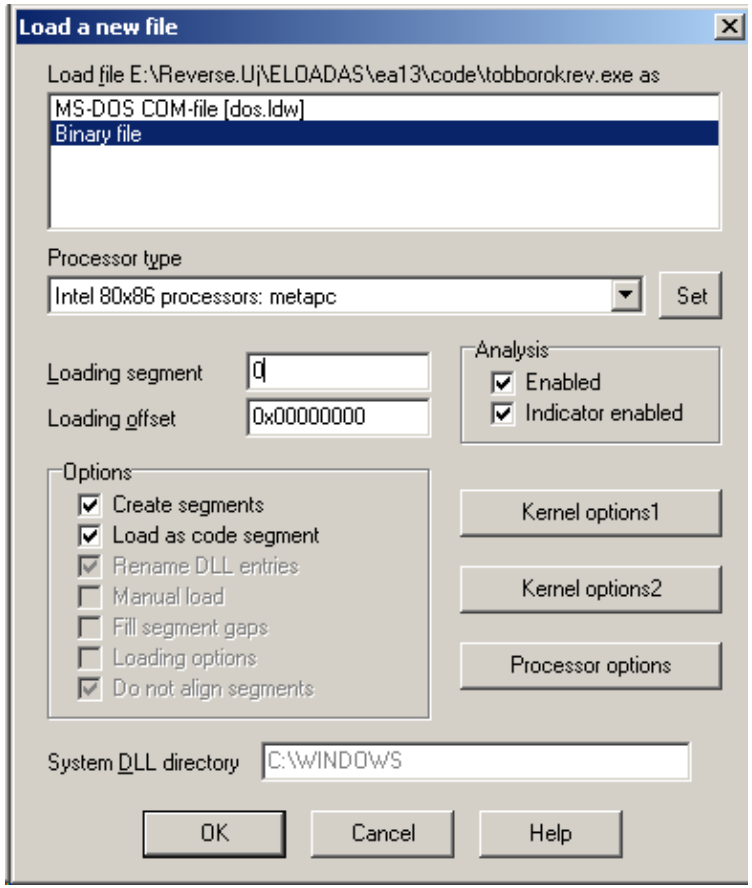
2012. december 9.

Section 1

IDA

Bináris fájlformátum visszafejtése

Az egyszerűség kedvéért nevezzük át egy PE programban lévő MZ szignaturát RE-re. Ekkor az IDA nem ismeri fel, így bináris fájlként tudjuk beolvasni.



Struktúrák definiálása

Definiáljuk a következő struktúrákat az IDA-ban (Shift+F9):

- DOS header
- NT header
- Section header

Dos header definiálása

seg000:00000000 címhez rendeljük hozzá az <Edit> <Struct var> vagy Alt-Q gombbal a dos_header struktúrát.

```
seg000:00000000     dos_header <4552h, 90h, 3, 0, 4, 0, 0FFFFh, 0, 0B8h,\
seg000:00000000                  0, 0, 0, 40h, 0, 0, 0, 0, 0, 0D8h>
seg000:00000040     db   0Eh
```

Az e_lfanew tagváltozó értéke 0xD8, ami a fájl elejétől mérve az NT header kezdete.

NT header definiálása

seg000:000000D8 címhez rendeljük hozzá az <Edit> <Struct var> vagy Alt-Q gombbal a nt_header struktúrát.

```
seg000:000000D8  nt_header <4550h, 14Ch, 3, 50B3F04Ch, 0, 0, 0E0h, \
seg000:000000D8                103h, 10Bh, 9, 0, 8C00h, 5400h, 0, 1469h,\
seg000:000000D8                1000h, 0A000h, 400000h, 1000h, 200h>
seg000:00000118  db     5
```

Az OptionalHeader az NT headerben a 0x18 offseten kezdődik, és a mérete 0xe0. Ezért az első szegmens header tábla a 0xd8+0x18+0xe0 = 0x1d0 helyen kezdődik. A szegmensek száma 3.

Section header definiálása

seg000:000001d0 címhez rendeljük hozzá az <Edit> <Struct var> vagy Alt-Q gombbal a section_header struktúrát. Mivel 3 section van, ezért a műveletet ismételjük meg még kétszer.

```
seg000:000001D0  section_header <'.text', 8B04h, 1000h, 8C00h, 400h, \
seg000:000001D0                  0, 0, 0, 0, 60000020h>
seg000:000001F8  section_header <'.rdata', 26B4h, 0A000h, 2800h, 9000h, \
seg000:000001F8                  0,  0, 0, 0, 40000040h>
seg000:00000220  section_header <'.data', 2B48h, 0D000h, 1000h, 0B800h, \
seg000:00000220                   0, 0, 0, 0, 0C0000040h>
seg000:00000248      db     0
```

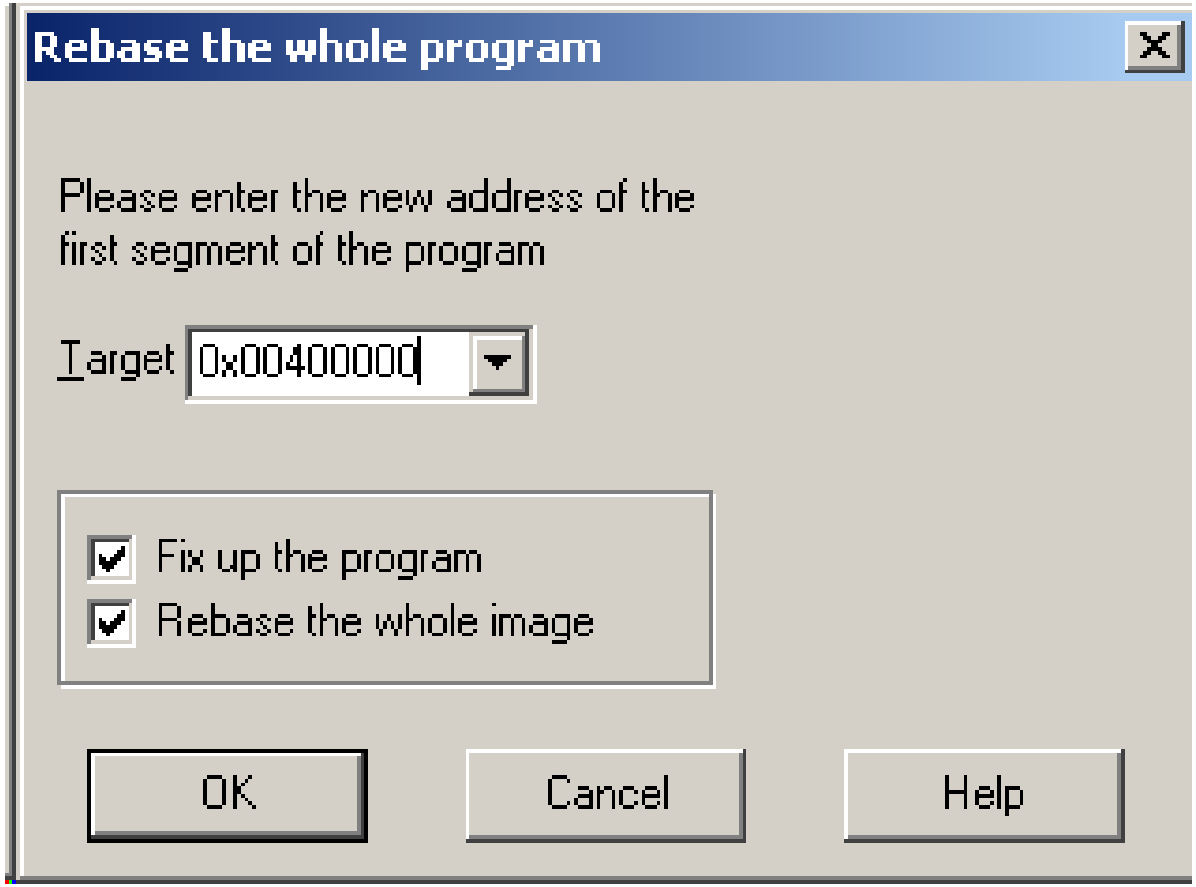
Memóriabeli helyek meghatározása

A programot a 0 címtől kezdve töltöttük be, és a section-ok sem 4096 byte többszörösén, azaz nem laphatáron kezdődnek.

- Program kezdete: OptionalHeader.BaseOfImage 0x00400000.
- .text section
  - kezdete a fájlban PointereToRawData = 0x400
  - mérete a fájlban SizeOfRawData = 0x8c00
  - memóriában a helye RVA-ban VirtualAddress 0x1000
- .rdata section
  - kezdete a fájlban PointereToRawData = 0x9000
  - mérete a fájlban SizeOfRawData = 0x2800
  - memóriában a helye RVA-ban VirtualAddress 0x0A000
- .data section
  - kezdete a fájlban PointereToRawData = 0x0b800
  - mérete a fájlban SizeOfRawData = 0x1000
  - memóriában a helye RVA-ban VirtualAddress 0x0D000

Program betöltési helyének megváltoztatása

<Edit> <Rebase program...>



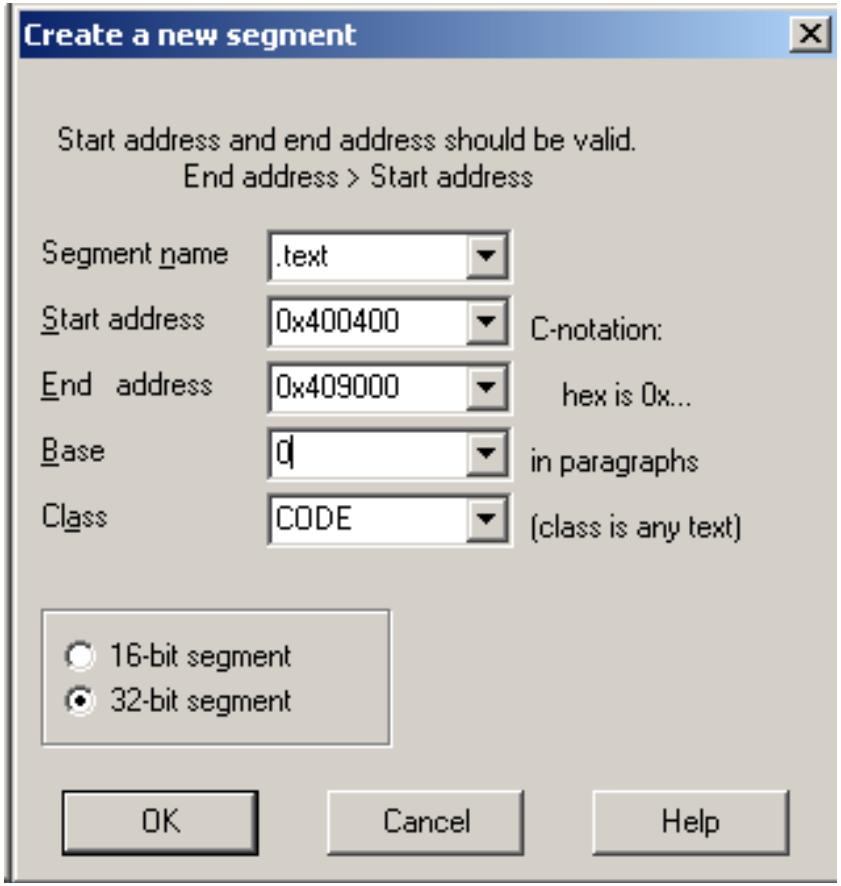
Új 3 segment létrehozása

Szegmensek kezdete és vége:

- .text 0x00400400-0x00409000
- .rdata 0x00409000-0x0040B800
- .data 0x0040B800-0x0040C800

Segment-ek létrehozása

Új segment-eket a <Edit> <Create new segment> menüpont kiválasztásával lehet létrehozni.



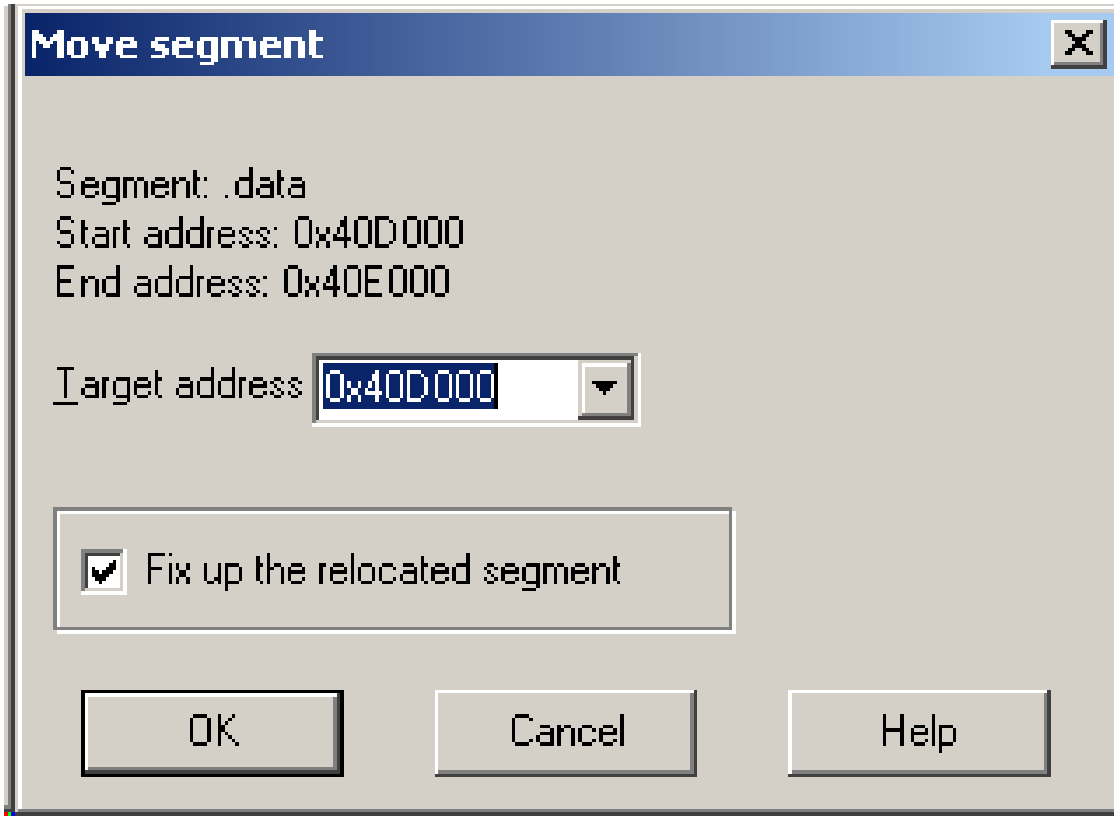
Section-ok elhelyezkedése

<View> <Open subviews> <Segments> vagy Shift-F7 gombbal megnézhetjük a definiált segmenseket .

seg000	00400000	00400400	?	?	?	.	L	**byte**	0001	**public**	**CODE**	32	0000
.text	00400400	00409000	?	?	?	.	.	**byte**	0000	**public**	**CODE**	32	FFFFFFFF
.rdata	00409000	0040B800	?	?	?	.	.	**byte**	0000	**public**	RDATA	32	FFFFFFFF
.data	0040B800	0040C800	?	?	?	.	.	**byte**	0000	**public**	DATA	32	FFFFFFFF

Szegmensek eltolása

Érdemes a legnagyobb című szegmenstől visszafele haladni, és az <Edit> <Move current segment> paranccsal a szegmens headerben megadott Virtual Address, és a program kezdőcímének megfelelően, az IDA adatbázisban lévő adatokhoz új címet rendelni.



Kód visszafejtés

NT header Opcional Header részében az Address of Entry Point értéke 0x1469A. Ehhez hozzáadva a program kezdőcímét (0x400000) megkapjuk az első végrehajtandó utasítás címét. Ettől a címtől kell az <Edit> <Code> vagy a C betű segítségével a kódot visszafejteni.

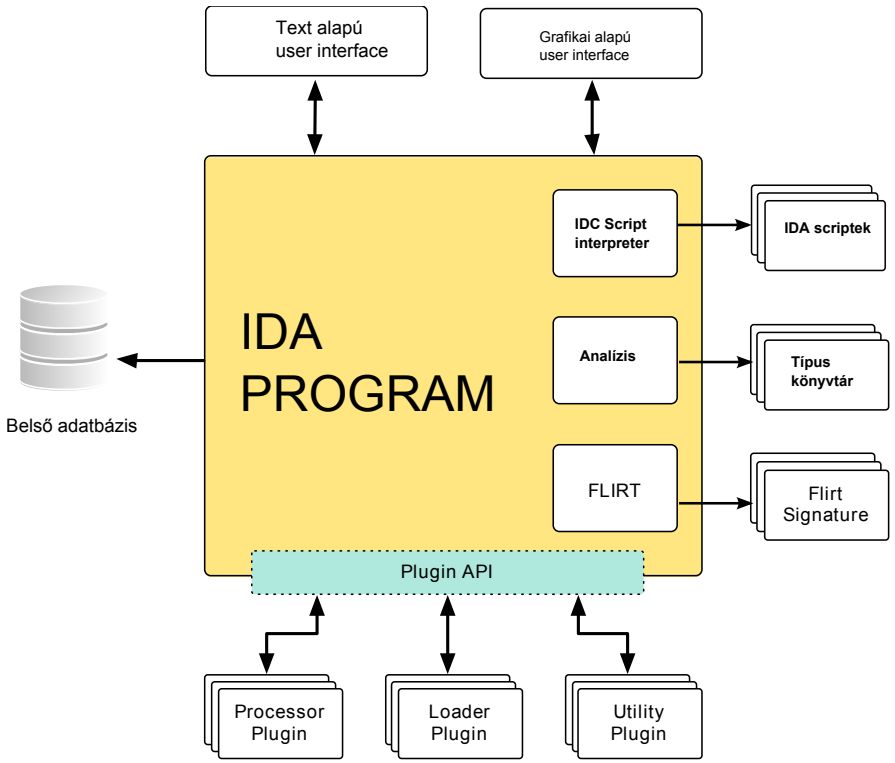
A kód minősége rosszabb, mint az automatikus visszafejtés esetén.

IDA programozási lehetőségek

- IDA script (nem elég rugalmas).
- IDA plugin rugalmas, de ismerni kell a C++ nyelvet. Az IDA API függvényei a következő feladatokat támogatják:
  - Fájl formátum felismerése és leképzése az IDA adatbázisba.
  - Különböző processzorok kódjának a disassemblálása.
  - Adat és vezérlésszerkezet analízis támogatása.
  - Grafikus megjelenítés a kódrészek kiemelésére, valamit szemléltető ábra, gráf készítés .
  - Saját analízishez kényelmes felhasználói interface létrehozása (Qt toolkit).
  - Új processzor utasításainak emulálásához szükséges rutinok létrehozása. Ezt a debugger modul fogja használni.
  - És még sok egyéb, talán még most nem is ismert funkciók létrehozása.

IDA plugin fajtái

- utility plugin
- loader plugin
- processzor plugin



IDA plugin élettartama

- IDA adatbázis betöltésétől a kilépésig;
- betöltő (loader) modul működési idejéig;
- processzor (visszafejtő) modul működési idejéig;
- igény szerint, interaktívan, vagy callback.

IDA Plugin

```
plugin_t __declspec(dllexport) PLUGIN =
{
  IDP_INTERFACE_VERSION,
  0,                    // plugin flags
  init,                 // inicializálás
  term,                 // terminálás. a fv  pointer lehet NULL is.
  run,                  // plugin rutin lefuttatás
  comment,              // plugin leírása
                        // látható a status sorban
                        // vagy tippként
  help,                 // többsoros leírása a plugin-ról
  wanted_name,          // plugin neve
  wanted_hotkey         // hotkey a plugin futtatásához
};
```

Plugin inicializálás

Ellenőrzés, hogy a plugin a megnyitott IDA adatbázishoz alkalmazható-e.

- Adott fájlformátumhoz készült.
- Adott CPU-t támogatja.
- IDA verzió megegyezik.
- GUI vagy text módban fut.
- Használ más plugint, és az be van-e töltve.
- és egyéb szempontok

```
int idaapi init(void)
{
    if( is_plugin_compatible_with_actual_database() )
    {
        return PLUGIN_OK; // usr activated
    }
    return PLUGIN_SKIP; // unload
}
```

Felhasználói Plugin aktiválás

- Hagyományos módon <Edit> <Plugin> <MyPlugin> <calls run()>
- Új menüpont hozzáadásával a (add_menu_item()) függvény segítségével.

```
inline bool add_menu_item(
        const char *menupath,
        const char *name,
        const char *hotkey,
        int flags,
        menu_item_callback_t* callback,
        void* ud );

typedf bool idaapi menu_item_callback_t(void* ud);
```

Plugin aktiválás események bekövetkezésére

Callback függvény regisztrálása:

```
// preprocesszor eseményre
hook_to_notification_point( HT_IDP, idp_callback, NULL);

// adatbázis változásra
hook_to_notification_point( HT_IDB, idb_callback, NULL);

// felhasználói interfész esetén
hook_to_notification_point( HT_UI, ui_callback, NULL);

// debugger eseményre
hook_to_notification_point( HT_DBG, dbg_callback, NULL);

typedf bool idaapi menu_item_callback_t(void* ud);
```

Új IDC függvény létrehozás

```
idc_func("MyFunc5", myfunc5, myfunc5_args);
```

Loader plugin létrehozás bináris fájl értelmezésére

Mire jó? Végtelen sok futtatható fájlformátum lehet.

- ROM-ban lévő kód;
- TCP csomagokban terjedő kódok, amit a tcpdump vagy egyéb programok segítségével fájlba menthetünk;
- önkicsomagoló tömörítő exe programok, amihez nekünk kell elvégezni az image létrehozását a visszafejtett algoritmus alapján;
- önkicsomagoló exe esetén, amikor rábízzuk a kicsomagolást a programra, azaz lefuttatjuk, és a memóriában keletkező kódot (processzhez tartozó lapokat) fájlba mentjük.
- stb.

Loader inicializálása

Előző diákon bemutatott, kézzel visszafejtett RE signature-val rendelkező, de ténylegesen PE fájlformátumhoz készítjük el a loader-t.

```
//
// Loader Module Descriptor Blocks
//
extern "C" loader_t LDSC = {
  IDP_INTERFACE_VERSION,
  0, /* no loader flags */
  accept_file,
  load_file,
  save_file,
  NULL,
  NULL,
};
```

accept_file

A függvény megnézi, hogy a fájl elején megtalálható-e az "RE" string.

```
int __stdcall accept_file(
        linput_t *li,
        char fileformatname[MAX_FILE_FORMAT_NAME],
        int n)
{
    uint32 magic;
    if( n || lread4bytes( li, &magic, false) )
         return 0;

    if (magic != RE_MAGIC ) return 0;

    // file has passed all
    qstrncpy(fileformatname, "REVERESE (PE) Image",
        MAX_FILE_FORMAT_NAME);

     // send messagess
     qsnprintf(fileformatname, MAX_FILE_FORMAT_NAME,
                "REVERSE Executable" );

     return ACCEPT_FIRST | 1;
}
```

save_file

Ezt a függvényt az IDA a <File> <Produce file> <Create exe file...> menü aktiválása esetén hívja meg. Célja az adatbázis alapján előállítani a futtatható fájt, de a kiírt adat a programozóra van bízva.

```
// Demo only
int idaapi save_file( FILE* fp,
      const char* fileformatname )
{
   qfwrite(fp, fileformatname, strlen(fileformatname) );
   // databese -> exec file
   // base2file(...);
   return 1;
}
```

Típusok megadása

Az 5.0 verziójú IDA nem ismeri a parse_decls() függvényt, ezért az IDA-ban definiált struktúra típust használtam, amit az IDA .til kiterjesztésű fájlban tárol. Ez a fájl csak akkor érhető el, amikor az IDA aktív, különben a .idb-be összecsomagolja a fájlokat. A .til fájlt az <IDA DIR>\til directory-ba kell másolni.

```
tid_t   dos_header_struct;
tid_t   nt_header_struct;
tid_t   opt_header_struct;
tid_t   section_header_struct;

void add_types()
{
    add_til( "reexe.til");   // til fájl olvasás
    dos_header_struct = til2idb(-1, "dos_header");
    nt_header_struct = til2idb(-1, "nt_header");
    opt_header_struct = til2idb(-1, "optional_header");
    section_header_struct = til2idb(-1, "section_header");
}
```

Header-ek beolvasás

A load_file () elsőnek a DOS header-t olvassa be, de a többi header struktúra beolvasása is hasonlóan történik.

```
void __stdcall load_file(
    linput_t *li,
    ushort /*neflag*/, const char * /*fileformatname*/)
{
    struct dos_header dhdr;
    uint32 pos=0
    add_types();
    // DOS header feldolgozás
    lread(li, &dhdr, sizeof( struct dos_header ) );
    mem2base( &dhdr, pos, pos+ sizeof( struct dos_header ), pos );

    if( !add_segm(0, pos, pos+sizeof( struct dos_header ), ".doshdr",
                  "DATA" ) ) {
       loader_failure();
    }
    segment_t *dos_seg = getseg( 0 );
    set_segm_addressing(dos_seg, 0 );  // 16 bit
    doStruct(pos, sizeof( struct dos_header ), dos_header_struct );
    ...
```

IDA függvények

- lread() fájlból adatokat olvas a memóriába.
- mem2base() memóriában lévő adatokat az IDA adatbázisba tölti.
- file2base() a fájlban lévő adatokat az IDA adatbázisba tölti.
- add_seg() szegmens létrehozás, interaktívan a <Create new segment> paranccsal adható meg.
- getseg() a paraméterként átadott lineáris cím alapján visszaadja az IDA adatbázisban lévő szegmens leírót.
- set_segm_addressing() Szegmens címzési módja 16, 32 vagy 64 bites. Kód visszafejtésénél nagyon fontos megadni.
- doStruct() Adatok struktúra típushoz rendelése, interaktívan az ALT-Q gombbal lehet megadni.

Fájban lévő szegmensek adatbázisba töltése

Kis DOS-os programocska betöltése.

```
pos += sizeof( struct dos_header ); // file pos léptetés

file2base( li , pos, pos, dhdr.e_lfanew, FILEREG_PATCHABLE );
if( !add_segm(0, pos, dhdr.e_lfanew, ".dosprg", "DATA" ) ) {
    loader_failure();
}
segment_t *dos_prg = getseg( pos );
set_segm_addressing(dos_prg, 0 );
pos = dhdr.e_lfanew;

...
```

Szegmensek mozgatása

Szegmensek laphatárra igazítása. Interaktívan az <Edit> <Move current segment> menüpont alatt érhetjük el ezt a funkciót.

```
for(int i=nthdr.NumberOfSection-1; i>= 0; i--)
{
    segment_t *s = getseg( pSec[i].PointereToRawData );
    move_segm( s, pSec[i].VirtualAddress, 0 );
}
```

Teljes program áthelyezése

Mivel az exe file a 0x400000 címhez képest tartalmazza a kódban a címeket, ezért az adatbázisba lévő kódot is át kell helyezni erre a címre. Program betöltési helyének megváltoztatása interaktívan az <Edit> <Rebase program...> menü segítségével történt.

```
// Rebase the whole program by 'delta' bytes
rebase_program( opthdr.BaseOfImage, 0 );

...
```

Belépési pont definiálás

Meg kell adni az első végrehajtandó utasítás címét, mert az IDA disassembler ettől a címtől kezdi visszafejteni a programot.

```
// Rebase the whole program by 'delta' bytes

// add entry point
add_entry(opthdr.AddressOfEntryPoint + opthdr.BaseOfImage,
        opthdr.AddressOfEntryPoint+opthdr.BaseOfImage,
        "_start", true );

// This function should be called only from the loader
//  to describe the input file.
create_filename_cmt();
...
```

Plugin installálás

A lefordított plugint a megfelelő IDA directory alá kell másolni.

- <IDA DIR>\plugin\ általános célú tool-ok
- <IDA DIR>\loaders\ fájlformátum értelmező plugin-ek
- <IDA DIR>\procs\ processzor assembly kódját visszafejtő plugin

Pin - A Dynamic Binary Instrumentation Tool

homepage:

`http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`

Bináris Instrumentation

Instrument – műszer, szerszám,
bináris instrumentation – kód mérése.

Előnyök:

- nyelvfüggetlen;
- gépi kód szintű;
- forrásfájl nélküli programokhoz is alkalmas

Az instrumentation történhet statikusan – futás előtti kódinjektálás, vagy dinamikusan – azaz futás közben.
Dinamikus módszer előnye:

- nem kell újrafordítani, újralinkelni;
- futás közben fedezi fel a kódot;
- alkalmas dinamikusan generált kódhoz is;
- futó program közben is lehet alkalmazni.

Néhány felhasználási terület

- kód tesztelésénél a kódlefedés mérésére;
- hívási fa generálás;
- memória szivárgás tesztelésére;
- párhuzamosan futó szálak vizsgálata;
- programrészek futási sebességének a mérése ;
- új gépi utasítások bevezetésének a tesztelése;
- visszfejtés esetén a megoldás ellenőrzése;
- utasítás, függvény lecserélése saját kódra.

PIN-tool futtatása

```
pin  -t  pintool.dll -- application
```

- pin – instrumentation motor;
- pintool.dll – felhasználó által írt instrumentation eszköz;
- application – tetszőlegesen kiválasztott vizsgálandó program;

Futó program instrumentálása

```
pin  -mt 0 -t pintool.dll -pid 1234
```

Instrumetation API részei

- *Instrumentation rutin* – program futása során az első alkalommal, amikor valahova kerül a vezérlés, hozzárendeli azt a rutint, ami minden esetben az utasítás végrehajtásakor le fog futni;
- *Analizís rutin* – az a kód, amit az *instrumentation rutin* hozzárendelt az utasításhoz. Az *instrumentált utasítások* végrehajtása során mindig meghívódik az analízis rutin;

Utasítás számlálás elve

```
counter++
push ebp
counter++
mov ebp,esp
counter++
sub esp,10h
counter++
mov dword ptr [ebp-4],0
counter++
mov eax,dword ptr [ebp+8]
counter++
cmp eax,0
counter++
jz L1
counter++
mov ecx,dword ptr [ebp-4]
counter++
jmp L2
```

Utasítás számlálás PIN tool-lal

```
#include <iostream>
#include "pin.h"

UINT64 icount = 0;

// analízis rutin
void docount() { icount++; }

// instrumentation rutin
void Instruction(INS ins, void *v)
{
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

void Fini(INT32 code, void *v)
{ std::cerr << "Count " << icount << endl; }

int main(int argc, char * argv[])
{
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```

Utasításvégrehajtás nyomkövetésének elve

```
print(EIP)
push ebp
print(EIP)
mov ebp,esp
print(EIP)
sub esp,10h
print(EIP)
mov dword ptr [ebp-4],0
print(EIP)
mov eax,dword ptr [ebp+8]
print(EIP)
cmp eax, 0
print(EIP)
jz L1
print(EIP)
mov ecx,dword ptr [ebp-4]
print(EIP)
jmp L2
```

Utasításvégrehajtás nyomkövetése PIN tool-lal

```
#include <iostream>
#include "pin.h"
FILE * trace;

// analízis rutin
void printip(void *ip) { fprintf(trace, "%p\n", ip);

// instrumentation rutin
void Instruction(INS ins, void *v)  {
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)printip,
            IARG_INST_PTR, IARG_END);

}

void Fini(INT32 code, void *v)  { fclose(trace); }

int main(int argc, char * argv[]) {
    trace = fopen("itrace.out", "w");
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instruction, 0);

    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```

Instrumentation finomsága

- *utasítást szintű*
- *alapblokk szintű* Az alapblokk fogalma más mint amit a fordítók elméletében használnak, mivel itt a blokk közepébe is be lehet ugrani. Azaz egy blokknak több belépési pontja is lehet, de csak egy kilépési pontja van. Ez érthető, mivel futás közben nem lehet megállapítani, hogy van-e más helyről is belépés a blokkba.
- *trace szintű* A trace végét feltétel nélküli vezérlésátadó utasítás zárja pl. **jmp** vagy **ret**.

```
sub edx,0FFh
cmp edx, esi
jle L1
```

```
mov edi,1
add eax, 010h
jmp L2
```

1 trace
2 Basic Block
6 utasítás

Gyorsabb utasítás számlálás

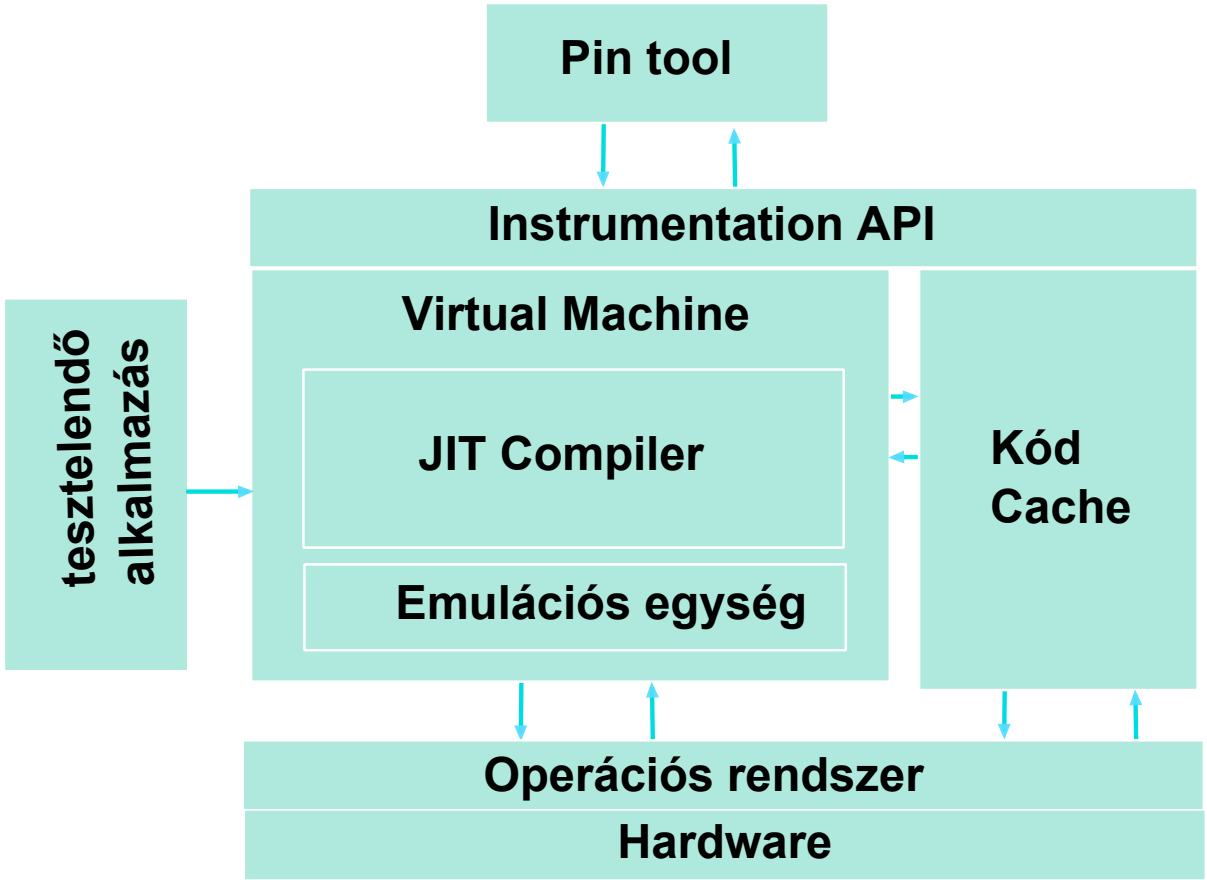
```
counter+=7
push ebp
mov ebp,esp
sub esp,10h
mov dword ptr [ebp-4],0
mov eax,dword ptr [ebp+8]
cmp eax, 0
jz L1
counter+=2
mov ecx,dword ptr [ebp-4]
jmp L2
```

Gyorsabb utasítás számlálás PIN tool-lal

```
#include <stdio.h>
#include "pin.H"
UINT64 icount = 0;
void docount(INT32 c) { icount += c; }

void Trace(TRACE trace, void *v) {
  for (BBL bbl = TRACE_BblHead(trace);
    BBL_Valid(bbl); bbl = BBL_Next(bbl)) {
    BBL_InsertCall(bbl, IPOINT_BEFORE, (AFUNPTR)docount,
          IARG_UINT32, BBL_NumIns(bbl), IARG_END);
  }
}

void Fini(INT32 code, void *v) {
  fprintf(stderr, "Count␣%lld\n", icount);
}

int main(int argc, char * argv[]) {
   PIN_Init(argc, argv);
   TRACE_AddInstrumentFunction(Trace, 0);
   PIN_AddFiniFunction(Fini, 0);
   PIN_StartProgram();
   return 0;
}
```

PIN architektúra

Úgy fogható fel mint egy virtuális gép, csak nem byte kódot, hanem gépi kódot értelmez.



Számított vezérlésátadási címek nyomkövetése

```
#include <stdio.h>
#include "pin.H"
FILE * trace;

VOID DumpJmpAddr(VOID * ip, VOID * addr)
{     fprintf(trace,"%p: %p\n", ip, addr ); }

VOID Instruction(INS ins, VOID *v) {
    if( INS_IsBranchOrCall(ins) && INS_OperandIsReg(ins, 0))      {
        INS_InsertCall(
              ins, IPOINT_BEFORE, (AFUNPTR)DumpJmpAddr,
              IARG_INST_PTR,
              IARG_BRANCH_TARGET_ADDR,
              IARG_END);
    }
}

VOID Fini(INT32 code, VOID *v) {
    fprintf(trace, "#eof\n");
    fclose(trace);
}

INT32 Usage() {
    PIN_ERROR( "This Pintool prints a trace of calculated call\n"
              + KNOB_BASE::StringKnobSummary() + "\n");
    return -1;
}
```

Dinamikus nyomkövetés eredményének felhasználása

Pin-tool output :

```
004036DA: 7C9132D9
004029CE: 0040157E
004029CE: 00404CE3
004029CE: 00405701
```

Az IDA által visszafejtett kód a PIN -tool által kiírt címen:

```
004029C8        mov     ecx, [esi]
004029CA        test    ecx, ecx
004029CC        jz      short loc_4029D0
004029CE        call    ecx
004029D0
004029D0 loc_4029D0:                    ; CODE XREF: sub_4029B7+15j
```

A hívott rutin, amit a rekurzívan pásztázó disassembler nem tudott feloldani, mert nem ismerte az **ecx** regiszter értékét.

```
0040157D sub_401578     endp
0040157E        db 0A1h ;
0040157F        db  40h ;
00401580        db 0FBh ;
```

Ajánlott olvasmány

Ami már nem fért bele az órába, de érdemes elolvasni:

`http://www.codeproject.com/Articles/30815/An-Anti-Reverse-Engineering-Guide`