



Cracking Using Partial Dumps: the case of Delphi

Shub-Nigurrath of ARTeam

Version 1.1 - December 2005

1.	Abstract	2
2.	Introduction to the problem.....	4
3.	Introducing the Partial Dumps	6
3.1.	Using DeDe.....	8
3.2.	Importing results into OllyDbg	12
4.	Patching the application	13
4.1.	Trial limitation	13
4.2.	Interventions on the code	13
4.2.1	Patch the onExit messagebox and expiration.....	13
4.2.2	Patching the AboutBox	16
4.2.3	Patching the Evaluation string in the Titlebar.....	19
5.	Loader's code	21
6.	Conclusions.....	22
	Appendix I. Delphi Compilation Process.....	23
	Appendix II. Script for reaching the OEP	25
7.	References.....	27
8.	History.....	27
9.	Greetings.....	27

Keywords

Loader, AsProtect, Delphi



1. Abstract

This tutorial introduces a novel technique (at least I never seen it around) that might be useful to understand the code of packed programs, written with non C/C++ languages. The technique results into a more readable assembler and definitely a faster way to understand where to patch the program in order to write an inline patch or a loader.

As usual I will provide sample code with this tutorial. All the sources have been tested with Win2000/XP and Visual Studio 6.0.

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.

Whenever used a commercial application is used just as a practical example, the used applications are chosen if they pass the following 3 conditions: it is a good example, there are already several patches around for the same version, the version used has been already updated. Our intention is to damage the less possible developers, but you know sometimes you have to go in the real world.

If you have ever seen the disassembler view of an application written with Delphi, you should already know that it is quite different to the same assembler of C/C++ programs (see [1] for a comparison). The Delphi compiler creates the assembler using different calling conventions, compiles the calls doing a completely different use of the data stack (let think that for example all the local or global variables are stored into the stack, relative to EBP: [ebp+xy] means that is pointer on global variable, [ebp-xy] means that it is pointer on local variable). See also Appendix I.

Moreover Delphi's assembler is generally built in a way that doesn't allow OllyDbg to build the calls, so whenever you stop at a specific handler the call stack is generally empty and you don't exactly know from where you're coming. The only information you can use are the return addresses from the data stack (see [2]). Despite it's often the only way to go, this is not the most efficient way to go often, one would have a better readable assembler.

Fortunately tools like DeDe [3] are thought to analyze Delphi programs in order to extract all those information that OllyDbg doesn't know how to get.

Up to now the situation is well known and you also much probably already know what I just told, because you already used DeDe (if not this is not the tutorial for it, see instead for example again at tutorial [1] to see how DeDe can improve the situation with OllyDbg).

Some problems arise when the Delphi application is protected or packed so as you cannot use DeDe on it, and you cannot also attach to the running process, because DeDe is not able to do it or because DeDe simply arrive too late on the application. With these situations you have to use the Partial Dumps mentioned in the title.

DeDe of course has a feature to attach to a running process to do extract the same information, but if the application is left running freely before DeDe has a the first chance to attach to it, you can simply arrive too late when most of the things have already been done by the application. The only



option you have is to attach to the application from the EP, or from the OEP. There are some differences then which vote for the use of Partial Dumps..

The tutorial will use SmartWhois [4] which is a perfect example for this technique.

Of course the same considerations can be taken to the VB world where instead of DeDe we would talk of P32Dasm or other exotic compiled languages.

Before continue reading I suggest that you have already understood how a program is loaded into memory, what is a loader, what it does and how and which are the differences among Debug and Standard Loaders. I suggest reading [5, 6, 7] to better understand the rest of this tutorial.

Have phun!

Shub-Nigurrath

December 2005



2. Introduction to the problem

We will start using our example SmartWhois, because as I already mentioned it suits our needs: it's a Delphi application, it's protected using AsProtect 2.11 SKE (see Figure 1).

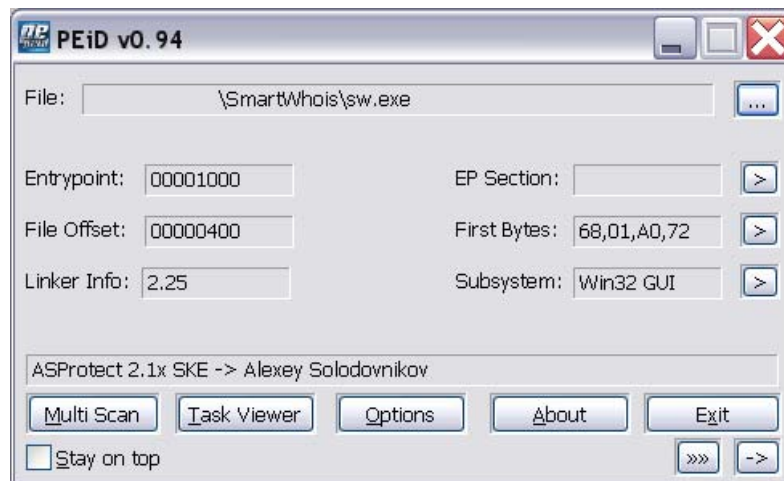


Figure 1 – The application is protected with AsProtect

You have as usual different ways to approach the problem, but these are the main two:

1. unpack AsProtect and restore and unpacked application, then (using DeDe) analyze it and patch. The problem is that of course you will have to distribute the whole executable, executable that will work only on the Windows family (2000,XP,...) on which you did the work (due to the OriginalFirstThunk)
2. an much more elegant way IMHO is to write a loader, which allows to distribute a smaller program, running on all systems, harder to reverse to understand what you did to crack the application and possibly working also on future versions.

It should be clear what we will do, a loader ;-)

I'm going to demonstrate that we will soon meet a problem: the assembler of the application is hard to understand because it's Delphi and DeDe cannot be of any help.

Run the application into OllyDbg and stop at the EP.

00401000	68 01A07200	PUSH sw.0072A001	
00401005	E8 01000000	CALL sw.0040100B	
0040100A	C3	RETN	
0040100B	C3	RETN	
0040100C	9A 08B59DCC 24	CALL FAR 5824:CC9DB508	Far call
00401013	DA0C54	FIMUL DWORD PTR SS:[ESP+EDX*2]	
00401016	0D FAC077D7	OR EAX,D777C0FA	
0040101B	7B C1	JPO SHORT sw.00400FDE	
0040101D	BA F96485C5	MOV EDX,C58564F9	
00401022	CA 46F6	RETF 0F646	Far return

Figure 2 - EP of the program

The AsProtect structure is almost the same of the version 2.0 for example here:

```
012BCC7F    BA 60CE2B01    MOV EDX,12BCE60    ; ASCII ".key"
012BCC84    B8 00000080    MOV EAX,80000000
```



Cracking Using Partial Dumps: the case of Delphi

```
012BCC89 E8 FACEFEFF CALL 012A9B88
012BCC8E 68 70CE2B01 PUSH 12BCE70 ; ASCII "regfile"
012BCC93 33C9 XOR ECX,ECX
012BCC95 BA 60CE2B01 MOV EDX,12BCE60 ; ASCII ".key"
```

The only anti-debut trick is the usual IsDebuggerPresent which is easily patched with an OllyDbg plug-in or manually.

Before pressing any key or in case OllyDbg doesn't stops at the EP, take care to set OllyDbg not to manage the exceptions for you as in Figure 3.

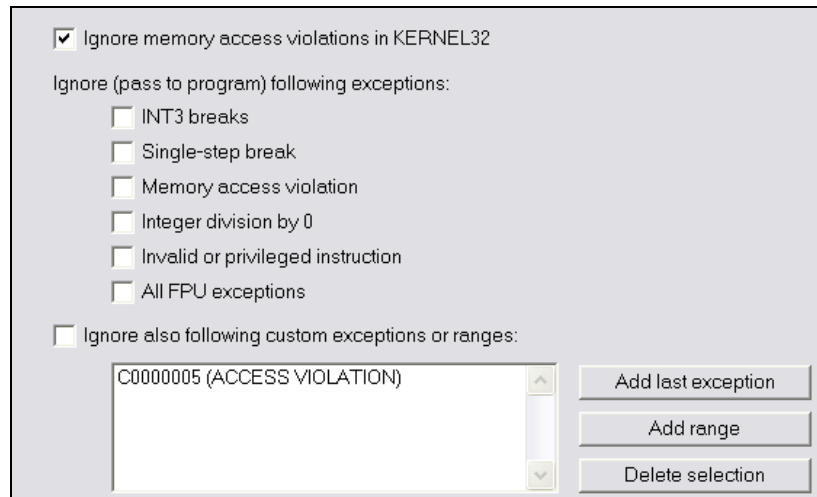


Figure 3 - OllyDbg Exception settings

Now we have to go to the unpacked application's code to see how it looks like. Follow these two steps:

1. when stopped for an exception into OllyDbg, press SHIFT-F9 to pass the exception to the program (it knows what to do with it anyway) and mentally count the number of exceptions before the program goes into the running mode and doesn't stops anymore
2. relaunch the application using CTRL-F2 (or menu) and stops at the last nth exception. You should see something as the following¹:

```
012BA6E9 C601 1E MOV BYTE PTR DS:[ECX],1E ; stopped here
012BA6EC 0807 OR BYTE PTR DS:[EDI],AL
012BA6EE 2C A7 SUB AL,0A7
012BA6F0 FD STD
012BA6F1 5B POP EBX ; 0012F870
012BA6F2 - E9 67648F06 JMP 07BB0B5E
```

NB: keep this piece of code away because we will need it later on when creating the loader.

3. Now we are able to discover that the original application was written with Delphi: while being stopped at the exception of point above, use "search for all referenced strings" in OllyDbg and you should get something clearly referring to Delphi², for example:


```
012A29E8 PUSH 12A2A68 ASCII "SOFTWARE\\Borland\\Delphi\\RTL"
```

¹ Addresses of where you will meet this code are of course computer dependant.

² This is not the only way, also the running application reveals it's Delphi nature. If you use a Spy utility to reveal applications windows' data, you will find for example that the main window's Class is TMainForm, which is typical of Delphi.



Now the application is ready to be executed in memory, because the AsProtect work is terminated (we are at its last exception remember). All it needs to do now is just pass the control to the application's code.³

To intercept the event what we have to do is to go into the memory view of OllyDbg (using ALT-M or the button ) and select the code section of our process (see Figure 4 address 401000) and Set there a Memory Breakpoint on Access.

00400000	00001000	SW	00400000 (itself)		PE header	Imag	R	RWE
00401000	0023A000	SW	00400000		code	Imag	R	RWE
0063B000	0000E000	SW	00400000		data	Imag	R	RWE
00649000	00006000	SW	00400000			Imag	R	RWE
0064F000	00004000	SW	00400000			Imag	R	RWE
00653000	00001000	SW	00400000			Imag	R	RWE
00654000	00001000	SW	00400000			Imag	R	RWE
00655000	00022000	SW	00400000			Imag	R	RWE
00677000	000B3000	SW	00400000	.rsrc	resources	Imag	R	RWE
0072A000	00027000	SW	00400000	.hala	imports,rels	Imag	R	RWE
00751000	00001000	SW	00400000	.adata		Imag	R	RWE

Figure 4 - memory map of the process

Once you did this you can continue to execute the application, paying attention to pass each single exception to the debugger process (otherwise it will terminate with an error).

The next stop is at the OEP.

```
0063A214  55          PUSH EBP
0063A215  8BEC        MOV EBP,ESP
0063A217  83C4 EC     ADD ESP,-14
0063A21A  53          PUSH EBX
0063A21B  33C0        XOR EAX,EAX
0063A21D  8945 EC     MOV DWORD PTR SS:[EBP-14],EAX
0063A220  B8 FC986300 MOV EAX,sw.006398FC
0063A225  E8 DACDCFF  CALL sw.00406F04
```

Now remove the memory on access breakpoint. See Appendix II to use a script instead.

3. Introducing the Partial Dumps

Once you are at the OEP try getting around the application, for example going around the most meaningful strings. If you have a little of sensibility for assembler you will see that it's full of apparently strange calls, movs and so on (more than usual I mean).

The call stack is often full of calls like where the destination address is taken from addresses tables As for example CALL DWORD PTR DS:[EBX+60] and the functions themselves are strange somehow..

I'm intentionally not clear because there are already around some good tutorials on Delphi (for example [1]) and because I really suggest doing this experiment realizing these differences on your own.

Now, in order to understand what I mean with Partial Dumps, dump the program (you should be stopped at the OEP, if not please do).

For example I used OllyDump as in Figure 5 (note, no rebuild of IAT).

³ ..and unfortunately manage most of the IAT entries

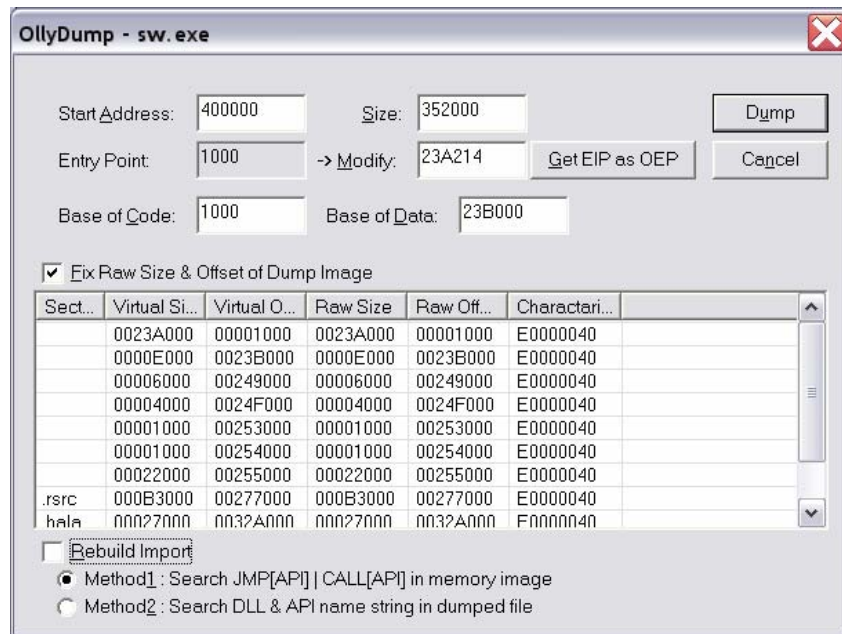


Figure 5 - OllyDump is ready to dump the application

Save it as dump.exe or whatever you like. Of course the program won't run because of the IAT elimination features of AsProtect and we did nothing to repair this IAT. So the IAT is still broken and the dump will not work at all, also ImpRec is of no use at this stage: we should have done something before reaching the OEP, but this is not a tutorial on MUPing AsProtect (see here if you want http://pnluck.altevista.org/soft-tute/Aspro_SKE_211_Esyst%28ingl%29.html) .

The dump is then “partial” because of the missing IAT, but even if it's not able to run the application it is instead good for a static code analyzer as DeDe. An indeed we will use this approach.

Before going on anyway do a run to ImpRec to fix a little the dump you just created, at least by the PE Header point of view.

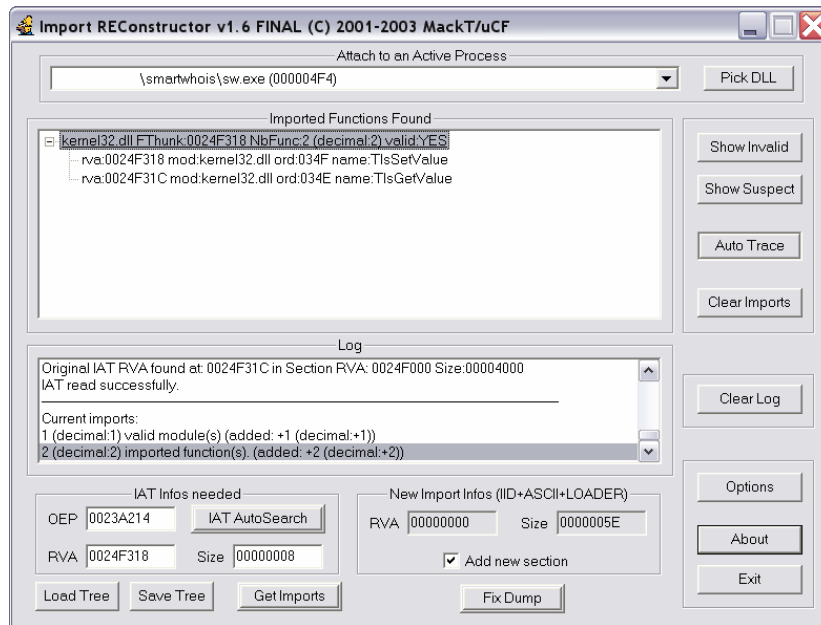


Figure 6 - ImpRec recovered only two imports from the process

As you can see from Figure 6, there are only two imports which have been recovered, this is because AsProtect does a really good work with IAT elimination, but we will not worry of this here. What we want it to use ImpRec to align sections and fix the PE Header so as to not give problems to DeDe.

3.1. Using DeDe

DeDe is an extremely powerful disassembler for Delphi programs. It uses the Delphi libraries used by the linker to understand where the signatures of the Delphi instructions inside the compiled program are. This, beside the Forms decompilation (more or less like in VB code) allows DeDe to decompile the graphical resources and insert comments about the original Delphi instructions into the assembler code.

DeDe is quite a well known and simple program and comes with several tutorials that should let you understand how to use it. I will go straight then to my final target without worrying of additional features DeDe might offer to you (just try playing with it, you're a reverser, not a dummy! ;-)).

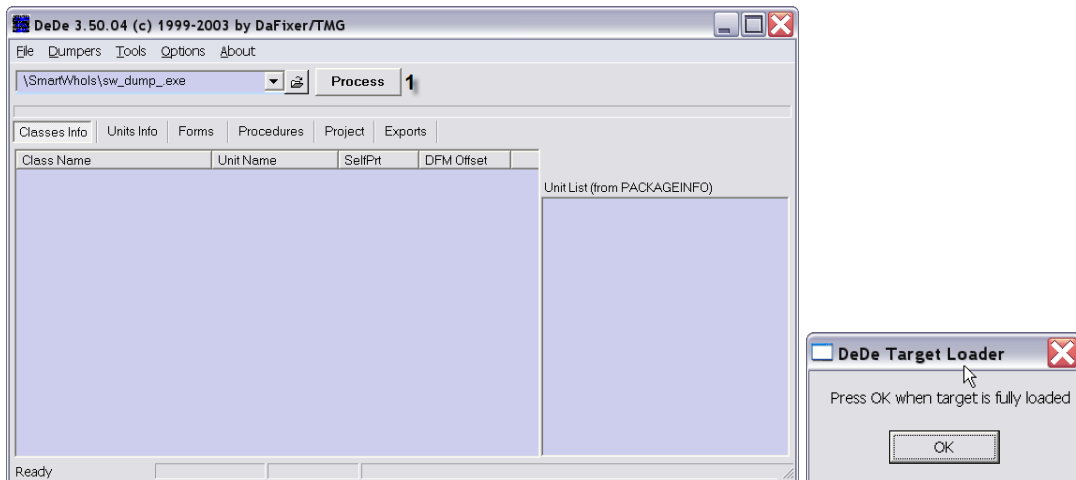
As I already told before DeDe has an "attach to active process" feature, which allows attaching to a running process's memory and do the decompilation. The main question some might have at this stage is then: Which is better to use DeDe attaching it to a process (already decompiled) or run it on a Partial Dump? This not indeed a dilemma because there are some differences which vote for the use of Partial Dumps

On the one hand, attaching to a running process might happen only when the application sets itself in the ready state (before things happens too quickly and you cannot be so fast) and generally you have the change to attach with DeDe when the initial applications settings have already been done. The DeDe analysis is then less accurate and often doesn't get all the possible information.

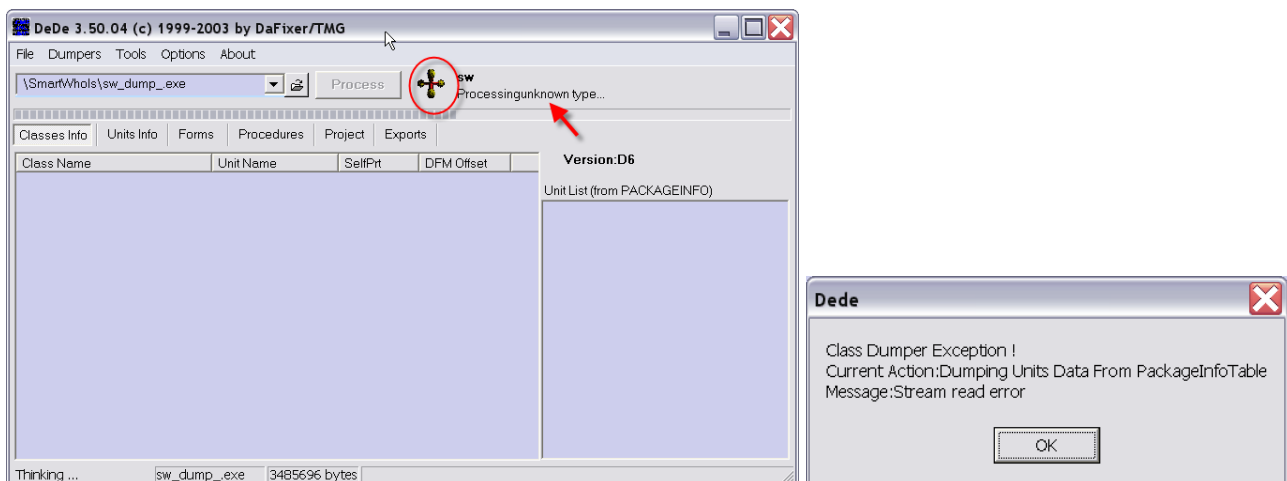
On the other hand if you would be able to open the application from DeDe (instead of attaching it) it would be better, because DeDe in that case would be able to get all the information out without sending the program into execution.



It's time to open DeDe⁴ and load the Partial Dump, fixed a little, we duly prepared at Section 3.

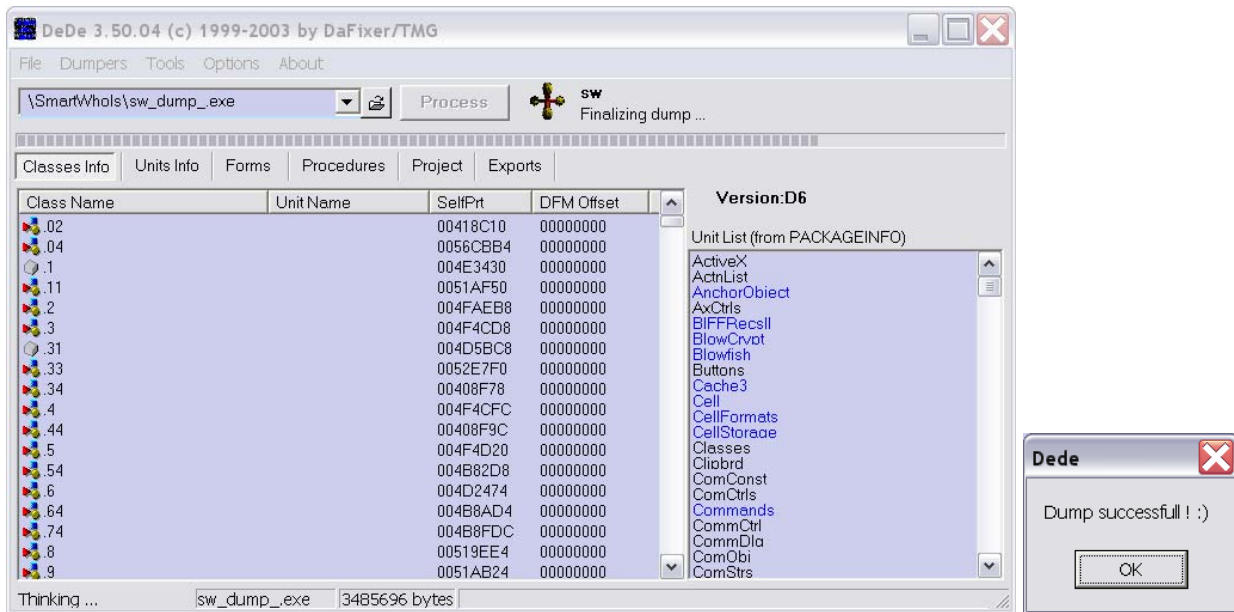


DeDe is processing things, even if with some error. These errors are irrelevant for us. For example note that if the application shows a system crash is normal because it's not a working dump, so ignore to let DeDe continue.

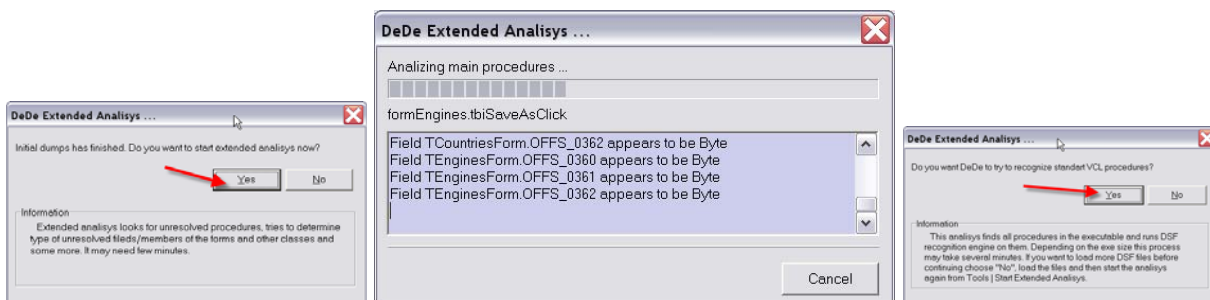


After a while the initial analysis is terminated..

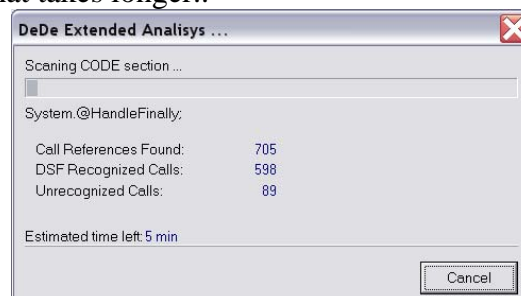
⁴ If the version of DeDe you downloaded is rather old you might see warnings telling you that some files are old, simply ignore them. Anyway these files are available on the web or from the installation of a Delphi compiler.



Of course you will have to do all the extended analysis DeDe proposes to you:



The last analysis is the one that takes longer..



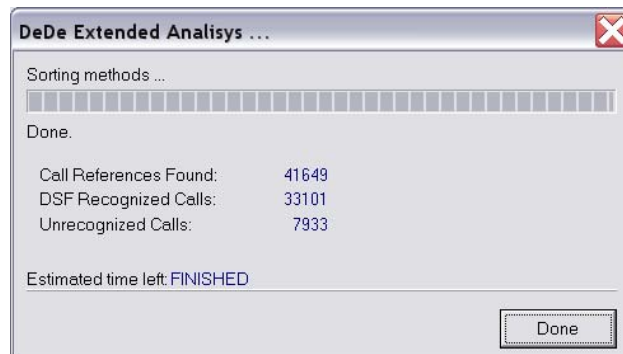
Meanwhile I can tell you why the partial dump you did is analyzed by DeDe, despite the errors it contains. The problem is that AsProtect is thought to remove the information required for a program to run, that are mainly the imports the system calls and so on. Being the Windows kernel written in C/C++ this also means for a C/C++ program that AsProtect will remove mostly all it's vital information, everything that's needed to understand what it does. There is an higher adherence between the asm code of a C/C++ program and the system.

With Delphi programs (or VB as well) instead the situation is different. There are big language specific libraries implementing the language functions, that are not directly translated into system API calls, but into long asm sources that might also eventually include a system call, but often it might not. This is also the reason why Delphi Asm code is less readable, because apparently does a lot of "strange" things.



AsProtect then is less effective for these binaries because it cannot remove to the DeDe inspections all the things that would have been better to remove.

DeDe is then able to read our partial dump, it doesn't worry if it can run or not (it doesn't worry too much, indeed), the information it requires to partially decompile the Delphi program are present and are automatically recognized.



DeDe recognized a lot of calls, well! Now we can start using DeDe, see the Forms, see the commented asm code and look at the events of any Form.

For us this is less interesting for the moment, because we want to do two things

1. dump all the decompilation results into text files for a later using
2. create a MAP file to be used inside OllyDbg (using MapConv, [6])

Press on Projects and select any folder you like, as in Figure 7, then press "Create files"

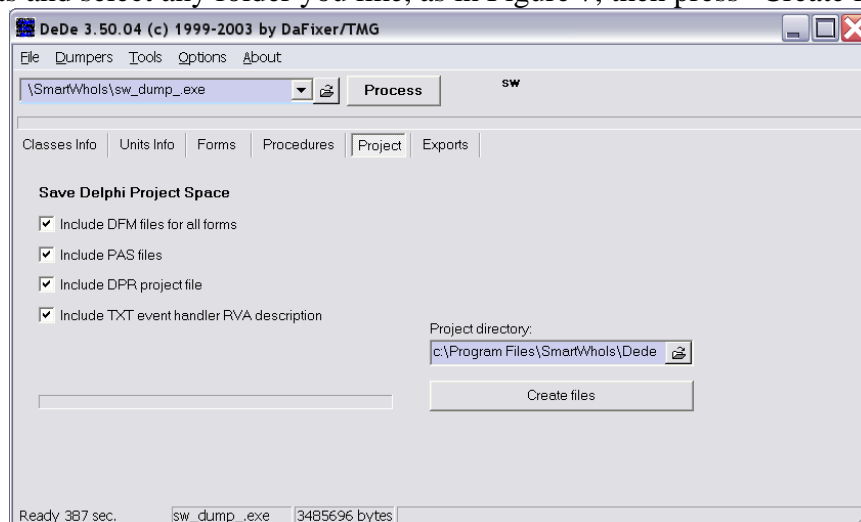


Figure 7 - Projects export for DeDe

It might happen that DeDe terminates this operation reporting that there have been some errors. If you go to see the .txt file where DeDe logged the errors, you can easily see that these errors are related to specific API calls and not so important for us.

Now do the last step with DeDe pressing "Exports" and setting things as in Figure 8 (except for the path which is as you like ;) and press "Create Export file".

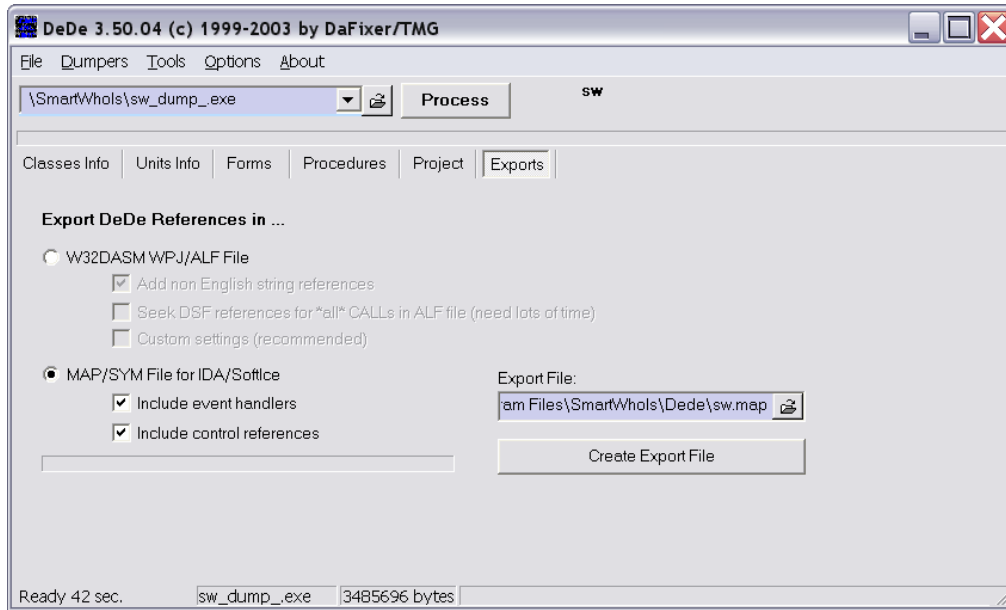


Figure 8 - Exports feature of DeDe

Then you have a good .map file you can import inside OllyDbg⁵.

Now you can close DeDe or leave it open, I prefer to close it and browse the saved files with textuuls searches, it's faster then navigating through the DeDe interface, and allows you to seek all the places where eventually a specific address or name is referenced

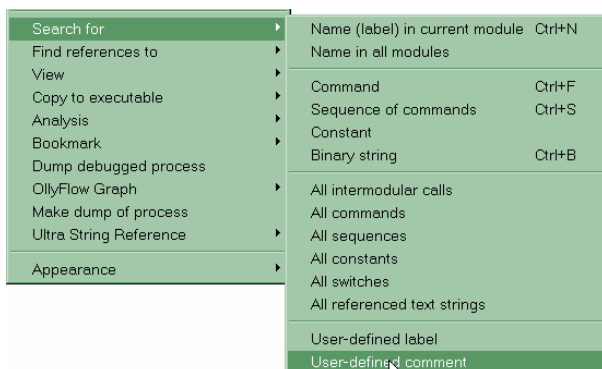
3.2. Importing results into OllyDbg

Once you have a valid map file created from DeDe and a valid Export of all its findings you are ready to return to Olly (we left it stopped at the application's OEP).



You will have to use plugin [6] to insert the content of the MAP file as comments and labels for Olly, using twice the plugin⁶.

The result is a much more readable asm code.



To understand the extension of what you just did try searching for all the user comments and labels with Olly!

Moreover DeDe produced for you some .pas files and a file called "events.txt" which is useful to search if a specific Delphi function is called and where it is.

At this stage you have the instruments to perform two opposite approaches:

⁵ Included also into this tutorial's archive

⁶ The MAP format is quite simple, just a textual list of addresses with the relative comments and names, it's a standard supported by several programs, including IDA.



1. Start from the events of a Form, which now are clearly commented digging down to see what happens, up to the patch point. *This is a top-down approach.*
2. Start from the Delphi functions (from the events.txt) and see who's calling them to climbing up to the patch point. *This is a bottom-up approach.*

During the patching of the application we will use both.

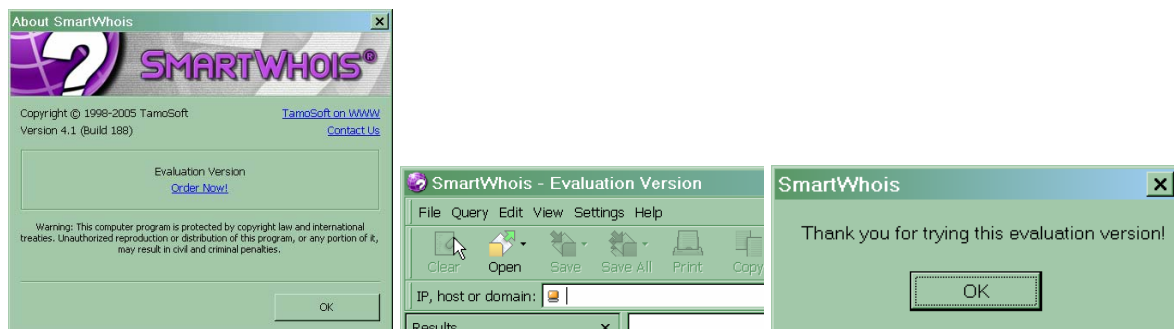
4. Patching the application

As already told the application used is SmartWhois [4] version 4.1. The application has been used because suites with the argument of the present tutorial; several patches are already around from independent sources; the application has been recently updated.

4.1. Trial limitation

The limits of the applications are quite classic:

1. trial limit of 30 days after which the application expires
2. Evaluation version indication on the titlebar
3. messagebox on exiting



4.2. Interventions on the code

The first consideration before approaching the problem is that the title bar is something containing the indication that the application is not registered, so is a good point where to dig in order to understand what happens there.

4.2.1 Patch the onExit messagebox and expiration

For what concerns the MessageBox on exit we can use the top-down approach⁷.

First of all we need to know which the class type of the main form is. We can use any Spy utility (e.g. Spy++ installed with VC6.0). The result is the one reported in Figure 9. The same approach can be used to find all the Class types of any interface components. The found name must be used to navigate the files created by DeDe.

⁷ Well this is a simple case and you can in a snap arrive to the point thanks to the MessageBox, but this is a general way to work.

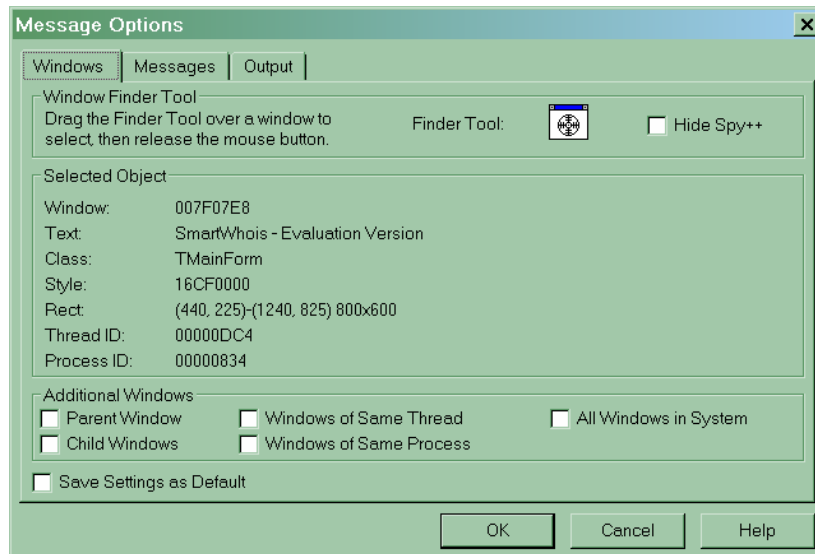


Figure 9 - Using Spy++ to find the Class type of the main window

The main form is of type TMainForm, then searching through all the files created by DeDe we find that it's into the file *formMain.pas* (obvious, but this is to describe the general procedure). Open it with a text editor and among the Form events it should be clear that there is a destroy procedure.

```
procedure TMainForm.FormDestroy(Sender : TObject);
begin
  (*
    00621BE0  55                push    ebp
    00621BE1  8BEC             mov     ebp, esp
    00621BE3  51              push    ecx
    00621BE4  8945FC           mov     [ebp-$04], eax
    00621BE7  8B45FC           mov     eax, [ebp-$04]

    00621BEA  E821040000       call    00622010
    00621BEF  55              push    ebp

    00621BF0  E857FFFFFF       call    00621B4C
    00621BF5  59              pop     ecx
    00621BF6  33C0            xor     eax, eax

    * Reference to GlobalVar_0064EC04
    00621BF8  A304EC6400       mov     dword ptr [$0064EC04], eax
    00621BFD  59              pop     ecx
    00621BFE  5D              pop     ebp
    00621BFF  C3              ret

  *)
end;
```

Well, place a BP on 00621BE0⁸.

Now, with Delphi as with MFC any application is included into a wrapper which implements the initial applications' logic. For Delphi this is the SystemForm, which creates the mainForm. Thus you have to also look at the *formSystem.pas*.

Open it and look the form events inside, there are two interesting:

```
00630620 procedure TSmartWhoisSystemForm.FormDestroy(Sender : TObject);
00630638 procedure TSmartWhoisSystemForm.FormCloseQuery(Sender : TObject);
```

⁸ Note that you should find there a comment imported from DeDe.



Place two BP at the beginning of each one and press RUN on OllyDbg. Press the [x] button inside SmartWhois and you should land at

```
00630638 > C680 28030000 01 MOV BYTE PTR DS:[EAX+328],1
; *TSmartWhoisSystemForm.OFFS_0328:Byte
```

which is the second function above, despite the comment⁹.

00630638	. C680 28030000 01	MOV BYTE PTR DS:[EAX+328],1	*TSmartWhoisSystemForm.OFFS_0328:Byte
0063063F	. 80B8 29030000 00	CMP BYTE PTR DS:[EAX+329],0	*TSmartWhoisSystemForm.OFFS_0329:Byte
00630646	75 35	JNZ SHORT sw.0063067D	
00630648	. 8B15 E0896400	MOV EDX,DWORD PTR DS:[6489E0]	sw.006481AC
0063064E	. 803A 00	CMP BYTE PTR DS:[EDX],0	
00630651	75 2A	JNZ SHORT sw.0063067D	
00630653	. 8B15 D8826400	MOV EDX,DWORD PTR DS:[6482D8]	sw.006481E8
00630659	. 803A 00	CMP BYTE PTR DS:[EDX],0	
0063065C	75 1F	JNZ SHORT sw.0063067D	
0063065E	. 8B15 B8876400	MOV EDX,DWORD PTR DS:[6487B8]	sw.00649C94
00630664	. 8B12	MOV EDX,DWORD PTR DS:[EDX]	
00630666	. 80BA 9C000000 00	CMP BYTE PTR DS:[EDX+9C],0	
0063066D	75 0E	JNZ SHORT sw.0063067D	
0063066F	. B1 01	MOV CL,1	
00630671	. B2 01	MOV DL,1	
00630673	. E8 A4000000	CALL sw.0063071C	does other things useful only for evaluation..
00630678	. E8 6F060000	CALL sw.00630CEC	this is the call that popup the messagebox

Figure 10 - call to the MessageBox on exit

If you look at the code, there's a simple call at 00630678 that is responsible for the call to the messagebox. You can patch the program as you like, modifying the above conditional jumps, nopping the call or digging inside the call. I patched the JNZ at 00630D0F.

Looking the code at 00630D0F it comes evident that there's a constant (006481E8) which is used to test if the application is expired or not: a value of 1 means expired; this conclusion is self evident looking at the following piece of code:

00630F5D	75 3F	JNZ SHORT sw.00630F9E	
00630F5F	. C605 E8816400 01	MOV BYTE PTR DS:[6481E8],1	
00630F66	. 56	PUSH ESI	
00630F67	. 6A 00	PUSH 0	
00630F69	. 8D4D F8	LEA ECX,[LOCAL_2]	
00630F6C	. A1 34866400	MOV EAX,DWORD PTR DS:[648634]	
00630F71	. 8B00	MOV EAX,DWORD PTR DS:[EAX]	sw.00630004
00630F73	. BA D40F6300	MOV EDX,sw.00630FD4	ASCII "S_EXP"
00630F78	. E8 E76BECFF	CALL sw.004F7B64	

A second patch is then to fix the JNZ at 00630F5D into a JMP, so as to avoid expiration.

The third patch is found looking where the other call at 00630673 in Figure 10 is called from: this call is in the same branch of the messagebox (which is called only here), so must have something connected to the trial status of the application.

Go to the first instruction of the call and press CTRL-R in order to see who's calling it. I will leave you away the effort of finding which the right call is, but the technique consist placing a BP on the first instruction of the call and move forward the clock. Now, press run (before you have to be at the OEP). You can easily see that the call is called by this expiration check:

⁹ This is due the fact that DeDe recognize the function in an early stage as a generic function and after as FormCloseQuery. Integrating Olly comments and DeDe files you solve the problem



006308D8	. A1 D8826400	MOV EAX,DWORD PTR DS:[6482D8]	
006308DD	8038 00	CMP BYTE PTR DS:[EAX],0	
006308E0	75 22	JNZ SHORT sw.00630904	
006308E2	E8 5D070000	CALL sw.00631044	
006308E7	84C0	TEST AL,AL	
006308E9	74 19	JE SHORT sw.00630904	
006308EB	B1 01	MOV CL,1	
006308ED	B2 01	MOV DL,1	
006308EF	8BC6	MOV EAX,ESI	
006308F1	E8 26FEFFFF	CALL sw.0063071C	
006308F6	33C0	XOR EAX,EAX	
006308F8	E8 3F060000	CALL sw.00630F3C	
006308FD	8BC6	MOV EAX,ESI	
006308FF	E8 54B0E3FF	CALL sw.0046B958	
00630904	> 8BC3	MOV EAX,EBX	
00630906	5E	POP ESI	sw.0046BA57
00630907	5B	POP EBX	sw.0046BA57
00630908	C3	RETN	

The JNZ at 006308E0 must be changes into a JMP. Moreover look at the test at 006308D8 just above, the constant involved (6482D8) is already referenced in Figure 10. Do you guess a specific meaning of it? Try it out.. ;-)

Meanwhile if you read this document having OllyDbg beside, you should still have it stopped at the following call:

```
00630673 . E8 A4000000 CALL sw.0063071C
```

Press F9 and if you followed my steps you should see the nag, press OK and land at the

```
00630620 > . A1 44856400 MOV EAX,DWORD PTR DS:[648544] ; <-
TSmartWhoisSystemForm@FormDestroy
```

Press F9 again and you land here:

```
00621BE0 >/. 55 PUSH EBP ; <-
TMainForm@FormDestroy
```

which is the last landing before terminating.

So the Delphi sequence of calls is:

```
TSmartWhoisSystemForm.FormCloseQuery(Sender : TObject);
TSmartWhoisSystemForm.FormDestroy(Sender : TObject);
TMainForm.FormDestroy(Sender : TObject);
```

This is an interesting thing, coming out only thanks to the integration of DeDe outs and OllyDbg.

4.2.2 Patching the AboutBox

To patch the AboutBox we will use again the top-down approach. The AboutBox, using again the Spy++ approach, is built using a TAboutForm.

NOTE

First of all restart the application, skip the initial exceptions and stop at the OEP as I already told



you before. Check that the comments inserted by DeDe are still in place and if not reapply them using MapConv, and check that the UDD file has been saved correctly by OllyDbg. Normally comments and labels remain, what you have to do again each time is the analysis of the code, using CTRL-A or the contextual menu in the asm view. By the way, if the code looks weird try removing this analysis.

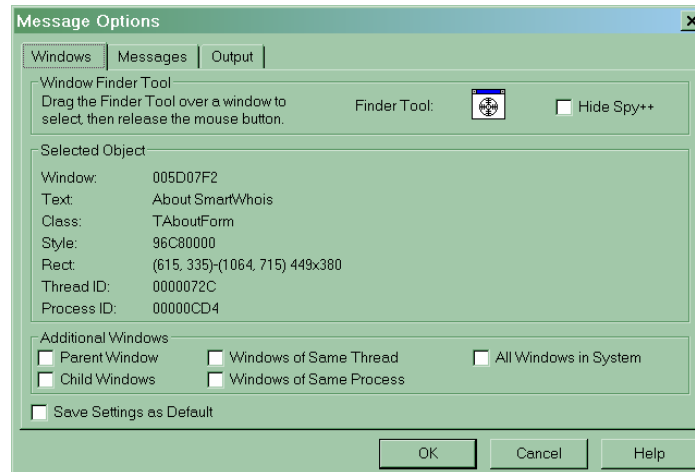


Figure 11 - Result of Spy++ on the AboutBox

We have to search the events of the TAboutForm in the saved DeDe files. We will find that this form has the following events:

TAboutForm.FormCreate	00602044
TAboutForm.FormShow	00602278
TAboutForm.LinkClick	00602590

We are interested in the FormCreate event at 00602044, Go there and place a BP. Thanks to DeDe comments the structure of the function is quite clear and it's also clear where to patch (see Figure 13). The label TAboutForm.lblLicensed is quite explanatory and what we have to do it so force it to appear.

```
00602093  0F84FF000000    JZ      00602198    ; NOP this JMP
0060209C  E8 3BFDFFFF    CALL  _sw.00601DDC    ; NOP this call
```

Note that the CALL _sw.00601DDC raises a recoverable exception in the program if not commented out. This happens because this call is responsible to assign a text to the lblLicensed label reporting the end of the support, based on the license number. We do not have any license number of course so you have to NOP this call or to fix it. I chose to NOP it also because of the nice result of Figure 12¹⁰.

¹⁰ 30 Dec 1900 is used as the base for calculations, so the value that was missing from the memory is the number of days starting from that day.



Figure 12 - patched AboutBox

NOTE

While you step the TAboutForm.FormCreate procedure, look also the data stack in Olly. You know that it contains also all the activation records of each call and also then the return addresses, the comments inserted from the MAP file are also there, transforming the whole call stack into a more talkative source of information.



00602044	55	PUSH EBP	<-TAboutForm@FormCreate
00602045	8BEC	MOV EBP,ESP	
00602047	6A 00	PUSH 0	
00602049	6A 00	PUSH 0	
0060204B	6A 00	PUSH 0	
0060204D	53	PUSH EBX	
0060204E	56	PUSH ESI	
0060204F	57	PUSH EDI	
00602050	8BD8	MOV EBX,EAX	
00602052	33C0	XOR EAX,EAX	
00602054	55	PUSH EBP	
00602055	68 6A226000	PUSH <sw.->System.@HandleFinally;	
0060205A	64:FF30	PUSH DWORD PTR FS:[EAX]	
0060205D	64:8920	MOV DWORD PTR FS:[EAX],ESP	
00602060	8BC3	MOV EAX,EBX	
00602062	E8 61020000	CALL sw.006022C8	
00602067	8BC3	MOV EAX,EBX	
00602069	E8 CE030000	CALL sw.0060243C	
0060206E	8BC3	MOV EAX,EBX	
00602070	E8 3F030000	CALL sw.006023B4	
00602075	8D45 FC	LEA EAX,[LOCAL.1]	
00602078	E8 6BFCFFFF	CALL sw.00601CE8	
0060207D	8B55 FC	MOV EDX,[LOCAL.1]	
00602080	8B83 FC020000	MOV EAX,DWORD PTR DS:[EBX+2FC]	*TAboutForm.lblVersion:TTamoLabel
00602086	E8 E1BEE4FF	CALL sw.0044DF6C	->Controls.TControl.SetText(TControl;TCaption);
0060208B	A1 A8856400	MOV EAX,DWORD PTR DS:[6485A8]	
00602090	8338 00	CMP DWORD PTR DS:[EAX],0	
00602093	0F84 FF000000	JE <sw.*lblLicensed:TTamoLabel>	
00602099	8D45 F8	LEA EAX,[LOCAL.2]	
0060209C	E8 3BFDFFFF	CALL sw.00601DDC	
006020A1	8B55 F8	MOV EDX,[LOCAL.2]	
006020A4	8B83 0C030000	MOV EAX,DWORD PTR DS:[EBX+30C]	*TAboutForm.lblLicensed:TTamoLabel
006020AA	E8 BDBEE4FF	CALL sw.0044DF6C	->Controls.TControl.SetText(TControl;TCaption);
006020AF	8D45 F4	LEA EAX,[LOCAL.3]	
006020B2	E8 A1FEFFFF	CALL sw.00601F58	
006020B7	8B55 F4	MOV EDX,[LOCAL.3]	
006020BA	8B83 10030000	MOV EAX,DWORD PTR DS:[EBX+310]	*TAboutForm.lblUpgrades:TTamoLabel
006020C0	E8 A7BEE4FF	CALL sw.0044DF6C	->Controls.TControl.SetText(TControl;TCaption);
006020C5	8B83 0C030000	MOV EAX,DWORD PTR DS:[EBX+30C]	*TAboutForm.lblLicensed:TTamoLabel
006020CB	8B40 48	MOV EAX,DWORD PTR DS:[EAX+48]	*TTamoLabel.OFFS_0048
006020CE	50	PUSH EAX	
006020CF	8B83 0C030000	MOV EAX,DWORD PTR DS:[EBX+30C]	*TAboutForm.lblLicensed:TTamoLabel
006020D5	8B40 4C	MOV EAX,DWORD PTR DS:[EAX+4C]	*TTamoLabel.OFFS_004C
006020D8	50	PUSH EAX	
006020D9	8BBB 08030000	MOV EDI,DWORD PTR DS:[EBX+308]	*TAboutForm.gbLicensed:TTamoGroupBox
006020DF	8B4F 4C	MOV ECX,DWORD PTR DS:[EDI+4C]	
006020E2	8B87 1C020000	MOV EAX,DWORD PTR DS:[EDI+21C]	
006020E8	D1F8	SAR EAX,1	
006020EA	79 03	JNS SHORT sw.006020EF	
006020EC	83D0 00	ADC EAX,0	
006020EF	2BC8	SUB ECX,EAX	
006020F1	8BB3 0C030000	MOV ESI,DWORD PTR DS:[EBX+30C]	*TAboutForm.lblLicensed:TTamoLabel
006020F7	2B4E 4C	SUB ECX,DWORD PTR DS:[ESI+4C]	
006020FA	8B83 10030000	MOV EAX,DWORD PTR DS:[EBX+310]	*lblUpgrades:TTamoLabel
00602100	2B48 4C	SUB ECX,DWORD PTR DS:[EAX+4C]	
00602103	83E9 04	SUB ECX,4	
00602106	D1F9	SAR ECX,1	
00602108	79 03	JNS SHORT <sw.*gbLicensed:TTamoGr	
0060210A	83D1 00	ADC ECX,0	
0060210D	8B83 08030000	MOV EAX,DWORD PTR DS:[EBX+308]	*gbLicensed:TTamoGroupBox
00602113	8B80 1C020000	MOV EAX,DWORD PTR DS:[EAX+21C]	*TamoGroupBox.OFFS_021C
00602119	D1F8	SAR EAX,1	

Figure 13 - First part of the TAboutForm.FormCreate procedure

4.2.3 Patching the Evaluation string in the Titlebar

To patch the titlebar (remove “Evaluation” string) we will use a bottom-up approach, because we do not exactly know where the title bar is set

Open the events.txt file and search for something containing the word “Set” and “Get”, we will find these Delphi APIs:

TControl.SetText(TControl;TCaption);	0044DF6C
TControl.GetText(TControl):TCaption;	0044DF3C



The concept is that the “Evaluation Version” must be got somewhere from the program’s resources, we want to intercept that event.

Place two breakpoints there and being stopped at the OEP with OllyDbg, press F9 to run the application.

You should land at the GetText function few times, each time the registry contain the string that is going to be got. Each time after the GetText function you stop at other breakpoint where there is SetText, it’s obvious that the got strings are then set.

Going on like this you should see before or later this situation:

EAX	010C2598
ECX	00000000
EDX	010CF6E0 ASCII "SmartWhois - Evaluation U
EBX	010C2598
ESP	0013FA74

This is what we were waiting for: use the call stack (ALT-K) and see who’s calling this piece of code.

0013FA74	00622AF5	? sw.0044DF6C	sw.00622AF0
0013FA98	00621B36	? sw.00622A8C	sw.00621B31

The function at 00622AF0 is called by the piece of code reported in Figure 14.

00622AA3	64:8920	MOV DWORD PTR FS:[EAX],ESP	
00622AA6	A1 00896400	MOV EAX,DWORD PTR DS:[648900]	check this location around ^_^
00622AAB	8338 00	CMP DWORD PTR DS:[EAX],0	
00622AAE	75 47	JNZ SHORT sw.00622AF7	JMP
00622AB0	8B15 80856400	MOV EDX,DWORD PTR DS:[648580]	sw.0064204C
00622AB6	8B12	MOV EDX,DWORD PTR DS:[EDX]	
00622AB8	8D45 F4	LEA EAX,DWORD PTR SS:[EBP-C]	ASCII " - "
00622ABB	B9 582B6200	MOV ECX,sw.00622B58	
00622AC0	E8 331FDEFF	CALL sw.004049F8	
00622AC5	8B55 F4	MOV EDX,DWORD PTR SS:[EBP-C]	
00622AC8	8D45 F8	LEA EAX,DWORD PTR SS:[EBP-8]	
00622ACB	E8 B424DEFF	CALL sw.00404F84	
00622AD0	8D45 F8	LEA EAX,DWORD PTR SS:[EBP-8]	
00622AD3	8B15 74886400	MOV EDX,DWORD PTR DS:[648874]	sw.00646AA4
00622AD9	8B12	MOV EDX,DWORD PTR DS:[EDX]	
00622ADB	E8 D024DEFF	CALL sw.00404FB0	
00622AE0	8B55 F8	MOV EDX,DWORD PTR SS:[EBP-8]	
00622AE3	8D45 FC	LEA EAX,DWORD PTR SS:[EBP-4]	
00622AE6	E8 891EDEFF	CALL sw.00404974	
00622AEB	8B55 FC	MOV EDX,DWORD PTR SS:[EBP-4]	
00622AEE	8BC3	MOV EAX,EBX	
00622AF0	E8 77B4E2FF	CALL sw.0044DF6C	call responsible of calling GetText

Figure 14 - Where the application's Titlebar is set

The patch should be self evident, transform the JNZ at 00622AAE into a JMP, or if you prefer rerun the application, land at the OEP, place a breakpoint at 622AA6 the value at 00648900 is copied to EAX and its address is used for the CMP: try searching for this constant through all the code, you should see another place where the “Evaluation Version” has been used..

NOTE

The patch did to the application is far from being complete and perfect; the target of this document is to teach about a specific technique rather than patching a program.



5. Loader's code

I will use for the loader of this application the usual Loader's framework we already released with document [5] (see also [7, 8] for AsProtect specific loaders). So I will only report here the specific code written for this application, assuming that you already know what I'm talking about. It's useful to use a known framework because this way the loader writing is the simplest task. You can use other loaders of course, written in assembler (there are some versions around derived from the document [5, 7, 8]).

```
<----- Start Code Snippet ----->
BOOL Loader::InitializePatchStack(growing_arraystack<Patch> &stkPatches)
{
    //NB 0x00 must explicitly converted to BYTE because otherwise the compiler confuses
    //it with a NULL pointer and doesn't know which constructor of class Patch to use.

    stkPatches.push(Patch(0x006308E0, (BYTE)0x75, (BYTE)0xEB)); //PATCH1
    stkPatches.push(Patch(0x00630F5D, (BYTE)0x75, (BYTE)0xEB)); //PATCH2
    stkPatches.push(Patch(0x00630D0F, (BYTE)0x75, (BYTE)0xEB)); //PATCH3

    stkPatches.push(Patch(0x0063066D, (BYTE)0x75, (BYTE)0xEB)); //PATCH FINAL NAG ON EXIT
    stkPatches.push(Patch(0x00622AAE, (BYTE)0x75, (BYTE)0xEB)); //PATCH "- Evaluation Version"

    //Patch for the AboutBox
    DWORD dwPatchaddrInj1[IMAXINDEXINJ1] = { 0x00602093, 0x00602094, 0x00602095, 0x00602096, 0x00602097,
                                              0x00602098 };
    int iOriDataInj1[IMAXINDEXINJ1] = { 0x0F, 0x84, 0xFF, 0x00, 0x00, 0x00 };
    int iPatchDataInj1[IMAXINDEXINJ1] = { 0x90, 0x90, 0x90, 0x90, 0x90, 0x90 };

    int idx=0;
    for (idx=0; idx<IMAXINDEXINJ1; idx++)
        stkPatches.push(Patch(dwPatchaddrInj1[idx], (BYTE)iOriDataInj1[idx], (BYTE)iPatchDataInj1[idx]));

    DWORD dwPatchaddrInj2[IMAXINDEXINJ2] = { 0x0060209C, 0x0060209D, 0x0060209E, 0x0060209F, 0x006020A0 };
    int iOriDataInj2[IMAXINDEXINJ2] = { 0xE8, 0x3B, 0xFD, 0xFF, 0xFF };
    int iPatchDataInj2[IMAXINDEXINJ2] = { 0x90, 0x90, 0x90, 0x90, 0x90 };

    for (idx=0; idx<IMAXINDEXINJ2; idx++)
        stkPatches.push(Patch(dwPatchaddrInj2[idx], (BYTE)iOriDataInj2[idx], (BYTE)iPatchDataInj2[idx]));

    //////////////////////////////////////

    return TRUE;
}
<----- End Code Snippet ----->
```

The function InitializePatchStack is quite obvious, nothing special to say, just you can find all the patches we told before..

```
<----- Start Code Snippet ----->
BOOL Loader::SetVictimDetails(TextString &victimFileName)
{
    #ifndef _DEBUG
    //just takes the real executing path from the first argument of the command line.
    //This is needed when then program is invoked with a different local path
    TextString path=TextString(__argv[0]);
    *(strrchr(path.c_str(), '\\'))='\\0';
    victimFileName=TextString(path+"\\_sw.exe");
    #else
    victimFileName=TextString("C:\\Program Files\\SmartWhois\\_sw.exe");
    #endif

    //Set this parameter to true when you want the loader to check the CRC of the file!
    SetVictimCRC(0x6cbf890a);

    SetCreateProcessFlags(DEBUG_PROCESS | DEBUG_ONLY_THIS_PROCESS | CREATE_NEW_CONSOLE);

    return TRUE;
}
<----- End Code Snippet ----->
```




The only remark for this function is that we have to give to the loader the correct path of the application. This application can be launched using a contextual menu so the local execution path is different each time: it is the one you were when you used the contextual menu. So you have to extract the real application's path using the commandline, the argv[0] is an excellent candidate, because contains the complete path of the program. At Point 1 we are then just extracting the path from argv[0].

The last important point for the loader is the GateProcedure condition, which is a specific memory location after testing which the loader can apply the patches.

I chose to use the last exception before the program is loaded, there are other alternatives of course.

At Section 2 I told you to remember a specific exception, the last:

```
012BA6E9    C601 1E      MOV BYTE PTR DS:[ECX],1E      ; stopped here
012BA6EC    0807        OR BYTE PTR DS:[EDI],AL
012BA6EE    2C A7       SUB AL,0A7
012BA6F0    FD         STD
012BA6F1    5B         POP EBX                      ; 0012F870
012BA6F2    - E9 67648F06 JMP 07BB0B5E
```

This piece of code is used to see if we are ready to patch: we are reading at 012BA6E9 (value which is pointed by EIP, being this the address where the exception occurs). If EIP+5 bytes forward there are these two bytes: 2CA7 we are ready to go.

```
<----- Start Code Snippet ----->
DWORD dwEIPOffset=0x05;

ReadProcessMemory(GetPI()->hProcess, (LPVOID)((victimContext.Eip) + dwEIPOffset), OridataRead,
                  BYTES2READATGATE, NULL);

//max index of the OridataRead must be BYTES2READATGATE-1
if ( (OridataRead[0] == 0x2C) && (OridataRead[1] == 0xA7) )
{
    // Key location found, now we can apply the patch
    #ifdef _DEBUG
    char str[256];
    sprintf(str,"Found Gate location at %X", (LPVOID)((victimContext.Eip) + dwEIPOffset));
    MessageBox(NULL, str, DEFAULT_MSG_CAPTION, MB_OK);
    #endif

    throw TRUE;
}
<----- End Code Snippet ----->
```

The whole loader's code is included into this tutorial's archive.

6. Conclusions

We went through several characteristics of Delphi and learnt how to use the Partial Dumps in conjunction with DeDe and OllyDbg and why it works with Delphi. The Partial Dumps technique allows taking advantages of DeDe work, also for packed applications, and has several advantages with respect to normally attaching DeDe to a running process, because that is not the ideal situation. The application already started to work and you can attach DeDe only when the initialization has terminated and the application is in the user-mode ready state. Stopping at the OEP instead, even without worrying of the lost IAT, gives you much more opportunities.

I suggest as usual to use this tutorial for learning more in depth how the operative system works and to use these examples to evolve your RCE techniques and not to crack programs.



Appendix I. Delphi Compilation Process

Delphi is not VB. This is perhaps one of the best things you could say about it. It's well-known that Delphi is fast and efficient, while still supporting the drag-and-drop "RAD" model of development. Probably the most overlooked aspect of Delphi is its elegance-- not just the language, but also the design of the compiler and the environment itself. Very few people realize that, apart from the differences between an interpreted and a compiled language, Delphi differs from VB in a more fundamental way: it is a "real" language, with a real compiler and class libraries, and as such is actually closer in philosophy to C and C++.

What makes C the language of choice for building operating systems, utilities and games? While C may have a reputation for being terse and cryptic, it allows for very low-level programming; and by low-level we mean "close to the hardware". Being close to the hardware makes C programs run very fast and very memory-efficient, which is an absolute requirement for applications where speed is a priority, like operating systems and high-performance games. C gives programmers almost-total control; because of this, it has been called a "portable assembly language".

The differences are by our point of view in the compilation process, which is different for C/C++ and Delphi. Figure 15 reports how C and C++ process input files to make the final executable. This is the generic C/C++ compilation process.

On GUI platforms like Windows, to facilitate the so-called "event-driven" model, and to handle graphical elements like icons, menus and dialog boxes, an additional step was added: Resource compilation. Resources are created using a media-dependent editor such as an icon or bitmap editor, or even a sound recorder, and then a resource script is created using a text editor, which is then compiled with a resource compiler (see Figure 16).

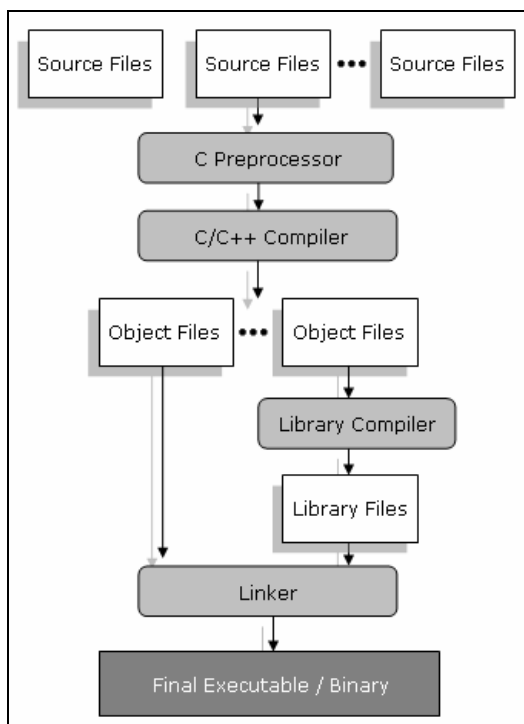


Figure 15 - C and C++ Compilation Process

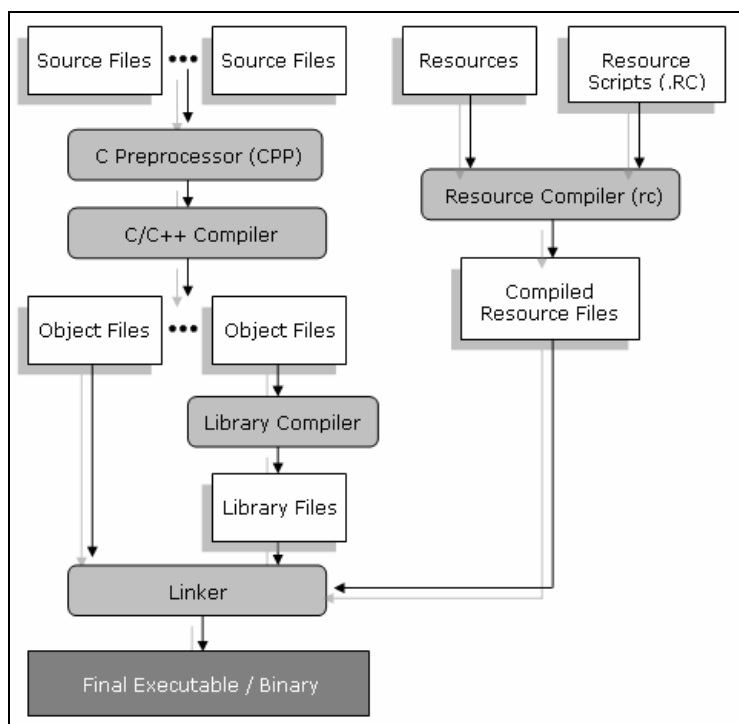


Figure 16 - MS VC Compilation Process



Figure 17 reports the Delphi equivalent, which Delphi performs behind-the-scenes when using the IDE.

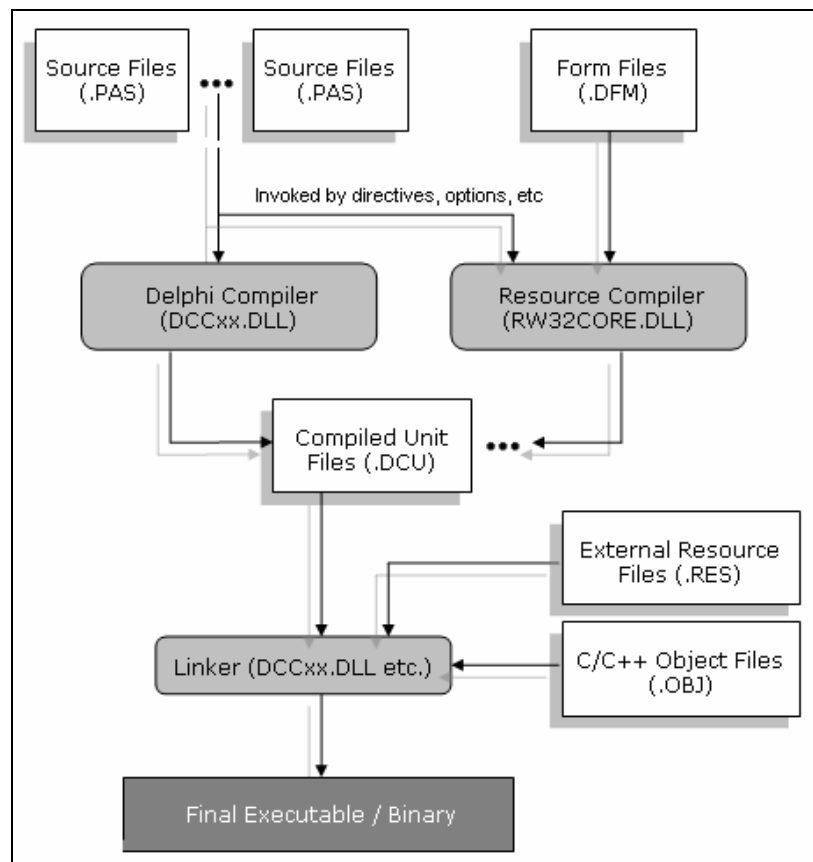


Figure 17 - Delphi Compilation Process

In a real compiler, source code is translated into machine code (binaries) using a compiler, and may take on an intermediate form of object files or library files. At the lowest level, the source language can interface with the operating system's API (e.g., the Win32 API in Windows, the standard C library in Unix, Int21h in DOS) and forms its link to the "real world". The language can provide this by providing mappings or wrappers to the API, such as Delphi's Windows.pas unit. Upon this basic level the higher-level libraries can be built; if the language is object-oriented, then a class library can be created. A good example of this is the Visual C++ Microsoft Foundation Classes (which has often been criticized for not being fully object-oriented). Another example is, of course, the Delphi VCL.

In contrast, a VB program, though it looks like a normal .EXE file, is merely an intermediate form of code (so-called 'p-code') for interpretation by a runtime system. VB source is "compiled" into this intermediate form and embedded within an EXE file. When the executable is run, the EXE loads the VB runtime VBRUNXXX.DLL, and which then interprets the instructions embedded within the EXE. In a sense, VB code is compiled for the VB virtual machine, analogous to the way the Java Virtual Machine runs its programs. This is the reason why apps written with VB, especially ones downloaded from the 'Net, are so prone to the "missing DLL" problem.



Appendix II. Script for reaching the OEP

Thanks to **britedream** I included also a script for the ODbgScript plugin [8] which would allow you to reach the OEP instantly. This script indeed should not only work with this specific AsProtect version but with all the AsProtect versions around. This script should work on old and new AsProtect protected target to find the OEP if there are no stolen bytes, otherwise it will land on the code section, right after the emulation of the stolen bytes. It makes life easier if you want to check the target few times to make a loader.

```
<----- Start Code Snippet ----->
cmp $VERSION,"1.41"
jae go
msg "Wrong Version"
ret

go:
gmi eip,CODEBASE
mov codeb,$RESULT
gmi eip,CODESIZE
mov codes,$RESULT
mov codet,codes
add codet,codeb
eoe chk
eob chk
mov fl,0
esto

chk:
mov eax1,eax
mov eax,ebp
cmp al,78
je flg
cmp al,90
je flg
mov eax,eax1
cnt:
esto

flg:
mov eax,eax1
cmp fl,0
jne found
mov start,esp
add start,14

loop:
add start,4
cmp start, ebp
je err
cmp [start],0
jne loop

nxt:
add start,4
cmp [start],0
je nxt
mov fl,start
jmp cnt

found:
mov start,[fl-4]
sub start,8
cmp [start],2
jne err
mov old,fl
sub old,4
mov new,[old]

here:
eob here
eoe here
cmp [old],new
jne test
esto
jmp here

err:
msg "Sorry , script isn't working"
```



```
ret

finshed:

bpmc
ret

test:
eoe test2
eob chkcode
bprm codeb, codes

test2:
esto
ret
chkcode:
mov ecx1, ecx
add ecx1, 4
cmp [ecx1], [esp]
jne finshed
bpmc
rtr

bprm codeb, codes
esto
ret
Created on 1/1/2006 by britedream
<----- End Code Snippet ----->
```

You can find the script also in this archive.



7. References

- [1] “Cracking Capturenprint 6.6.6.17”, Shub-nigurath, <http://tutorials.accessroot.com>
- [2] DeDe version 3.50.04.1635 with extensions,
<http://www.intechhosting.com/~access/ARTeam/tools/<DeDe.3.50.04.1635.zip>
- [3] SmartWhois, <http://www.tamos.com/products/smartwhois/>. The version used for the tutorial is here: <http://www.intechhosting.com/~access/ARTeam/tools/sw4.zip>
- [4] “Cracking with Loaders: Theory, General Approach and a Framework, Version 1.2”, Shub-Nigurath, ThunderPwr, <http://tutorials.accessroot.com> or on Code-Breakers Journal Vol.2 No.2
- [5] MapConv v1.4, http://www.intechhosting.com/~access/ARTeam/tools/MapConv_14.zip
- [6] “Writing Loader 2 Patch Apps Protected With Asprotect 2.0 V10”, Shub-Nigurath, Thunderpwr, <http://tutorials.accessroot.com>
- [7] “Writing Loader 2 Patch Apps Protected With Asprotect 1.2x And Earlier V10”, Shub-Nigurath, Thunderpwr, <http://tutorials.accessroot.com>
- [8] ODbgScript v1.41 VC6,
<http://www.intechhosting.com/~access/ARTeam/tools/ODbgScript.1.41.VC6.rar>

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.

8. History

- Version 1.0 – First public release!
- Version 1.1 – Added Appendix II

9. Greetings

I wish to tank all the ARTeam members, briedream and of course and who read the beta versions of this tutorial and contributed,.. and of course you, who are still alive at the end of this document!



<http://cracking.accessroot.com>