Private exe Protector unpacking

by deepzero, 2011

Scientia Potentia Est.

Table of Content

Contents

Foreword

Private exe Protector (PEP) is a lower end intermediate PE file protection and licensing solution. The price, 200$, is quiet high, which might be one of the reasons this protector is rarely being used. I chose it here because there is little to no documentation available on version 3.x. From the PEP homepage:

Private exe Protector (PEP) is a professional licensing, anti-tampering and software examination system. PEP works with traditional methods, such as file compression, code fragment encryption, metamorphic loading, protection from debugging and file tampering , and features new innovative techniques, including data protection with stolen resources technique and partial code execution on a virtual machine. Licensing functions can be automatically integrated into the protected program, which allows the end user to quickly and securly manage all licences issued with the built in licence manager. All in all, it is the ideal solution for software developers.

The main functionalities of protection are:

- Encryption, compression and protection of software
- Reliable protection of software against examination: PEP will protect your product against debugging, reverse assembling and other hazards.
- Integrated licensing system: license manager, user database management, key generation, binding of software to particular hardware.
- Unique software data protection system: PEP features stolen resources technique and counteraction to application memory dumping.
- Conversion and execution of program code fragment on a high-speed virtual machine, special markers for a deeper integration of protection into protected program.
- Complete control and flexibility of the protection option's set-up and configuration.
- Creation of trial applications with restriction in terms of the number of application launches, the number of days; the system of reminders and restrictions can be customized manually using built-in APIs.
- Maintenance of licence Blacklist
- No problems with antivirus software.
- Regular and significant updates of the protector.
- Support of all common programming languages from assembler, Delphi, C, etc. to script languages, such as Python, blitz3d, etc. Both .exe files and .dll files can be protected.
- Complete compatibility with any NT system: x32, x64 (2000, XP, Vista, 7, etc.), even with Linux (wine)

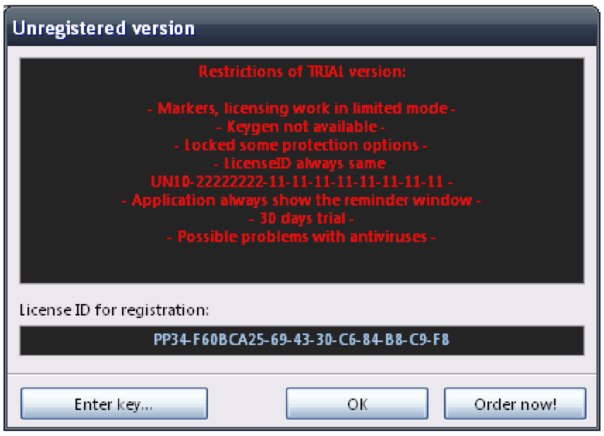
Tools used

OllyDebug 1.10, LordPe, ImpRec, CFF Explorer, PiD, ResHacker, HexEdit (all freeware).

OllyDebug plugins: ollyadvanced, MultiMate assembler, odbgscript, IDAficator,AnalyzeThis!.

Introduction

When execution the main .exe file, we are greeted by a startup nag, reminding us that we are running the “trial” version of the Protector.



Unregistered version

Restrictions of TRIAL version:

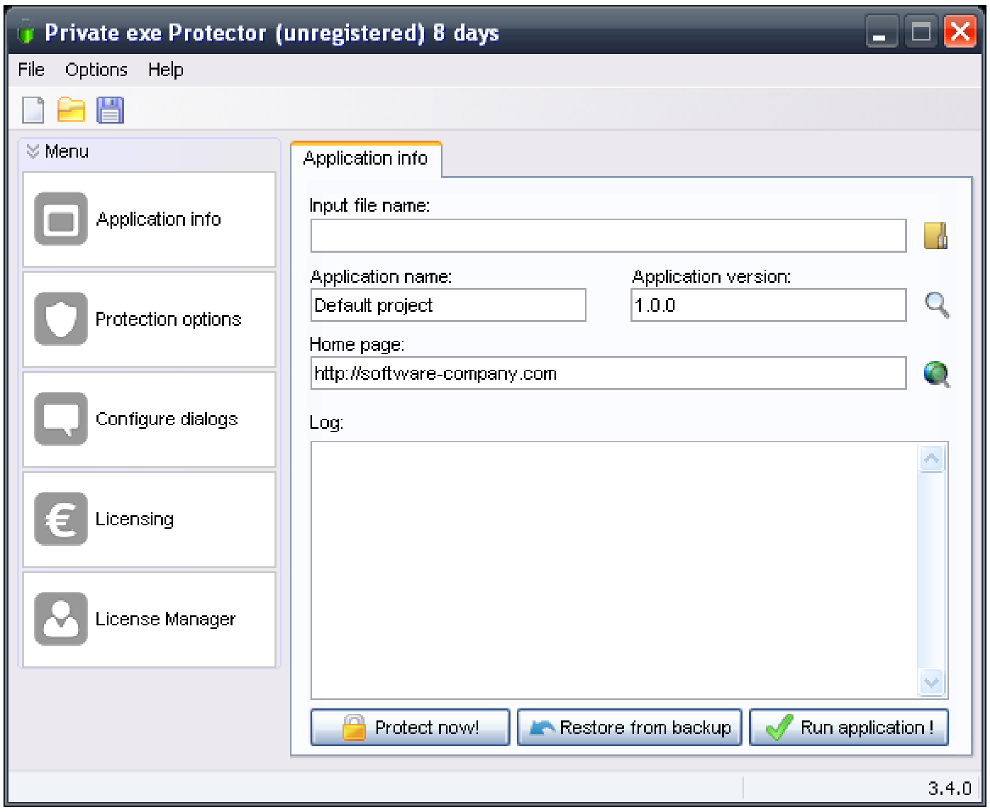
- Markers, licensing work in limited mode -
- Keygen not available -
- Locked some protection options -
- LicenseID always same
UN10-22222222-11-11-11-11-11-11-11-11 -
- Application always show the reminder window -
- 30 days trial -
- Possible problems with antiviruses -

License ID for registration:

PP34-F60BCA25-69-43-30-C6-84-B8-C9-F8

fig. 1 the startup nag

Despite there being an option to register the software, the “trial” version is actually a crippled “demo” version. Essential features of the program (e.g. key generation) are not implemented. We will prove this later. Once the program has started up, we see its interface:



Private exe Protector (unregistered) 8 days

Input file name:

Application name: Default project

Application version: 1.0.0

Home page: http://software-company.com

Log:

3.4.0

fig. 2 the PEP interface

Unfortunately, we notice that our RAM usage has increased significantly:

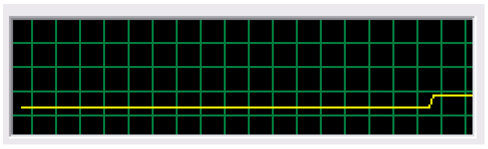


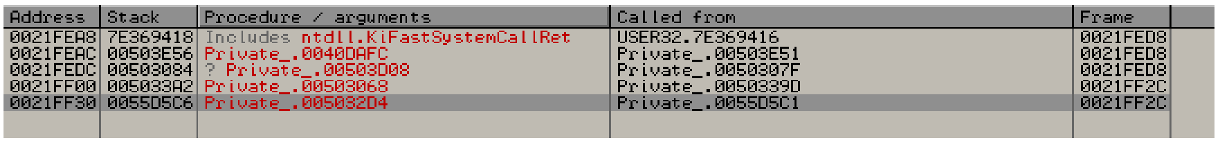
fig. 3 extensive memory usage - the anti-dump protection

This is because of PEPs “anti-dump” protection feature, which we will deal with in a second.

Locating the OEP

The first step of a successful unpack is usually locating the original entry point (OEP). As always, there are several ways to reach our goal. Let´s start with the most straightforward solution.

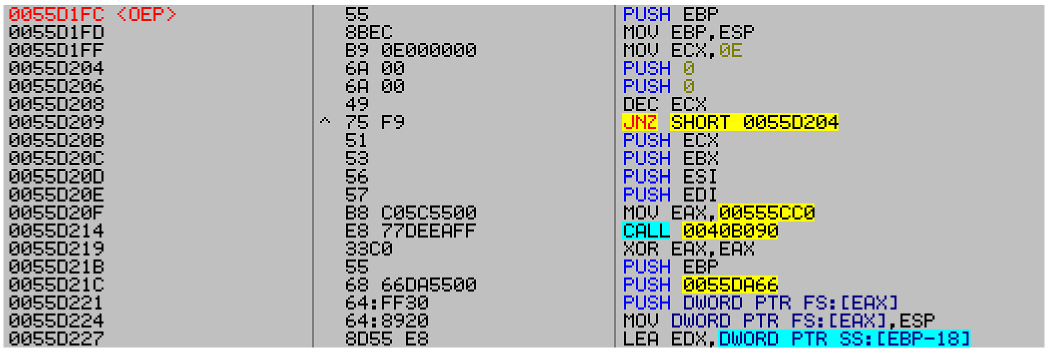
The idea is simple: run the application up to a point where you can be sure execution control has been handed over to the protected application, then examine the (call-)stack for clues. Obviously, the main form is displayed by the application, so we can simply pause the application in our debugger as soon as the main form has loaded. Note that the nag is NOT part of the program, but rather part of the protection system and thus displayed by the protector`s code *before* reaching the OEP.



```
Address  Stack    Procedure / arguments               Called from            Frame
0021FEA8 7E369418 Includes ntdll.KiFastSystemCallRet  USER32.7E369416        0021FED8
0021FEAC 00503E56 Private_.0040DAFC                   Private_.00503E51      0021FED8
0021FEDC 00503084 ? Private_.00503D08                 Private_.0050307F      0021FED8
0021FF00 005033A2 Private_.00503068                   Private_.0050339D      0021FF2C
0021FF30 0055D5C6 Private_.005032D4                   Private_.0055D5C1      0021FF2C
```

fig. 4 the call stack

Clearly, Virtual Address (VA) 0x0055d5c1 is what we are looking for – and indeed: this address lies within the OEP function. The actual OEP is thus the start of the function, VA 0x55D1FC:



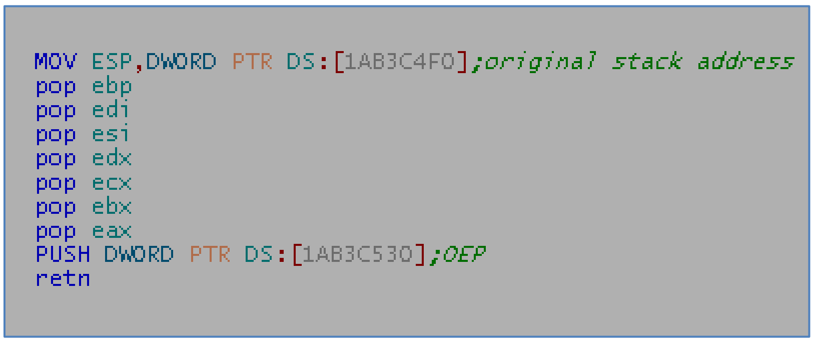
```
0055D1FC <OEP>    55             PUSH EBP
0055D1FD          8BEC           MOV EBP,ESP
0055D1FF          B9 0E000000    MOV ECX,0E
0055D204          6A 00          PUSH 0
0055D206          6A 00          PUSH 0
0055D208          49             DEC ECX
0055D209        ^ 75 F9          JNZ SHORT 0055D204
0055D20B          51             PUSH ECX
0055D20C          53             PUSH EBX
0055D20D          56             PUSH ESI
0055D20E          57             PUSH EDI
0055D20F          B8 C05C5500    MOV EAX,00555CC0
0055D214          E8 77DEEAFF    CALL 0040B090
0055D219          33C0           XOR EAX,EAX
0055D21B          55             PUSH EBP
0055D21C          68 66DA5500    PUSH 0055DA66
0055D221          64:FF30        PUSH DWORD PTR FS:[EAX]
0055D224          64:8920        MOV DWORD PTR FS:[EAX],ESP
0055D227          8D55 E8        LEA EDX,DWORD PTR SS:[EBP-18]
```

fig. 5 OEP

The OEP is a clean standard Delphi OEP and not virtualized, obfuscated or stolen, which is why it can also be easily located by a signature scan.

However, how does the Protector “know” where the OEP is? Let`s investigate this.

A run trace reveals the OEP is reached stepping over a RETN instruction at VA 0x1AB50B98. This is the point where the execution is handed over to the protected application. An analysis of the previous (obfuscated) code confirms this:



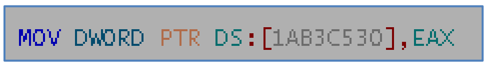
```
MOV ESP,DWORD PTR DS:[1AB3C4F0];original stack address
pop ebp
pop edi
pop esi
pop edx
pop ecx
pop ebx
pop eax
PUSH DWORD PTR DS:[1AB3C530];OEP
retn
```

fig. 6 handing over control the main program

The above code has been deobfuscated. First, the stack is reset to its original address, then the general purpose CPU registers are restored, lastly the OEP is pushed to the stack, so that the code flow continues execution there.

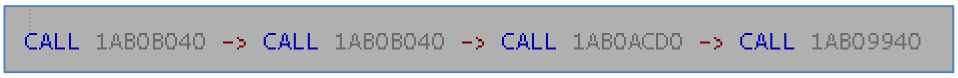
So far, so good. But how was the OEP address retrieved?

Right before the “lead-out” of the protector’s code, it`s written to the stack:



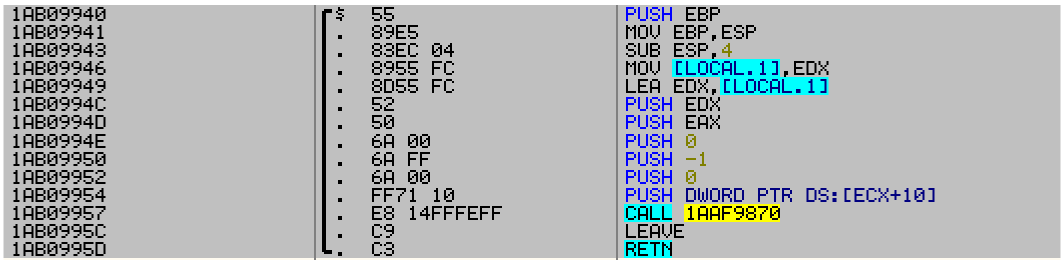
```
MOV DWORD PTR DS:[1AB3C530],EAX
```

Two commands before this line of code, EAX is set in these calls:



```
CALL 1AB0B040 -> CALL 1AB0B040 -> CALL 1AB0ACD0 -> CALL 1AB09940
```

The routine at 0x1AB09940 is short and simple:



```
1AB09940  $ 55             PUSH EBP
1AB09941  . 89E5           MOV EBP,ESP
1AB09943  . 83EC 04        SUB ESP,4
1AB09946  . 8955 FC        MOV [LOCAL.1],EDX
1AB09949  . 8D55 FC        LEA EDX,[LOCAL.1]
1AB0994C  . 52             PUSH EDX
1AB0994D  . 50             PUSH EAX
1AB0994E  . 6A 00          PUSH 0
1AB09950  . 6A FF          PUSH -1
1AB09952  . 6A 00          PUSH 0
1AB09954  . FF71 10        PUSH DWORD PTR DS:[ECX+10]
1AB09957  . E8 14FFFEFF    CALL 1AAF9870
1AB0995C  . C9             LEAVE
1AB0995D  . C3             RETN
```

fig. 7call to CryptDecrypt() WinAPI

The call is an API call, which directly calls the WinCrypt API „CryptDecrypt()“ through the Protectors internal IAT. The internal IAT (only used by protector code) has no further obfuscation or redirection.