



Unpacking With Tracers II

+NCR/CRC! [ReVeRsEr] of ARTeam

Version 1.0 – January 2006

1.	Abstract	2
2.	Working With Tracers	3
3.	References	35
4.	Conclusions	36
5.	History	36
6.	Greetings	36

Keywords

Tracers.



1. Abstract

Hi folks!! How are you doing?. I'm back again writing a tutorial but this time with a goal that I hope to be able to achieve sometime in the near future. I pretend to follow (if he allows me to do so) the steps of the Master **AkirA**. This guy, a good friend of mine, cracker, programmer and researcher has opened to us a wide range of possibilities we must take advantage of. Everybody in CracksLatinoS! 2005 know who I am talking about, but some people know him specially for his excellent work with Xprotector and Themida.

AkirA and his tracers have given us many new possibilities when it comes to face our beloved packers, either new ones or not so new ones. Up to now, the tracers are the best tool to do 70%, if not more, of the work. In this "short" tutorial we'll see how to blow up a packer and how to play with it. I can't remember if it was RLP, Hygyn's, SCP or PE Mutilator, but it's a packer nonetheless with an easy redirection of the import table. But the important thing is that we'll see how to use the tracers and how to apply them to different cases and we'll realize how they make our life easier, specially with the import table (IAT).

Before starting, as AkirA once said, the tracers are useful with most of packers nowadays, even the new ones that come out, 'cause few programmers now them yet. But if we start using them and make them known, we'll see that things will change. It's a fact that packers are more and more complicated, with more anti-debug and anti-dump tricks, redirections of the import table almost impossible to figure out, and many more difficulties. But with this tool, things are much easier.

Well, enough said, let's start working and see what happens...

As usual I will provide sample code with this tutorial, and non-commercial sample victims. All the sources have been tested with WinXP 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

First of all, we'll define the basic idea of a tracer, which is the name that AkirA gave to them.

A tracer consists basically of an injector (exe) and a library (dll).

The injector creates a suspended process, which in our case will be the packed program and then it injects a remote thread in that process. That remote thread is nothing but our DLL which has a hook. That hook can intercepts functions which can be the usual ones or even Zw functions or a ntcalls. We can even dump sections, log any instructions we want, etc. All in all, we can do whatever we want with to the program once we have it under control, thanks to the hook.

I don't pretend to explain all the theory here, you can study AkirA's tutorials for that matter. I've only used them and tried the ideas shown there to see if they worked. And they did! They're pretty powerful.

Those who want the whole theory about tracers should read the tutorials here:

www.iespana.es/OllyDbg under AkirA's section, all of them are there, the one about Xprotector, Themida, ApiWrapper, and the rest of them.

In this tutorial we'll use the type 4.c tracer, because the type 4.b had a bug when working with kernel32.dll; which was solved by [kaos_xlro](#).

We'll also use a couple of crackmes, one packed and the original without packing so that we can compare to see if things are going well. Our main goal isn't unpacking but to show how well the tracers work and what a fantastic tool they are to fight packers. They can even be used with programs that aren't packed, but that's another story.

Now, let's go. First of all let's see how we would unpack a program without using tracers. We load the program in our OllyGhost modified with the patches for VP, plugins for anti-debug,... Well, what we always use in unpacking tutorials. I've said this many times, so I hope you know what I'm talking about.

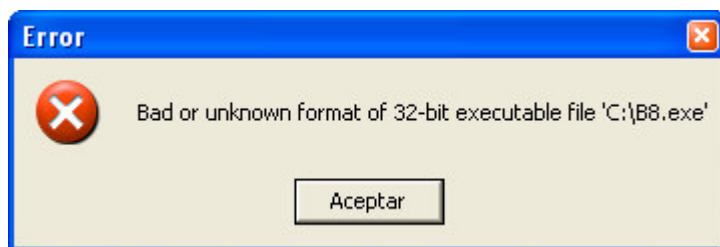


Figure 1 – Error on PE-HEADER.

Soon after the start, we see this warning message in Olly that prevents us from loading the program. As you can see, it stops at System StartUp Breakpoint:

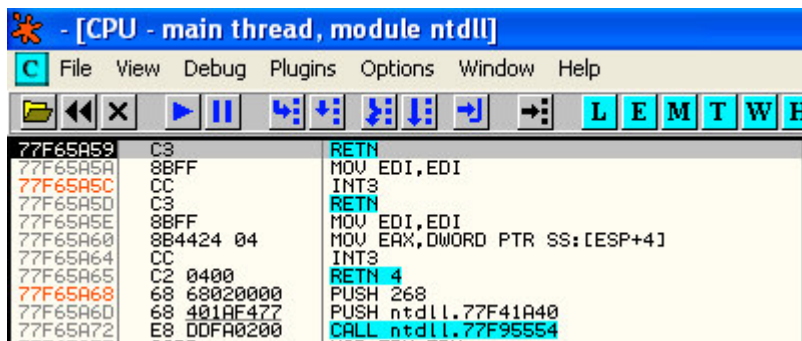


Figure 2 – System Start Up Breakpoint.

This happens because the value in NumOfRvaAndSizes has been modified on purpose by the packer itself. In general, in many packers this value is used to decrypt the last portion of code before jumping to the OEP and perform a CRC32. In this case, the value has been changed to 40h:

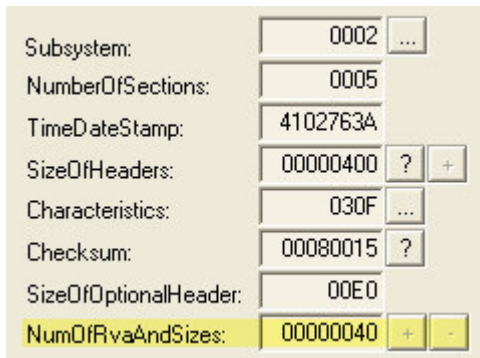


Figure 3 – NumOfRvaAndSizes.

The default value in Windows executables is 10h and every program, supposedly, should have that value. But as I said, sometimes this value is changed with evil purposes in mind ;)

If we also look at the MemoryMap, we'll see that many of the sections of the program haven't been loaded. Only the PE-HEADER and another section:

00370000	00041000				Map	R	R
003C0000	00001000				Priv	RWE	RWE
00400000	0007E000	B8		PE header	Imag	R	RWE
0055F000	00021000				Priv	RW	Gua: RW
77E40000	00001000	kernel32		PE header	Imag	R	RWE
77E41000	00076000	kernel32	.text	code, import:	Imag	R	RWE
77EB7000	00003000	kernel32	.data	data	Imag	R	RWE
77EBA000	00073000	kernel32	.rsrc	resources	Imag	R	RWE
77F20000	00006000	kernel32	.reloc	relocations	Imag	R	RWF

Figure 4 – Sections & PE-HEADER.



And if we open the original crackme with a PE Editor, we see that there are a few more sections that should be loaded:

[Section Table]						
Name	VOffset	VSize	ROffset	RSize	Flags	
.text	00001000	00000506	00000400	00000600	E0000020	
.rdata	00002000	000001AA	00000A00	00000200	C0000040	
.data	00003000	00000590	00000C00	00000400	C0000040	
.src	00004000	00072E58	00001000	00073000	40000040	
	00077000	000066D9	00074000	000016D9	E0000020	

Figure 5 – Sections.

The view is too fuzzy by now. What can we do? Well, I, personally, would start trying things and perhaps do some unorthodox thing to unpack the crackme. Many of us have done this sometime, but today we will resist the temptation and we will try this new tool we've heard about some time ago that maybe we don't fully understand but we know that it works and that's enough. At the end of the tutorial you will be able to judge if it was worthwhile or not.

Before using our tracers, let's try something. Let's change the value in NumOfRvaAndSizes to 10h and load the Olly again.

First we change the value using any PE Editor like LordPE:

Subsystem:	0002	...
NumberOfSections:	0005	
TimeDateStamp:	4102763A	
SizeOfHeaders:	00000400	? +
Characteristics:	030F	...
Checksum:	00080015	?
SizeOfOptionalHeader:	00E0	
NumOfRvaAndSizes:	00000010	+ -

Figure 6 – Correct NumOfRvaAndSizes.

Now we load the program in Olly and we see the usual warning we like so much:

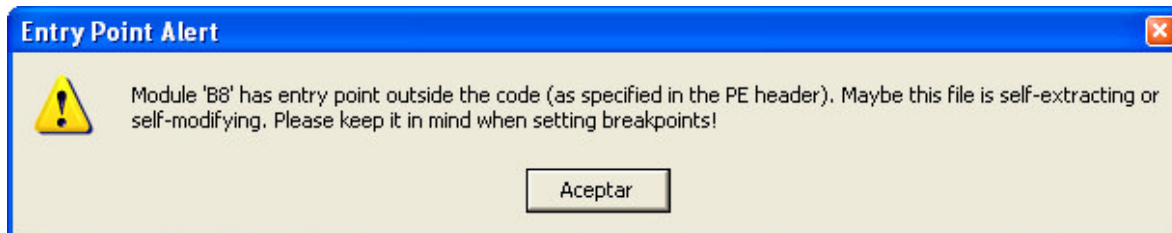


Figure 7 – EntryPoint Outside the code



You missed it, didn't you? XD.

And then this annoying message pops up:

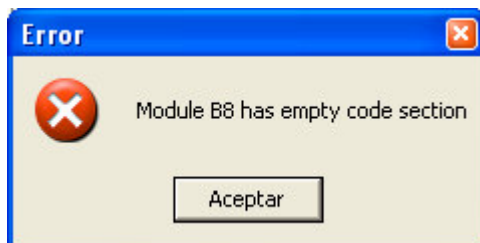


Figure 8 – Empty code section

We press <Aceptar> and now it seems we're at the EP:

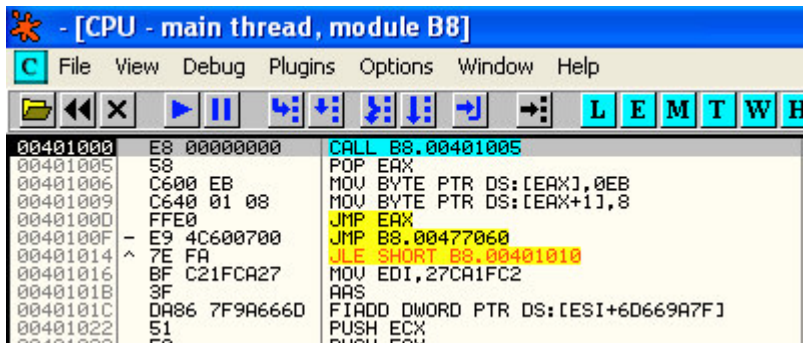


Figure 9 – EntryPoint.

I configure my Olly so that it doesn't stop at the exceptions:

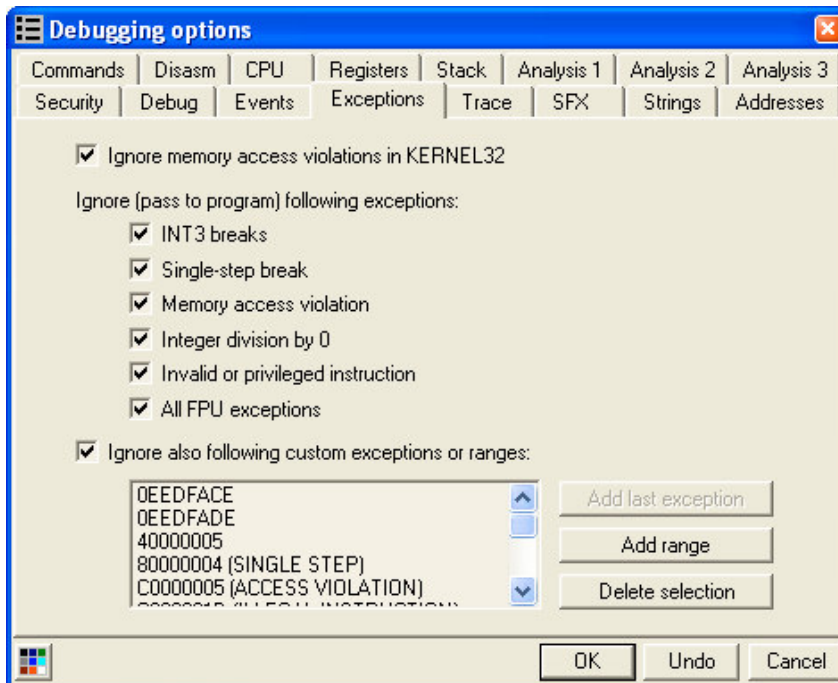


Figure 10 – Debugging Options.



The I press F9 (Run) and the program starts beautifully like if nothing had happened. So, that value in NumOfRvaAndSizes was there just to piss us off. But be careful, in many cases couldn't be like that. An example is my friend **marciano's** packer. As I said before, it uses that value to unpack a section of the code and to perform some CRC32s. Actually, this value is used to decrypt a part of the code.

Now, let's uncheck all the boxes that we checked before in the exceptions tab and restart Olly by pressing <Ctrl+F2>. We'll make use of the exceptions method to get to the OEP. I assume that you know how this works, so I won't go through it in detail. The only thing I'll tell you is that the last exception before the program starts is (in my computer):

0047846E	30C3	XOR BL,AL
00478470	36:E0 FF	LOOPNE SHORT B8.00478472
00478473	04 24	ADD AL,24
00478475	60	PUSHAD
00478476	8A85 E62F4000	MOV AL,BYTE PTR SS:[EBP+402FE6]
0047847C	0AC0	OR AL,AL
0047847E	75 03	JNZ SHORT B8.00478483
00478480	61	POPAD
00478481	C3	RETN
00478482	6983 BD372D40	IMUL EAX,DWORD PTR DS:[EBX+402D37BD],27
0047848C	61	POPAD
0047848D	C3	RETN
0047848E	BE 00004000	MOV ESI,B8.00400000
00478490	00400000	MOV EAX,DWORD PTR DS:[ESI+00000000]

Figure 11 – Last Exception.

An the OEP is:

00401000	6A 00	PUSH 0
00401002	E8 BD040000	CALL B8.004014C4
00401007	A3 78324000	MOV DWORD PTR DS:[403278],EAX
0040100C	E8 EF040000	CALL B8.00401500
00401011	6A 00	PUSH 0
00401013	68 2E104000	PUSH B8.0040102E
00401018	6A 00	PUSH 0
0040101A	6A 65	PUSH 65
0040101C	FF35 78324000	PUSH DWORD PTR DS:[403278]
00401022	E8 A9040000	CALL B8.004014D0
00401027	6A 00	PUSH 0

Figure 12 – Original Entry Point.

OEP == 401000.

If we look for the table of jumps of the IAT, we'll see the values in the jumps that have been redirected:

Figure 13 – Opcodes to search.



004014BE	-	FF25 10204000	JMP DWORD PTR DS:[402010]
004014C4	-	FF25 0C204000	JMP DWORD PTR DS:[40200C]
004014CA	-	FF25 08204000	JMP DWORD PTR DS:[402008]
004014D0	-	FF25 20204000	JMP DWORD PTR DS:[402020]
004014D6	-	FF25 18204000	JMP DWORD PTR DS:[402018]
004014DC	-	FF25 1C204000	JMP DWORD PTR DS:[40201C]
004014E2	-	FF25 34204000	JMP DWORD PTR DS:[402034]
004014E8	-	FF25 24204000	JMP DWORD PTR DS:[402024]
004014EE	-	FF25 28204000	JMP DWORD PTR DS:[402028]
004014F4	-	FF25 2C204000	JMP DWORD PTR DS:[40202C]
004014FA	-	FF25 30204000	JMP DWORD PTR DS:[402030]
00401500	-	FF25 00204000	JMP DWORD PTR DS:[402000]

Figure 14 – Table of JMP's.

If we select one of those jumps and click on Follow in Dump, we'll see all the redirected addresses of the IAT:

Address	Hex dump	ASCII
00402000	50 0A 9A 00 00 00 00 00 00 00 9A 00 F0 00 9A 00	P.Ü.....Ü.-.Ü.
00402010	E0 01 9A 00 00 00 00 00 00 02 9A 00 C0 03 9A 00	000.....\$00.L00.
00402020	B0 04 9A 00 00 05 9A 00 90 06 9A 00 80 07 9A 00	000.000.000.000.000.
00402030	70 08 9A 00 00 09 9A 00 00 00 00 00 94 20 00 00	000.000.000.000.000.000.000.000.000.000.
00402040	00 00 00 00 00 00 00 00 F6 20 00 00 08 20 00 00÷..000.
00402050	A4 20 00 00 00 00 00 00 00 00 00 00 7A 21 00 00	000.000.000.000.000.000.000.000.000.000.
00402060	18 20 00 00 8C 20 00 00 00 00 00 00 00 00 00 00	000.000.000.000.000.000.000.000.000.000.

Figure 15 – IAT With Redirected Addresses.

Our job now would be to find the section of the code in which the packer stores all these “bad” values. That's not difficult and you'll see that it's right here:

00477B4D	8F02	POP DWORD PTR DS:[EDX]	009A0000
00477B4F	83F0 01	XOR EAX,1	
00477B52	EB 01	JMP SHORT B8.00477B55	
00477B54	40	INC EAX	
00477B55	83F0 01	XOR EAX,1	
00477B58	60	PUSHAD	
00477B59	8BD8	MOV EBX,EAX	
00477B5B	8BFE	MOV EDI,ESI	
00477B5D	8985 75204000	MOV DWORD PTR SS:[EBP+402075],EAX	
00477B63	E8 5FBFFFF	CALL B8.004776C6	
00477B68	61	POPAD	
00477B69	8147 04 F00000	ADD DWORD PTR DS:[EDI+4],0F0	

Figure 16 – Redirecting Addresses.

It'll take an address out of the Stack and it'll store it in the memory address pointed by EDX:

0012FB84	009A0000	
0012FB88	00477C52	B8.00477C52
0012FB8C	004020EC	ASCII "lstrlenA"
0012FB90	004020EC	ASCII "lstrlenA"
0012FB94	00477C52	B8.00477C52
0012FB98	00075659	
0012FB9C	0012FBB0	
0012FBA0	77E40000	kernel32.77E40000
0012FBA4	00402008	B8.00402008
0012FBA8	00402094	B8.00402094
0012FBAC	77E560E1	kernel32.lstrlenA
0012FBB0	004000C0	ASCII "PE"
0012FBB4	00400258	B8.00400258
0012FBB8	00075659	
0012FBBC	0012FBD0	

Figure 17 – Stack.

We can even see the name of the redirected function in the Stack. And if we look at the registers, there's no doubt. We have all the info needed to write an script:



Registers (FPU)		
EAX	77E560E1	kernel32.lstrlenA
ECX	00402094	88.00402094
EDX	00402008	88.00402008
EBX	77E40000	kernel32.77E40000
ESP	0012FB84	
EBP	00075659	
ESI	009A0000	
EDI	004786D8	88.004786D8
EIP	00477B4D	88.00477B4D

Figure 18 – Registers.

We could write a script like the following one to easily repair the IAT.

----- SCRIPT TO REPAIR THE TABLE -----

```
start:
bphws 40eb4d,"x"
eob break
run
```

```
break:
cmp eip,40eb4d
jne final
bphwc 40eb4d
mov [esp],eax
jmp start
```

```
final:
ret
```

This is the way we usually do it. But, what if the redirection is like the one in Execryptor? Or if there's too much antidebug to trace the program? Or if there's too much polymorphism and also mixed with JunkCode? Or even worse, the new built-in metamorphism in ASProtect? No doubt that the job would be much more difficult and we would spend several hours in front of the computer trying to repair the IAT or trying to find the OEP. Actually, when I wrote the tutorial of the Unofficial Execryptor Crackme, I repair the IAT by hand and I traced one by one more than 200 entries and I swore that I wouldn't do it again (although I did, he-he).

Well, this is like those TV ads where they say, "Buy it now and solve all your problems". In this case the product are the TRACERS.

Okay, this is the code of the injector:

----- 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 ElJMP=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=strlen(buffer);
    WriteFile(hWriterFile, buffer,Tamano,&v0, NULL);
    WriteFile(hWriterFile, " ",1,&v0, NULL);
    wsprintf(buffer,"%x",DirApi);
    Tamano=strlen(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_READWR
ITE);

    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;
    ElJMP=CalculaJMP(DirVolcarInfo,(DWORD)PtrSegundaTabla);

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

    memcpy((void *)PtrSegundaTabla,&ElJMP,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 ( 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 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 = 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 K13;
        popad;

K13:
        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 -----

Now it's time to compile the dll. And in most of the cases, we are ready to work with this.

One thing we must take into account is that my injector will load the program from C:\ and the name of the executable must be B8.exe. and the dll that's going to be injected will only intercept functions from user32.dll. We'll see later on how to do it with the rest of modules of the OS.

Let's rename our packed program to B8.exe and save it in C:\:



Figure 19 – Renamed EXE.

Now we place the injector.exe and the import.dll in the same folder and we run the injector and see that a DOS console is loaded and a messagebox pops up:



Figure 20 – MessageBoxA.

The title of the messagebox says "PRUEBA: kernel32.dll" but in reality it deals with user32.dll, I forgot to change the title, he-he.

We press <Aceptar> and the program is loaded. Now we close it and take a look at the log.txt:

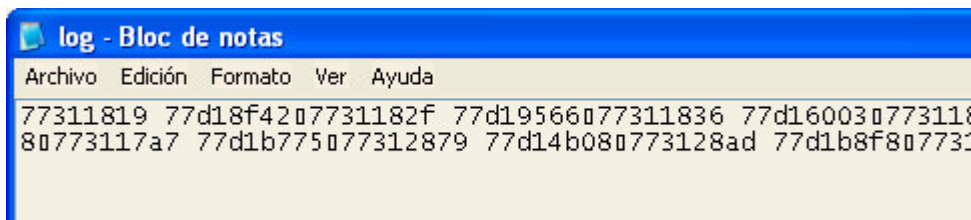
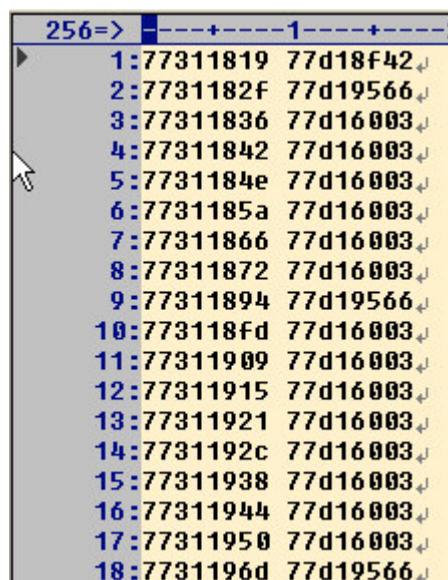


Figure 21 – Notepad.

Don't be scared!! We see many addresses with no apparent meaning, but if we look closer and format the output a little bit we'll find a treasure. In order to visualize the document better, I'll use a text editor called "Dana":



	256=>	-----1-----
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
13:	77311921	77d16003
14:	7731192c	77d16003
15:	77311938	77d16003
16:	77311944	77d16003
17:	77311950	77d16003
18:	7731196d	77d19566

Figure 22 – DANA Editor.

These are the first lines of the log.txt, let's take a look at the last ones:



98:	77312c05	77d1b8f8
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:	40117a	77d1d9df

Figure 22 – CALL's from our program.

There we are, each and every call that the program has made and the functions called. This isn't like the ApiWrapper, in that tracer we intercepted the moment in which the packer wrote the values in the table, or more precisely, the moment in which the packer took the addresses of the functions from a module in order to apply to them the corresponding algorithms to redirect the table. Here we can see that each call that the crackme made to a function from user32.dll has been intercepted. If any function is not used when the crackme is loaded but rather when the user clicks on a button for instance, it won't show up and we will have to force its execution with the corresponding action. That's why when the crackme is loaded and it's hooked by the dll, we should play with it and click every button, etc., so that as many functions as possible are called. However, we'll see how to get functions that are not executed at loading time and we'll try to write an script to automatize such a task. But first, let's see what we got.

Let's open an Olly and let's load any program and in the OEP let's assemble direct jumps to each and every one of the addresses that we saw before, in this way: JMP ADDRESS_OF_THE_MODULE. Mine are like this:



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 ADC99177	JMP user32.EndDialog
00401032	-	4C	TNC FRP

Figure 23 – JMP's from the Log.

We see that all of the addresses are pointing to user32.dll. So far, so good. Now let's load the original crackme in Olly and let's see which are its user32.dll functions:

004014BE	-	FF25 10204001	JMP DWORD PTR DS:[<&kernel32.ExitProcess>]	kernel32.ExitProcess
004014C4	-	FF25 0C204001	JMP DWORD PTR DS:[<&kernel32.GetModuleHandleA>]	kernel32.GetModuleHandleA
004014CA	-	FF25 08204001	JMP DWORD PTR DS:[<&kernel32.lstrlenA>]	kernel32.lstrlenA
004014D0	-	FF25 20204001	JMP DWORD PTR DS:[<&user32.DialogBoxParamA>]	user32.DialogBoxParamA
004014D6	-	FF25 18204001	JMP DWORD PTR DS:[<&user32.EndDialog>]	user32.EndDialog
004014DC	-	FF25 1C204001	JMP DWORD PTR DS:[<&user32.GetDlgItem>]	user32.GetDlgItem
004014E2	-	FF25 34204001	JMP DWORD PTR DS:[<&user32.GetDlgItemTextA>]	user32.GetDlgItemTextA
004014E8	-	FF25 24204001	JMP DWORD PTR DS:[<&user32.LoadBitmapA>]	user32.LoadBitmapA
004014EE	-	FF25 28204001	JMP DWORD PTR DS:[<&user32.LoadIconA>]	user32.LoadIconA
004014F4	-	FF25 2C204001	JMP DWORD PTR DS:[<&user32.MessageBoxA>]	user32.MessageBoxA
004014FA	-	FF25 30204001	JMP DWORD PTR DS:[<&user32.SendMessageA>]	user32.SendMessageA
00401500	-	FF25 00204001	JMP DWORD PTR DS:[<&comctl32.InitCommonControls>]	comctl32.InitCommonControls

Figure 24 –Table of JMP's from Original Crackme.

Well, I think that there's no doubt that it works;) We see that only MessageBoxA is missing from the list:

- 1- 401027 77d35694 --> DialogBoxParamA
- 2- 40104f 77d19710 --> LoadIconA
- 3- 401069 77d15f0e --> SendMessageA
- 4- 401076 77d180bc --> GetDlgItem
- 5- 401088 77d180bc --> GetDlgItem
- 6- 40109a 77d367ec --> LoadBitmapA
- 7- 4010ad 77d15f0e --> SendMessageA
- 8- 4010bd 77d367ec --> LoadBitmapA
- 9- 4010d0 77d15f0e --> SendMessageA
- 10- 40117a 77d1d9df --> EndDialog

But if we run the tracer again and when the crackme is loaded we go to the "About" window and then we take a look at the log.txt:

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:	40117a	77d1d9df	↓

Figure 25 – New Log.



Let's compare to the original crackme and see what's at 401169:

00401154	75 2F	JNZ SHORT original.00401185	
00401156	6A 00	PUSH 0	
00401158	68 88304000	PUSH original.00403088	
0040115D	68 95304000	PUSH original.00403095	
00401162	6A 00	PUSH 0	
00401164	E8 8B030000	CALL <JMP.&user32.MessageBoxA>	
00401169	EB 1A	JMP SHORT original.00401185	

Style = MB_OK!MB_APPLMODAL
Title = "Acerca de..."
Text = "[Crackme v2.0 por Yllera]"
hOwner = NULL
MessageBoxA

Figure 26 – MessageBoxA function.

MAGIC!!!, It's MessageBoxA, he-he. Now we have the complete list of functions, have we?:

- 1- 401027 77d35694 --> DialogBoxParamA
- 2- 40104f 77d19710 --> LoadIconA
- 3- 401069 77d15f0e --> SendMessageA
- 4- 401076 77d180bc --> GetDlgItem
- 5- 401088 77d180bc --> GetDlgItem
- 6- 40109a 77d367ec --> LoadBitmapA
- 7- 4010ad 77d15f0e --> SendMessageA
- 8- 4010bd 77d367ec --> LoadBitmapA
- 9- 4010d0 77d15f0e --> SendMessageA
- 10- 40117a 77d1d9df --> EndDialog
- 11- 401169 77d3b00a --> MessageBoxA

Well, not quite, GetDlgItemTextA is missing. We can get this one the same way that MessageBoxA:

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	:4010fe	77d365b2	↓
109	:401112	77d365b2	↓
110	:40117a	77d1d9df	↓

Figure 27 – New Log.

004010F5	FF 75 08	PUSH DWORD PTR SS:[EBP+8]	
004010F9	E8 E4030000	CALL <JMP.&user32.GetDlgItemTextA>	
004010FE	6A 40	PUSH 40	
00401100	68 7C324000	PUSH original.0040327C	
00401105	68 ED030000	PUSH 3ED	
0040110A	FF 75 08	PUSH DWORD PTR SS:[EBP+8]	
0040110D	E8 D0030000	CALL <JMP.&user32.GetDlgItemTextA>	
00401112	68 7C324000	PUSH original.0040327C	
00401117	E8 AE030000	CALL <JMP.&kernel32.lstrlenA>	
0040111C	0BC0	OR EAX,EAX	
0040111E	75 02	JNZ SHORT original.00401122	

Figure 28 – GetDlgItemTextA function.



Don't worry, there's a way to get this address without knowing which function's been called, which is what we're really interested in. Up to now, we're only testing the results of the tracer.

Okay, we have all the functions from user32.dll, let's get now the ones from kernel32.dll. Here's a problem that many of you may have suffered. I've asked the people in the list many times that when they send sources attached to the tutorials, they send also the compiled exe or the dll because not all of us have the same compilers and not all of us compile the same way. I was asking for the compiled injectors and dlls for more than a month 'cause I didn't have Visual C++ at home, but nobody answered. Finally, **Furious Logic** and **kaos_xlro** helped me and sent them over. Furious Logic sent all the tracers and dlls in the Xprotector and Themida tutorials and kaos_xlro sent me the tracer 4.c and the dlls of the ApiWrapper (Thanks a lot to both of you!!!).

Once we have the injector.exe and the corresponding dll, it's easy to change it so that it does whatever we want without having to re-compile it. What I did was to load the dll in Olly with the dlls loader and to modify the code that loads the module we want to hook. This is the line in the source code:

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

We can find this line easily in the dll, either looking at the String References or looking at the code:

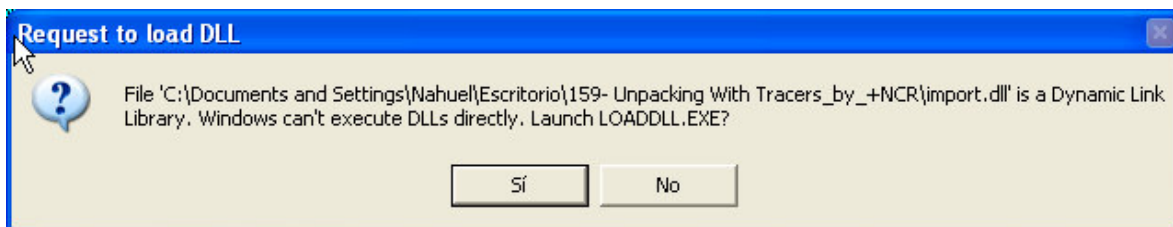


Figure 29 – Loading the Import.dll.

When we load a dll, Olly tells us that it cannot load it directly and asks us whether we want to do it via LOADDLL.EXE. We say YES and continue:

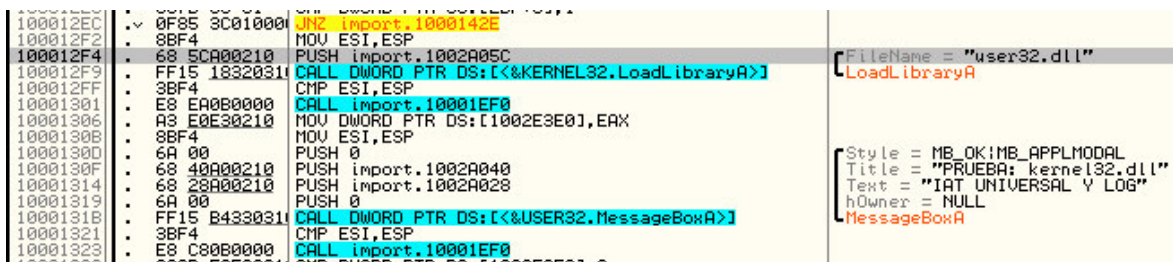


Figure 30 – Changing the dll to load.

There you are, the code to be modified. We select the parameter of LoadLibraryA and click on Follow in Dump – Immediate Constant:

Address	Hex dump	ASCII
1002A05C	75 73 65 72 33 32 2E 64 6C 6C 00 00 00 00 00 00	user32.dll.....
1002A06C	3F 3F 5F 43 40 5F 00 00 00 00 00 00 00 00 00 00	??_C@.....
1002A07C	00 00 00 00 00 00 00 00 6C 6F 67 2E 74 78 74 00log.txt.
1002A08C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1002A09C	AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA

Figure 31 – Dump.



Now we only have to change 'user32.dll' by 'kernel32.dll' and remember to write a null character at the end of the string to avoid an error. Then we save the changes and it's done, without compiling:

Address	Hex dump	ASCII
1002A05C	6B 65 72 6E 65 6C 33 32 2E 64 6C 6C 00 00 00 00	kernel32.dll...
1002A06C	3F 3F 5F 43 40 5F 00 00 00 00 00 00 00 00 00 00	??_C@_.....
1002A07C	00 00 00 00 00 00 00 00 6C 6F 67 2E 74 78 74 00log.txt.
1002A08C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Figure 32 – Now we load the kernel32.dll.

Now we run the injector again and take a look at the log.txt:

```
17:77312a04 77e5b332↓
18:77312a19 77e5b332↓
19:77312a36 77e5b332↓
20:4774da 77e4169e↓
21:401007 77e5ad86↓
22:40102e 77e598fd↓
23:<
```

Figure 33 – Log.

We do the same thing as before, we open Olly and load any program and assemble direct jumps to these addresses:

00401000	- E9 9906A477	JMP kernel32.VirtualProtect
00401005	- E9 7C9DA577	JMP kernel32.GetModuleHandleA
0040100A	- E9 EE88A577	JMP kernel32.ExitProcess
0040100F	90	NOP
00401010	90	NOP

Figure 34 – Functions from kernel32.dll.

We can see that there's an error just by looking at the log.txt 'cause the first address is 4774DA and this address isn't in our code section. Let's take a look at the original program:

00400000	00001000	original	PE header	Image	R	RWE
00401000	00001000	original .text	code	Image	R	RWE
00402000	00001000	original .rdata	imports	Image	R	RWE
00403000	00001000	original .data	data	Image	R	RWE
00404000	00073000	original .rsrc	resources	Image	R	RWE
00480000	00004000			Map	R E	R E
00540000	00002000			Map	R E	R E
00550000	00103000			Map	R	R
00660000	00085000			Map	R E	R E
00960000	00001000			Priv	RW	RW
00970000	00001000			Priv	RWE	RWE
00A5F000	00021000			Priv	RW GUA	RW

Figure 35 –MemoryMap.

Besides, if we look at the table of jumps of the original program, we'll see that VirtualProtect isn't there:

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

Figure 36 – Table of Jmp's of the original crackme.



Those are the only three functions that the program imports from kernel32.dll.

There's only `lstrlenA` left, but as we know, the more we play with the hooked crackme, the more functions we'll have in the log. So we load the crackme again with the injector.exe and we play for a while with the edits and when we look at the log again, the missing function is already there:

```
19:77312a36 77e5b332 ↓
20:4774da 77e4169e ↓
21:401007 77e5ad86 ↓
22:40111c 77e560e1 ↓
23:40102e 77e598fd ↓
24:<
```

Figure 37 – Log of Kernel32.dll.

00401100	. 68 7C324000	PUSH original.0040327C
00401105	. 68 ED030000	PUSH 3ED
0040110A	. FF75 08	PUSH DWORD PTR SS:[EBP+8]
0040110D	. E8 D0030000	CALL <JMP.&user32.GetDlgItemTextA>
00401112	. 68 7C324000	PUSH original.0040327C
00401117	. E8 AE030000	CALL <JMP.&kernel32.lstrlenA>
0040111C	. 0BC0	OR EAX, EAX
0040111E	. 75 02	JNZ SHORT original.00401122
00401120	. EB C1	JMP SHORT original.004010E3
00401122	. E8 67000000	CALL original.0040118E

Figure 38 – `lstrlenA` function.

Now we have our list of kernel32.dll functions:

```
401007 77e5ad86 --> GetModuleHandleA
40111c 77e560e1 --> lstrlenA
40102e 77e598fd --> ExitProcess
```

The only one left is `comctl32.dll`, but we know how to solve that. We modify the dll and change 'kernel32.dll' by 'comctl32.dll'. We run the injector again and take a look at the log.txt:

```
256=> 1:401011 773138b1 ↓
2:<
```

Figure 39 – Log of comctl32.dll.

00401000	- E9 AC28F176	JMP comctl32.InitCommonControls
00401005	90	NOP
00401006	90	NOP
00401007	A3 78324000	MOV DWORD PTR DS:[403278],EAX
0040100C	E8 EF040000	CALL <JMP.&comctl32.InitCommonCon

Figure 40 – `InitCommonControls` function.

```
401011 773138b1 --> InitCommonControls
```

We already have all the functions imported by the program without having to trace or skip any anti-debuggers. But although these tracers make our life easier, they



are not fool-proof. I've tested this tracer 4.c with different versions of ASProtect, PESPIN with CopymenII and ExeCryptor and they haven't got a single good entry. For instance, with some versions of ASProtect (1.31 or 2.0), we got this message when we run the injector:

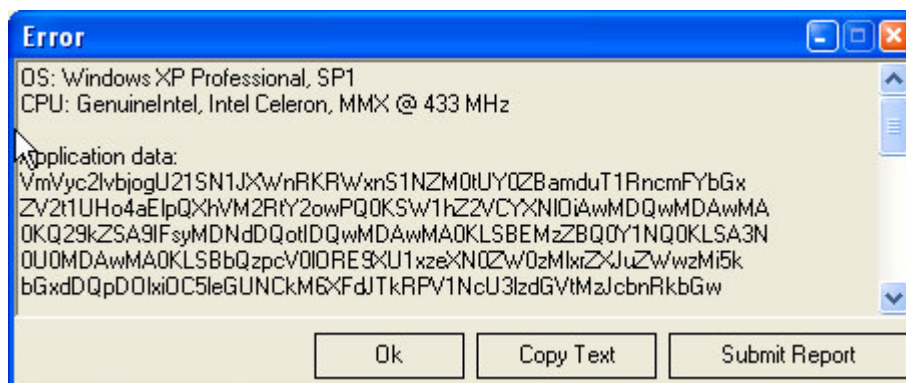


Figure 41 – Error in ASProtect 2.xx.

But with ASProtect 2.0 Build 10.20, it doesn't happen. Another example is the Official Execryptor Crackme that we saw. In that case, it doesn't get any function. But with the latest version of Execryptor I haven't had any problems, it got me all the functions:

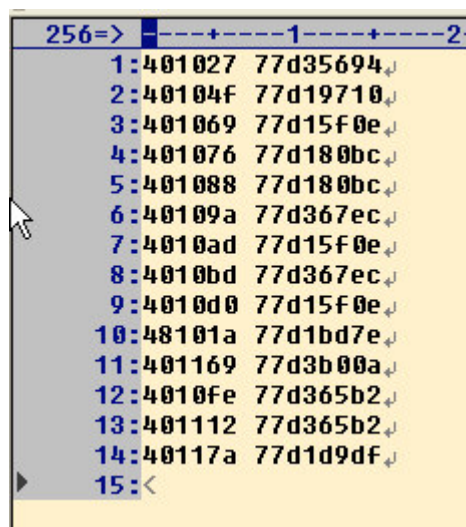


Figure 42 – Log from a crackme packed with ExeCryptor.

Those functions are from the CrackME 2 by Yllera packed (FULL) with Execryptor version 2.1.15 (Licensed Version).

NOTE: When I say that the tracer 4.c isn't able to get any function, I don't mean that it doesn't work, but that the tracer doesn't work when we only run it, because with such special packers as ASProtect or ExeCryptor or some with CopyMenII, the redirections of the table are very tricky and go beyond our imagination XD. But this tracer has a log function that logs everything in the txt file and gives us the tool to get all the functions that don't show up by themselves. We'll force their execution and we'll see that we can get all of them.

Let's keep on working with the packed crackme that we had before.



Now, imagine that some functions aren't in the log for some reason that's beyond our control. How can we get those functions? Well, as AkirA says in his tutorial, we'll force their execution.

For this task we need one library for each module of the OS. But, as we saw, we can modify that library and we don't need to compile many of them.

In this case, we'll get some functions from user32.dll.

First of all we run the injector.exe and the messagebox pops up:



Figure 43 – MessageBoxA.

Then, it loads the packed program:



Figure 44 – Packed Crackme.

Now we open an Olly and we attach the B8.exe:

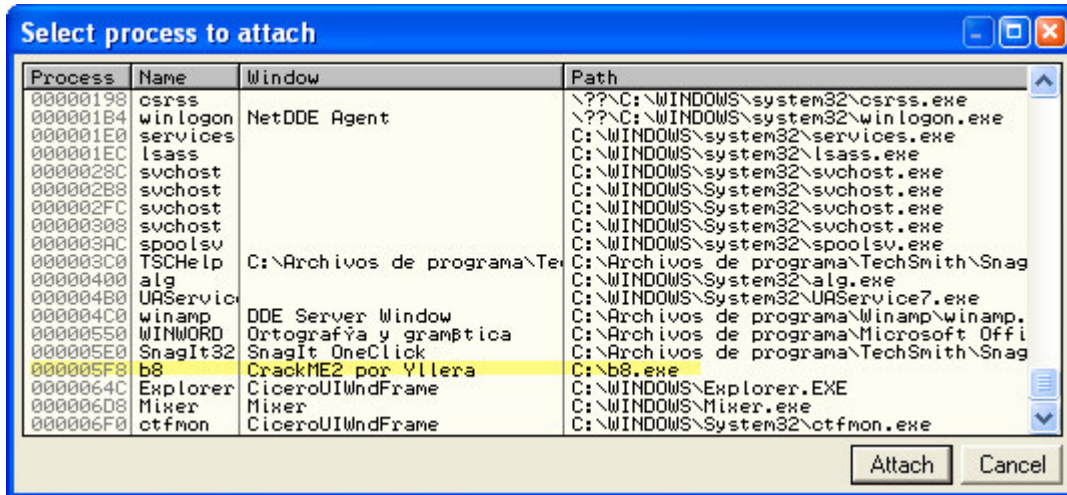


Figure 45 – Attaching.



Now we go to the MemoryMap and look for the beginning of the code section of the program:

00400000	00001000	b8		PE header	Imag	RW	RWE
00401000	00001000	b8	.text		Imag	RW	RWE
00402000	00001000	b8	.rdata		Imag	RW	RWE
00403000	00001000	b8	.data		Imag	RW	RWE
00404000	00073000	b8	.rsrc	resources	Imag	RW	RWE
00477000	00007000	b8		imports	Imag	RW	RWE

Figure 46 – MemoryMap .

In this case, we know that it begins at 401000 but in other cases we won't know. However, very often it's the section next to the PE-HEADER. So, we go to the beginning of that section, which in our case is precisely the OEP:

00401000	6A 00	PUSH 0
00401002	E8 BD040000	CALL b8.004014C4
00401007	A3 78324000	MOV DWORD PTR DS:[403278],EAX
0040100C	E8 EF040000	CALL b8.00401500
00401011	6A 00	PUSH 0
00401013	68 2E104000	PUSH b8.0040102E
00401018	6A 00	PUSH 0
0040101A	6A 65	PUSH 65
0040101C	FF35 78324000	PUSH DWORD PTR DS:[403278]
00401022	E8 A9040000	CALL b8.004014D0

Figure 47 –OEP .

Now, we look for the table of jumps of the IAT by doing a binary search of the opcodes of the jumps (0xFF25):

Figure 48 –Searching Opcodes .

004014BE	-	FF25 10204000	JMP DWORD PTR DS:[402010]
004014C4	-	FF25 0C204000	JMP DWORD PTR DS:[40200C]
004014CA	-	FF25 08204000	JMP DWORD PTR DS:[402008]
004014D0	-	FF25 20204000	JMP DWORD PTR DS:[402020]
004014D6	-	FF25 18204000	JMP DWORD PTR DS:[402018]
004014DC	-	FF25 1C204000	JMP DWORD PTR DS:[40201C]
004014E2	-	FF25 34204000	JMP DWORD PTR DS:[402034]
004014E8	-	FF25 24204000	JMP DWORD PTR DS:[402024]
004014EE	-	FF25 28204000	JMP DWORD PTR DS:[402028]
004014F4	-	FF25 2C204000	JMP DWORD PTR DS:[40202C]
004014FA	-	FF25 30204000	JMP DWORD PTR DS:[402030]
00401500	-	FF25 00204000	JMP DWORD PTR DS:[402000]

Figure 49 –Table of JMP's .

Good, we can see that all the entries have been redirected.

Before, we said that this dll had a log function that wrote the addresses in the log.txt. Well, this function works along with a table with 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 exports table of each module that is loaded with the dll. In this case we'll find all the functions from the exports 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.

This table is easy to find, we only need to go to the Memory Map and put the OPCODES of the instructions, but using wildcards, because we don't know the specific addresses or the address of the log function:

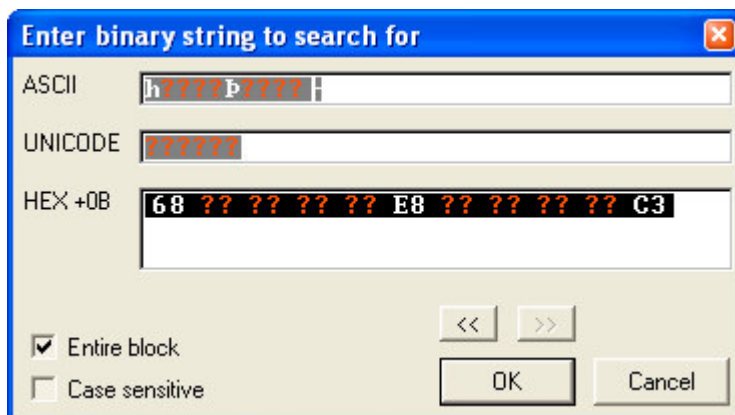


Figure 50 –Searching the log function .

The first occurrence is this:

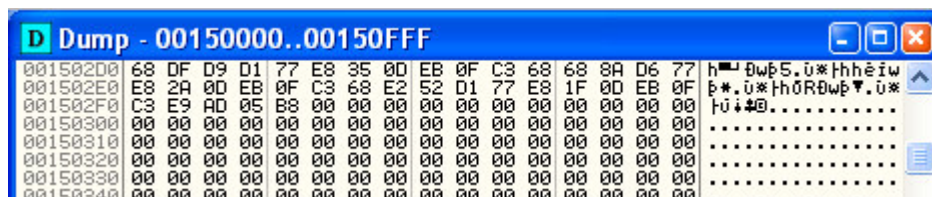


Figure 51 –Dump.

001502D0	68 DF09D177	PUSH 77D1D90F
001502D5	E8 350DEB0F	CALL import.1000100F
001502DA	C3	RETN
001502DB	68 688AD677	PUSH 77D68A68
001502E0	E8 2A0DEB0F	CALL import.1000100F
001502E5	C3	RETN
001502E6	68 E252D177	PUSH 77D152E2
001502EB	E8 1F0DEB0F	CALL import.1000100F
001502F0	C3	RETN
001502F1	- E9 AD05B800	JMP EndTask
001502F6	0000	0000 RVTF PTR DS: F0X1 01

Figure 52 –Beginning of table.



And the rest of the table is at 0CD0000:

00CD0000	68 6FB0D177	PUSH 77D18D6F
00CD0005	E8 0510330F	CALL import.1000100F
00CD000A	C3	RETN
00CD000B	68 90E1D377	PUSH 77D3E190
00CD0010	E8 FA0F330F	CALL import.1000100F
00CD0015	C3	RETN
00CD0016	68 BD92D177	PUSH 77D192BD
00CD001B	E8 EF0F330F	CALL import.1000100F
00CD0020	C3	RETN
00CD0021	68 3192D677	PUSH 77D69231
00CD0026	E8 E40F330F	CALL import.1000100F
00CD002B	C3	RETN
00CD002C	68 01BAD577	PUSH 77D5BA01
00CD0031	E8 D90F330F	CALL import.1000100F
00CD0036	C3	RETN
00CD0037	68 F7BFD177	PUSH 77D1BFF7
00CD003C	E8 CE0F330F	CALL import.1000100F
00CD0041	C3	RETN
00CD0042	68 2F62D677	PUSH 77D6622F
00CD0047	E8 C30F330F	CALL import.1000100F
00CD004C	C3	RETN
00CD004D	68 745ED677	PUSH 77D65E74
00CD0052	E8 B80F330F	CALL import.1000100F
00CD0057	C3	RETN
00CD0058	68 6A70D377	PUSH 77D3706A
00CD005D	E8 AD0F330F	CALL import.1000100F
00CD0062	C3	RETN
00CD0063	68 B779D277	PUSH 77D279B7
00CD0068	E8 A20F330F	CALL import.1000100F
00CD006D	C3	RETN
00CD006E	68 0CB7D577	PUSH 77D5B70C
00CD0073	E8 970F330F	CALL import.1000100F
00CD0078	C3	RETN

Figure 53 – The rest of the table.

FUNCION_LOG in our case begins at 1000100F:

10001004	CC	INT3
10001005	\$v E9 C6020000	JMP import.10001200
1000100A	\$v E9 F10B0000	JMP import.10001C00
1000100F	\$v E9 6C000000	JMP import.10001080
10001014	\$v E9 A7040000	JMP import.100014C0
10001019	\$v E9 22000000	JMP import.10001040
1000101E	CC	INT3
1000101F	CC	INT3

Figure 54 – The start of Function Log.

As you can see, I already put a BPX with F2 in the jump that takes us to the function, because we're going to need it.

10001080	> 55	PUSH EBP
10001081	. 8BEC	MOV EBP,ESP
10001083	. 83EC 40	SUB ESP,40
10001086	. 53	PUSH EBX
10001087	. 56	PUSH ESI
10001088	. 57	PUSH EDI
10001089	. 8D7D C0	LEA EDI,DWORD PTR SS:[EBP-40]
1000108C	. B9 10000000	MOV ECX,10
10001091	. B8 CCCCCC	MOV EAX,CCCCCC
10001096	. F3:AB	REP STOS DWORD PTR ES:[EDI]
10001098	. 60	PUSHAD

Figure 55 – Function Log.

That's the beginning of the log function, but we're interested in the end of the function. So we must put a BPX in the final RET of the function:

10001247	. 5E	POP ESI
10001248	. 5B	POP EBX
10001249	. 83C4 40	ADD ESP,40
1000124C	. 3BEC	CMP EBP,ESP
1000124E	. E8 9D0C0000	CALL import.10001EF0
10001253	. 8BE5	MOV ESP,EBP
10001255	. 5D	POP EBP
10001256	. C3	RETN
10001257	CC	INT3
10001258	CC	INT3

Figure 56 – The RET of the function.



Now let's go back to the table of jumps of the program.

Now we need some previous info to work, because if we start here, we won't know what functions does the program use and, even if we knew using the KAM for instance, we should also know where the jumps to user32.dll, kernel32.dll, etc., are. We'll use this when there are only a few functions missing and we'll use the first mechanism to find the rest of them. Otherwise it won't make much sense to do all the work manually, that would be the Unofficial ExeCryptor Crackme.

For instance, let's look for all the functions that weren't in the log and that we had to get by playing with the crack for a while. If my memory serves me well, they were MessageBoxA, GetDlgItemTextA (from user32.dll) and strlenA (from kernel32.dll).

Let's suppose that we have the complete table already and only the above functions missing.

Let's mark in the picture the jumps that are still left to be repaired:

004014BD	CC	INT3	
004014BE	- FF25 10204000	JMP DWORD PTR DS:[402010]	
004014C4	- FF25 0C204000	JMP DWORD PTR DS:[40200C]	
004014CA	- FF25 08204000	JMP DWORD PTR DS:[402008]	
004014D0	- FF25 20204000	JMP DWORD PTR DS:[402020]	
004014D6	- FF25 18204000	JMP DWORD PTR DS:[402018]	
004014DC	- FF25 1C204000	JMP DWORD PTR DS:[40201C]	
004014E2	- FF25 34204000	JMP DWORD PTR DS:[402034]	
004014E8	- FF25 24204000	JMP DWORD PTR DS:[402024]	
004014EE	- FF25 28204000	JMP DWORD PTR DS:[402028]	
004014F4	- FF25 2C204000	JMP DWORD PTR DS:[40202C]	
004014FA	- FF25 30204000	JMP DWORD PTR DS:[402030]	
00401500	- FF25 00204000	JMP DWORD PTR DS:[402000]	

← una
← dos
← tres

Figure 57 – The Missing functions.

As I said before, we assume that we have the rest of the functions and we know which ones are missing.

If we have the rest of the functions, we know where the jumps to the functions of kernel32, user32, etc., are. In this case, we know that the first two were from kernel32, then the one missing and then the jumps to the functions from user32.dll. So this jump is either the last from kernel32 or the first from user32. Which one do we choose? Well, let's try. I have the dll which intercepts functions from user32.dll loaded, so I'll go for the other two first 'cause they're very likely to be from user32.dll since they're among the others. So we'll start with "tres" and "dos" and then we'll go to "uno".

At the end of this table of jumps we have some free space in which we'll write a JMP to "tres":

00401508	0000	ADD BYTE PTR DS:[EAX],AL
0040150A	0000	ADD BYTE PTR DS:[EAX],AL
0040150C	EB E6	JMP SHORT b8.004014F4
0040150E	0000	ADD BYTE PTR DS:[EAX],AL
00401510	0000	ADD BYTE PTR DS:[EAX],AL
00401512	0000	ADD BYTE PTR DS:[EAX],AL

Figure 58 – Writing a JMP to "tres".

We change the EIP by doing a NewOriginHere at the jump:

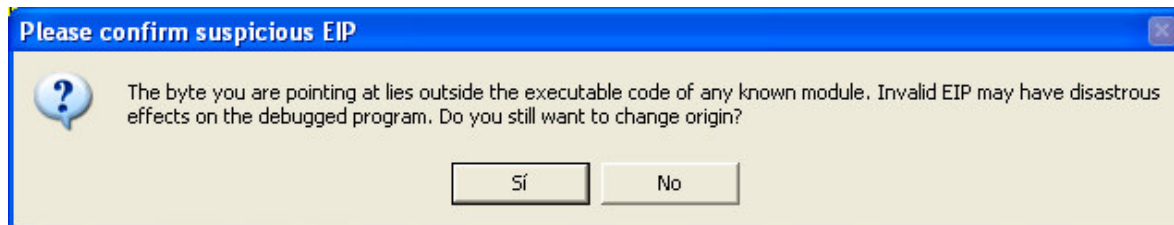


Figure 59 – Changing the EIP.



00401508	0000	ADD BYTE PTR DS:[EAX],AL
0040150A	0000	ADD BYTE PTR DS:[EAX],AL
0040150C	EB E6	JMP SHORT b8.004014F4
0040150E	0000	ADD BYTE PTR DS:[EAX],AL
00401510	0000	ADD BYTE PTR DS:[EAX],AL
00401512	0000	ADD BYTE PTR DS:[EAX],AL

Figure 60 – Writing a JMP.

Now we press F9 (Run) and the program stops at the first of the two BPXs:

1000100F	E9 6C000000	JMP import.10001080
10001014	E9 A7040000	JMP import.100014C0
10001019	E9 22000000	JMP import.10001040
1000101E	CC	INT3
1000101F	CC	INT3
10001020	CC	INT3

Figure 61 – Stop in the BPX.

If we look at the Stack, we can see that between both returns there's an address:

001EFFC4	0015078A	RETURN to 0015078A from import.1000100F
001EFFC8	77D3B00A	USER32.77D3B00A
001EFFCC	77F4F31F	RETURN to ntdll.77F4F31F from ntdll.DebugBreakPoint
001EFFD0	00000005	
001EFFD4	00000004	

Figure 62 – Stack – Address of the function.

Let's copy that address and let's assemble, in another Olly, a jump to it:

00401000	- E9 05A09377	JMP user32.MessageBoxA
00401005	90	NOP
00401006	90	NOP
00401007	A3 78324000	MOV DWORD PTR DS:[403278],EAX

Figure 63 – JMP to MessageBoxA.

Now we press F9 again and it stops at the RET of the LOG function:

10001248	5B	POP EBX
10001249	83C4 40	ADD ESP,40
1000124C	3BEC	CMP EBX,ESP
1000124E	E8 90C00000	CALL import.10001EF0
10001253	8BE5	MOV ESP,EBP
10001255	5D	POP EBP
10001256	C3	RET
10001257	CC	INT3

Figure 64 – We stop in the RET.

We change the EIP in the Stack so that it returns to the jump:

001EFFC4	0040150C	b8.0040150C
001EFFC8	77D3B00A	USER32.77D3B00A
001EFFCC	77F4F31F	RETURN to ntdll.77F4F31F from ntdll.DebugBreakPoint
001EFFD0	00000005	
001EFFD4	00000004	
001EFFD8	00000001	

Figure 65 – We can change the return of the call.



00401508	0000	ADD BYTE PTR DS:[EAX],AL
0040150A	0000	ADD BYTE PTR DS:[EAX],AL
0040150C	EB E6	JMP SHORT b8.004014F4
0040150E	0000	ADD BYTE PTR DS:[EAX],AL
00401510	0000	ADD BYTE PTR DS:[EAX],AL
00401512	0000	ADD BYTE PTR DS:[EAX],AL
00401514	0000	ADD BYTE PTR DS:[EAX],AL

Figure 66 – JMP to JMPs table.

Of course, I put a BPX beforehand so that I don't miss the execution ;)

Now I change the address of the jump and point it to “dos”, that is, JMP 4014E2:

00401508	0000	ADD BYTE PTR DS:[EAX],AL
0040150A	0000	ADD BYTE PTR DS:[EAX],AL
0040150C	EB 04	JMP SHORT b8.004014E2
0040150E	0000	ADD BYTE PTR DS:[EAX],AL
00401510	0000	ADD BYTE PTR DS:[EAX],AL

Figure 67 – JMP to JMPs table.

We press F9 (Run) again and it stops at the log function.

And we see the address of the function in the Stack again:

001EFFC0	0015096A	RETURN to 0015096A from import.1000100F
001EFFC4	77D365B2	USER32.77D365B2
001EFFC8	77D3B00A	USER32.77D3B00A
001EFFCC	77F4F31F	RETURN to ntdll.77F4F31F from ntdll.DbgBreakPoint
001EFFD0	00000005	
001EFFD4	00000004	

Figure 68 – Stack of Log Function.

Now, in another Olly we assemble a JMP to this address (JMP 77D365B2):

00401000	- E9 05A09377	JMP user32.MessageBoxA
00401005	- E9 A8559377	JMP user32.GetDlgItemTextA
0040100A	90	NOP
0040100B	90	NOP
0040100C	. E8 EF040000	CALL <JMP.&comctl32.InitCommonC

Figure 69 – Writing a JMP's.

For the last function, the best thing would be to modify the source so that it loads the more dlls possible, or at least the best known ones (kernel32.dll, user32.dll, comctl32.dll, etc.,). So I made a table for each module. Otherwise we would have to try with each dll until we find the right one.

In this case, our only chance is to change the dll to look for functions from kernel32.dll, 'cause Olly, when we repeat the procedure, stays “Running” and nothing happens. So this could be a good sign to change dlls, he-he.

We proceed as before, we modify the dll to look for functions from kernel32.dll, we run the injector, we attach the program and we assemble the jump to the function missing:

0040150C	0000	ADD BYTE PTR DS:[EAX],AL
0040150E	0000	ADD BYTE PTR DS:[EAX],AL
00401510	0000	ADD BYTE PTR DS:[EAX],AL
00401512	EB B6	JMP SHORT b8.004014CA
00401514	0000	ADD BYTE PTR DS:[EAX],AL
00401516	0000	ADD BYTE PTR DS:[EAX],AL
00401518	0000	ADD BYTE PTR DS:[EAX],AL

Figure 70 – Writing a JMP, the last one.



We change the EIP and press F9 (Run) and it stops at the beginning of the LOG function:

1000100F	E9 6C000000	JMP import.10001030
10001014	E9 A7040000	JMP import.100014C0
10001019	E9 22000000	JMP import.10001040
1000101E	CC	INT3
1000101F	CC	TNT3

Figure 71 – Stop in the BPX.

And if we look at the Stack, we have the address of the missing function:

001E5000	0015000A	RETURN to 0015000A from import.1000100F
001E5008	77E560E1	kernel32.77E560E1
001E500C	77F4F31F	RETURN to ntdll.77F4F31F from ntdll.DebugBreakPoint
001E5010	00000005	
001E5014	00000004	
001E5018	00000003	

Figure 72 – Stack of LOG function.

We go to another Olly and assemble a jump to that address:

00401000	- E9 05A09377	JMP user32.MessageBoxA
00401005	- E9 A8559377	JMP user32.GetDlgItemTextA
0040100A	- E9 D250A577	JMP kernel32.lstrlenA
0040100F	90	NOP
00401010	90	NOP
00401011	6A 00	PUSH 0
00401012	20 2E104000	PUSH eax

Figure 73 – Writing more JMPs.

With this, we have the complete table, but it still takes us long to fill the table entry by entry. So my idea isn't other but to automatize this last mechanism proposed by AkirA via an inject which will change the jump so that it points to each of the jumps in the table. Then, when it stops at the log function, it'll take the address from the Stack and store it in the corresponding memory address. Then it'll change the return of the function and repeat everything all over again. As I said, this is not too complicated but the more dlls are loaded in memory, the better. So that every function could be found in some table, otherwise we won't know where the functions of each module begin. So the first thing is to modify the source of the tracer 4.c to do this. I'm not sure how to do that, probably just adding more lines like:

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

will suffice. If any of the list members (kaos_xlro or AkirA) can send me the source, I'll continue with this.

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 is the end. This new method is very powerful but try it by your self =)

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

Of course, the first thanyou goes to AkirA for giving us this new possibility and second and not less important, to kaos_xlro who was the one that taught me and helped me with this subject. Also the people from VirtualMachine. I also want to say hello to k0rt, ELVIS[em], Joe Cracker, SyXe and EagleCrack who I always find in MSN, although I'm always about to leave (sorry for that). I also say hello to Ricardo with whom I dreamed last night. I very much look forward to meet him and I'm travelling to Buenos Aires, Argentina soon.

And of course to everybody in the CracksLatinoS! 2005 list and all the people in ARTeam, no exceptions.

See you soon.

+NCR/CRC! [ReVeRsEr]

Saturday, November 12th, 2005



ncr.iespana.es