



Unpacking and dumping ExeCryptor, and coding loader

deroko/ARTeam

Version 1.0 – January 2006

1.	Abstract.....	2
2.	Unpacking and dumping.....	3
3.	Rebuilding imports.....	11
4.	Removing limitations	16
5.	Loader : at the boundary of perfection	21
6.	Conclusion	30
7.	References.....	30
8.	Greetings.....	31

Keywords

execryptor, unpacking, dumping, loader



1. Abstract

ExeCryptor is one of the best protections available on the market, very difficult to dump and unpack, but with some knowledge it can be done. It will change entrypoint with meta code which is luckily for us executed in range of our target, also it has some other (dis?)advantages like resolving APIs whenever they are called using it's own GetProcAddress by hash, it will emulate real GetProcAddress and some other nasty tricks. As far as I know there are only couple of tutorials about unpacking and dumping ExeCryptor, actually only 2 of them by PakMan/SND and pnluck.

How did I get this software anyway? One night I was chatting with my friends at IRC and someone mentioned buffer overflows in [Golden FTP pro 2.7.0](#),[11] according to my friend this software has way too many buffer overflows, so to prevent evil hackers from finding them, and writing advisories, and exploits for this crappie software, author decided to protect it with ExeCryptor and make software harder to analyze and find buffer overflows. Lame, don't you think so? Heh wait till you see key check routine. Simple algo with Sleep(3000) to simulate long long and very complex key check routine. I hope that new advisories won't come up soon :D

For this tutorial we are gona need target : [Golden FTP pro 2.7.0](#) packed with unknown version of ExeCryptor. This target hates olly, terminates it whenever olly is active, has .code section checksum checking so no patches can be applied permanently, unless we unpack it.

Tools:

- PEiD v0.94
- SoftICE
- IceExt 0.67
- Olly
- hiew
- tasm32/tlink32 [9]
- Oepfinder vX.Y.Z [oepfinder]

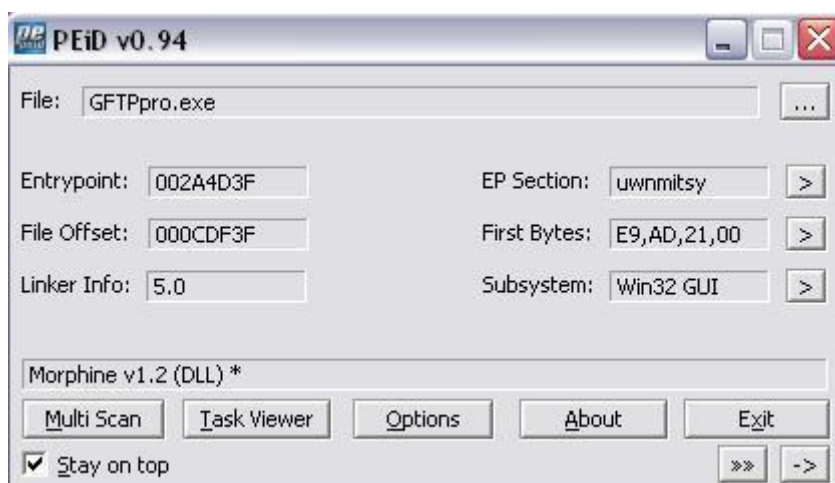
NOTE: for everything shown here you may use Olly but it was boring to attach to target each time I needed to check something during my little research. If anyone is curios about my configuration it is win xp sp0 (no sp) with SoftICE 4.3.2...

S verom u Boga, deroko/ARTeam

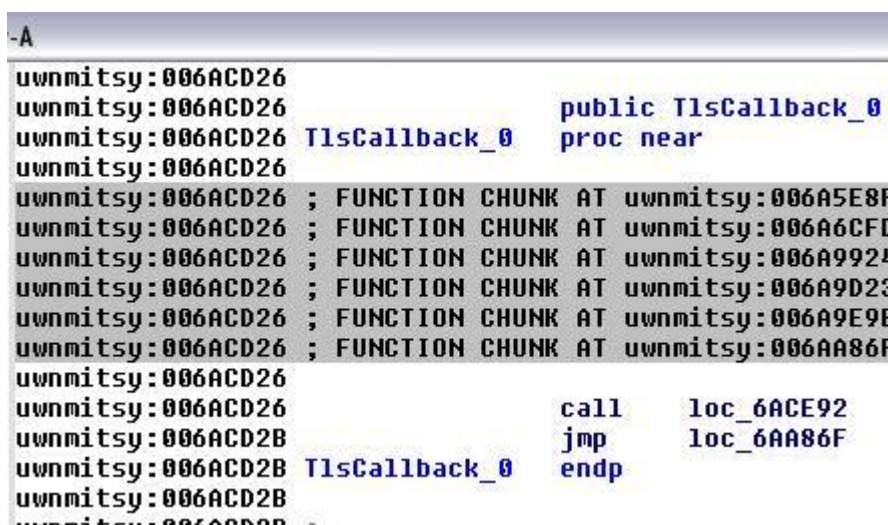


2. Unpacking and dumping

First things first, so we scan our target with PEiD:



Well heh, not Morphine due to TLS callback and heavy anti-debug:



Now we are sure that protector used here is execryptor. TLS callbacks are executed before each thread including primary thread. First coder that took advantage of TLS callbacks is roy'g'biv member of 29a[1] with his Shrug virus for x32/x64/ia64. Latter on, in the book by Peter Szor[4] was stated that Symantec has discovered this method even before roy g biv, but I doubt. I bet it took them more then 2-3 days to figure what is going on.

Oki, during past few days, I've coded special tool to break at entrypoint of ExeCryptor packed app, and also some other protectors such as Armadillo, ASProtect, enigma etc... This tool will be used here to defeat execryptor and you can find it in oepfinder folder.



First we fire up oepfinder to check if execryptor detects us:



And we click on trace, we should get 3 MessageBoxes informing us about EIP in our range :



We ignore first two, because they are not what we are looking for, and we wait 3rd:



If you press ok, Trial reminder will popup, so at this point we click on **Cancel**





Memorize above bytes (FF 25), attach to process and restore them:

```
EAX=00000000 EBX=7FFDE000 ECX=0059B74F EDX=0059B74F ESI=00009F33
EDI=00000000 EBP=0013FFF0 ESP=0013FFB8 EIP=004C51CA DS:004EA3AC=005D25E9
CS=001B DS=0023 SS=0023 ES=0023 FS=003B GS=0000

0010:0013FFB8 005A14C6 00000000 0059B74F 7C816D4F ..Z...O.Y.Omü!
0010:0013FFC8 00000000 00009F33 7FFDE000 8054B038 ...3j...α.Δ8.TG
0010:0013FFD8 0013FFC8 821563C8 FFFFFFFF 7C8399F3 ...c.e...a!
0010:0013FFE8 7C816D58 00000000 00000000 00000000 Xü!...Actx...
0010:0013FFF8 006A4D3F 00000000 78746341 00000020 ?MJ...y$.
0010:00140008 00000001 00002498 000000C4 00000000 ....4.....
0010:00140018 00000020 00000000 00000014 00000001 .....
0010:00140028 00000006 00000034 00000114 00000001 .....
0010:00140038 00000000 00000000 00000000 00000000 .....
0010:00140048 00000000 00000002 00000000 00000000 .....
0010:00140058 00000000 000000214 0000019C 00000000 .....
0010:00140068 2D59495B 0000003B0 00000032 000003E4 Iiv-...2...Σ...
0010:00140078 000002D2 00000000 830202E4 000006B8 .....Σ...a...

==> 004C51CA JMP [004EA3AC]
001B:004C51D0 JMP [004EA3B0]
001B:004C51D6 JMP [004EA3B4]
001B:004C51DC JMP [004EA3B8]
001B:004C51E2 JMP [004EA3BC]
001B:004C51E8 JMP [004EA3C0]
001B:004C51EE JMP [004EA3C4]
001B:004C51F4 JMP [004EA3C8]
001B:004C51FA JMP [004EA3CC]
001B:004C5200 JMP [004EA3D0]
001B:004C5206 JMP [004EA3D4]
001B:004C520C JMP [004EA3D8]
001B:004C5212 JMP [004EA3DC]
001B:004C5218 JMP [004EA3E0]
001B:004C521E JMP [004EA3E4]
001B:004C5224 JMP [004EA3E8]
001B:004C522A JMP [004EA3EC]
```

Oki we are at JMP table to obfuscated IAT. Before we proceed I will in short terms explain IAT obfuscation used by execryptor in this target. Each jmp to obfuscated IAT is actually jmp to ExeCryptors code. ExeCryptor is trying to avoid easy recovering of IAT by patching some jmp, like we do with Armadillo packed applications. Instead, whenever some API is needed it will be resolved and stored in IAT at runtime. Set BPX at [4C51CAh](#) and press F5 once in debugger:

```
001B:004C51C4 JMP [004EA3A8]
001B:004C51CA JMP [004EA3B0] GetModuleHandleA
001B:004C51D0 JMP [004EA3B4]
001B:004C51D6 JMP [004EA3B8]
001B:004C51DC JMP [004EA3BC]
```

You see? Only used APIs are being resolved, but I'm gona talk about this more when we come to Import rebuilding (that is gona be nice and long importrec plug-in coding [3]).

Now if you take a look at stack you will see that return address is pointing to ExeCryptor's code, don't afraid, set BPX at [4C51CAh](#) and you will break once more from execryptor code, 3rd break is the one that will take us back to .code section so we can easily determine what compiler is used to compile this app, and with some luck we will be able to restore stolen bytes and fix this target:

Keep pressing F5 in sice or F9 in olly till you see that return address is : [4A7608h](#).



```
001B:004A75D9 AND     DWORD PTR [EAX+000000141],-03
001B:004A75E0 MOV     EAX,004A75D4
001B:004A75E5 CMP     EAX,[004D4D70]
001B:004A75EB JZ      +004A75F3
001B:004A75ED CALL   [004D4D70]
001B:004A75F3 CALL   +004A7628
001B:004A75F8 RET
001B:004A75F9 NOP
001B:004A75FA NOP
001B:004A75FB NOP
001B:004A75FC PUSH   004D4E48 ; "__CPPdebugHook"
001B:004A7601 PUSH   00
001B:004A7603 CALL   KERNEL32!GetModuleHandleA
001B:004A7608 PUSH   EAX
001B:004A7609 CALL   +004C51DC
001B:004A760E MOV     [004E4118],EAX
001B:004A7613 CMP     DWORD PTR [004E4118],00
001B:004A761A JNZ     +004A7626
001B:004A761C MOV     DWORD PTR [004E4118],004C65C8
001B:004A7626 RET
001B:004A7628 NOP
001B:004A7629 PUSH   EBP
001B:004A762A MOV     EBP,ESP
001B:004A762B ADD     ESP,24
001B:004A762E MOV     EAX,004D4E78
001B:004A7632 PUSH   EBX
001B:004A7634 PUSH   ESI
001B:004A7635 PUSH   EDI
001B:004A7636 CALL   +004A6AC8
001B:004A763B CALL   +004B1140
001B:004A7640 AND     DWORD PTR [EAX+000000141],-03
001B:004A7647 MOV     EDX,[004E4118]
001B:004A764D CMP     DWORD PTR [EDX],02
001B:004A7650 JNZ     +004A7664
001B:004A7652 PUSH   DWORD PTR [004D4D6C]
001B:004A7658 PUSH   01
001B:004A765A PUSH   04
(C:\WINDOWS\system32\cmd.exe) - PID(0028) - C:\Program Files\ExeCryptor\text+000065D9
```

If you take closer look at this picture you might notice “__CPPdebugHook” which tells us that this is nothing more than Borland C/C++ compiled application. How do I know that? Well when you use tasm32 and bcc32 for all your programs you get familiar with code compiled with Borland compilers.

Keep going till you get here, just step over each procedure that is on your way:

```
001B:004ACC2E JMP     +004ACC31
001B:004ACC30 INC     EBX
001B:004ACC31 MOV     AL,[EBX]
001B:004ACC33 TEST    AL,AL
001B:004ACC35 JZ      +004ACC3B
001B:004ACC37 CMP     AL,20
001B:004ACC39 JZ      +004ACC30
001B:004ACC3B CMP     AL,09
001B:004ACC3D JZ      +004ACC30
001B:004ACC3F CALL   +004ACC44
001B:004ACC44 PUSH   EAX
001B:004ACC45 PUSH   EBX
001B:004ACC46 PUSH   00
001B:004ACC48 PUSH   00
001B:004ACC4A CALL   KERNEL32!GetModuleHandleA
001B:004ACC4F PUSH   EAX
001B:004ACC50 CALL   [ESI+18]
001B:004ACC53 ADD     ESP,10
001B:004ACC55 PUSH   EAX
001B:004ACC57 CALL   +004ABDA0
001B:004ACC5C POP     ECX
001B:004ACC5D JMP     +004ACC80
001B:004ACC5F MOV     EDX,[004E4514]
001B:004ACC65 PUSH   EDX
001B:004ACC66 MOV     ECX,[004E4510]
001B:004ACC6C PUSH   ECX
001B:004ACC6D MOV     EAX,[004E450C]
001B:004ACC72 PUSH   EAX
001B:004ACC73 CALL   [ESI+18]
001B:004ACC75 ADD     ESP,0C
```

call to main()

Nice, we have located call to main() in Borland produced applications, as I told you, I know all of this stuff so this target was perfect to write one ExeCryptor unpacking tutorial. Oki, go to the 401000h and you will see Borland C/C++ generated entrypoint:

```
001B:00401000 JMP     +00401012
001B:00401002 BOUND   DI,[EDX]
001B:00401005 INC     EBX
001B:00401006 SUB     EBP,[EBX]
001B:00401008 DEC     EAX
001B:00401009 DEC     EDI
001B:0040100A DEC     EDI
001B:0040100B DEC     EBX
001B:0040100C NOP
001B:0040100D JMP     +008C75DA
001B:00401012 JMP     +0059B74F
001B:00401014 CALL   +005D1840
```



Now, I will show you simple application compiled with bcc32:

00401000	JMP SHORT bla.00401012	
00401002	DB 66	CHAR 'f'
00401003	DB 62	CHAR 'b'
00401004	DB 3A	CHAR '.'
00401005	DB 43	CHAR 'C'
00401006	DB 2B	CHAR '+'
00401007	DB 2B	CHAR '+'
00401008	DB 48	CHAR 'H'
00401009	DB 4F	CHAR 'O'
0040100A	DB 4F	CHAR 'O'
0040100B	DB 4B	CHAR 'K'
0040100C	NOP	
0040100D	DB E9	
0040100E	DD OFFSET bla.CPPdebugHook	
00401012	MOV EAX,DWORD PTR DS:[40910F]	
00401017	SHL EAX,2	
0040101A	MOV DWORD PTR DS:[409113],EAX	
0040101F	PUSH EDX	
00401020	PUSH 0	
00401022	CALL <JMP.&KERNEL32.GetModuleHandleA>	[pModule = NULL GetModuleHandleA
00401027	MOV EDX,EAX	
00401029	CALL bla.004020BC	
0040102E	POP EDX	
0040102F	CALL bla.00401458	
00401034	CALL bla.004020C0	
00401039	PUSH 0	[Arg1 = 00000000 bla.00402CD8
0040103B	CALL bla.00402CD8	
00401040	POP ECX	
00401041	PUSH bla.004090B8	
00401046	PUSH 0	
00401048	CALL <JMP.&KERNEL32.GetModuleHandleA>	[pModule = NULL GetModuleHandleA
0040104D	MOV DWORD PTR DS:[409117],EAX	
00401052	PUSH 0	
00401054	JMP bla.00406788	

Take a closer look at both pictures, one from softice, and one from olly and debugged simple “Hello world” program. Do you see that bytes from 401000h – 401012h are the same? And when did oepfinder break? Yes, at first access to code section and that is call to GetModuleHandleA (jmp from 4C51CAh) so we are sure that oepfinder will take us to the right spot, if we step into and from first call to GetModuleHandleA we will be in ExeCryptor morphed entrypoint. But that is not necessary right now because jmp at 401012h from our target will take us to morphed code and we can execute it without even worrying what instructions where there. In short, those were 2 calls to GetModuleHandleA and two calls to procedures in code section itself, you can find them easily but that is not necessary here. Also note that there is push at 401041h and at 401052h, jmp at 401054 will take us to procedure that I will name “prepare to call main()” because I have no idea how to call it.

Once we break at 4ACC4Fh or any address that is within our “prepare to call main()” we can scroll up till we see push ebp at:

```
001B:004ACB03 NOP
001B:004ACB04 PUSH EBP
001B:004ACB05 MOV EBP,ESP
001B:004ACB07 ADD ESP,-0C
001B:004ACB0A PUSH EBX
001B:004ACB0B PUSH ESI
001B:004ACB0C PUSH EDI
001B:004ACB0D MOV ESI,[EBP+08]
001B:004ACB10 MOV EAX,[ESI+10]
001B:004ACB13 AND EAX,0
```

Now, repeat same steps with oepfinder, once we break on 1st call to GetModuleHandleA set: BPX 4ACB04h and run code till we hit our breakpoint so we can examine stack and determine what is the value pushed on stack before jmp to “prepare to call main()”:



```
0010:0013FFBC 00000000 004C6564 7C816D4F 00000000 B...deL.Omü!...
0010:0013FFCC 00000042 7FFD8000 8054B038 0013FFC8 B...S.Δ8.TG
0010:0013FFDC 81EE3340 FFFFFFFF 7C8399F3 7C816D58 03ëü...?Xmü!
0010:0013FFEC 00000000 00000000 00000000 006A4D3F .....?MJ.
PROT32-

001B:004ACB03 NOP
001B:004ACB04 PUSH EBP
001B:004ACB05 MOV EBP,ESP
001B:004ACB07 ADD ESP,-0C
001B:004ACB0A PUSH EBX
001B:004ACB0B PUSH ESI
001B:004ACB0C PUSH EDI
001B:004ACB0D MOV ESI,[EBP+08]
001B:004ACB10 MOV EAX,[ESI+10]
001B:004ACB13 AND EBX,01
```

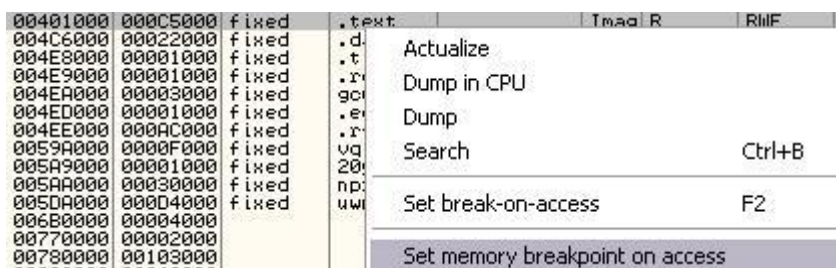
Now you see that last 3 instructions before “prepare to call main()” are:

```
push 4C6564h
push 0
jmp 4ACB04h <jmp _preparate_to_call_main>
```

Only thing that is left to go over is to figure what are those calls between two calls to GetModuleHandleA. Here is how we are going to find them. NOTE: this is not required for dump to work because that jmp at 401012h will take us to morphed stolen code and that’s all we need.

Two calls are located like this:

We open our dump without imports and step into jmp at 401012h, once we step into jmp set Memory breakpoint on access on .code section:



Run code, and first break is jmp [GetModuleHandleA], no need to show it, 3rd stop is this:

```
004A6428 8BC2 MOV EAX,EDX
004A642A 803D AC654C00 0 CMP BYTE PTR DS:[4C65AC],0
004A6431 75 25 JNZ SHORT fixed.004A6458
004A6433 803D AD654C00 0 CMP BYTE PTR DS:[4C65AD],0
004A643A 74 14 JE SHORT fixed.004A6450
004A643C 803D AD654C00 0 CMP BYTE PTR DS:[4C65AD],0
004A6443 74 13 JE SHORT fixed.004A6458
004A6445 8B15 908D4D00 MOV EDX,DWORD PTR DS:[4D8D90]
004A644B 833A 00 CMP DWORD PTR DS:[EDX],0
004A644E 75 08 JNZ SHORT fixed.004A6458
004A6450 8B0D 908D4D00 MOV ECX,DWORD PTR DS:[4D8D90]
004A6456 8901 MOV DWORD PTR DS:[ECX],EAX
004A6458 A1 BC8D4D00 MOV EAX,DWORD PTR DS:[4D8DBC]
004A645D C600 01 MOV BYTE PTR DS:[EAX],1
004A6460 E8 FFF4FFFF CALL fixed.004A5964
004A6465 C3 RETN
```

Run till retn and after that we are again in ExeCryptor code, 2 memory breakpoints following will hit dummy procedures (2 rets) and 3rd memory break point will stop here:



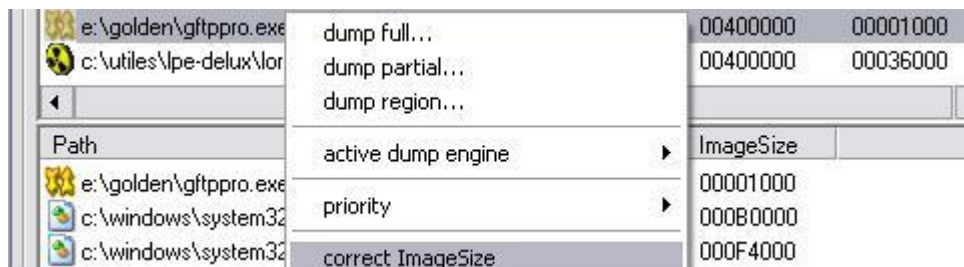
```
004A76D4 55          PUSH EBP
004A76D5 8BEC        MOV EBP,ESP
004A76D7 83C4 F8     ADD ESP,-8
004A76DA 53          PUSH EBX
004A76DB 8B5D 08     MOV EBX,DWORD PTR SS:[EBP+8]
004A76DE 85DB        TEST EBX,EBX
004A76E0 0F95C0     SETNE AL
004A76E3 83E0 01     AND EAX,1
004A76E6 85DB        TEST EBX,EBX
004A76E8 75 11       JNZ SHORT fixed.004A76FB
004A76EA 85DB        TEST EBX,EBX
004A76EC 75 0D       JNZ SHORT fixed.004A76FB
004A76EE 8D55 F8     LEA EDX,DWORD PTR SS:[EBP-8]
004A76F1 52          PUSH EDX
004A76F2 E8 29000000 CALL fixed.004A7720
004A76F7 59          POP ECX
004A76F8 8B5D FC     MOV EBX,DWORD PTR SS:[EBP-4]
004A76FB 68 9C000000 PUSH 9C
004A7700 53          PUSH EBX
004A7701 E8 3A9AF5FF CALL fixed.00401140
004A7706 81C0 10000000 ADD EAX,10
004A770C 50          PUSH EAX
004A770D E8 DAEFFFFF CALL fixed.004A66EC
004A7712 83C4 0C     ADD ESP,0C
004A7715 E8 42FEFFFF CALL fixed.004A755C
004A771A 5B          POP EBX
004A771B 59          POP ECX
004A771C 59          POP ECX
004A771D 5D          POP EBP
004A771E C3          RETN
```

Next break should be `jmp [GetModuleHandleA]` and then, bingo, you break at “prepare to call `main()`”.

Ok if you want you may rebuild stolen bytes from OEP but that is not necessary. When is good time to dump target? At 3rd `MessageBoxA` again, because that is 1st call to `GetModuleHandleA` after EP. So here we go again:

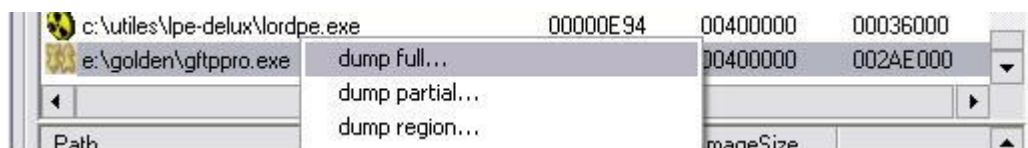


Fire up LordPE and dump target (don't forget to adjust image size):





Now, just dump application:



Oh yes, don't forget to restore FF 25 at 4C51CAh and to set entrypoint in dump to 1000h.

That's all about dumping this application, now only thing that is left to go over is import reconstruction which is hard if you have no idea how it works or medium if you have idea how to fix them. Ok hold one now we are moving to imports.



3. Rebuilding imports

ExeCryptor has nice import protection, and will not allow us to use importrec at the point when we reach good OEP, some APIs will be resolved whenever they are called. Some APIs are/will be resolved at the moment of terminating this process(not all) so you should set hardware break point on execution on ExitProcess. Once we break on ExitProcess, run importrec and resolve APIs that are called. Oki? Well hmmm no. We are gona code our own importrec plugin to repair all those obfuscated calls [3].

Whenever API is called, ExeCryptor will use it's own GetProcAddress to find API, resolving APIs by hand is crazy. Good thing with ExeCryptor is that all of it's code is located in image range so our tracer should return valid pointer only when EIP is not in image range. Tracer is included with this document [execryptor-imprecplugin], I can't guarantee that it will work with all targets, but it will work for this one for sure.

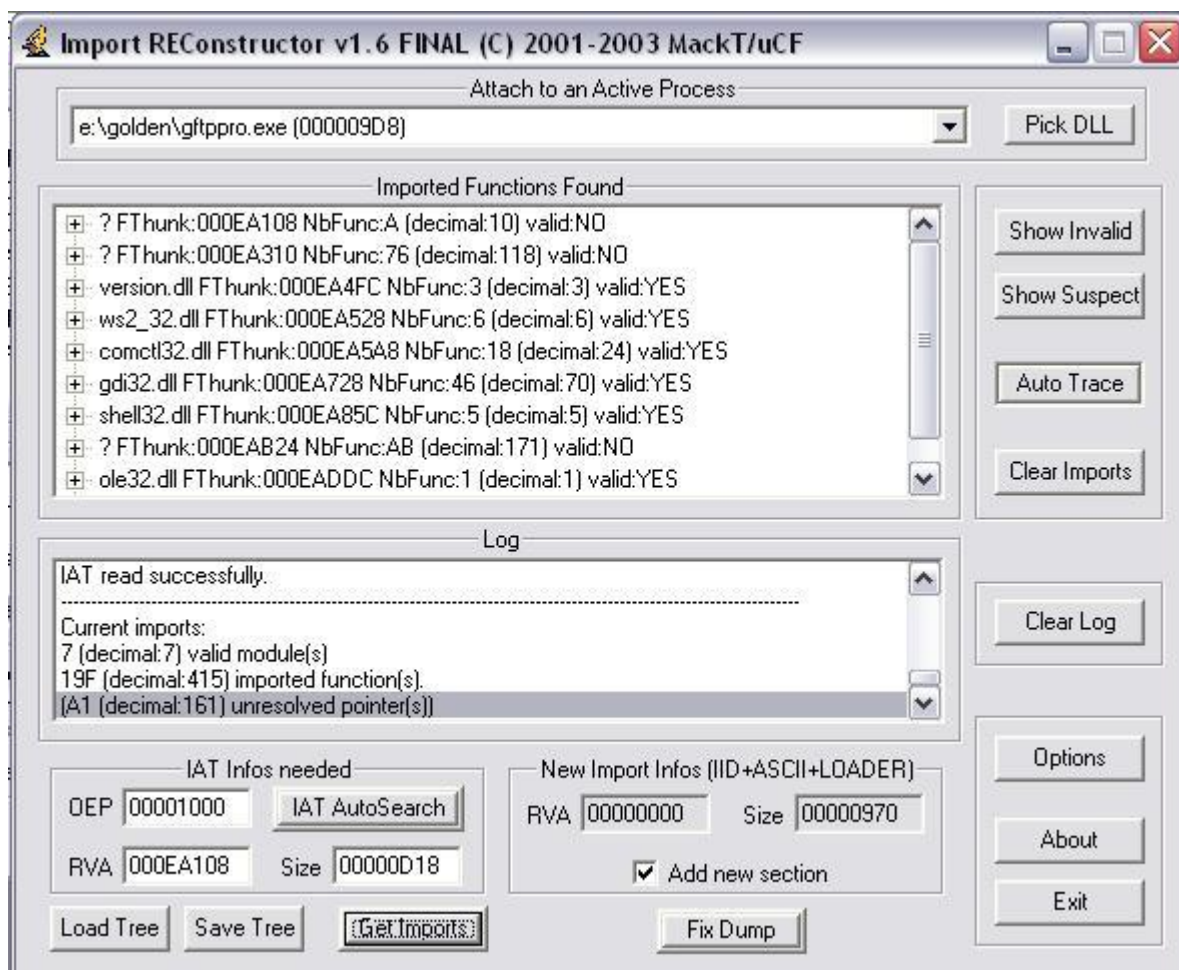
Import starts from here:

```
001B:004C50A3 RET
001B:004C50A4 JMP [004EA108]
001B:004C50AA JMP [004EA10C]
001B:004C50B0 JMP [004EA110]
001B:004C50B6 JMP [004EA114]
001B:004C50BC JMP [004EA118]
001B:004C50C2 JMP [004EA11C]
001B:004C50C8 JMP [004EA120]
001B:004C50CE JMP [004EA124]
001B:004C50D4 JMP [004EA128]
```

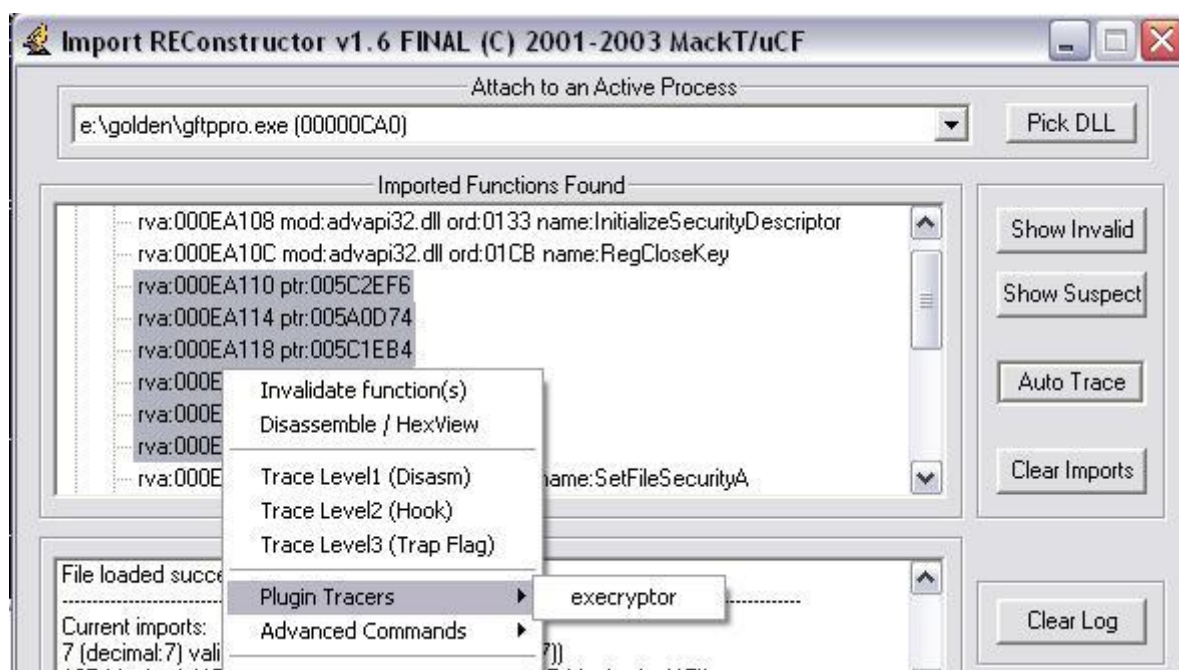
Ends here:

```
001B:004C5A26 JMP [004EADDC]
001B:004C5A2C JMP [004EADCC]
001B:004C5A32 INT
001B:004C5A34 JMP [004EADDC]
001B:004C5A36 INT
001B:004C5A38 INT
001B:004C5A3C JMP [004EAE04]
001B:004C5A42 JMP [004EAE08]
001B:004C5A48 JMP [004EAE0C]
001B:004C5A4E JMP [004EAE10]
001B:004C5A54 JMP [004EAE14]
001B:004C5A5A JMP [004EAE18]
001B:004C5A60 JMP [004EAE1C]
001B:004C5A66 INT
```

So fire up importrec and for IAT Rva type: [EA108](#) and for size [D18h](#), adjust oep to [1000h](#) and click on Get Imports you will get output similar to one shown on picture:

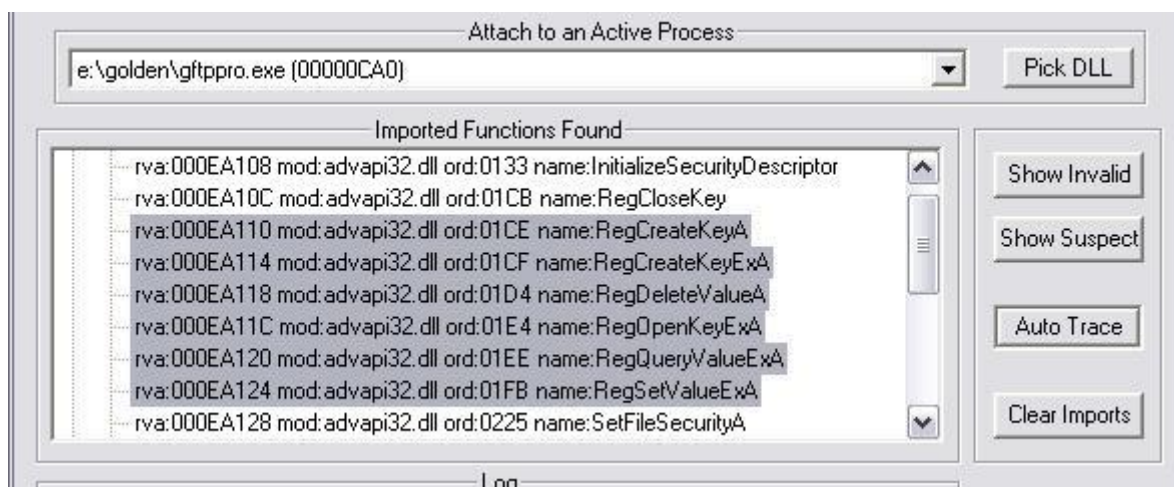


Heh, some are not resolved, no problemo at all, no problemo, my importrec plug-in will do all dirty job for us... hehe [3]:



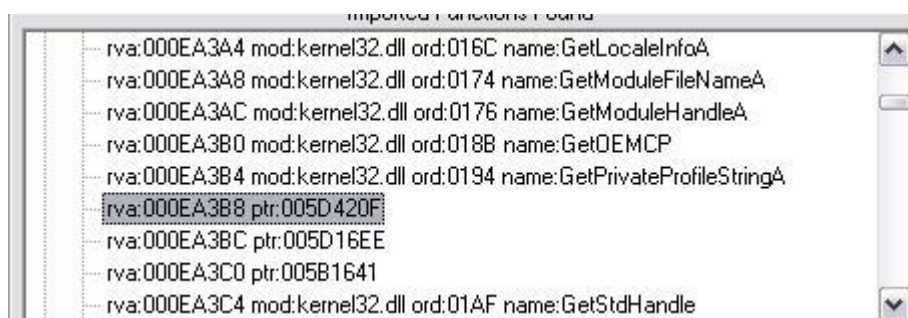


Run plug-in and wait a little bit:



Oh shit, imports are recovered :D

You may select all invalid ptrs and run plug-in on them and your GFTPPro will crash trying to resolve this one:



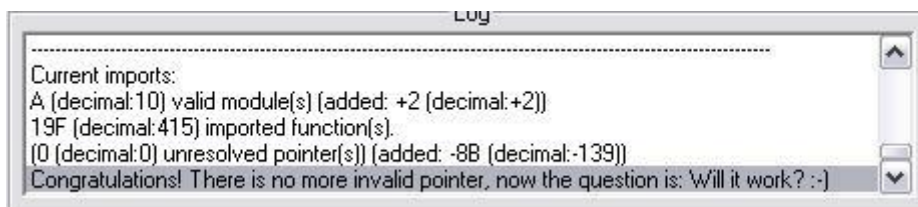
Well that one is rewritten(emulated) GetProcAddress so we have to insert it manually:





Now select all invalid ptrs again and run plug-in, oki you might wanna take cigarette break before this ends, it will take 2-3 mins to resolve all of them.

And after 2-3 mins:



Fix import in dump file as you have done many many times, open it in olly and examine IAT:

Fixed IAT start:

004C509C	832D 70754E00 0	SUB	DWORD	PTR	DS:[4E7570],1	
004C50A3	C3	RETN				
004C50A4	-FF25 08A14E00	JMP	DWORD	PTR	DS:[&advapi32.InitializeSecurityDescriptorA]	advapi32.InitializeSecurityDescriptorA
004C50A9	-FF25 0CA14E00	JMP	DWORD	PTR	DS:[&advapi32.RegCloseKey]	advapi32.RegCloseKey
004C50B0	-FF25 10A14E00	JMP	DWORD	PTR	DS:[&advapi32.RegCreateKeyA]	advapi32.RegCreateKeyA
004C50B6	-FF25 14A14E00	JMP	DWORD	PTR	DS:[&advapi32.RegCreateKeyExA]	advapi32.RegCreateKeyExA
004C50BC	-FF25 18A14E00	JMP	DWORD	PTR	DS:[&advapi32.RegDeleteValueA]	advapi32.RegDeleteValueA
004C50C2	-FF25 1CA14E00	JMP	DWORD	PTR	DS:[&advapi32.RegOpenKeyExA]	advapi32.RegOpenKeyExA
004C50C8	-FF25 20A14E00	JMP	DWORD	PTR	DS:[&advapi32.RegQueryValueExA]	advapi32.RegQueryValueExA
004C50CE	-FF25 24A14E00	JMP	DWORD	PTR	DS:[&advapi32.RegSetValueExA]	advapi32.RegSetValueExA
004C50D4	-FF25 28A14E00	JMP	DWORD	PTR	DS:[&advapi32.SetFileSecurityA]	advapi32.SetFileSecurityA
004C50DA	-FF25 2CA14E00	JMP	DWORD	PTR	DS:[&advapi32.SetSecurityDescriptorDacl]	advapi32.SetSecurityDescriptorDacl
004C50E0	-FF25 10A34E00	JMP	DWORD	PTR	DS:[&kernel32.CloseHandle]	kernel32.CloseHandle
004C50E6	-FF25 14A34E00	JMP	DWORD	PTR	DS:[&kernel32.CompareFileTime]	kernel32.CompareFileTime
004C50EC	-FF25 18A34E00	JMP	DWORD	PTR	DS:[&kernel32.CompareStringA]	kernel32.CompareStringA
004C50F2	-FF25 1CA34E00	JMP	DWORD	PTR	DS:[&kernel32.CreateDirectoryA]	kernel32.CreateDirectoryA
004C50F8	-FF25 20A34E00	JMP	DWORD	PTR	DS:[&kernel32.CreateEventA]	kernel32.CreateEventA
004C50FE	-FF25 24A34E00	JMP	DWORD	PTR	DS:[&kernel32.CreateFileA]	kernel32.CreateFileA
004C5104	-FF25 28A34E00	JMP	DWORD	PTR	DS:[&kernel32.CreateMutexA]	kernel32.CreateMutexA
004C510A	-FF25 2CA34E00	JMP	DWORD	PTR	DS:[&kernel32.CreateThread]	kernel32.CreateThread

End of fixed IAT:

004C5A34	-FF25 0CA04E00	JMP	DWORD	PTR	DS:[&ole32.IsEqualGUID]	ole32.IsEqualGUID
004C5A3A	CC	INT3				
004C5A3B	CC	INT3				
004C5A3C	-FF25 04AE4E00	JMP	DWORD	PTR	DS:[&oleaut32.SysAllocStringLen]	oleaut32.SysAllocStringLen
004C5A42	-FF25 08AE4E00	JMP	DWORD	PTR	DS:[&oleaut32.SysFreeString]	oleaut32.SysFreeString
004C5A48	-FF25 0CAE4E00	JMP	DWORD	PTR	DS:[&oleaut32.SysReAllocStringLen]	oleaut32.SysReAllocStringLen
004C5A4E	-FF25 10AE4E00	JMP	DWORD	PTR	DS:[&oleaut32.SysStringLen]	oleaut32.SysStringLen
004C5A54	-FF25 14AE4E00	JMP	DWORD	PTR	DS:[&oleaut32.VariantChangeTypeEx]	oleaut32.VariantChangeTypeEx
004C5A5A	-FF25 18AE4E00	JMP	DWORD	PTR	DS:[&oleaut32.VariantClear]	oleaut32.VariantClear
004C5A60	-FF25 1CAE4E00	JMP	DWORD	PTR	DS:[&oleaut32.VariantCopyInd]	oleaut32.VariantCopyInd
004C5A66	CC	INT3				
004C5A67	CC	INT3				
004C5A68	0000	0000	BYTE	PTR	DS:[EAX],0	

Well, now lets try our app to see if it really works. Shall we?



Well what the hell, we nailed one more protection with our tools. Hence but this porgy has it's own key check procedure, and also nag is still displayed so we will have to patch it. If you are wondering what this plug-in does, it is simple, it works like nonintrusive debugger to detect when EIP is outside of given range. In this example given range is ImageBase and ImageBase+SizeOfImage, so when ever eip is not in our range we have API. This can be recoded to take snapshot of all loaded modules and to check when EIP is in range of any of them. Sometimes, some protectors may use dummy API calls to foobar tracers, but we can easily avoid this, checking if EIP is in dummy API. For more about this plugins I suggest you to read my solution to warrantyVoider's crackme [3].

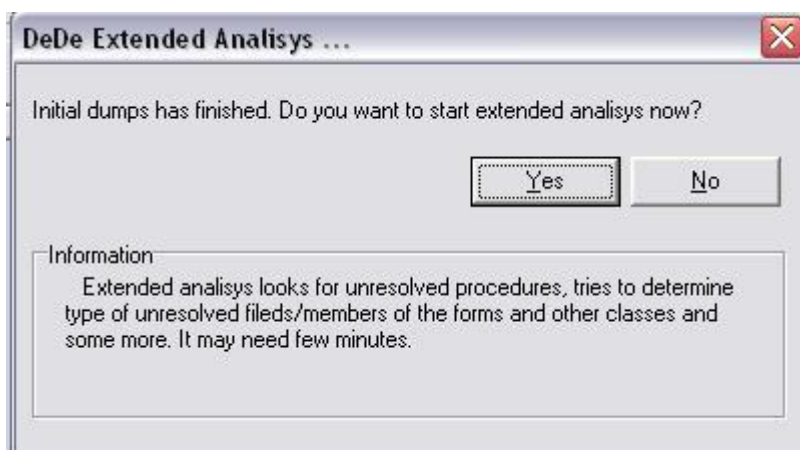


4. Removing limitations

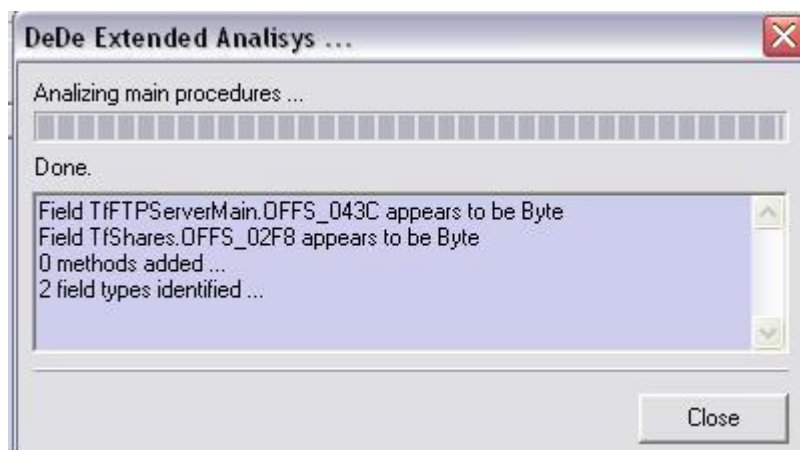
Oki we wanna remove NAG and also to make About box to display our name and email. If you are lazy as I'm than you will patch it. This application can be analyzed using DeDe [2] and, also, using partial dumps so we can code loader without unpacking. Lets find Registration form:



Then we select all possible analyzes available in DeDe:

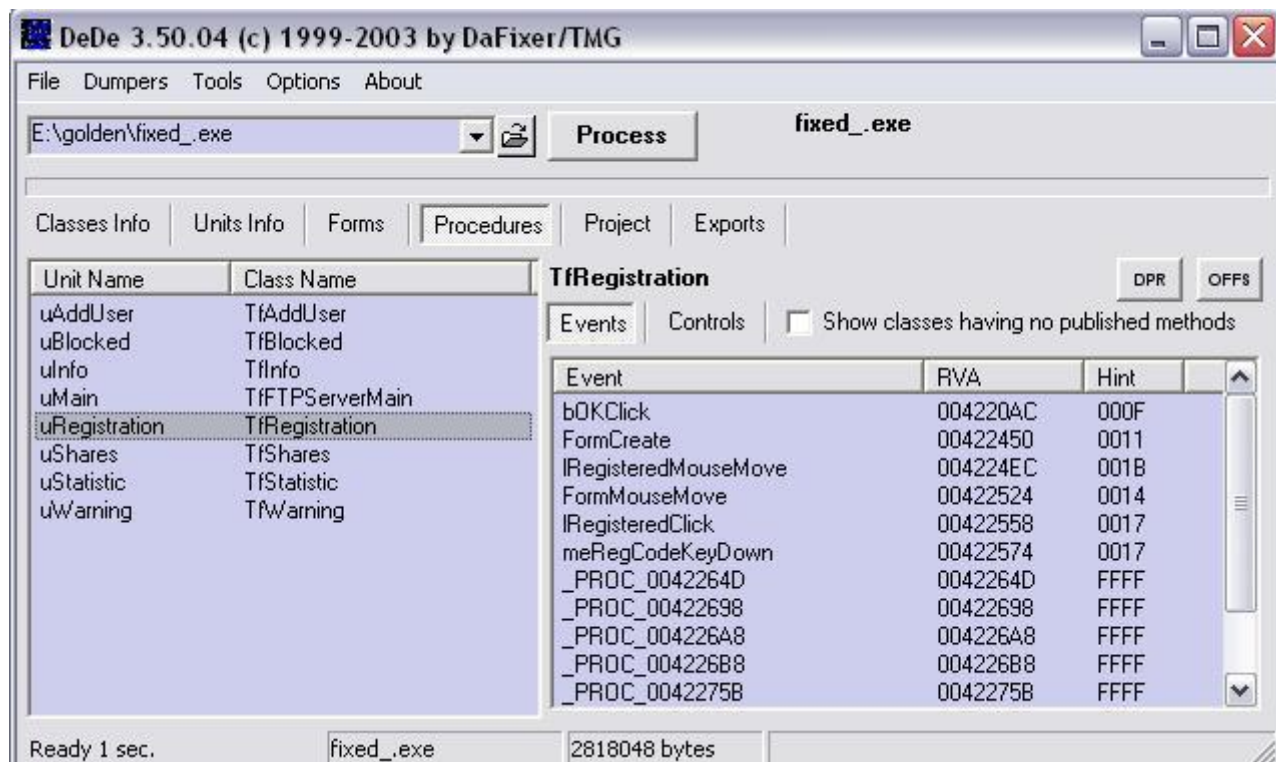


And we click yes:



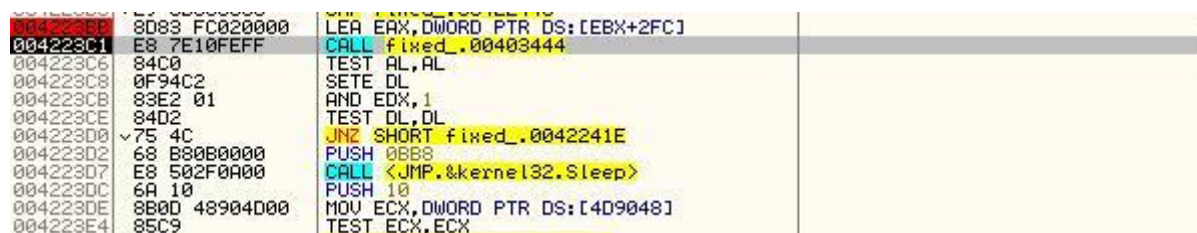


Take a look at Procedures:



Open fixed file in olly and go to [4220ACh](#) and set breakpoint:

go go go go go and check this :



Hahahahahaahahaha Sleep(3000); pro, no way :D Sorry I had to show this, I tough that there is really complex key check routine but what the hell we use some comparison and then we use Sleep to make people think that there is some mad key check procedure... Damn :D If we skip Sleep leetnes by changing Z flag at [4223D0h](#) then our “registered” button will disappear. Take care with proc that is called before JNZ.

If you wanna get key than reverse this a little bit, or just patch it. Well we are gona patch this baby, removing nag also removes “Register” button from main form... huhu...

Load file in IDA and look at main procedure a little bit, then produce nms file and load it in softice via i2s plug-in or produce map file and load it in olly using GODup plug-in. Take a cigarette break again till IDA analyses this.



Oki we are back from cigarette break and ready to continue, we open file in IDA and lets see if key check procedure is called only when we type our serial or there are more references to this proc:

```
.text:00403444
.text:00403444
.text:00403444 sub_403444 proc far ; CODE XREF: .text:00404F5F↓
.text:00403444 ; .text:00404FB7↓p ...
.text:00403444
.text:00403444 var_10 = dword ptr -10h
.text:00403444 var_C = dword ptr -0Ch
.text:00403444 var_4 = dword ptr -4
```

Well seems like it is called 2 times and guess what? Both times from main(), focus on the first reference:

```
.text:00404F5A mov eax, offset unk_40A25F
.text:00404F5F call near ptr sub_403444
.text:00404F64 test al, al
.text:00404F66 jz short loc_404FC9
.text:00404F68 call sub_403600
.text:00404F6D mov word ptr [ebp-274h], 98h
.text:00404F76 cmp eax, 2
.text:00404F79 jnz short loc_404FAF
.text:00404F7B mov eax, 1
.text:00404F80 mov edx, 2
.text:00404F85 push eax
```

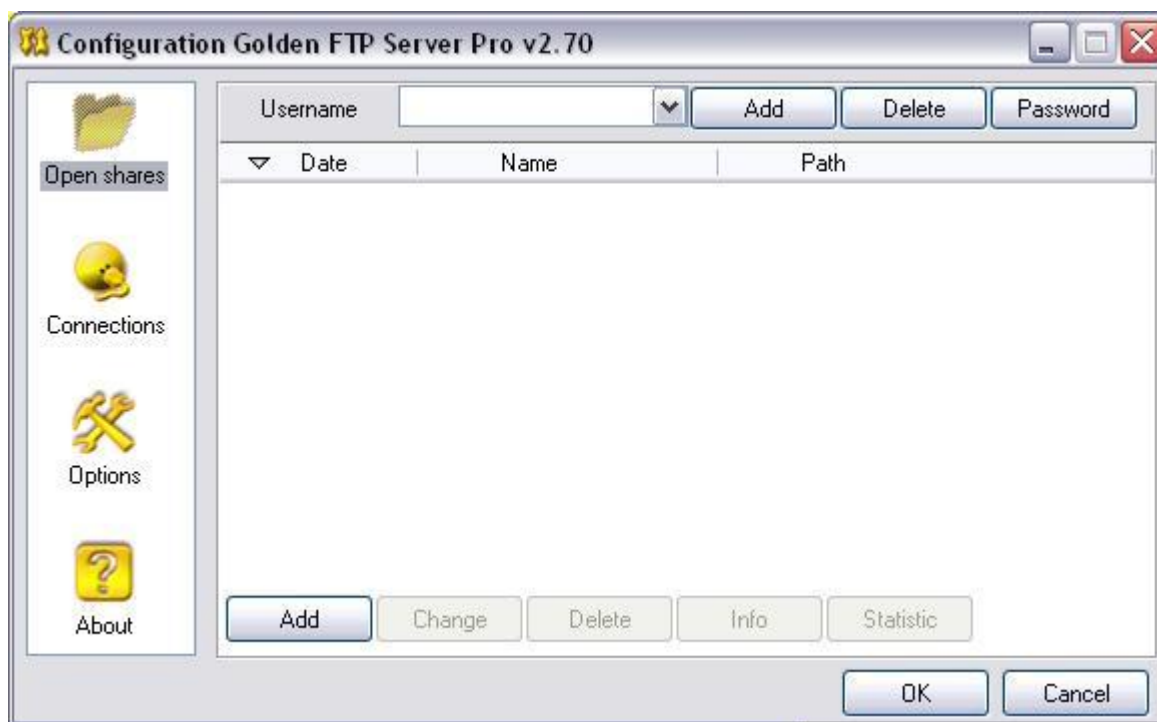
If al is 0 than progy will take us here:

```
.text:00404FC9 loc_404FC9:
.text:00404FC9 mov byte_4C6844, 1
.text:00404FDB
```

Simple enough if registered than byte at 4C6844 is set to 1, else it remains 0. Lets try this, from olly:

00404F5A	B8 5FA24D00	MOV EAX, fixed_.0040A25F	ASCII "oooooooooooooooooooo"
00404F5F	E8 E0E4FFFF	CALL fixed_.00403444	
00404F64	84C0	TEST AL, AL	
00404F66	74 61	JE SHORT fixed_.00404FC9	
00404F68	E8 93E6FFFF	CALL fixed_.00403600	
00404F6D	66:C785 8CFDFF	MOV WORD PTR SS:[EBP-274], 98	
00404F76	83F8 02	CMP EAX, 2	
00404F79	75 34	JNZ SHORT fixed_.00404FAF	
00404F7B	B8 01000000	MOV EAX, 1	
00404F80	

Change Z flag or patch je to jmp and voila:



Where is “Register” button, where is NAG? Hehe, nailed...

One more thing left to fix:



Open IDA and scan for “unregistered” references:

```
.text:00410039      push     4CB2E9h
.text:0041003E      push     offset unk_4DA27B
.text:00410043      call    lstrcmpi
.text:00410048      test    eax, eax
.text:0041004A      jz      short loc_41005F
.text:0041004C      push    offset aUnregistered_1 ; "unregistered"
.text:00410051      push    offset unk_4DA443
.text:00410056      call    lstrcmpi
.text:0041005B      test    eax, eax
.text:0041005D      jnz     short loc_410087
```

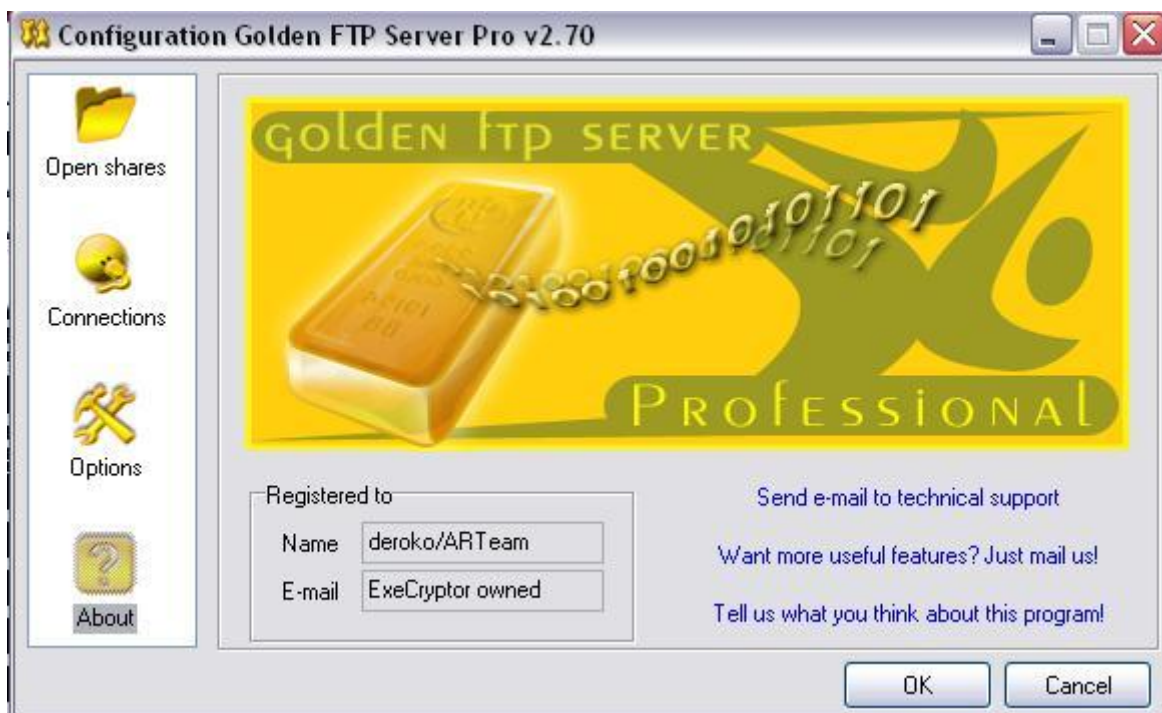
Oki offset 4DA27Bh is for email and offset 4DA443h is for Name so we have to apply 3 patches here:

1. 404F66h - should be modified to jmp
2. 4DA27Bh - store your email (don't be stupid ==)
3. 4DA443h - store your name

Steps 2 and 3 should be inlined.



And after a little bit of modification and inline patch for About box:



Well that's it, no more registration, unpacked and patched...



5. Loader : at the boundary of perfection

I want to write loader for execryptor packed app but I can't use standard loader nor debug loader. Why is that?

1. standard loader can't hook anything when primary thread is suspended because no k32 nor any other dlls are loaded, and we can't force them to load by hooking entry point because TLS will detect us (hook tls?)

2. debug loader can be used with some extra code to make it stealth, but if I think a little bit what should I add to it, than, heh, too much coding for simple patch.

New era rises, and nonintrusive debug loaders are coming to my mind.

Nonintrusive debuggers are stealth [7,12] and very powerful. They run in context of debugged process, there is no need to use debug loop, no need to use Write/ReadProcessMemory, nor Get/SetThreadContext etc... I will skip loader theory because Shub-Nigurrath and Thunderpwr did awesome job with coding loader tutorial series [5,6]

Nonintrusive debugger works by hooking KiUserExceptionDispatcher and examining EXCEPTION code to determine if particular exception is caused by debugger itself (int 3h, int 1h, T flag) or exception is caused by debugged program in which case we pass exception to thread exception handler.

KiUserExceptionDispatcher:

```
mov     ecx, [esp+4]           ptr to CONTEXT
mov     ebx, [esp]            ptr to exception_code
push    ecx
push    ebx
call    sub_7C9377C1           go for thread exception handler
or      al, al                handled?
jz      short loc_7C90EB0A     no, go to RaiseException
pop     ebx
pop     ecx
push    0
push    ecx                   ptr to CONTEXT
call    ZwContinue            set thread Context
jmp     short loc_7C90EB15
```

At this point we are interested in 1st 7 bytes of KiUserExceptionDispatcher that we are going to replace with simple hook:

```
push    nonintrusive_address
ret
```

nonintrusive:

```
mov     ecx, [esp+4]
mov     ebx, [esp]
nonintrusive_tracer/debugger comes after this
```



ecx and ebx hold all important data for our debugger, if exception caused by program is the one that we don't care (e.g. Exception caused by protected application) then we will simply return execution to KiUserExceptionDispatcher+7. If we deal with exception we will call NtContinue same way as KiUserExceptionDispatcher calls it. If you are planning to develop debugger using this technique than my advice would be also to hook call to NtContinue in KiUserExceptionDispatcher so you can set T flag in context.context_eflags and always have control over your target. This was implemented by me in nonintrusive importrec plug-in for warrantyVoider's The Amazing Picture Downloader crackme [3] from www.crackmes.de. Yet again I'm mentioning this crackme because it was really hard to dump and fix it, but it was enjoyable :D

For start I will code one small nonintrusive debugger that will inform user about exceptions in our target process. So you get basic idea how to write code like this. [example]

I will cover only main parts of this code, and I've skipped part that uses GetProcAddress to find needed APIs.

```
<++>
    push    offset pi nfo
    push    offset si nfo
    push    0
    push    0
    push    CREATE_SUSPENDED
    push    0
    push    0
    push    0
    push    0
    push    offset prog
    call    CreateProcessA
; first we create process with primary thread suspended

    push    PAGE_EXECUTE_READWRITE
    push    MEM_COMMIT
    push    1000h
    push    0
    push    pi nfo.pi_hProcess
    call    VirtualAllocEx ; then we allocate some memory in target
    mov     mhandle, eax

; then we adjust some hooks, hook to my memory buffer
    mov     dword ptr[hook+1], eax
    mov     eax, ki user
    add     eax, 7
; and we store return to KiUserExceptionDispatcher+7 in our
; nonintrusive debugger
    mov     dword ptr[goback+1], eax

; change protection of ntdll.KiUserExceptionDispatcher
    push    offset dummy
    push    PAGE_EXECUTE_READWRITE
    push    1000h
    push    ki user
    push    pi nfo.pi_hProcess
    call    VirtualProtectEx
```



```
; store hook in Ki UserExceptionDispatcher
push    0
push    6
push    offset hook
push    ki user
push    pi nfo. pi _hProcess
call    WriteProcessMemory

; copy nonintrusive debugger into allocated memory block
push    0
push    size_nonintrusive
push    offset nonintrusive
push    mhandle
push    pi nfo. pi _hProcess
call    WriteProcessMemory

; and resume suspended thread
push    pi nfo. pi _hThread
call    ResumeThread

push    pi nfo. pi _hThread
call    CloseHandle
push    pi nfo. pi _hProcess
call    CloseHandle

push    0
call    ExitProcess

hook: push    10000001h
      ret

nonintrusive:
; mov in ecx context and in ebx pointer to exception_code
mov     ecx, [esp+4]
mov     ebx, [esp]

; save all register and get delta offset in ebp
pushad
call    delta
delta: pop    ebp
      sub    ebp, offset delta
; we check if this is first exception so we don't use LoadLibraryA
; to many times
cmp     [ebp+first], 0
jne     __skip

lea     eax, [ebp+user]
push    eax
call    [ebp+LoadLibraryA]
inc     [ebp+first]

__skip:
; only inform user about exception
lea     eax, [ebp+exception]
call    [ebp+MessageBoxA], 0, eax, 0, 0
```



```
; restore registers
popad
; go back to KiUserExceptionDispatcher+7
; push 10000001h is patched from loader before copying
goback:
push    10000001h
ret

; needed data for this dummy nonintrusive tracer
first          dd      0
MessageBoxA     dd      ?
LoadLibraryA    dd      ?
exception       db      "exception occurred", 0
user           db      "user32.dll", 0

size_nonintrusive = $-nonintrusive
```

<++>

If we start debugger.exe, 2 message boxes will popup.

Here is output from SoftICE of KiUserExceptionDispatcher:

```
ntdll!KiUserExceptionDispatcher
001B:7C90EAE3 PUSH     00150000
001B:7C90EAE4 RET
001B:7C90EAE5 AND     AL, 51
001B:7C90EAE6 PUSH    EBX
001B:7C90EAE7 CALL    7C90377C1
001B:7C90EAE8 OR     AL, AL
001B:7C90EAE9 JZ      7C90EB0A
001B:7C90EAEA POP     EBX
001B:7C90EAEB POP     ECX
001B:7C90EAC0 PUSH    00
001B:7C90EAC1 PUSH    ECX
001B:7C90EAC2 CALL    ntdll!NtContinue
001B:7C90EAC3 JMP     7C90EB15
```

And my nonintrusive tracer:

```
001B:00150000 MOV     ECX, [ESP+04]
001B:00150004 MOV     EBX, [ESP]
001B:00150007 PUSHAD
001B:00150008 CALL    0015000D
001B:00150009 POP     EBP
001B:0015000A SUB     EBX, 0040112D
001B:0015000B CMPEB PTR [EBP+0040116A], 00
001B:0015000C JNZ     00150030
001B:0015000D LEA     EAX, [EBP+00401188]
001B:0015000E CALL    [EBP+00401172]
001B:0015000F INC     DWORD PTR [EBP+0040116A]
001B:00150010 LEA     EAX, [EBP+00401176]
001B:00150011 PUSH    00
001B:00150012 PUSH    00
001B:00150013 PUSH    EAX
001B:00150014 PUSH    00
001B:00150015 CALL    [EBP+0040116E]
001B:00150016 POPAD
001B:00150017 PUSH    7C90EAF3
001B:00150018 RET
```

At line [150044h](#) you see push KiUserExceptionDispatcher+7 followed by ret.

This is the simplest implementation of nonintrusive debugger. It can't get simpler than this.

Oki, now when you are sure that you understand above code and you are capable to write your own simple nonintrusive tracer then you are ready for exeCryptor loader coding, but this kind of loader will also work for some other protectors (asprotect, armadillo, but you can use standard loaders for those or debug loader).



Also note that template that is introduced here, and source code provided [loader] can be used whenever you get good place with my oepfinder using “Stealth Mode” and “Extra fast” checked.

Now we know how nonintrusive tracers work. For our loader to break at good place (when code is decrypted and decompressed, 3rd MessageBoxA), we are gonna use similar algo as one implemented in my oepfinder.

First we set PAGE_GUARD on code section of our target, then we hook KiUserExceptionDispatcher as shown above and we copy our nonintrusive tracer to allocated buffer, and resume thread. If we did everything as we have planned then there will be no error. I'll tell you that tracing and debugging this code is not an easy task if you use olly, with softice it is easy by adding int 3h in suspicious parts of code and using [i3here on](#).

```
<+>
patch_addr      equ      404f66h
useraddr        equ      4da443h
email_addr      equ      4da27bh
<+>
```

Needed constants for loader, [patch_addr](#) is addr of magic jz. [useraddr](#) and [emailaddr](#) are addresses of a buffers that hold name and email address of registered user.

What is trick with this application? It has code integrity check, if we patch anything in code section we are doomed, you will get nice messageBox informing you that “File is corrupted”, so no permanent patches could be applied. My solution was to break at the point when code is decrypted and to set int 3h at [404F66h](#), when I hit my breakpoint, restore original byte (74h) and redirect EIP to good address ([404FC9h](#)).

Secret of my engine is to catch whenever EIP is in guessed range using PAGE_GUARD, this time we will catch PAGE_GUARD from nonintrusive debugger and we are waiting for 3rd occurrence of EIP in guessed range (remember 3rd MessageBoxA?).

I will skip part of loader that hooks KiUserExceptionDispatcher because it is not difficult at all (same as one in our example except I use VirtualProtectEx to set PAGE_GUARD on guessed range), instead I will focus on real engine hidden here:

```
<+>
nonintrusive:
    ; get ptr to exception_code and context
    ; stuff that is replaced with our hook
    mov     ecx, [esp+4]
    mov     ebx, [esp]

    pushad
    call    delta
delta:    pop     ebp
    sub     ebp, offset delta

    ; check if needed DLLs are loaded
    ; if not, we load kernel32.dll and user32.dll
```



```
; also note that we are within TLS callback so user32.dll is not
; loaded at this point (kernel32.dll is here just for fun)
cmp     [ebp+dllloaded], 0
jne     __skip_dll_load
pushad
lea     eax, [ebp+szdi kernel 32]
push    eax
call    dword ptr[ebp+LoadLibraryA]

; change protection on CreateThread so it can be hooked,
; like this we make out nonintrusive debug loader thread safe
lea     eax, [ebp+dummy1]
call    dword ptr[ebp+VirtualProtect], [ebp+createthread],
1000h, PAGE_EXECUTE_READWRITE, eax
mov     edi, [ebp+createthread]
; hook CreateThread, because this is not thread-safe tracer
; all we do here is to make CreateThread return whenever it is
; called with simple ret 18h
push    dword ptr[edi]
pop     dword ptr[ebp+oldcreatethread]
mov     dword ptr[edi], 18c2h

inc     [ebp+dllloaded]
popad

__skip_dll_load:
; check exception_code
cmp     dword ptr[ebx], EXCEPTION_GUARD_PAGE
je      __check_eip
cmp     dword ptr[ebx], EXCEPTION_BREAKPOINT
je      __breakpoint

; if not my exception I set PAGE_GUARD on code section
__setgp:
lea     eax, [ebp+dummy1]
push    eax
push    PAGE_EXECUTE_READWRITE or PAGE_GUARD
push    [ebp+range1]
push    [ebp+basestart1]
call    dword ptr[ebp+VirtualProtect]
popad
; and go back to KiUserExceptionDispatcher+7
jmp     __goback

; there is only one int3h that I'm looking for and that is the
; one at 404F66h
__breakpoint:
mov     esi, [ecx.context_eip]
cmp     esi, patch_addr
; if not my int3h, then it is nothing more than exception caused
; by protected app so we set page_guard and continue execution
; from KiUserExceptionDispatcher+7
jne     __setgp
mov     esi, patch_addr
; restore original opcode
mov     byte ptr[esi], 74h
```



```
; instead of changing Z flag, redirect EIP to 404FC9h
mov     [ecx.context_eip], 404fc9h
```

```
; and store our signature in this application : D
```

```
lea     esi, [ebp+crackedby]
mov     edi, useraddr
cld
mov     ecx, 13
rep     movsb
lea     esi, [ebp+email]
mov     edi, emailaddr
mov     ecx, 21
rep     movsb
```

```
; restore bytes in KiUserExceptionDispatcher
```

```
lea     esi, [ebp+nonintrusive]
mov     edi, [ebp+kiuser]
mov     ecx, 7
rep     movsb
```

```
; restore bytes in hooked CreateThread
```

```
mov     edi, dword ptr[ebp+createthread]
push    dword ptr[ebp+oldcreatethread]
pop     dword ptr[edi]
```

```
popad
jmp     __myntcontinue
```

```
; here is where we check if eip is in a guessed range
; if you plan to use this code on some other target
; don't forget to modify range.
```

```
__check_eip:
mov     esi, [ecx.context_eip]
cmp     esi, [ebp+basestart1]
jb      __notoep
cmp     esi, [ebp+topend1]
jnb     __notoep
```

```
; we need 3rd stop in code section
```

```
inc     [ebp+eipnrangecount]
cmp     [ebp+eipnrangecount], 3
jne     __notoep
```

```
; if it is 3rd break in code, store int 3h at jz address
```

```
mov     esi, patch_addr
mov     byte ptr[esi], 0cch
jmp     __notoep
```

```
pushad
```

```
; delete TLS callback if any, so oilly can be attached w/o problemo
; at this point we need ImageBase which is store in Peb.ImageBase
```

```
mov     eax, dword ptr fs:[30h] ; PEB
mov     eax, dword ptr [eax+8] ; ImageBase
mov     ebx, eax
add     ebx, dword ptr[ebx+3ch] ; PE header
```



Unpacking and dumping ExeCryptor, and coding loader

```
cmp [ebx.NT_Optional_Header.OH_DirectoryEntries.DE_TLS.DD_VirtualAddress], 0
je __@@3

mov ecx, [ebx.NT_Optional_Header.OH_DirectoryEntries.DE_TLS.DD_VirtualAddress]
add ecx, eax
pushad
lea eax, [ebp+dummy1]
call dword ptr[ebp+VirtualProtect], ecx, 100h, RWX, eax
popad
mov dword ptr[ecx+0ch], 0 ; delete TLS call back ptr...

__@@3:
popad

__notoep:
; remove PAGE_GUARD
lea eax, [ebp+dummy1]
push eax
push PAGE_EXECUTE_READWRITE
push [ebp+range1]
push [ebp+basestart1]
call dword ptr[ebp+VirtualProtect]

popad
jmp __myntcontinue

__myntcontinue:
push 0
push ecx
call del ta2
del ta2: pop ebp
sub ebp, o del ta2
call dword ptr[ebp+NtContinue]

nop ; no return from here

__goback:
; patched from loader with KiUserExceptionDispatcher+7
push 10000001h
ret

; and some data needed by nonintrusive loader

NtContinue dd 0
int3heap dd 0
VirtualProtect dd 0
LoadLibraryA dd 0
createthread dd 0
kiuser dd 0
dllloaded dd 0
basestart1 dd 0
topend1 dd 0
range1 dd 0
dummy1 dd 0
oldopcode db 0
eipinrange dd 0 ; we stop on 3!!!
```



Unpacking and dumping ExeCryptor, and coding loader

```
ol dcreatethread      dd      0
szdi kernel 32        db      "kernel 32. dl l ", 0
crackedby             db      "ExeCryptoreeee", 0
email                db      "loader is working", 0
si zenonintrusive     =      $-nonintrusive
<++>
```

If we compile and run this code we will see that our loader is working, take a look:



Well as you can see code is very simple, but effective, this kind of loader can be used to patch armadillo, asprotect, engima, execryptor really fast and safe.

Note that we can use drX registers to set hardware breakpoints on execution and gain control without even patching nor changing code. In this way we will also avoid .code integrity check..

S verom u Boga, deroko/ARTeam



6. Conclusion

Well as you can see coding loader isn't hard at all, with a little bit of patience and thinking we defeated one of the best protections available. ExeCryptor is very strong and good protection, hard to break and I'm sure that any application protected with it, won't be easy to unpack. I will not provide you with binary of loader, because I don't wanna see millions of cracks on warez sites using my loader. Similar thing happened to Registry Mechanic after condzero released his tutorial on muping Armadillo based VB targets with a twist [10].

Please note that this text is for educational and research purposes only.

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.

7. References

- [1] "Thread Local Storage : The Hidden Entry Point", rog g biv, #8 29a e-zine
- [2] "Cracking using partial dumps", Shub-Nigurrath, <http://tutorials.accessroot.com>
- [3] "solution to warrantyVoider's The Amazing Picture Downloader", deroko, http://www.crackmes.de/users/warrantyvoider/the_amazing_picture_downloader/
- [4] "The Art of Computer Virus Research and Defense", Peter Szor, Symantec Press
- [5] "Writing Loaders For Unknown Packers", Shub-Nigurrath, <http://tutorials.accessroot.com>
- [6] "Cracking With Loaders Theory and General Approach and Framework", Shub-Nigurrath and Thunderpwr, <http://tutorials.accessroot.com>
- [7] "A non-intrusive debugger for Windows NT", Ryan S. Wallach, http://www.usenix.org/events/usenix-win2000/full_papers/wallach/wallach.pdf
- [8] "Weakness Of The Windows Api Part1", Gabri3l, <http://tutorials.accessroot.com>
- [9] "Unpacking Armadilloed DLL and tools", deroko, <http://tutorials.accessroot.com>
here you will find all needed stuff to compile my sources including tasm32/tlink32
- [10] "Registry Mechanic", Condzero, <http://tutorials.accessroot.com>
- [11] "Golden FTP pro v2.70", Target, <http://www.intechhosting.com/~access/ARTeam/tools/golden-ftp-server-pro.zip>
- [12] "EXCEPCIONES, SEH Y TRAZADORES 7", Akira, http://www.intechhoting.com/~access/ARTeam/tutorials/non_arteam/akira/Estudio_de_las_Excpciones,_las_SEH_y_los_trazadores_7.pdf.rar
- [13] "Bypass Hardware Breakpoint Protection", Mattwood FRET, http://reverseengineering.online.fr/spip/IMG/pdf/Bypass_Hardware_Breakpoint_Protection.pdf
- [14] "Log DRX operations via DR7 General Detection(GD) bit", yates, <http://www.yates2k.net>



8. Greetings

I wish to tank all the ARTeam members for sharing their knowledge, to 29a virus writing group for one of the best e-zines, and to all great coders... and of course you for reading this tutorial.



<http://cracking.accessroot.com>