



# Unpacking With Tracers I

## +NCR/CRC! [ReVeRsEr] of ARTeam

Version 1.0 – January 2006

1.	Abstract.....	2
2.	Working With Tracers.....	3
3.	References.....	24
4.	Conclusions.....	24
5.	History.....	24
6.	Greetings.....	24

### Keywords

Tracers.



### 1. Abstract

Hi you all!!!. This is my first tutorial for a non spanish group. So forgive me if something's a little difficult to understand. Besides, this is my first tutorial for such an important and well known cracking group as [ARTeam](#).

Well, sorry but I haven't introduced myself. My name's [+NCR/CRC! \[ReVeRsEr\]](#) and I'm a member of CracksLatinoS! 2005. Many of you surely know that cracking list, by one of its founders, the Master Ricardo Narvaja. He, besides CracksLatinoS, has also been a member of lists like [\[aRC\]](#), [\[RVLCN\]](#) and [PDAToolBoxHispano](#).

Well, I hope my nerves won't let me down and that the work will be interesting enough for you.

In this tutorial we'll do some unpacking using Tracers, a new technique developed by [AkirA](#) (a friend of mine and author of tutorials about Xprotector and Themida, You surely know him ;) and now implemented by yours truly.

It'll be an easy unpacking, but what's important are the tracers.

Before commencing I warn you that my way of working will be the same as in CracksLatinoS. The ones that already know me are surely used to it. And for the new ones, I hope you find it interesting.

Now, to the tutorial...

As usual I will provide sample code with this tutorial, and non-commercial sample victims. All the sources have been tested with Win2XP and Visual Studio 6.0 (Visual C++).

The techniques described here are general and not specific to any commercial applications. The whole document must be intended as a document on programming advanced techniques, how you will use these information will be totally up to your responsibility.



## 2. Working With Tracers

As I said before, we're going to perform an easy unpacking of a packer that completely destroys the IAT as well as the table of jumps. We'll automatize things very little 'cause our objective is to show the technique and besides, all these things are still in a testing phase since I asked [kaos\\_xlro](#) to make some changes in the source programs. So it's still a long way to go ;)

This is my second tutorial on tracers, the first one was written for CracksLatinoS and if anyone wants to download it, you can go to Ricardo Narvaja's FTP or just drop me a line.

Let's start by introducing the program, this is my Guinea pig:



Figure 1 – My guinea pig =).

First of all, let's use the **RDG Packer Detector** to identify the packer:

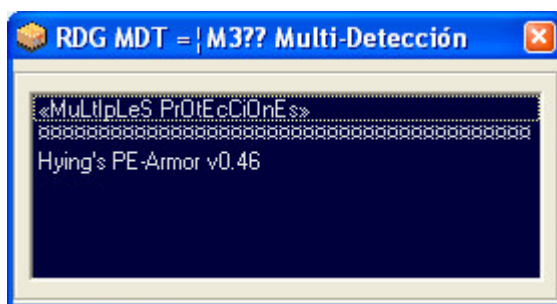


Figure 2 – Hying's PE-Armor v0.46.

It tells us that it's Hying's PE Armor v0.46. Now, let's try with PEiD to see if we can find the OEP:



Figure 3 – The real OEP?.



Well, it seems that this packed program's OEP is at 401000. Let's check it out with Olly.

I open my OllyDbg, with the patch for VProtect, Anti-CAPTION and CLASS WINDOW and the HideDebugger plugin activated with all its options, and I load the program:

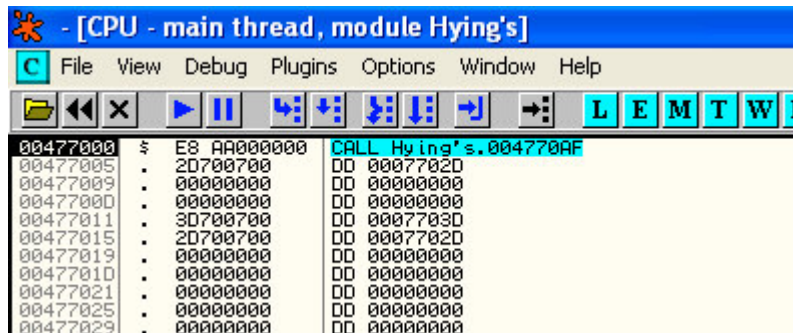


Figure 4 – EntryPoint.

There you can see the packer's EntryPoint (EP). Now, let's try to get to the OEP. But first, let's configure Olly in the following way: We go to Options – Debugging Options – Exceptions and we only check the 'Ignore Memory Access Violations in KERNEL32.dll' box:

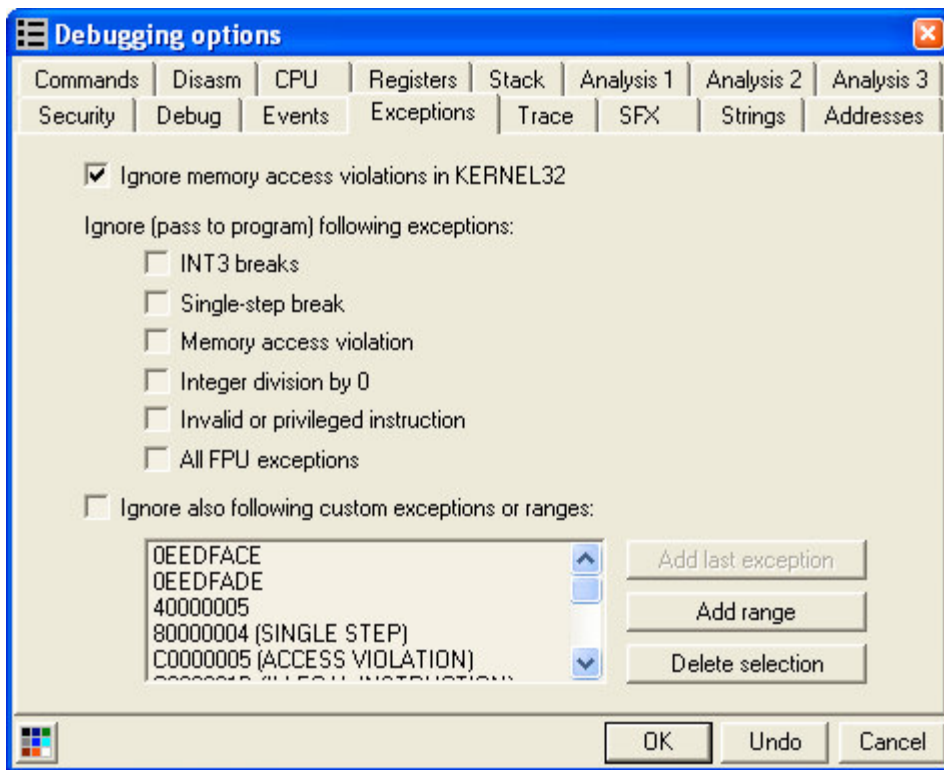


Figure 5 – Configuring OllyDbg.



Then, I press F9 (Run) and an exception occurs:

00580052	AD	LODS DWORD PTR DS:[ESI]
00580053	CD 20	INT 20
00580055	B9 04000000	MOV ECX, 4
0058005A	E8 1F000000	CALL 0058007E
0058005F	EB FA	JMP SHORT 0058005B
00580061	E8 16000000	CALL 0058007C
00580066	E9 EBF80000	JMP 0058F956

Figure 6 – Exception.

Access violation when reading [00000000] - use Shift+F7/F8/F9 to pass exception to program

Figure 7 – An Access Violation.

In order to pass the exception we only need to press <Shift+F7> and then F9 again. Another exception occurs and I again pass it by pressing <Shift+F7> followed by F9. If we do this several times, we can see that many more exceptions occur. So we repeat the above process until the program is running (it hasn't any anti-debug tricks). In that very moment, I press the minus sign key ('—') to go back in Olly and I see what the last exception was (befote the program run) so that I can write a script later on with OllyScript that will leave us in that last exception without working too much:

start:

eob break

run

break:

cmp eip,581fe8

je final

esti

jmp start

final:

ret

So we run the script and the program stops just in that last exception:

OllyScript	Run script...
OllyUni	Abort
Breakpoint Manager	Pause
Anti Anti BPM	Resume
OllyDbg PE Dumper	Step
UnhandledExceptionFilter 0.22p	
Ultra String Reference	About

Figure 8 – Running a script.

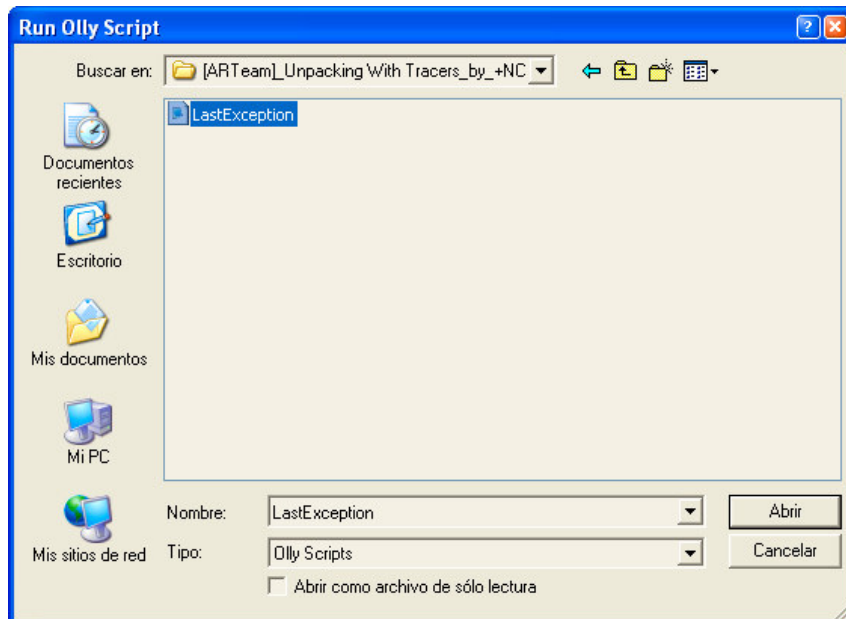


Figure 9 – Selecting the script.

And soon after, we get the warning telling us that the script finished its work:

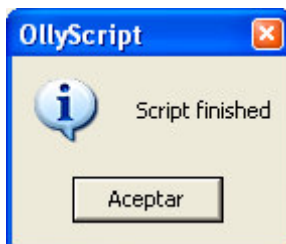


Figure 10 – Script finished.

And we are at the last exception, that in my case is:

00581FE8	AD	LODS DWORD PTR DS:[ESI]
00581FE9	CD 20	INT 20
00581FEB	E8 07000000	CALL 00581FF7
00581FF0	C783 83C013EB	MOV DWORD PTR DS:[EBX+EB13C083],2EB580B
00581FFA	CD 20	INT 20
00581FFC	83C0 02	ADD EAX,2
00581FFF	EB 01	JMP SHORT 00582002
00582001	E9 50C3E8E8	JMP E940E356

Figure 11 – My last exception.

Access violation when reading [00000000] - use Shift+F7/F8/F9 to pass exception to program

Figure 12 –An access violation.

I pass this last exception with <Shift+F7> and then I go to the Memory Map by pressing <Alt+M> in order to put a BPM on Access in the code section of the program, the one just below the PE-HEADER:



003F0000	00001000				Priv	RWE	RWE
00400000	00001000	Hying's		PE header	Imag	R	RWE
00401000	00003000	Hying's		code	Imag	R	RWE
00404000	00073000	Hying's		data,resource	Imag	R	RWE
00477000	00001000	Hying's		SFX,imports	Imag	R	RWE
00580000	00003000				Priv	RW	RW
77E40000	00001000	kernel32		PE header	Imag	R	RWE
77E41000	00076000	kernel32	.text	code,import	Imag	R	RWE

Figure 13 –Memory Map.

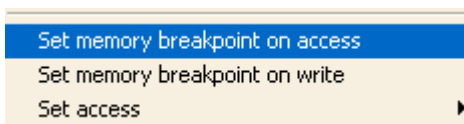


Figure 14 –We set a BPM on Access.

When I press F9 (Run) I see that the program stops here because of the BPM:

00477105	A4	MOV BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
00477106	B3 02	MOV BL,2
00477108	E8 6D000000	CALL Hying's.0047717A
0047710D	73 F6	JNB SHORT Hying's.00477105
0047710F	33C9	XOR ECX,ECX
00477111	E8 64000000	CALL Hying's.0047717A
00477116	73 1C	JNB SHORT Hying's.00477134
00477118	33C0	XOR EAX,EAX
0047711A	E8 5B000000	CALL Hying's.0047717A
0047711F	73 23	JNB SHORT Hying's.00477144
00477121	B3 02	MOV BL,2
00477123	41	INC ECX
00477124	DA 1A	MOVB AL,1A

Figure 15 – Stop because of the BPM.

Memory breakpoint when reading [00401000]

Figure 16 – The BPM.

But here the packer is just about to begin to decrypt the code section, so it'll take us some time to get to the OEP. So I open another OllyDbg, this time one patched to find OEPs (you can download it from Ricardo Narvaja's FTP), which is nothing but an OllyDbg with patched jumps so that it only stops because of Memory Breakpoints, in particular we're interested in the Executing ones because, as we know, the code section is executable and we can know the first instruction executed and therefore we can know where the OEP is.

So I open my OEPs finder OllyDbg, load the program and put a BPM on Access in the code section as we did before. In a few seconds I'm at the first Breakpoint on execution:

00401000	6A 00	PUSH 0
00401002	E8 BD040000	CALL Hying's.004014C4
00401007	A3 78324000	MOV DWORD PTR DS:[403278],EAX
0040100C	E8 EF040000	CALL Hying's.00401500
00401011	6A 00	PUSH 0
00401013	68 2E104000	PUSH Hying's.0040102E
00401018	6A 00	PUSH 0
0040101A	6A 65	PUSH 65
0040101C	FF35 78324000	PUSH DWORD PTR DS:[403278]
00401022	E8 A9040000	CALL Hying's.004014D0
00401027	6A 00	PUSH 0
00401029	E8 90040000	CALL Hying's.004014BE

Figure 17 – The first BPM on Execution.

Memory breakpoint when executing [00401000]

Figure 18 – The first BPM on Execution.





Well, this looks like an OEP and there's no Stolen Bytes or anything like that. It's the typical MASM32 OEP.

An alternative technique to get to the OEP would've been to put the BPM in the code section and to save the TRACE of the program in a file. Then, once the program is running, we can look what the first MemoryBreakpoint on execution was, and that'd be our OEP.

Okay, now we search through the table of jumps of the IAT by looking at their OPCODES (in this case they are JMPs, 0xFF25). And we see that all of them have been replaced by CALLs that point to the same section of the packer in which, after a couple of operations with an auxiliary table, the address of the function called is obtained.

In my case, I find the table of JMPs and changed them to CALLs by pressing <ENTER> over any of the jumps that we can see below the OEP:

004014BE	90	NOP
004014BF	E8 8E0D1800	CALL 00582252
004014C4	90	NOP
004014C5	E8 880D1800	CALL 00582252
004014CA	90	NOP
004014CB	E8 820D1800	CALL 00582252
004014D0	90	NOP
004014D1	E8 7C0D1800	CALL 00582252
004014D6	90	NOP
004014D7	E8 760D1800	CALL 00582252
004014DC	90	NOP
004014DD	E8 700D1800	CALL 00582252
004014E2	90	NOP
004014E3	E8 6A0D1800	CALL 00582252
004014E8	90	NOP
004014E9	E8 640D1800	CALL 00582252
004014EE	90	NOP
004014EF	E8 5E0D1800	CALL 00582252
004014F4	90	NOP
004014F5	E8 580D1800	CALL 00582252
004014FA	90	NOP
004014FB	E8 520D1800	CALL 00582252
00401500	90	NOP
00401501	E8 4C0D1800	CALL 00582252

Figure 19 – The JMP table was overwrite.

There we can see our table of jumps, all of them changed to direct CALLs to a section of the packer responsible of the redirection of the table.

Repairing this redirection of code by means of a script or an injection of code can be very tedious, although not impossible, and could take us too long. However, with the tracers, it'll take us seconds to know which function corresponds to which CALL.

But you're probably wonder what tracers are, how can we use them and how do we write them.

Well, basically a tracer consists of two elements, an \*.exe called injector.exe and a \*.dll.

The injector.exe is the one who loads the program and injects in it the library (\*.dll). Then, this library intercepts data from our process, for example ntcalls, imported functions, Zw functions, it can also dump sections, etc., We can do anything we wish with these tracers.





In particular, this tracer called 4.c, will intercept the library we specify, for instance user32.dll and it'll take from it the export table and put it in the memory of our process with all its functions. Besides, we'll have a log function that will log the address from where a function is called along with the function itself. Putting the export table in our process means that when a function is called, the call won't be to the original library but to our table that will have the following format:

```
PUSH ADDRESS_OF_THE_FUNCTION  
CALL LOG_FUNCTION  
RET
```

ADDRESS\_OF\_THE\_FUNCTION: is the actual address of the function got from the export table of each module that is loaded with the dll. In this case we'll find all the functions from the export table of user32.dll. Then, when one of these functions is called from the program, it isn't called directly to the module, but to our dll that will intercept the call and will log the point from which the call was made as well as the function called.

Okay, in order to compile the \*.dll and the injector.exe you must have Visual C++ 6.0.

### ----- CODE OF THE INJECTOR -----

```
#include "stdio.h"  
#include <windows.h>  
  
int main()  
{  
    HANDLE    hModule;  
    char      *DLLFile="import.dll";  
    char      *path="C:\\b8.exe";  
  
    STARTUPINFO  SInfo;  
    PROCESS_INFORMATION PInfo;  
  
    int LenWrite;  
    char * AllocMem;  
    HANDLE hThread;  
    DWORD Result;  
    PTHREAD_START_ROUTINE Injector;  
    FARPROC pLoadLibrary=NULL;  
  
    LenWrite = strlen(DLLFile) + 1;  
  
    GetStartupInfo(&SInfo);  
    CreateProcess(path, NULL, NULL, NULL, FALSE, CREATE_SUSPENDED, NULL,  
    NULL,    &SInfo,&PInfo);  
  
    HModule=PInfo.hProcess;  
  
    AllocMem = (char *) VirtualAllocEx(hModule,NULL, LenWrite,  
    MEM_COMMIT,PAGE_READWRITE);
```



```
WriteProcessMemory(hModule, AllocMem , DLLFile, LenWrite, NULL);

pLoadLibrary = (FARPROC) GetProcAddress(GetModuleHandle("kernel32.dll"),
"LoadLibraryA");

Injector = (PTHREAD_START_ROUTINE) pLoadLibrary;
if(!Injector)
{
    printf("Cannot inject THREAD_START_ROUTINE\n");
    return 0;
}

hThread = CreateRemoteThread(hModule, NULL, 0, Injector,
(void *) AllocMem, 0, NULL);

if(!hThread)
{
    printf("Cannot create THREAD_START_ROUTINE\n");
    return 0;
}

Result = WaitForSingleObject(hThread, INFINITE);

if(Result==WAIT_ABANDONED || Result==WAIT_TIMEOUT ||
Result==WAIT_FAILED)
{
    printf("WaitForSingleObject bad result\n");
    return 0;
}
VirtualFreeEx(hModule, (void *) AllocMem, 0, MEM_DECOMMIT);

if(hThread!=NULL)
    CloseHandle(hThread);

ResumeThread(PInfo.hThread);

ExitProcess(1);

return 1;
}
```

---

Once we've compiled this source in Visual C++ 6.0, it's time to write the dll.

----- TRAZADOR 4.C -----

```
#include <windows.h>
#include <stdio.h>
```



```
#include <stdlib.h>
#include <string.h>

#define scast static_cast
#define rcast reinterpret_cast

bool Instalar();
bool DesInstalar();

//bool NtdllApi; //Esta variable del codigo en asm

HMODULE module;
const BYTE* imageBase;
const IMAGE_DOS_HEADER* dosHeader;
const IMAGE_NT_HEADERS* winHeader;

const IMAGE_DATA_DIRECTORY* exportDataDir;
DWORD exportRVA;
DWORD exportSize;

DWORD exportBegin;
DWORD exportEnd;
const IMAGE_EXPORT_DIRECTORY* exportDir;

DWORD* funcTable;
const DWORD* nameTable;
const WORD* ordinalTable;

const DWORD* nameTableBegin, *nameTableEnd, *nameTableIter;
const WORD* ordinalTableIter;
DWORD NumeroFunciones=0;

void * PrimeraTabla;
DWORD * PtrPrimeraTabla;

void * SegundaTabla;
char * PtrSegundaTabla;

DWORD Auxiliar1=0;
DWORD dwIdOld;
HANDLE hProc;

bool flag=false;

#define MAXLOGS 2000
DWORD contador=0;
DWORD EI JMP=0;
DWORD DirVolcarInfo;

#define MAXAPIS 2000
```



```
DWORD APIS[MAXAPIS]={0};  
DWORD ContadorApis=0;  
unsigned int i=0;
```

```
DWORD CalculaJMP(DWORD MiFuncion, DWORD DirActual)  
{  
DWORD jmpaddr = MiFuncion - (DWORD)DirActual - 5;  
return jmpaddr;  
}
```

```
HANDLE hWriterFile;  
DWORD v0=0;  
char buffer[50]={0};
```

```
DWORD DirApi=0;  
DWORD DirRetorno=0;  
DWORD Tamano=0;
```

```
void VolcarInfo()  
{  
    _asm  
    {  
        pushad;  
        mov eax,esp;  
        add eax,0x74;  
        mov eax,[eax];  
        mov DirApi,eax;  
        mov eax,esp;  
        add eax,0x78;  
        mov eax,[eax];  
        mov DirRetorno,eax;  
        popad;  
    }  
  
    for(i=0;i<ContadorApis;i++)  
    {  
        if(APIS[i]==DirRetorno)  
            return;  
    }  
  
    if(ContadorApis==MAXAPIS)  
        return;  
  
    APIS[ContadorApis]=DirRetorno;  
    ContadorApis++;  
  
    if(contador<MAXLOGS)  
    {  
        wsprintf(buffer,"%x",DirRetorno);  
        Tamano=lstlen(buffer);
```



```
WriteFile(hWriterFile, buffer,Tamano,&v0, NULL);
WriteFile(hWriterFile, " ",1,&v0, NULL);
wsprintf(buffer,"%x",DirApi);
Tamano=lstlen(buffer);
WriteFile(hWriterFile, buffer,Tamano,&v0, NULL);
WriteFile(hWriterFile, "\n",1,&v0, NULL);
}
}
```

```
BOOL APIENTRY DllMain( HINSTANCE hInstance, DWORD ul_reason_for_call, LPVOID
lpReserved)
```

```
{
    if(ul_reason_for_call == DLL_PROCESS_ATTACH)
    {
        module = LoadLibrary( "user32.dll" );
        MessageBox(0,"IAT UNIVERSAL Y LOG","PRUEBA:
kernel32.dll",MB_OK);

        if ( module == NULL )
            return 1;

        // get headers
        imageBase = rcast<const BYTE*>( module );
        dosHeader = rcast<const IMAGE_DOS_HEADER*>( module );
        winHeader = rcast<const IMAGE_NT_HEADERS*>( imageBase + dosHeader-
>e_lfanew );

        // find the export data dir
        exportDataDir = &winHeader->OptionalHeader.DataDirectory[
IMAGE_DIRECTORY_ENTRY_EXPORT ];
        exportRVA = exportDataDir->VirtualAddress;
        exportSize = exportDataDir->Size;

        if ( exportRVA == 0 )
            return 1;

        // find the export dir
        exportBegin = exportRVA;
        exportEnd = exportBegin + exportSize;
        exportDir = rcast<const IMAGE_EXPORT_DIRECTORY*>( imageBase +
exportBegin );

        // get the export function/name/ordinal tables
        funcTable = (DWORD*)( imageBase + exportDir->AddressOfFunctions);
        nameTable = (DWORD*)( imageBase + exportDir->AddressOfNames );
        ordinalTable = rcast<const WORD*>( imageBase + exportDir-
>AddressOfNameOrdinals );
        NumeroFunciones = exportDir->NumberOfNames;
    }
}
```



```
        NumeroFunciones++;
        Instalar();
    }

    if(ul_reason_for_call == DLL_PROCESS_DETACH)
    {
        DesInstalar();
    }

    return TRUE;
}

bool Instalar()
{
    ////////////second pass to build table 1//////////
    //I save the real addresses to recover them later on

    hWriterFile=CreateFile("log.txt",                                GENERIC_WRITE,
FILE_SHARE_READ,0,OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL,0);
    SetEndOfFile(hWriterFile);

    PrimeraTabla=
    VirtualAlloc(NULL,(NumeroFunciones*4),MEM_COMMIT,PAGE_EXECUTE_READWRITE);
    PtrPrimeraTabla= (DWORD *) PrimeraTabla;

    nameTableBegin = nameTableIter = nameTable;
    nameTableEnd = nameTableBegin + exportDir->NumberOfNames;
    ordinalTableIter = ordinalTable;

    for ( ; nameTableIter != nameTableEnd; ++nameTableIter, ++ordinalTableIter )
    {
        const char* funcName = rcast<const char*> ( imageBase + *nameTableIter );

        if ( strcmp( funcName, "??_C@_", 6 ) == 0 )
            continue; // just a string, skip

        DWORD funcAddress = funcTable[ *ordinalTableIter ];
        funcAddress += rcast<DWORD>( imageBase );
        char * AlaFuncion=(char *)funcAddress;
        DWORD CAlaFuncion;

        //This is the new code. Here I compare the content of the API
        //that I read with the string "Ntdll", in reality only 'Ntdl', but
        //in hexa: 0x4C44544E
        //The same with the rest of the passes to the table.

        //The problem was that I was reading false APIs from kernel32
        //Instead of to an API, the table was pointing to a string
    }
}
```



```
//like "Ntdll.Rtl..."
```

```
memcpy( &CAlaFuncion,AlaFuncion, 4);  
if ( CAlaFuncion == 0x4C44544E )  
    continue;
```

```
/*_asm {  
    pushad;  
    mov eax, AlaFuncion;  
    mov eax, [eax];  
    cmp eax, 0x4C44544E;  
    mov NtdllApi, 0;  
    je K13;  
    popad;
```

K13:

```
    jne K14;  
    mov NtdllApi, 1;  
    popad;
```

K14:

```
    }  
  
    if (NtdllApi)  
        continue;*/
```

```
    funcAddress = funcTable[ *ordinalTableIter ];  
    (*PtrPrimeraTabla)=funcAddress;  
    PtrPrimeraTabla++;
```

```
    }  
    ////////////End of the second pass////////////////////////////////////
```

```
    ////////////third pass to build the table 2////////////////////////////////
```

```
    //I build tables with opcode 0x68 (push) api address and opcode 0xc3 retn  
    //I reserve room for the second table
```

```
    SegundaTabla=  
VirtualAlloc(NULL,(NumeroFunciones*11),MEM_COMMIT,PAGE_EXECUTE_READWRITE);
```

```
PtrSegundaTabla= (char *) SegundaTabla;  
nameTableBegin = nameTableIter = nameTable;  
nameTableEnd = nameTableBegin + exportDir->NumberOfNames;  
ordinalTableIter = ordinalTable;
```

```
for ( ; nameTableIter != nameTableEnd; ++nameTableIter, ++ordinalTableIter )  
{  
    const char* funcName = rcast<const char*> ( imageBase + *nameTableIter );  
    if ( strcmp( funcName, "??_C@_", 6 ) == 0 )
```





**continue; // just a string, skip**

```
DWORD funcAddress = funcTable[ *ordinalTableIter ];  
funcAddress += rcast<DWORD>( imageBase );  
char * AlaFuncion=(char *)funcAddress;  
DWORD CAlaFuncion;
```

```
memcpy( &CAlaFuncion,AlaFuncion, 4);  
if ( CAlaFuncion == 0x4C44544E )  
    continue;
```

```
/*_asm {  
    pushad;  
    mov eax, AlaFuncion;  
    mov eax, [eax];  
    cmp eax, 0x4C44544E;  
    mov NtdllApi, 0;  
    je K15;  
    popad;
```

**K15:**

```
    jne K16;  
    mov NtdllApi, 1;  
    popad;
```

**K16:**

```
}
```

```
if (NtdllApi)  
    continue;*/
```

```
(*PtrSegundaTabla)=(char)0x68;  
PtrSegundaTabla++;
```

```
memcpy((void *)PtrSegundaTabla,&funcAddress,4);  
PtrSegundaTabla+=4;  
DirVolcarInfo=(DWORD)VolcarInfo;  
EIJMP=CalculaJMP(DirVolcarInfo,(DWORD)PtrSegundaTabla);
```

```
(*PtrSegundaTabla)=(char)0xE8;  
PtrSegundaTabla++;
```

```
memcpy((void *)PtrSegundaTabla,&EIJMP,4);  
PtrSegundaTabla+=4;
```

```
(*PtrSegundaTabla)=(char)0xC3;  
PtrSegundaTabla++;
```

```
}  
//////////End of the third pass//////////
```

```
//////////fourth pass to change the export table//////////
```



```
PtrSegundaTabla= (char *) SegundaTabla;
Auxiliar1=0;
dwIdOld= GetCurrentProcessId();
hProc = OpenProcess(PROCESS_ALL_ACCESS, 0, dwIdOld);

nameTableBegin = nameTableIter = nameTable;
nameTableEnd = nameTableBegin + exportDir->NumberOfNames;
ordinalTableIter = ordinalTable;

for ( ; nameTableIter != nameTableEnd; ++nameTableIter, ++ordinalTableIter )
{
    const char* funcName = rcast<const char*> ( imageBase + *nameTableIter );

    if ( strncmp( funcName, "??_C@_", 6 ) == 0 )
        continue; // just a string, skip

    DWORD funcAddress = funcTable[ *ordinalTableIter ];
    funcAddress += rcast<DWORD>( imageBase );
    char * AlaFuncion=(char *)funcAddress;
    DWORD CAlaFuncion;

    memcpy( &CAlaFuncion,AlaFuncion, 4);
    if ( CAlaFuncion == 0x4C44544E )
        continue;

    /*_asm {
        pushad;
        mov eax, AlaFuncion;
        mov eax, [eax];
        cmp eax, 0x4C44544E;
        mov NtdllApi, 0;
        je K17;
        popad;

K17:

        jne K18;
        mov NtdllApi, 1;
        popad;

K18:

    }

    if (NtdllApi)
        continue;*/

    Auxiliar1=(DWORD)PtrSegundaTabla;
    Auxiliar1=Auxiliar1-rcast<DWORD>( imageBase );// quito la imagen base
    VirtualProtectEx(hProc,(void *)&funcTable[ *ordinalTableIter ], 4,
    PAGE_EXECUTE_READWRITE, &dwIdOld);
```



```
funcTable[ *ordinalTableIter ]=Auxiliar1;
VirtualProtectEx(hProc, (void *)&funcTable[ *ordinalTableIter], 4, dwIdOld,
&dwIdOld);
    PtrSegundaTabla+=11;
}
//////////End of the fourth pass//////////

flag=true;
return 1;
}

bool DesInstalar()
{
    if(flag==false)
        return 1;

    //////////fifth pass to recover the original table//////////

    PtrPrimeraTabla= (DWORD *) PrimeraTabla;
    Auxiliar1=0;
    dwIdOld= GetCurrentProcessId();;
    hProc = OpenProcess(PROCESS_ALL_ACCESS, 0, dwIdOld);

    nameTableBegin = nameTableIter = nameTable;
    nameTableEnd = nameTableBegin + exportDir->NumberOfNames;
    ordinalTableIter = ordinalTable;

    for ( ; nameTableIter != nameTableEnd; ++nameTableIter, ++ordinalTableIter )
    {
        const char* funcName = reinterpret_cast<const char*>( imageBase + *nameTableIter );

        if ( strcmp( funcName, "??_C@_", 6 ) == 0 )
            continue; // just a string, skip

        DWORD funcAddress = funcTable[ *ordinalTableIter ];
        funcAddress += reinterpret_cast<DWORD>( imageBase );
        char * AlaFuncion=(char *)funcAddress;
        DWORD CAlaFuncion;

        memcpy( &CAlaFuncion,AlaFuncion, 4);
        if ( CAlaFuncion == 0x4C44544E )
            continue;

        /*_asm {
            pushad;
            mov eax, AlaFuncion;
            mov eax, [eax];
            cmp eax, 0x4C44544E;
            mov NtdllApi, 0;
            je K13;
        }
    }
}
```



```
K13:      popad;

      jne K14;
      mov NtdllApi, 1;
      popad;

K14:      }

      if (NtdllApi)
          continue;*/

      Auxiliar1=(DWORD)(*PtrPrimeraTabla);
      VirtualProtectEx(hProc,(void *)&funcTable[ *ordinalTableIter ],4,
PAGE_EXECUTE_READWRITE, &dwIdOld);

      funcTable[ *ordinalTableIter ]=Auxiliar1;
      VirtualProtectEx(hProc, (void *)&funcTable[ *ordinalTableIter ],4, dwIdOld,
&dwIdOld);
      PtrPrimeraTabla++;
  }
  ////////////End of the fifth pass////////////////////////////////////

  return 1;
}
```

----- END OF TRACER 4.C -----

Once we have both sources compiled, we proceed like this:

We rename our packed program to B8.exe (it must be called like this. If you want, you can modify the source and choose another name for the executable). Then we save it in C:\\:

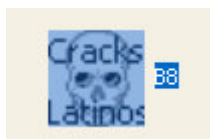


Figure 20 – Rename the packed file.

Now we place the injector.exe and the import.dll in the same folder.

In this case we're going to intercept functions from user32.dll; if we want to intercept functions from another library, we just have to change the following line in the dll source code and write the name of the new library:

```
module = LoadLibrary( "user32.dll" );
```

Then you compile again and it's done. If you don't feel like compiling, you can change it with an hex editor, or even with OllyDbg as I usually do;)



Now, we only have to run the injector.exe and see the results:



Figure 21 – A little MessageBoxA.

This messagebox pops up along with a DOS console and once we press <Aceptar> the program will be loaded. You should notice that sometimes the compilers include functions that are never used, or perhaps the program doesn't use all the functions when it's loaded, only the ones needed to be loaded. This type of tracer intercepts the functions that are called when the program is loaded. For the rest of them you must resort to the alternative method explained in the first tutorial and also written by AkirA. That is, to play a little bit with the program before pressing <Aceptar> (pushing buttons, opening menus, etc.,) so that as many functions as possible are called. Easy, isn't it?

Then, we press <Aceptar> in the above messagebox, the program is loaded, and we can look into the log.txt file that has been created. I open it with Notepad for instance or with any other text editor like Dana in order to see the data in a more convenient way:

```
1:77311819 77d18f42
2:7731182f 77d19566
3:77311836 77d16003
4:77311842 77d16003
5:7731184e 77d16003
6:7731185a 77d16003
7:77311866 77d16003
8:77311872 77d16003
9:77311894 77d19566
10:773118fd 77d16003
11:77311909 77d16003
12:77311915 77d16003
```

Figure 22 – My Log.

These are the first functions that the dll intercepted, but we're not interested 'cause they are functions of the packer. The important ones are at the bottom of the file:

```
98:77312005 77d100f0
99:401027 77d35694
100:40104f 77d19710
101:401069 77d15f0e
102:401076 77d180bc
103:401088 77d180bc
104:40109a 77d367ec
105:4010ad 77d15f0e
106:4010bd 77d367ec
107:4010d0 77d15f0e
108:401169 77d3b00a
109:4010fe 77d365b2
110:401112 77d365b2
111:40117a 77d1d9df
112:77311a82 77d1b775
```

Figure 23 – My Log; functions from the packed file.



There we are, the calls to all the functions that the program imports from user32.dll. In order to know to which one corresponds each of the addresses, we simply open another OllyDbg, load any program and we assemble a JMP ADDRESS for each one and they'll show up like magic. These are the ones I found:

00401000	- E9 8F469377	JMP USER32.DialogBoxParamA
00401005	- E9 06879177	JMP USER32.LoadIconA
0040100A	- E9 FF4E9177	JMP USER32.SendMessageA
0040100F	- E9 A8709177	JMP USER32.GetDlgItem
00401014	- E9 A3709177	JMP USER32.GetDlgItem
00401019	- E9 CE579377	JMP USER32.LoadBitmapA
0040101E	- E9 EB4E9177	JMP USER32.SendMessageA
00401023	- E9 C4579377	JMP USER32.LoadBitmapA
00401028	- E9 E14E9177	JMP USER32.SendMessageA
0040102D	- E9 089F9377	JMP USER32.MessageBoxA
00401032	- E9 7B559377	JMP USER32.GetDlgItemTextA
00401037	- E9 76559377	JMP USER32.GetDlgItemTextA
0040103C	- E9 9EC99177	JMP USER32.EndDialog
00401041	90	NOP
00401042	90	NOP
00401043	90	NOP
00401044	90	NOP
00401045	90	NOP

Figure 24 – JMPs to log functions.

All of them. Some are duplicated but it doesn't matter. So we already know which are the functions imported from user32.dll. We also now from what addresses they were called, so we can fix the CALLs easily and substitute them for the corresponding JMPs.

With a program like KAM we can know what libraries does the program load and therefore, what libraries we should intercept. In this example only user32.dll, kernel32.dll and comctl32.dll are used.

Let's look now for the kernel32.dll ones. We change 'user32.dll' for 'kernel32.dll' in the line mentioned above either with an hexadecimal editor or with OllyDbg. Then we run the injector.exe again and after that, we take a look at the log.txt:

```
21:77312d30 77e50332
22:401007 77e5ad86
23:40111c 77e560e1
24:40102e 77e598fd
25:<
```

Figure 25 – Functions from kernel32.dll.

These are the three functions that are imported from kernel32.dll. So, to find out which functions are, we open an OllyDbg and load any program in order to assemble the JMPs ADDRESS like we did before:

00401000	- E9 8190A577	JMP kernel32.GetModuleHandleA
00401005	- E9 0750A577	JMP kernel32.lstrlenA
0040100A	- E9 EE88A577	JMP kernel32.ExitProcess
0040100F	90	NOP
00401010	90	NOP

Figure 26 – JMP's to kernel functions.

Only the functions from comctl32.dll are left.



We do the same work and we get this:

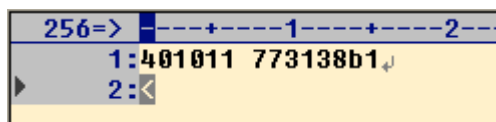


Figure 27 – My log of comctl32.dll.

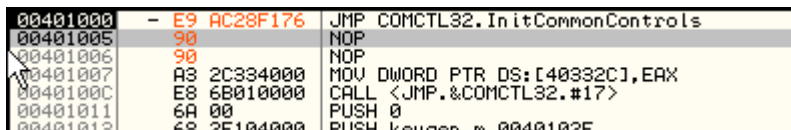


Figure 28 – JMP to comctl32 function.

Well, we have all our imported functions, now all there's left to do is to fix the IAT the way you like it best.

Since we haven't finished yet the piece of code to fix the table more elegantly, we'll do it in a somewhat unorthodox manner. I'm talking about the table of jumps 'cause the IAT is alright since this packer doesn't mess it up. Studying the routine that redirects the table a little bit, we found it easily:

Address	Hex dump	ASCII
00402000	B1 38 31 77 00 00 00 00 E1 60 E5 77 86 AD E5 77	81w...B*6w3+6w
00402010	FD 98 E5 77 00 00 00 00 DF 09 D1 77 BC 80 D1 77	2y8w...B*6w3+6w
00402020	94 56 D3 77 EC 67 D3 77 10 97 D1 77 0A B0 D3 77	6U6w3+6w...B*6w3+6w
00402030	0E 5F D1 77 B2 65 D3 77 00 00 00 00 00 00 00 00	8_0w3+6w...B*6w3+6w
00402040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00402050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00402060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00402070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

Figure 29 – The Import Address Table.

We can see that it begins at 402000 and it ends at 40203C, so the SIZE would be END – BEGINNING = 40203C – 402000 = 3C. We already know what the OEP is, so all there's left is to dump it, for example with OllyDump:

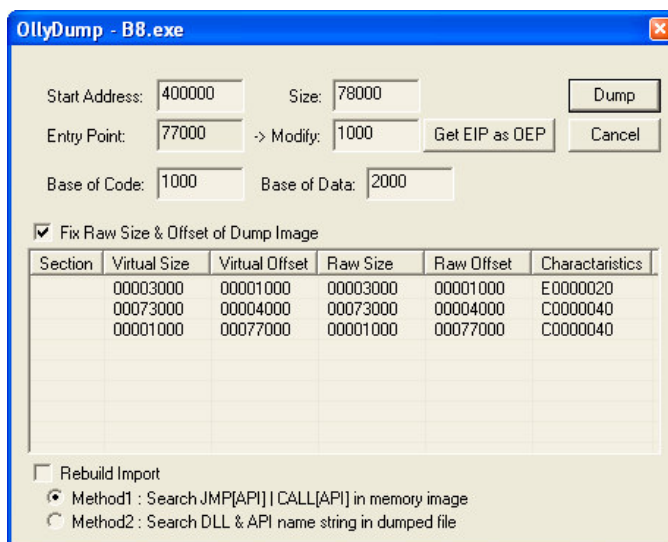


Figure 30 – Dumping with OllyDump.





Then we open the IREC (Import REConstructor) in order to repair the table once and for all ;):



Figure 31 – IAT info.



Figure 32 – Import functions.

Then we press the button <Fix Dump>, select the default dumping method and bye-bye Hying's.

All there's left is to change every CALL of the messed-up table of jumps for his corresponding JMP to the address in the IAT. Since we have the addresses of the functions, the IAT ready and the addresses from which the functions are called, we take for instance the CALL to DialogBoxParamA, we press <ENTER>, get to the CALL in the table of jumps, assemble a JMP and the first one's fixed. Proceeding the same way with the rest of them, we'll have our table ready soon:

004014BD	CC	INT3	
004014BE	- FF25 10204000	JMP DWORD PTR DS:[<&kernel32.ExitProcess	kernel32.ExitProcess
004014C4	- FF25 0C204000	JMP DWORD PTR DS:[<&kernel32.GetModuleHandleA	kernel32.GetModuleHandleA
004014CA	- FF25 08204000	JMP DWORD PTR DS:[<&kernel32.lstrlenA	kernel32.lstrlenA
004014D0	- FF25 20204000	JMP DWORD PTR DS:[<&user32.DialogBoxParamA	USER32.DialogBoxParamA
004014D6	- FF25 18204000	JMP DWORD PTR DS:[<&user32.EndDialog	USER32.EndDialog
004014DC	- FF25 1C204000	JMP DWORD PTR DS:[<&user32.GetDlgItem	USER32.GetDlgItem
004014E2	- FF25 34204000	JMP DWORD PTR DS:[<&user32.GetDlgItemTextA	USER32.GetDlgItemTextA
004014E8	- FF25 24204000	JMP DWORD PTR DS:[<&user32.LoadBitmapA	USER32.LoadBitmapA
004014EE	- FF25 28204000	JMP DWORD PTR DS:[<&user32.LoadIconA	USER32.LoadIconA
004014F4	- FF25 2C204000	JMP DWORD PTR DS:[<&user32.MessageBoxA	USER32.MessageBoxA
004014FA	- FF25 30204000	JMP DWORD PTR DS:[<&user32.SendMessageA	USER32.SendMessageA
00401500	- FF25 00204000	JMP DWORD PTR DS:[<&comctl32.InitCommon	comctl32.InitCommonControls
00401506	00	DB 00	
00401507	00	DB 00	

Figure 33 – The JMP table.

We only have to save the changes and to run the program to see if after this mess the poor thing works, he-he.

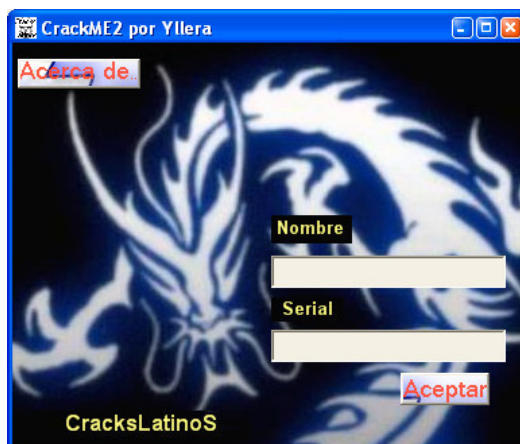


Figure 34 – Unpacked program.



Yep, it Works. So, mission accomplished.

### 3. References

- [1] "AkirATrazadores", AkirA , <http://www.iespana.es/OllyDbg>
- [2] "AkirAApiWrapper" , AkirA, <http://www.iespana.es/OllyDbg>
- [3] "AkirAExecryptor", AkirA, <http://www.iespana.es/OllyDbg>
- [4] "AkirAObsidium", AkirA, <http://www.iespana.es/OllyDbg>
- [5] "Inyeccion corriendo e inyeccion por registro\_kaos", kaos\_xlro, CracksLatinoS! List
- [6] "Trazador 4c\_kaos", kaos\_xlro, CracksLatinoS! List

### 4. Conclusions

Well, this was another example of the potencial of the tracers. In the CracksLatinoS list we keep trying to find new uses and thinking how to improve them to make things easier. All I have to say is that this last type of tracer isn't only useful for easy packers like this one, but also for Execryptor, with which we've had very good results, ASProtect, Obsidium, SoftWrap, PE-LOCK, tElock, PESpin, etc.,. I still have to test them with some other packers but I'm sure they'll work.

All in all, the thing's just born and we have to let it mature but it works fine as you've just seen.

Well, I guess this is the end of the tutorial. I hope that you like it and that you let me write a second one. I did my best to publish it in this terrific group of Crackers.

**All the code provided with this tutorial is free for public use, just make a greetz to the authors and the ARTeam if you find it useful to use. Don't use these concepts for making illegal operation, all the info here reported are only meant for studying and to help having a better knowledge of application code security techniques.**

### 5. History

- Version 1.0 – First public release!

### 6. Greetings

I can't go without thanking [AkirA](#), [kaos\\_xlro](#), [Ricardo Narvaja](#) and a special mention to [HouDiNi](#) that was the friend who translated this writing for me. Let's see if we can also translate the first one to complete the job.

Thank you all, folks, thanks to the ARTeam.

See you next time.

**+NCR/CRC! [ReVeRsEr]**

Thursday, November 17th, 2005



@: [reversing\\_ar@yahoo.com.ar](mailto:reversing_ar@yahoo.com.ar)

MSN: [nahuelriva@hotmail.com](mailto:nahuelriva@hotmail.com)

WWW: <http://ncr.iespana.es> (will be available soon)



<http://ncr.iespana.es>