© The CodeBreakers-Journal, Vol. 1, No. 2 (2004)
http://www.CodeBreakers-Journal.com

AsProtected Notepad!

Author: E. Labir

Abstract

When i started to read tutorials about dumping i saw that i could only expect something like:

```
use tracex N times
this is your EP...
dump with procdump (or other tool)
use the utility XXXX to reconstruct the imports
```

What can i learn from it? NOTHING!. I don't wanna have a magic recipe working only on obsolete versions of the protectors, that's not enough, i want to learn.

Along this paper, i'll try to show common methods used by the anti-crack ppl to prevent from dumping or debugging. The general method of this paper is to debug into looking for all anti-debug, get rid of it and then, only then, analyze to understand how to dump (and how to prepare for dumping, as well).

As an example, i will refer to the demo version of Asprotect (we will protect some common files like Notepad or Regedit) but there is NOT information about any commercial target (don't waste your time asking me for that).

DISCLAIMER: what you do with this knowledge is ONLY your responsability.

Keywords: *Reverse Code Engineering, Protection-Techniques, Analysis, Evaluation of Effecience of Software-Protection-Systems*

Contents

I. TARGETS

For ellaborating this tut i've used the demo version of asprotect to protect Notepad.I as commented above, i won't give any info commercial targets. The purpose of this paper is to explain how your software can be in a better way, not to crack any particular real-life target. Your first task is to download the demo of Asprotect from its homepage and to read its help, this will give you a good idea of how it works.

When using the demo version i've set all options available:

```
Resource Protection = YES
Use Max. compresion = YES
Anti-debugger = YES
Checksum = YES
Trial info ... limited trial:
Number of days = 30
Number of executions = 10
Reminder message = YES
Expiration date = 2003, december the 31th
```

After pressing on the protection button you should see more or less the following:

```
" Use CRC check protection...
 Use anti-debugger protection...
 Use 30 trial days limitation...
 Use expiration date (31/12/03)...
 Use 10 executions limitation
 Use reminder
 Use built-in dialogs...
 Protection done ..."

 File size: ... compressed to ..., Ratio: ..%"
```

II. INTRODUCTION

Asprotect has different features, for example:

- You can choose to encrypt parts of your code.
- It protects your imports table against dumping.
- Adds tons of garbage code to make debugging pretty annoying
- Adds anti-debug, checksums, hides your entry point.
- Uses the windows registry to store the time-limit and registration information
- Let's the programmer to use some special procedures, f.e. GetRegistrationInformation().

In this paper i won't examine the RSA protection, this is a loss of time. Even if i'd be able to find a mistake into it, this could be corrected in a matter of seconds. The problem with using cryptography is very simple:

- If you encrypt an important part of the program nobody can properly test the demo version.
- If you only encrypt the "save" procedure i will substitute your encrypted code by my own one.

So cryptography isn't so relevant.

Tools you need:

```
A debugger, i use Olly
A hex editor, i use hiew
Procdump (only to dump from memory)
```

Tools you do NOT need:

```
Imports reconstructors
```

Some words about how to proceed:

Observe that, when unpacking a target, you don't need to restrict yourself to an only "victim":

- Program a pretty simple app and protect it, find its EP (f.e. show a msg just at the EP will help). You can, and must, leave clues on your code (recognizable strings and stuff) leading you to the info you need. This will help a lot if you attach your debugger to the target.
- Same with other simple targets, like notepad. Do your conclusions still hold?.
- Try several real-life targets and compare.
- Add/remove different protections (with/without time-trials, with/without compressing the .rsrc,...)

It will be much easier if you do it in an scalated way, from the easiest to the most difficult. This strategy suggests that offering "demo" versions of the protector that semi-protect your programs it's NOT a good idea, don't you agree?. Amazingly enough, nobody seems to follow this strategy...

III. ANTI-DISSAMBLER

The best way to understand it it's to deal with an example, let's debug together some instructions of the Asprotected Notepad to see how it works:

```
01001000    $ 68 01300101      PUSH pnotepad.01013001
01001005    . E8 01000000      CALL pnotepad.0100100B
0100100A    . C3               RETN
0100100B    $ C3               RETN
```

As you see, Asprotect has used some rets to jump. When you see a RET just look at the stack to see where you're gonna jump.

```
01013001     60                 PUSHAD ; preserve all registers
01013002     E8 03000000        CALL pnotepad.0101300A ;
01013007    -E9 EB045D45        JMP 465E34F7 ;
0101300C     55                 PUSH EBP ;
```

First comment is that doing the calls stepping over is a bad idea, do it with CALL pnotepad.0101300A and you will see that you arrive to the first exception. Well, you have done about 50 calls to different APIs, unpacked lots of strings,... without noticing it. Next time, promise you will step into. Look again at the call, 0101300A is NOT the beginning of an instruction in your dissambler. Therefore you're gonna jump into the middle of an instruction.

```
PUSHAD ; execute this
CALL pnotepad.0101300A ; debug into
```

Now, you step into the call, and you're at 0101300A:

```
0101300A     5D                 POP EBP             ; pnotepad.01013007
0101300B     45                 INC EBP
0101300C     55                 PUSH EBP
0101300D     C3                 RETN
```

After the jump, all the dissambled code has drastically changed in front of your eyes. This has been got by doing the following:

```
pushad
call _below

db E9h ; garbage inserted between the instructions
db EBh ;
db 04h ;

_below: pop ebp
inc ebp
...
```

The dissambler, before to run into that jump, has taken the E9 and has seen this is the start of a jump. Therefore has dissambled this as a jump and so all your code below the call is wrongly dissambled. Am afraid there's no cure for this, you have to get used to not to know what's gonna be the next instruction you will run into (you can "go to" this next instruction and you will see what's gonna be executed, but this can be done only a few times cos is pretty time-consuming).

Some obvious advises:

- When you see a look to some API have a look at its parameters (on the stack)
- All instructions having FS:[XXh] are either SEH related or anti-debug.
- All instrutions of the kind ”modify [esi]”, or other register, where esi points somewhere inside your target are probably gonna compute a checksum or simply unpacking something.
- Once you’re sure the code run by a call doesn’t contain anything interesting you can step over it.
- Make a list of secure breakpoints - bpx that don’t bother anything - and proceed as follows:

```
Set the first bpx.
Run your code till the first bpx.
Remove the first breakpoint.
Set the second.
...
```

IV. SELF-DECRYPTION

Let’s get a feeling of how Asprotect works.

As expected, there isn’t any useful string reference at the beginning. Run the code and you’ll recieve your first exception:

```
0084377C     3100               XOR DWORD PTR DS:[EAX],EAX ; eax = 0
```

Now, let’s have a look at the current string references. I’ve selected these ones for you:

```
"kernel32.dll",
"GetProcAddress",
"Protection Error",
"regfile",
"Software\ASProtect\SpecData"
```

The first 3 are associated to the call the app does to IsDebuggerPresent, the ”regfile” has to be with the keyfile and the later is a registry key where Asprotect keeps the trial information.

And let’s also have a look at intermodular calls the debugger can find:

```
USER32.MessageBoxA
ADVAPI32.RegQueryValueExA
KERNEL32.GetSystemTime
KERNEL32.CreateFileA
KERNEL32.GetProcAddress
KERNEL32.CreateFileA
...
```

Their use is far from evident, now since we see that the offsets where the app does the call are consecutive, we go there and examine it. As you can well imagine we have there the "IAT" of Asprotect. Asprotect has constructed its own IAT and left them in an array of redirected jumps:

```
008350C8    -FF25 B8818400      JMP DWORD PTR DS:[8481B8]  ; ADVAPI32.RegSetValueA
008350CE     8BC0               MOV EAX,EAX
008350D0    -FF25 B4818400      JMP DWORD PTR DS:[8481B4]  ; ADVAPI32.RegSetValueExA
008350D6     8BC0               MOV EAX,EAX
....
```

It's also interesting to note that if you examine the imports table of Asprotect you will only find GetModuleHandleA and GetProcAddress, this 2 APIs are enough to locate any other one we need.

Debugging into - as we're gonna do - requires being able to overcome all "infinite" loops generated by Asprotect, the next lines try to enlighten this question.

How to recognize the loops and overcome them

The loops use to satisfy some patterns and are easiy to identify. I've examined several targets without surprises, some examples:

1) Push/Pop loops. They do something like this:

```
_iterate:
mov ecx, [ebx+edx] ; edx = counter, initially set to FFFFFFFF
; ebx = pointer to the code to start the decryption from
....  ; code to compute new value
push ecx ; push new value
... ;
pop [ebx+edx] ; new instruction!
sub edx, 4 ; decrease counter
;
cmp edx, -320 ; compare the counter to a negative value
jnz _iterate ; another iteration
jmp _continue ; leave loop (a jump or any other instruction)
```

How to ovecome them: Simply set a bpx on "`jmp _continue`", to check it's right compute the final value of edx and set a bpx on condition EDX == final value. Other protectors generate more "accurate loops", so the instruction "`jmp_continue`" is kept encrypted as well, and then you have to set a bpx on "edx == 320".

2) This example, also from protected NOTEPAD, shows a more complicated loop based on a double push/pop structure:

```
_RealIterate: ; start of the loop
push [ebx+ecx] ;
xor ax, 80BC ;
pop edx ; edx = [ebx+ecx]
call _down ; pushes a dword and jumps below, inside the loop
;
....  ; not executed instructions, anti-dissambler
;
_down:  add esi, 71741184 ; meaningless
pop esi ; esi = offset after the call
;
xor ecx, 55baa1f6 ; compute new value
xor edx, 181419f7 ;
sub edx, 2c828064 ;

;
... ; useless computations
;
push edx ; push new value
... ;
pop [ebx+ecx] ; set new opcode
... ;
cmp ebx, -674 ; check counter
jnz _iterate ;
; here goes your safe bpx
_iterate:
... ; trash code
jmp _RealIterate
```

As you see, the couple push [ebx+ecx] and pop [ebx+ecx] deflates Asprotect. You can see too that you have, as i commented above, a safe bpx after ”`jnz _iterate`”.

General notes about the loops:

All loops start to decrypt from the end of the code backwards till they ”almost” reach the current loop (mind the ”almost”). The reason to start backwards is that you can’t set a bpx inmediately after the loop, at least you’re sure it won’t affect the decryption.

Some packers do the next mistake:

```
jnz _offset1
jmp _offset2
```

When you arrive to the jump it’s rather usual to see on the debugger ”jump is taken” , this will surely mean that we’re into a loop. Set a bpx there and continue debugging into, next time you appear into your bpx you will be totally sure you’re in a loop. Bpx ”`jmp _offset2`” and you have it.

This loops, are really effective against tracers. Simply mind that the tracer is a big factor times slower than simply running the code... Try yourself this one, not traced doesn't delay your app:

```
xor eax, eax
_iterate:
inc eax
cmp eax, 05FFFFFh
jne _iterate
```

BTW: If you see your tracer doesn't seem to advance, pause it and you will see its current status, help it by overcoming by hand the loop.

BTW: Sometimes, you will see this kind of "rare" conditional jumps

```
jpe  xxxxxxxxh ; jump if parity...
                jno  xxxxxxxxh ; jump if not overflow..
... ; other pretty uncommon jumps
```

As you know, this conditional jumps are much harder to code than the standard "jnz", so you should suspect. It's very weird to find one of this jumps used to make a loop, be careful.

Back to Asprotect:

Needless to say, Asprotect generates lots of junk code to hide the loops. It's also interesting that Asprotect generates instructions like "int 20",... and others that is pretty evident won't be run by the program. The idea is that the cracker shouldn't distinguish the real code from the junk code, right?. Why to generate "int 20"?. I will explain below how to overcome a possible "prefetch queue" trick using this mistake.

Locating kernel32

We commented above that Asprotect has only GetModuleHandleA and GetProcAddress into its imports, therefore it needs to locate a lot of APIs. Asprotect uses some standard algorithms to do this job:

Now, we have to pay attention to the interval between the 3rd and 4th loops. We can see a very standard kernel32 location algorithm that essenctially does the following:

```
mov eax, [esp+24]   ; this was the first dword on the stack at the beginning of execution,
                        ; it contained a pointer inside kernel32
and eax, FFFF0000 ;
add eax, 10000 ; round eax to the memory alignment and prepare it for the loop

_next: sub eax, 10000 ; the image base of kernel32 has to be a multiple of 10000
cmp eax, 'MZ'; ms-dos stub?
jne _next ;  nope, continue searching
;  yes, kernel32 image base found
```

After locating kernel32 you can see it finds GetProcAddress and uses it for locating some basic APIs.

Locating the ”basic” APIs

Locating the APIs is done next to locate kernel32 image base. The algorithm uses a pre-computed CRC32 of the names of the APIs to locate them into the exports. You will see that if the pre-computed CRC32 agrees with the one he computes from the string into the exports then he stores this address, more or less, we could describe it as follows:

```
esi = pointer to string with the name of the next API at kernel32' exports
call CRC32
cmp eax, edi ; eax = return code
; edi = CRC32 of the API we look for
jne _continueSearch ; takes the next string into the exports and tries its CRC32
je  _storeOffset
```

The APIs he locates now are simply the following: GetModuleHandleA, LoadLibraryExA, VirtualAllocEx and VirtualFreeEx. This are the APIs it needs for a start.

(if you’re at Regedit, now EIP == 42279B)

Example:

```
006E17B1    8B33           MOV ESI,DWORD PTR DS:[EBX] ; read CRC of API to look for
006E17B3    89B5 6B030000  MOV DWORD PTR SS:[EBP+36B],ESI ; input for the next procedure
006E17B9    E8 0B000000    CALL XXXXXX.006E17C9 ; find api
006E17BE    AB             STOS DWORD PTR ES:[EDI] ; store offset of the API
006E17BF    83C3 04        ADD EBX,4 ; next API to locate
006E17C2    833B 00        CMP DWORD PTR DS:[EBX],0 ; more APIs to look for?
006E17C5   ^75 EA          JNZ SHORT XXXXXX.006E17B1 ; yes, continue search
006E17C7    61             POPAD ; nope, restore registers and continue
006E17C8    C3             RETN ;
```

This way, he locates the following ”basic” APIs from kernel32 (saves their addresses for later use):

Pay attention on the use of the CRC, this makes more difficult to know what’s the API Asprotect looks for. Unfortunately, the algorithm has a security mistake: some moment, somewhere, you’ll have a register pointing to the name of the API... later you’ll see more names over and over again... then it’ll be evident that it’s locating the APIs. Fixing this ”security” problem is hard, if you try to import it by ordinal then you can have compatibility problems (a good method seems to code yourself a procedure to find the APIs by ordinal or name).

V. THROWING EXCEPTIONS TO THE DEBUGGER

@DAEMON: You're gonna see, man, that there's still some ppl in the world who haven't read your articles...

For this part of the tutor, you will need basic knowledge of the SEH. Look at DAEMON's site at www.anticrack.de for some comprehensive guides. I guess you know that when an exception occurrs the SEH is called, but only if we're not debugging. If we're debugging the exception is sent to the debugger and then we ourselfs have to decide what to do with this.

Provoking lots of exceptions as anti-debug is a must, the advantages are:

- Debugging becames much more time-consuming
- Most exceptions have to be solved ”by hand”, cos the debugger is unable to handle them properly
- Some exceptions are really hard to understand

The demo version of ASPROTECT uses only 2 kind of Exception tricks, let's review them:

TRICK 1. CHANGING EIP

You can find it at the first exception generated by Asprotect:

```
0084377C     3100               XOR DWORD PTR DS:[EAX],EAX
```

Set a bpx on the SEH handler, let's recall that you have the SEH at fs:[0] (probably, you will see the current SEH handler at the stack marked by your debugger), and pass the exception to it. This is what Asprotect does:

```
push _return ; first instruction at the SEH, your breakpoint
inc [esp] ; increase the return offset
ret ; ret used as jump, we jump to 1+[esp]

_return+1:
mov eax, [esp+0Ch] ; eax = pointer to the context structure
add [eax+b8],2 ; increase by 2 two the value of EIP (consult the context definition)
xor eax, eax ; prepare to continue exception
ret ; get out from handler, should jump to Except1 + 2, i.e. the next instruction
; to "xor [eax], eax" (which has length 2).
```

Note that, increasing into 2 the EIP register will lead us to the next instruction to the one which provoked the exception. Note too that, at this point, the context of the current thread will be restored before to continue execution (excluding EIP, that has been increased by 2).

Conclusion: To overcome this trick NOP out the instruction provoking the exception and continue running the program.

After this anti-debug trick ASPROTECT removes the current handler, and sets another one for the next exception:

```
pop fs:[0]
pop eax ; esp increases in 4, eax is a trash register
push value1
...
push value7
ret ; ret used as jump, we go to value7.
```

TRICK 2. CHANGE EIP + DEBUG REGISTERS

Proceed as above, into the SEH you can see the following instructions:

```
seh:
mov eax, [eax+C] ; pointer to context
add [eax+b8],2 ; EIP = next instruction to the one provoking the exception

; till here nothing special
push ecx ; saves ecx
xor ecx, ecx ; overwrites the debug registers
mov [eax+4], ecx ; DR0 = 0
mov [eax+8], ecx ; DR1 = 0
mov [eax+C], ecx ; DR2 = 0
mov [eax+10], ecx ; DR3 = 0
mov [eax+18], 155 ; DR7 = 155
pop ecx ; restores ecx, equilibrates the stack
xor eax, eax ; return from the handler + restore the context
; The debug registers will be set to the new values (0,...,155)
ret ; and we'll continue execution into the next instrucion
```

Note that he doesn't overwrite DR6, i think he should set DR6 = 0. The Debug registers are used by our debugger and so we must NOT let him to touch them. Fortunately, he doesn't check the value of the Debug Registers after returning from the SEH. Therefore, it's safe to NOP this trick too.

Asprotect generates about 25 exceptions of the kind of trick1 and trick2 (too few and too easy).

IMPROVING THE EXCEPTIONS TRICKS

First of all, it's naive to generate always the exceptions with exactly the same instructions. For example, we can do:

```
way 1)
db FFFFh ; invalid opcode

way 2)
int 3 ; debug exception
nop ;
way 3)
xor eax, eax ; overwritting an illegal memory reference (not always xor [eax], eax)
pop [eax] ;
```

And a very very long etc... Also, in the SEH itself there's no need of changing only EIP. As you know, the SEH gives us kinda ring0 access to the context, Why not to change the debug and stack registers as well?

```
push ebp
mov ebp, esp

mov eax, [ebp+10h] ; now eax points to the context structure
; first, overwritte the debug registers
mov [eax+4], ecx ; DR0 = 0
mov [eax+8], ecx ; DR1 = 0
mov [eax+C], ecx ; DR2 = 0
mov [eax+10], ecx ; DR3 = 0
mov [eax+18], 155 ; DR7 = 155

; now, change ESP, EBP, EIP
mov edx, [new_esp]
mov dword ptr [eax+0C4h], edx ; change the value of ESP on return from the SEH
mov edx, [new_ebp]
mov dword ptr [eax+0B4h], edx ; change the value of EBP on return from the SEH
mov edx, [new_eip]
mov dword ptr [eax+0B8h], edx ; change the value of EIP on return from the SEH

xor eax, eax ; prepare to return from exception and continue the program
pop ebp
ret
```

Now, let's review the attack we've suffered: it has overwritten the debug registers and also changed the values of ESP, EBP, EIP. Thefore to overcome the exception we should do, instead of passing it to the debugger (amazingly, Olly survives to this exception so simply pass it):

```
nop the exception
jump to EIP
set esp, ebp to the new values
```

With this, you've got rid of this trick, as well. There're much more sophisticated tricks (consult DAEMON's cave, is he and not me who should write about this stuff).

VI. CHECKSUMS

A. CHECKSUM OF THE FILE NOTEPAD.EXE

After touching with procdump, open+save it is enough cos it changes the last access to the file, the asprotected app detects the file has been changed and stops working showing a message box "file corrupted, please run a virus check...".

Now, we proceed to pass exceptions (overcoming all tricks) till we're displayed the message box. We make a note of the last exception we passed, starting again to debug into for locating it: You inmediately arrive to the following instructions:

```
00843406     3100                 XOR DWORD PTR DS:[EAX],EAX ; exception
00843408     64:8F05 00000000     POP DWORD PTR FS:[0] ; remove seh
0084340F     58                   POP EAX ;
00843410     A1 647E8400          MOV EAX,DWORD PTR DS:[847E64] ; eax = precomputed checksum
00843415     3B45 FC              CMP EAX,DWORD PTR SS:[EBP-4] ; ok?
00843418     74 44                JE SHORT 0084345E ; yes, go to ...
```

If you look at the values of the registers at 00843415 you can see they look more or less like:

```
EAX = D3F1E6B1, [EBP-4] = 989F8C34.
```

Is not difficult to imagine what's he comparing there: neither pointers to a dll, nor small user varibles,.. Trying for different modifications of the same app yields the answer (one of the values remains fixed, the good one, and the other randomly changes). Moreover, we can see - some lines below - the string "file corrupted" on the screen. Of course, to bypass the checksum simply set the value of eax to match [ebp-4].

This checksum has simply taken into account the value for the protected file pnotepad.exe (BTW: i know it cos i've hooked the APIs for mapping files into memory and i've traced back the value of the checksum, you can do it yourself but mind that it takes pretty long), therefore one would expect to find checksums of the memory image of notepad as well... yes, there're some of them :-)

BTW: D3+F1+E6+B1, 98+9F+8C+34 are also useful as checksums (fail with a probability 1/100). And is pretty easy to make a mistake and think that they're indices of a loop or something else. If you think you need more accuracy into your checking then use 4 of them, D3F1E6B1 and 989F8C34 are TOO evident.

Enumerating all checksums, appart from not teaching too much, is pretty lengthy. Therefore, we'll simply examine the core ideas involved into it.

B. CHECKSUMS OF THE MEMORY IMAGE

I must admit this section is only done for fun, cos getting trapped into some of the checksums of the protected notepad is not easy at all (remove all your bpx after using them, that's all).

First of all, a simple question: Before to jump to the protected app, don't you think we could check its memory image? Yes, seems a good idea. The demo version of asprotect does. To find this checksums, or to find any other checksum, we proceed modifying the memory image and running it.

Before the last exception, we arrive to a point where we see another "not desired" exception: "don't know how to step cos memory at [xxxxxxxxh] is not readable".

So we proceed going again till the last exception before this crash and debug into... We need to locate a loop computing the checksum, or something like that, and comparing to the old (stored) value. What Asprotect does is to compute several checksums of its memory image to look for changes, if a change is found then it won't return correctly from the following instructions:

```
xor [esp], eax ; xor with the computed checksum
ret ; crash if changed!
```

Observe that, since the value of the checksum changes as random as you modify the image this will surely produce a "random" crash.

For finding faster where this checksums are i debugged both an altered target and a non-altered one, i recommend you to do the same when dealing with this kind of problems. As you can observe, it's important to remove the bpx you set to avoid the loops and so on, this way Asprotect doesn't find any change (neither you see the trick, that will remain hybernating for the next time).

Now, let's see how the protection works, this is one of the checksums i mentioned above (you have 3 calls to this checksum procedure):

```
0083AEF4     68 D76A3C93        PUSH 933C6AD7
0083AEF9     68 DC150000        PUSH 15DC
0083AEFE     68 14990000        PUSH 9914
0083AF03     68 00A00100        PUSH 1A000
0083AF08     FF35 D4748400      PUSH DWORD PTR DS:[8474D4]
0083AF0E     E8 F5E2FFFF        CALL 00839208
```

You can see what's it "checksuming" setting a bpx at:

```
0083923D     66:8B13            MOV DX,WORD PTR DS:[EBX]
```

and examining the contents of ebx. So, the call returns a value into eax that is the checksum of the memory image and then does the following:

```
0083AF13     310424             XOR DWORD PTR SS:[ESP],EAX
0083AF16     8B05 D4748400      MOV EAX,DWORD PTR DS:[8474D4]
0083AF1C     010424             ADD DWORD PTR SS:[ESP],EAX
0083AF1F     C3                 RETN
```

If the checksum is not correct it'll make a mess.

VII. MORE ANTI-DEBUG: IsDebuggerPresent, FS:[18],...

Don't you feel a bit dissapointed if your target doesn't call IsDebuggerPresent? So do i...

Some moment, you're gonna be prompted a message box saying "debugger detected, ...". Then is the moment to start debugging into from the last exception onwards to see where we have been fooled:

```
008434B8     3100          XOR DWORD PTR DS:[EAX],EAX ; last exception before the msgbox
008434BA     EB 01         JMP SHORT 008434BD
008434BC     68 648F0500   PUSH 58F64
```

If you are a bit patient, there's a moment when you arrive to:

```
00840F1E     68 8C0F8400   PUSH 840F8C                 ; ASCII "kernel32.dll"
00840F23     E8 3842FFFF   CALL 00835160               ; JMP to kernel32.GetModuleHandleA
00840F28     8BD8          MOV EBX,EAX
```

This will simply get the value of the image base of kernel32, later:

```
00840F2D     B8 A40F8400         MOV EAX,840FA4            ; ASCII "HrFgavcc`wVtbtmf}"
00840F32     E8 E5FDFFFF         CALL 00840D1C
```

This is a decryption procedure, now pay attention on the following fact:

```
HrFgavcc`wVtbtmf} = 17 chars
IsDebuggerPresent = 17 chars
```

Got it?. It's not a bad idea to pad the name of the API so its length don't make us to suspect.

```
00840F3F     50                  PUSH EAX     ; offest of name of the API
00840F40     53                  PUSH EBX            ; image base of kernel32
00840F41     E8 2242FFFF         CALL 00835168       ; JMP to kernel32.GetProcAddress
```

Finally, it checks if the API has been found and ,if so, calls it:

```
00840F46     8BF8                MOV EDI,EAX          ; kernel32.IsDebuggerPresent
00840F48     89FE                MOV ESI,EDI
00840F4A     85FF                TEST EDI,EDI
00840F4C     74 06               JE SHORT 00840F54
00840F4E     FFD6                CALL ESI      ; call IsDebuggerPresent
```

Overcoming this call to IsDebuggerPresent is pretty simple, just change the return to zeroe or simply patch the API at kernel32 (this is what the API does under WinXP):

```
MOV EAX,[DWORD FS:18]
MOV EAX,[DWORD DS:EAX+30]
MOV EAX, 0 ; here goes your PATCH
RETN
```

ADDING OTHER TRICKS

As you seen, the anti-debug into the demo version isn't so strong. One should think of adding new tricks, but adding them correctly meaning this:

Analyzing a real-real life target i came across this exception:

```
CALL 010309E0
MOV ECX,DWORD PTR DS:[EBX+8]
ADD ECX,108
MOV EAX,ECX
MOV EDX,DWORD PTR DS:[EAX] ; exception, eax not readable!
```

So now i went inmediately to the SEH. Since i didn't see it into the stack i went to examine fs:[0], but then i was surprised cos it pointed to kernel32. Why?. Obviously, if Asprotect would have placed here an anti-debug SEH trick then we'd have the SEH installed somewhere else therefore this wasn't that. If it wasn't a SEH trick then this instruction shouldn't have been run, at least with that value on eax, so we've been fooled at some moment. Now, it's time to go back (till the previous exception) and start debugging into...

```
mov eax, fs:[30]
movzx eax, byte ptr [eax+2]
or al, al
je ...
```

You have to admit that this is a bit shocking, right?. The correct way of checking if this is anti-debug is to program it yourself and see what happens when you're debugging in contrast to when you aren't. Since this is the first time, i will give you the assembler source for this trick (compile with TASM):

```
.386
.Model Flat ,StdCall

extrn     ExitProcess: PROC
extrn     MessageBoxA: PROC

.Data

szZeroe db 'zeroe',0
szNotZeroe db 'NOT zeroe',0

.Code
Main:
push 0 ; for the message box
push 0

mov eax, fs:[30h]
movzx eax, byte ptr [eax+2]
or al, al ; checking the return

je _zeroe
push offset szNotZeroe
jmp _MessageBoxA

_zeroe:
```

```
push offset szZeroe

_MessageBoxA:
push 0
call MessageBoxA


push 0
call ExitProcess
End Main                        ;End of code, Main is the entry point
```

Now, proceeding to debug and to run without debugging you can see that AL satisfies:

```
al = 0 Not Debugged
al = 1 Debugged
```

And so you have to clear the value of eax to continue debuggig by the right branch of code. You also have other TEB (or PEB) based tricks - check Daemon's cave, as usual - you can inject in your code, one of the simplest being simply to do the same that IsDebuggerPresent does:

```
mov eax, fs:[18h]
mov eax, [eax+30h]
        movzx eax, byte ptr [eax+2]
test eax, eax
je ...
```

Proceeding as above, do it yourself, you can see that: eax = 0 Not Debugged, eax = 1 Debugged

Some comments:

- There is no need of crashing the app inmediately after detecting you're being debugged, delay it for a while (doing it a.s.a.p means i will have a very accurate idea of where to look for the trick). Also, have always set a SEH - to confuse the cracker - and check into it if the exception code corresponds to a "good" (provoked by you) exception or a bad one
- Use some anti-dissambler to hide this fs:[XX] based tricks, i almost suffered a heart attack when i first saw it on the screen (do it for me, i'm too young to die).
- You are allowed to put the same trick at different places... sometimes not-being cracked is just a matter of boring to death your cracker...

VIII. HOOKING

In this section we'll try to see how to understand the internals of the protector, since this would need a lot of space i'm gonna limit myself to the registry access (closely related to the time-limits). Once we know where's all the anti-debug is very trivial to extract the info you need, just observe which are the imports of the target - also those dynamically loaded - and hook them all. While reading this section, it's a good idea to open regedit and verify yourself the changes.

How to:

For example, if you're interested in knowing about the time-limits then you should hook all APIs imported by the protector that could read from files, the registry or ask for this information. The next, are found at the demo version of Asprotect:

```
Advapi32:
RegCloseKeyA
RegEnumKeyExA
RegOpenKeyExA
RegQueryInfoKeyExA
RegQueryValueExA
RegSetValueA
RegSetValueExA
```

Now, you simply have to let the program run with the hooked APIs and take note of what you see:

```
 call to RegOpenKey for opening HKEY_CURRENT_USER\Software\Borland\Locales with KEY_ALL_ACCESS.
 call to RegOpenKey for opening HKEY_CURRENT_USER\Software\Delphi\Locales with KEY_ALL_ACCESS.
(nothing matters if they're not at your OS)

 call to RegOpenKeyExA to open HKEY_CURRENT_USER\Software\Asprotect\SpecData with access KEY_READ
 call to RegQueryValueExA, value name = "A93471655372341". This call is done with Buffer = NULL and
pBufferSize != NULL , this will return the length of the registry key. Observe that "A93471655372341"
has already been computed.

 call to RegOpenKeyExA to open HKEY_CURRENT_USER\Software\Asprotect\SpecData with access KEY_READ
 call to RegQueryValueExA, value name = "A93471655372341". This time, the call retrieves the value.
...
 call to RegSetValueA, to set a new value for HKEY_CURRENT_USER\Software\Asprotect\SpecData

 call to RegSetValueA, to set the value of HKEY_CURRENT_ROOT\.key to null. Why to null? Simply cos it
want to set a new value and so it deletes it.
 call to RegOpenKeyExA, to open with KEY_SET_VALUE access HKEY_CURRENT_ROOT\.key.
 call to RegSetValue, to set the value of  HKEY_CURRENT_ROOT\.key to "regfile".
...
```

BTW: sometimes, setting hooks directly on the system dll's, you will see calls like: 0006F8D8 772AD1DA /CALL to RegOpenKeyExA from SHLWAPI.772AD1D4 Obviously, you're not interested into this cos we only want to know those calls done FROM the protector.

I don't want to enlarge too much this section but, hooking the access to the registry, you can get things like: Asprotect uses `HKEY_CURRENT_USER\Software\Asprotect\SpecData` to keep its registration info, this value is updated every time the target is run (and Asprotect needs so). The value of this key is encrypted. We can also find an easy weakness on this, f.e.: bypassing the time-limits is very easy, just delete the key fron the registry and hook the calls to GetSystemTime and GetLocalTime. We can also hook the whole registry key generation process and generate all kind of keys for all machines and times. Of course, you don't need this all if you know how to dump :D

It's also much fun to hook GetModuleHandle:

```
0006FF64   0084C07A  /CALL to GetModuleHandleA from 0084C074 ; find the image base of kernel32
0006FF68   0084C79C  \pModule = "kernel32.dll"
...
0006FF64   0084C4AC  /CALL to GetModuleHandleA from 0084C4A6 ; find the image base of user32
0006FF68   008484BC  \pModule = "user32.dll"
...
0006FE24   0084129F  /CALL to GetModuleHandleA ; image base of the calling program
0006FE28   00000000  \pModule = NULL
```

debugging a bit after returning from GetModuleHandleA (we're inside Asprotect):

```
0084129F     68 AA128400        PUSH 8412AA ;
008412A4     E9 85000000        JMP 0084132E ;
0084132E     5A                 POP EDX                         ; 008412AA
0084132F     5B                 POP EBX ;
00841330     68 37138400        PUSH 841337       ;
00841335     C3                 RETN ; ret used as jump
00841337     8943 F6            MOV DWORD PTR DS:[EBX-A],EAX         ; pnotepad.01000000
```

Where [EBX-A] is an offset inside Asprotect. Later, if we see this the very same value into the target's unpacked code - do a find for the constant - we know we have to type there a call to GetModuleHandleA. With a bit more of patiente we find a call already INSIDE the unpacked target, this returns to:

```
00841483     5D                 POP EBP ; restore ebp
00841484     C2 0400            RETN 4 ; jmp to NOTEPAD.01006C4E
...
01006C4C     . FFD7             CALL EDI ;  we come from this call
01006C4E     . 50               PUSH EAX                       ; |Arg1 = 01000000
01006C4F     . E8 ADBBFFFF      CALL pnotepad.01002801         ; \pnotepad.01002801
```

Now, hook "call edi" and you will see it does "call 00841460", there we find this:

```
00841460     55                 PUSH EBP
00841461     8BEC               MOV EBP,ESP
00841463     8B45 08            MOV EAX,DWORD PTR SS:[EBP+8]
00841466     85C0               TEST EAX,EAX
00841468     75 13              JNZ SHORT 0084147D
0084146A     813D 787A8400 0000 CMP DWORD PTR DS:[847A78],400000
00841474     75 07              JNZ SHORT 0084147D
00841476     A1 787A8400        MOV EAX,DWORD PTR DS:[847A78]
0084147B     EB 06              JMP SHORT 00841483
0084147D     50                 PUSH EAX
0084147E     E8 DD3CFFFF        CALL 00835160     ; JMP to kernel32.GetModuleHandleA
00841483     5D                 POP EBP
00841484     C2 0400            RETN 4
```

This is a call to GetModuleHandle with a couple of tests to see if Asprotect is still there. Just replace the call by a mov eax, `[pre_computed_module_handle]` and the job's done.

IX. HOW TO FIND THE ENTRY POINT

A. DEMO VERSION OF ASPROTECT

For now, you have enough information to overcome all tricks: checksum, IsDebuggerPresent and, of course, the exceptions. So you have to count the number of exceptions you have till the protected app loads, restart and start debugging from the last one. Just look for a jump taking you to a far address.

General notes about the entry point:

This is pretty evident, but let's do some simple observations.

1) Many programs have the following instructions at the very beginning:
```
push ebp
mov ebp, esp
sub esp, ?? ; constant to allocate local storage into the stack
push ebp ; preserve registers, you can find this at Delphi programs
push esi
push edi
```
2) You should see calls to well known procedures at the very beginning (after pressing "Analyze" in the debugger), for example:
```
From kernel32, user32,...

GetModuleHandleA
FindWindow
ShowWindow
GetVersion(Ex)
GetCommandLine
...

From MSVCRT (Microsoft Visual C++ Runtime Library)

__set_app_type
__p__f_mode
__setusermatherr
...
```
(Complete this list yourself)
3) We have to get to this piece of code by a far jump, this jump can be achieved in many ways, f.e:
```
mov eax, [variable] ; precalculated entry point
push eax ; eax = 401038h
ret ; push the return and use ret to do the jump
```
You will probably have some popad nearby, may be inmediately before, to restore all registers before to jump to the real entry point.
4) Is possible that, even after a far jump like, we haven't reached the entry point (try to distinguish it by the calls as mentioned in 2). In this case, the exepacker has probably placed there a decryption routine to confusse you and you will jump to the real EP after decrypting something (this decryption routine could be place at the PE cave, i.e. between the section headers and the start of the code).
5) Since you know where exactly is all the anti-debug you can use the "tracer" of your debugger, this will save a lot of time (but you don't need it, takes 30 minutes to find the EP):
In Olly, this is done as follows:
```
Debug,  Set Condition,  EIP is in range ...
Now, go to Trace Into.
```

Case One: Finding the Entry Point for Regedit (demo version of Asprotect, Win98)

Of course, you shouldn't get info the non-protected Regedit cos you don't have it in real life. Well, as i said, go to the last exception and NOP it. Now, let's start debugging till we find the right EP (entry point). Some tips:

```
don't pay attention on any short jump
don't pay attetion in any call that doesn't lead far away
when you find a ret, pay attention on [esp]
remember the advises about the loops
```

This is the final exception (you're inside a heap):

```
00BC3033    3100                    XOR DWORD PTR DS:[EAX],EAX
00BC3035    64:8F05 00000000        POP DWORD PTR FS:[0]
00BC303C    58                      POP EAX
00BC303D    833D 847EBC00 00        CMP DWORD PTR DS:[XXXXXXXXh],0
00BC3044    74 14                   JE SHORT 00BC305A
```

Important note: the polymorphism of Asprotect isn't very high, you will find the same kind of "start" in every target you examine (full version of Asprotect is included).

NOP the "xor [eax], eax" and let's debug. To save some time we'll trace over some calls, f.e.:

```
00BC3050    BA 04000000             MOV EDX,4
00BC3055    E8 6EDAFFFF             CALL 00BC0AC8 ; Trace OVER
... ; Trace OVER the call to message box
; displaying "this is un unregistered..."

00BD75FF    58                      POP EAX
00BD7600    E8 0A000000             CALL 00BD760F ; Trace Ove
```

r

At this point, you will see that Regedit loads, so we have to trace INTO the last call. (applying this trick we'll save up a lot of time :-), he,he..). So, if you have stepped into you must find this call:

```
00BD61C8    81C6 8A7BA105           ADD ESI,5A17B8A
00BD61CE    E8 0E000000             CALL 00BD61E1 ; step into too
```

Now, it isn't too difficult to find out a decryption loop:

```
00BD5B90    FF341A                  PUSH DWORD PTR DS:[EDX+EBX]
...
00BD5BDE    893413                  MOV DWORD PTR DS:[EBX+EDX],ESI
...
00BD5BE6    83EA 04                 SUB EDX,4 ; comparison
00BD5BE9    0FBFC8                  MOVSX ECX,AX
00BD5BEC    81FA 54EBFFFF           CMP EDX,-14AC ; standard loop, as we saw
00BD5BF2    0F85 19000000           JNZ 00BD5C11
```

Very standard, as you know. And, after this decryption loop the final jump:

```
00BD5C85    894424 1C                MOV DWORD PTR SS:[ESP+1C],EAX  ; AREGEDIT.0040B747
00BD5C89    61                       POPAD
00BD5C8A    50                       PUSH EAX
00BD5C8B    C3                       RETN
```

The ”mov [esp+1c], eax” is done to preserve the value of eax after ”popad”. You are at the entry point:

```
0040B747   /. 55                     PUSH EBP
0040B748   |. 8BEC                   MOV EBP,ESP
0040B74A   |. 83EC 1C                SUB ESP,1C
0040B74D   |. 53                     PUSH EBX
0040B74E   |. 56                     PUSH ESI
```

Press Analyze and it will be totally evident. Let me warn you: the full version of Asprotect protects much better the entry point, we'll see it later.

Case Two: Finding the EP for Notepad (Demo version of Asprotect, WinXP)

As we did above, go to the last exception:

```
00843033     3100               XOR DWORD PTR DS:[EAX],EAX
00843035     64:8F05 00000000   POP DWORD PTR FS:[0]
0084303C     58                 POP EAX
0084303D     833D 847E8400 00   CMP DWORD PTR DS:[847E84],0
00843044     74 14              JE SHORT 0084305A
...
0084306B     6A 00              PUSH 0
0084306D     E8 5E21FFFF        CALL 008351D0                    ; JMP to USER32.MessageBoxA
...
0085A2A8     81C6 D20FB455      ADD ESI,55B40FD2
0085A2AE     E8 08000000        CALL 0085A2BB ; debug into
...
0085A2D6     8BC2               MOV EAX,EDX
0085A2D8     FF343A             PUSH DWORD PTR DS:[EDX+EDI] ; start of decryption
...
0085A309     890C17             MOV DWORD PTR DS:[EDI+EDX],ECX ; end of decryption
...
0085A316     81FA 54EBFFFF      CMP EDX,-14AC ; final comparison of the loop
0085A31C     0F85 0F000000      JNZ 0085A331 ;
...
0085A39A     894424 1C          MOV DWORD PTR SS:[ESP+1C],EAX   ; pnotepad.01006AE0
0085A39E     61                 POPAD ; restore all the registers
0085A39F     50                 PUSH EAX ;
0085A3A0     C3                 RETN ; jump to notepad
... ;
01006AE0     . 6A 70            PUSH 70 ; entry point!
01006AE2     . 68 88180001      PUSH pnotepad.01001888 ;
01006AE7     . E8 BC010000      CALL pnotepad.01006CA8 ;
```

So now we know how to get the Entry Point of every target. Great!. To dump the file simply change the first instructions by an infinite loop, ”jmp 01006AE0” for notepad, and run it. Go to ProcDump and you can Dump it, after having done so, remember to redirect the entry point to its true value and also to change the infinite loop by the original instructions.

Of course, hidding the EP can be done definetly better, let’s review an example. First thing to note in the example is that there’s a moment when you see the same ”exit” than in the demo version of Asprotect, but it’s only trying to confuse you... nice psychologic-trick:

```
0105E6D3      75 07                       JNZ SHORT 0105E6DC ; jump IS ALWAYS taken
0105E6D5      894424 1C                   MOV [DWORD SS:ESP+1C],EAX ; got it???
0105E6D9      61                          POPAD
0105E6DA      50                          PUSH EAX
0105E6DB      C3                          RETN
```

Here you naively set a bpx at the mov [esp+1c], eax and run it... but it fails. In this case, it was obvious that it was a trick cos the instructions before the conditional jump made impossible to reach it [yet another mistake], this should be avoided. This same target has some very interesting polymorphic code to protect the EP, for example:

```
0105E846      8D6424 4E    LEA ESP,[DWORD SS:ESP+4E]
0105E84A      F3:          PREFIX REP:                  ; Superfluous prefix
0105E84B      EB 02        JMP SHORT 0105E84F
...
01056BB6    F3:            PREFIX REP:                    ; Superfluous prefix
01056BB7    EB 02          JMP SHORT 01056BBB
01056BB9    CD 20          INT 20
```

Playing with esp makes a sense when you hear that Win9x crashes if esp-X doesn’t point to a valid address. I can’t tell you much more, cos i’m now under WinXP, but you have a very interesting thread at, of course, board.anticrack.de about why this happens [look for a common thread from Merlin and Drizz, about December the first]. There’s also an interesting point where you reach the following:

```
01056BF2    66:8135 FB6B0501 E1EF    XOR WORD PTR DS:[1056BFB],0EFE1
01056BFB    0AED                     OR CH,CH
01056BFD    CD 20                    INT 20
```

There you can see that 1056BFB is the NEXT instruction to the current one, when xored it becames:

```
01056BFB    EB 02                    JMP SHORT 01056BFF
```

I guess you’ve already heard about the prefetch-queue trick (if not, look for it and read). Is that? The answer is not and is indeed pretty easy to deduce. Suppose the instructions are run ”as they are”, then:

```
XOR dword ptr [_next],0EFE1 ; not really executed cos it was in the CPU cache
_next: OR ch, ch ; executed
INT 20 ; priviledged instruction!, crashes in all WinNT
```

BTW: correctly coding a prefetch queue trick for the pentiums is a pretty difficult task. The prefetch queue hasn’t why to store exactly the next instrucions, nothing to be with the old times...

Look at fs:[0] too... not SEH has been set and there're aren't more anti-debug tricks (make sure debugging from the last exception). Debugging more you'd reach a point like this:

```
01056AF9    9D                        POPFD ; still in the protector
01056AFA    5F                        POP EDI
01056AFB    59                        POP ECX
01056AFC    C3                        RETN ; far jump to 004072DC
...
04072DC   -FF25 1C436200             JMP DWORD PTR DS:[62431C] ; jumps to 01041C64
...
01041C64    55                        PUSH EBP ; we're in the protector
01041C65    8BEC                      MOV EBP,ESP ;
01041C67    8B45 08                   MOV EAX,DWORD PTR SS:[EBP+8] ; we move zeroe!!
01041C6E    813D A47A0401 0000>       CMP DWORD PTR DS:[1047AA4],400000  ; ASCII "MZP"
01041C78     75 07                    JNZ SHORT 01041C81 ; not taken
01041C7A     A1 A47A0401              MOV EAX,DWORD PTR DS:[1047AA4] ; [1047AA4] = 400000
01041C7F     EB 06                    JMP SHORT 01041C87
01041C87     5D                       POP EBP
01041C88     C2 0400                  RETN 4 ; jump to 004073B1
```

So this was simply a trick to confuse us, the protector has simply jumped for a moment to our code, gone back and checked some values. Let's do the "ret":

```
...7
004073B1   |. A3 68066200             MOV DWORD PTR DS:[620668],EAX      ;  target.00400000
004073B6   |. A1 68066200             MOV EAX,DWORD PTR DS:[620668]
004073BB   |. A3 D0E06000             MOV DWORD PTR DS:[60E0D0],EAX
004073C0   |. 33C0                    XOR EAX,EAX
004073C2   |. A3 D4E06000             MOV DWORD PTR DS:[60E0D4],EAX
004073C7   |. 33C0                    XOR EAX,EAX
004073C9   |. A3 D8E06000             MOV DWORD PTR DS:[60E0D8],EAX
004073CE   |. E8 C1FFFFFF             CALL target.00407394
004073D3   |. BA CCE06000             MOV EDX, target.0060E0CC
004073D8   |. 8BC3                    MOV EAX,EBX
004073DA   |. E8 25D6FFFF             CALL target.00404A04
004073DF   |. 5B                      POP EBX
004073E0   \. C3                      RETN
```

Now, it's a crucial momoment for us: We have to decide where's gonna be the entry point. For that, we need to debug into - yes, again - those calls CALL target.00407394 and CALL target.00404A04 to be sure we're not in another trick. Also, feeling a bit curious can help: If the entry point is 004073B1, why the **** is this in the middle of a procedure?. Let's see what's above 004073B1:

```
004073A0    /$ 53               PUSH EBX
004073A1    |. 8BD8             MOV EBX,EAX
004073A3    |. 33C0             XOR EAX,EAX
004073A5    |. A3 C4E06000      MOV DWORD PTR DS:[60E0C4],EAX
004073AA    |. 6A 00            PUSH 0
004073AC    |. E8 2BFFFFFF      CALL target.004072DC
```

Now two questions: Is 004073A0 called from somwhere else?. Nope (if so i hardly believe this was the EP). What does CALL target.004072DC?. Let's follow it:

```
004072DC     $-FF25 1C436200    JMP DWORD PTR DS:[62431C]  ; [62431C] = 01040C64


01041C64     55                 PUSH EBP
01041C65     8BEC               MOV EBP,ESP
01041C67     8B45 08            MOV EAX,DWORD PTR SS:[EBP+8]
01041C6A     85C0               TEST EAX,EAX
01041C6C     75 13              JNZ SHORT 01041C81
01041C6E     813D A47A0401 0000 CMP DWORD PTR DS:[1047AA4],400000     ; ASCII "MZP"
01041C78     75 07              JNZ SHORT 01041C81
01041C7A     A1 A47A0401        MOV EAX,DWORD PTR DS:[1047AA4]
01041C7F     EB 06              JMP SHORT 01041C87
01041C81     50                 PUSH EAX
01041C82     E8 3135FFFF        CALL 010351B8       ; JMP to kernel32.GetModuleHandleA
01041C87     5D                 POP EBP
01041C88     C2 0400            RETN 4
```

Get's the image base of the target calling GetModuleHandleA. Therefore, our entry point is 004073A0 but we also have to change CALL target.004072DC by a CALL GetModuleHandleA. This is a pretty good trick...

By debugging into the target you can ensure you don't jump back to the protector and therefore make sure you are into the protected app. Also you should see some calls to known APIs, as we commented. Likely, all faked API calls are loaded by the protector (using GetProcAddress or by other means).

X. PREPARING FOR DUMPING

First of all, let's compare Notepad with the protected Notepad to get some info:

Notepad:

```
Entry Point = 6AE0, Size Of Image = 13000, image base = 1000000, size of code = 6E00,
size of initialized data = 9400

name virtual size virtual offset raw size raw offset characteristics

.txt 6D72 1000 6E00 400 60000020
.data 1BA8 8000 600 7200 C0000040
.rsrc 8D10 A000 8E00 7800 40000040
```

Asprotected Notepad:

```
Entry Point = 1000, Size Of Image = 2C000, image base = 1000000, size of code = 6E00
size of initialized data = 9400

name virtual size virtual offset raw size raw offset characteristics

7000 1000 3C00 400 C0000040
2000 8000 200 4000 C0000040
.rsrc 9000 A000 1000 4200 C0000040
18000 13000 17E00 5200 C0000040
1000 2B000 0 1D000 C0000040
```

Now, some observations:

- Asprotect has added two new sections at the end and deleted the names of them all (excluding the .rsrc section).
- All virtual sizes have been "rounded up" (not really, cos this size has to be a multiple of the memory page) 3. All virtual offsets have been preserved (this is compelling if the target hasn't .reloc)
- The raw sizes and offsets have been modified cos the program has been compressed (the raw sizes are smaller).
- The characteristics have been set to C0000040 (there's no need to touch them at least you use softice)

Let's have a look at the directory table:

```
  NOTEPAD  PROTECTED
RVA Size RVA Size
-----------------------------------------------------------------------------
Export Table 0 0 0 0
Import Table 6D20 C8 13A38 224
Resource A000 8D10 A000 8D10
Exception 0 0 0 0
Security 0 0 0 0
Relocations 0 0 139C4 8
Debug Datas 1340 1C 1340 1C
Description 0 0 0 0
Global Ptr 0 0 0 0
Tls Table 0 0 0 0
Load Config 0 0 0 0
Bound Import 258 D0 0 0
Import Address Table 1000 324 0 0
```

Time for observations...

- The resources are "ok", also the debug datas.
- The IAT and "bound import" have been set to zero, the import table has been moved inside one of the sections added by Asprotect. This way, Asprotect will use its own imports at startup (GetModuleHandle and GetProcAddress) and will reconstruct the ones of the protected app.

So now we're able to deduce all we have to do after having dumped the asprotected target:

- Change the EP to the original one, we know it cos it was got into the first part of this tutorial and restore the original instructions at it (we'll also need to call GetModuleHandle, etc...)
- Change the raw sizes to the virtual sizes and the raw offets to the virtual offsets/sizes respectively. [BTW: Win2k requests the PE header to be perfect, don't think you've done it if it works in Win9x]
- Reconstruct the imports (the most difficult by far!)
- Reallocate the resources (only to be able to manipulate them with a res-editor, read below)

Observe that there's no need to remove the last two sections. Indeed, the "datas" of the resources are in one of those... Needless to say, is much more ellegant to restore the resouces and then remove the two sections added by our friend Asprotect.

BTW [small info, how to dump]:

I guess you already know how to do it but, just in case. To dump the protected app you have to set an infinite loop at the EP (you know where this is from the previous sections), run it, go to procdump and dump. Don't forget to overwritte the infinite loop with the original instructions at the (restored) Entry Point. Sometimes, the target checks the values of the registers at the entry point (the anti-crack pass it some given values), take care with it.

XI. RECONSTRUCTING THE IMPORTS

BTW: You need background on the imports for this section. There's another tutor on the imports (mine too) in this same edition of CodeBreakers, you have there the explainations you need. I also like, available at www.anticrack.de, "PE Files Import Table Rebuilding" by TiTi/BLiZZARD

A. How to protect your ImportsTable

Say... there're several levels of protection that are currently used (as far as i know). Let's see it:

1) Removing the OriginalFirstThunk: Let's have a glance at this example, the `IMPORT_IMAGE_DESCRIPTORs` start at 0000C1E8h:

```
OriginalFirstThunk: removed (set to zeroe).
Timestamp, ForwarderChain: not used
Name of dll: BA C2 00 00 (ok)
FirstThunk: C5 C2 00 00 (ok)
```

The IAT, starting after the null `IMPORT_IMAGE_DESCRIPTOR`, has the addresses of the APIs imported by the target (the first being 77E7897F and the last 77F19FE6).

```
0000C1E8h : 00 00 00 00 00 00 00 00 00 00 00 00 BA C2 00 00  ----
0000C1F8h : 38 C2 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ----
0000C208h : C5 C2 00 00 44 C2 00 00 00 00 00 00 00 00 00 00  --------
0000C218h : 00 00 00 00 D2 C2 00 00 54 C2 00 00 00 00 00 00  --------
0000C228h : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000C238h : 7F 89 E7 77 4C BC E8 77 00 00 00 00 E6 9F F1 77  ------------
0000C248h : 1A 38 F1 77 10 40 F1 77 00 00 00 00 4F 1E D8 77  ------------
0000C258h : 00 00 00 00 00 00 4D 65 73 73 61 67 65 42 6F 78  MessageBox
0000C268h : 41 00 00 00 77 73 70 72 69 6E 74 66 41 00 00 00  AwsprintfA
0000C278h : 45 78 69 74 50 72 6F 63 65 73 73 00 00 00 4C 6F  ExitProcessLo
0000C288h : 61 64 4C 69 62 72 61 72 79 41 00 00 00 00 47 65  adLibraryAGe
0000C298h : 74 50 72 6F 63 41 64 64 72 65 73 73 00 00 00 00  tProcAddress
0000C2A8h : 47 65 74 4F 70 65 6E 46 69 6C 65 4E 61 6D 65 41  GetOpenFileNameA
0000C2B8h : 00 00 55 53 45 52 33 32 2E 64 6C 6C 00 4B 45 52  USER32.dllKER
```

This is "easy" to fix, take the addresses of the IAT and look for the APIs corresponding to each one. Now, substitute the addresses by the RVA pointing to the name of the API. The OriginalFirstThunk can be set to the FirstThunk. You have a complete description of how to do it at "PE Files Import Table Rebuilding".

2) Removing the names of the APIs:

```
0000C1E8h : 00 00 00 00 00 00 00 00 00 00 00 00 BA C2 00 00  ----
0000C1F8h : 38 C2 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ----
0000C208h : C5 C2 00 00 44 C2 00 00 00 00 00 00 00 00 00 00  --------
0000C218h : 00 00 00 00 D2 C2 00 00 54 C2 00 00 00 00 00 00  --------
0000C228h : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000C238h : 7F 89 E7 77 4C BC E8 77 00 00 00 00 E6 9F F1 77  ------------
0000C248h : 1A 38 F1 77 10 40 F1 77 00 00 00 00 4F 1E D8 77  ------------
0000C258h : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ----------
0000C268h : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ----------------
```

This is essentially the same case than in 2.2.1. You only have to write the strings where you have some room and now you are at case 2.2.1 (take an address, check the API corresponding to it, add the string, overwrite the address with the RVA, next address...)

3) Diverting the calls to the protector: Have a look at the IAT of Notepad (protected with our friend Asprotect)

```
01001050  F9 89 C5 77 F1 E7 C5 77 AC 24 C4 77 D7 40 C4 77
01001060  1D 53 C4 77 CE 48 C4 77 89 28 C4 77 85 3B C4 77
01001070  FF 1E C4 77 6D DF C4 77 01 16 C5 77 A3 16 C5 77
01001080  F6 5E C4 77 B0 1B C4 77 00 00 00 00 78 04 85 00
01001090  8C 04 85 00 98 04 85 00 54 03 85 00 AC 04 85 00
010010A0  C0 04 85 00 CC 04 85 00 FC 96 85 00 D8 04 85 00
010010B0  EC 04 85 00 F8 04 85 00 1C 05 85 00 30 05 85 00
010010C0  40 05 85 00 50 05 85 00 64 05 85 00 D8 05 E6 77
```

As you see, some of the addresses have been substituted by offsets inside Asprotect. Let's examine one of them, 00850564:

```
00850564     55                 PUSH EBP
00850565     8BEC               MOV EBP,ESP
00850567     83EC 18            SUB ESP,18
0085056A     64:A1 18000000     MOV EAX,DWORD PTR FS:[18]
00850570    -E9 200A6077        JMP kernel32.77E50F95
```

This seem to be the entry of a procedure but that jump, JMP kernel32.77E50F95, Where does it go?. Just go to the exports of kernel32 and you will find the answer:

```
API: Name=GetLocalTime, address = 77E50F89
API: Name=SizeOfResource, address = 77E5105F
```

Therefore, we're jumping some point in the middle of GetLocalTime. Now, if we examine the first instructions of this API:

```
77E50F89       55                 PUSH EBP
77E50F8A     . 8BEC               MOV EBP,ESP
77E50F8C     . 83EC 18            SUB ESP,18
77E50F8F     . 64:A1 18000000     MOV EAX,DWORD PTR FS:[18]
77E50F95     . 8B40 30            MOV EAX,DWORD PTR DS:[EAX+30]
```

Everthing is now clear, the protector has emulated the first instructions of the API and then jumped to the next one. However, the protection of the imports in the demo version is extremely lame:

(Snapshot of the `IMAGE_IMPORT_DESCRIPTORs`)

```
01006D20  88 70 00 00 FF FF FF FF FF FF FF FF A4 71 00 00
01006D30  A0 12 00 00 3C 6F 00 00 FF FF FF FF FF FF FF FF
01006D40  F2 71 00 00 54 11 00 00 78 70 00 00 FF FF FF FF
```

(snapshot of the RVAs given by the OriginalFirstThunk)

```
01006DE0                          9A 73 00 00 AE 73 00 00
01006DF0  BC 73 00 00 CC 73 00 00 DC 73 00 00 F0 73 00 00
01006E00  88 73 00 00 00 00 00 00 40 72 00 00 00 00 00 00
```

(snapshot of the names)

```
01007100                                    08 00 47 65               .Ge
01007110  74 46 69 6C 65 54 69 74 6C 65 57 00 04 00 43 6F  tFileTitleW..Co
01007120  6D 6D 44 6C 67 45 78 74 65 6E 64 65 64 45 72 72  mmDlgExtendedErr
01007130  6F 72                                            or
```

As you see, everything is totally intact. Therefore, we only have to take the RVAs to the API names pointed by the OriginalFirstThunk and paste it over the IAT... It's amazing to find this mistake into the demo version.

4) Adding polymorphism: I've already seen this into one protector... the idea is to "mutate" the emulated instructions of the API so it's much more difficult to recover where are we jumping. Funny enough, the protector where i first saw it forgets to remove the hints from the imports table and this makes the effort useless in many cases. Needless to say, for breaking 2.2.4 you need to be able to code an emulator so you can recover the offset where you're jumping (takes too long by hand). This can be terribly difficult, right?.

5) Miscellaneous tricks.: Sometimes, when you think you've successfully reconstructed the imports you load the app and it crashes. Then, after some headaches, you decide to trace into the target and you end up discovering one of this tricks:

6) Total substitution of an API: This is only possible with some APIs, for example GetModuleHandleA and GetVersion, that return a predictable value. Let's see how to do it with GetModuleHandleA:

The protector looks for the following:

```
push 0
call GetModuleHandleA
mov dword ptr [mod_handle], eax
```

And substitutes it by something like:

```
nop ; "do nothing code"
... ;
nop ;
mov eax, dword ptr [009F0A88] ; [009F0A88] = value returned by GetModuleHandleA
mov dword ptr [mod_handle], eax ;
```

Obviously, the protector will call GetModuleHandleA at startup and will store at 009F0A88 the return (so we can hook all calls done by the protector to know what are the procedures it wants to fake).

7) Kernel32.dll is not writtable: If you recall, the first instructions of the APIs where emulated, right?. For some APIs is possible to use this simple fact:

```
The instructions emulating the API can ge modified (they are inside the protector)
The APIs inside kernel32, or any other DLL, can’t be overwritten.
```

A real-life example: This is a procedure that get’s called some moment into the target, it should return the same that GetVersion but let’s see what happens...

```
PUSH ECX ; entry
PUSH EDX ;
SUB ESP,94 ; allocate local storage
CMP [43D1FC],-1 ; [43D1FC] = to 2 if not dumped
JNZ _compare ; jumps if not dumped
MOV [ESP],94 ; esp points to kernel32
MOV EAX,ESP ;
PUSH EAX ;
CALL [_GetVersionEx] ; call GetVersionEx
MOV EAX,[ESP+10] ; overwrittes kernel32
MOV [43D1FC],EAX
MOV EAX,[ESP+4]
MOV [461228],EAX
_compare: CMP [43D1FC],2 ; test version
JNZ _bye
MOV EAX,[461228] ; real return
JMP _restore ; bye!
_bye: XOR EAX,EAX ;
_restore: ADD ESP,94 ; retore esp
POP EDX ; restore registers
POP ECX
RETN
```

As you see, if the protector is still there we’ll have [43D1FC] = -1 and so the procedure simply returns. Otherwise, the behaviour can be very unpredictable: when debugging you can simply overcome the kernel overwritting (depending on several factors: your debugger configuration, if you’re in ring0 or not,...), if you do so then you correctly call GetVersionEx and correctly bypass the `_compare`. Later, when you run it without your debugger it crashes.

8) Moving away a whole procedure: This is also a hard trick: You can simply take a whole procedure and ”move” it to the protectors code, for example if you see you have:

```
mov eax, []
call 401700
...
mov eax, []
call 401700
...

410700: push ebp
mov ebp,esp
...
push eax
call LoadResourceA
...
pop ebp
ret
```

Then you move the procedure to other offset inside your code and update the calls. This can’t be done with every procedure (do to the need of relocating some instructions).

B. Tips for finding the tricks

- Use the debugger’s tracer to trace into, you can add the condition ”EIP is inside the protectors code”. When you see a crash, or something that you think is strange, trace back to see what’s happened (in olly: Options, debugging options, trace, log commands and set it to log enough number of them).
- Trace into for a while from the entry point onwards to see what are the values of the different [XXXXXXXXh], some of them will surely be set by the protector itself. Substitute them by the correspondant call to the API, never by the hardcoded value cos this will crash in other Wins, for example:

  `mov eax, [XXXXXXXXh] ; [XXXXXXXXh] = ImageBase`

  has to be swapped by ”push 0; call GetModuleHandleA”.
- compare the not-dumped program with the dumped one, check the returns of the procedures to see if they math (of course, some procedures allocating memory and so on will return different values).

XII. THE DEATH OF ALL TRICKS

Let's think for a moment about how the protector builds the imports: Somewhere, it has encrypted the API names and the offsets where each of the APIs' addresses has to be stamped. So, one by one, it will take this API names and will do something like:

```
push offset ApiName ; ApiName db 'ApiName',0
push handleToDll ; handle to, f.e., kernel32.dll
call GetProcAddress ; get the address
push eax ;
call HideApiCall ;  divert the call to the protectors code (or other trick)
mov [IAT.ApiOffest], eax ;  set the value of the IAT to the diverted jump
...
```

Of course, all the ImportsTable has been deleted (hints included!, Pavol) and so we only have a lot of diverted calls to some mutated entries at the IAT... we need a new approach.

The idea is very simple:

1) Hook GetProcAddress
2) Trace until you see where is stored the address
3) If it uses some "masking" process, HideApiCall above, nop it if you can (this way, you will have a not-dumped version with all the imports completely unprotected).

Now, with the list of API names and the offsets where they have to be stamped its completely trivial to reconstruct the ImportsTable.

Let's outline how to do it with NOTEPAD protected with Asprotect: We hook GetProcAddress, overcome all anti-debug and voila!

```
0006FDDC   008425A1  /CALL to GetProcAddress from 0084259C
0006FDE0   76360000  |hModule = 76360000 (comdlg32)
0006FDE4   0006FE2B  \ProcNameOrOrdinal = "PageSetupDlgW"
```

Now, debug until we return from the call to GetProcAddress, you're here:

```
008425A1     8945 F8           MOV DWORD PTR SS:[EBP-8],EAX    ; comdlg32.PageSetupDlgW
```

This wasn't what you look for, cos [EBP-8] is on the stack and it has to be stored into the IAT of the target:

```
008425A4     837D F8 00        CMP DWORD PTR SS:[EBP-8],0 ; API found?
008425A8     0F85 DA000000     JNZ 00842688 ; yes, go to store it
... ; debug into
0084299D     8902              MOV DWORD PTR DS:[EDX],EAX
                               ; eax = comdlg32.PageSetupDlgW
```

Bingo!, edx = 010012A0 (the image base of notepad is 01000000). Proceeding in this way we have:

```
offset     api
--------        ------------
010012A0 PageSetupDlgW
010012A4 FindTextW
010012A8 PrintDlgExW
...
```

As you see, and expected, they all are consecutive offsets (cos is the part of the IAT corresponding to one single DLL). Later you will see the rest of the APIs on the screen so you only need to be patient.

I wanna show you a bit more, when loading the APIs of kernel32:

```
0006FDDC   008425A1  /CALL to GetProcAddress from 0084259C ; recover MapViewOfFile
0006FDE0   77E40000  |hModule = 77E40000 (kernel32)
0006FDE4   0006FE2B  \ProcNameOrOrdinal = "MapViewOfFile"
```

Now, debug into...

```
008425A1     8945 F8            MOV DWORD PTR SS:[EBP-8],EAX  ; kernel32.MapViewOfFile
008425A4     837D F8 00         CMP DWORD PTR SS:[EBP-8],0       ;
008425A8     0F85 DA000000      JNZ 00842688       ; jump is taken
...       ; a loop, some rets,...

00842921     E8 7EFEFFFF        CALL 008427A4       ; YOU ARE HERE!
00842926     8B17               MOV EDX,DWORD PTR DS:[EDI]
00842928     8902               MOV DWORD PTR DS:[EDX],EAX
```

You can see, eax = address of MapViewOfFile inmediately before to do the call. If you step OVER the call you can see that now eax = 00850354, this is an address inside Asprotect. Indeed, this is the address that will be set into the IAT instead of the good one at kernel32. Now, go to 00850354:

```
00850354     55                 PUSH EBP
00850355     8BEC               MOV EBP,ESP
00850357     6A 00              PUSH 0
00850359     FF75 18            PUSH DWORD PTR SS:[EBP+18]
0085035C     FF75 14            PUSH DWORD PTR SS:[EBP+14]
0085035F     FF75 10            PUSH DWORD PTR SS:[EBP+10]
00850362     FF75 0C            PUSH DWORD PTR SS:[EBP+C]
00850365     FF75 08            PUSH DWORD PTR SS:[EBP+8]
00850368     68 8A4DE577        PUSH 77E54D8A
0085036D     C3                 RETN
```

Isn't it cute? 008427A4 is the procedure to hide the API entry. At this point, we have this nice shortcut: NOP the call, let the program to load. All the addresses at the IAT are the right ones. Now, you can use on of those import reconstructors you like so much.

So we proceed as shown and obtain all datas we need:

```
SHELL32:
DragFinish at 01001154
DragQueryFile at 01001158
...
```

Don't forget, that each DLL has to be finished with a null 32-bit pointer. At some moment, you can observe some discontinuity, for example:

```
CloseHandle at 01002000
GetLocaleInfoA at 01002008 (eight bytes after CloseHandle)
```

This probably means that the protector is gonna compute the address to place at 01002004 by other means and will fill it later. Set a bpx "on Write" at that address and proceed as usual (find a lower bound to debug from, etc...).

Real-life advise: When dealing with real-life targets one has to add up to a few hundreds of APIs, the next two tips will help:

- You can use SnippetCreator to avoid typing all the names of the APIs. Indeed, if you see that most of the APIs are stored in alphabetical order (this is pretty likely) you can go to thank (your) God cos SnippetCreator always sets them into this the very same order. Therefore, you have to do the following:
  Use SnippetCreator to construct the imports table, regardless of the first thunk and IAT Change the RVAs to the 4 or 5 APIs that aren't into their right positions (you won't have many more) You know the value of the FirstThunk for each DLL (in the example, 010012A0 - `IMAGE_BASE` for comdlg32), so you can go there and paste all the RVAs to the API names. Correct the `IMPORT_IMAGE_DESCRIPTORs`
  Job's done.
- SnippetCreator has a bug, it crashes if you try to add many APIs from two different DLLs (try yourself). To solve it, an easy (but not very ellegant) way is:
  1) Take the target, add all the APIs from the first DLL
  2) Set the ImportsTable and IAT to Zeroe at the directory table (PE header)
  3) Use ImpFake32k, coded by Pegasus, to add the "minimum" imports table to your app.
  4) Repeat the steps to add now all APIs from the second DLL. etc...

  At the end, you have to carefully glue all the pieces of the IAT and correct the ImportsTable, but it doesn't take much time.

Finally, a simple note: you're probably thinking this method is really long and time-consuming, wrong. Take InternetExplorer and count the number of imports it has, its ¡ 100. Even if you think of a target having 500 imports this takes a short time: 500*(30 seconds for clicking on each one) = 4.16 hours... and this is really pesimistic. Coding a little program that let's you to click on the Api name, as Snippet does, and places it in the order you choose (and not in alpha order) is an almost trivial modification of Snippet Creator and it should be easy (i don't know if there's a tool doing this, plz drop me a line if you know).

XIII. IMPROVEMENTS FOR THE PROTECTION

I know, you can try to call GetProcAddress + 8, or simply try to make use of ntdll.LdrGetProcedureAddress or even try to use an undocumented syscall (assuming some risks). But if i can debug your code... nothing will stop me. The matter is, that when the protector looks for the APIs - if done by name - i clearly see on the screen lots of API names pointed by my registers, you don't need too much imagination to guess what's up. Importing by ordinals is less stable and can lead to some "crash", my sugesttion is: try to import by ordinal and if it fails then try to do it by name but NEVER use GetProcessAddress or any other api: code yourself an algorithm to look for them.

XIV. FINAL NOTE

We saw above that there's an easy way of letting to construct non-protected imports so we could simply dump and use an imports reconstructor, f.e. revirgin to name one. So, Why to look for this new method?. Well, there's a strong reason for this: If you let the protector to run till completion then you don't really know what you're gonna have to face, defeating to hook GetProcAddress requires a lot of recoding and its by far more difficult.

XV. RESOURCES

Well, resources are easy to protect: moving them away. This works cos they don't need a .reloc section for its correct "moving". Protectors, use to take profit of this fact and simply take all resources (not the headers but, say, the drawings of the icons and stuff) and place them in other section INSIDE the protector. This way, removing the protector leads to loosing all resources.

If you look at the .rsrc sections you will see that the structure, i.e. the "tree", of the resources has been preserved but their datas have been moved to one of the new inserted sections. There you will find your resources but there's yet another problem... their encrypted...

Let's have a look at the resources as seen by Procdump, we take as example Notepad and examine the menu:

Asprotect:

```
ResDir (MENU) Entries:01 (Named:00, ID:01) TimeDate:00000000 Vers:4.00
    ResDir (1) Entries:01 (Named:00, ID:01) TimeDate:00000000 Vers:4.00
        ID: 00000C0A  DataEntryOffs: 00000440
        DataRVA: 10750  DataSize: 00400  CodePage: 4E4
```

Notepad:

```
ResDir (MENU) Entries:01 (Named:00, ID:01) TimeDate:00000000 Vers:4.00
    ResDir (1) Entries:01 (Named:00, ID:01) TimeDate:00000000 Vers:4.00
        ID: 00000C0A  DataEntryOffs: 00000440
        DataRVA: 10750  DataSize: 00400  CodePage: 4E4
```

As you can see the menu is fine, but if now examine the icons (i only paste the first) we see the following:

Asprotected:

```
 ResDir (ICON) Entries:09 (Named:00, ID:09) TimeDate:00000000 Vers:4.00
    ResDir (1) Entries:01 (Named:00, ID:01) TimeDate:00000000 Vers:4.00
        ID: 00000C0A  DataEntryOffs: 000003B0
        DataRVA: 19E80  DataSize: 00668  CodePage: 4E4
```

Notepad:

```
 ResDir (ICON) Entries:09 (Named:00, ID:09) TimeDate:00000000 Vers:4.00
    ResDir (1) Entries:01 (Named:00, ID:01) TimeDate:00000000 Vers:4.00
        ID: 00000C0A  DataEntryOffs: 000003B0
        DataRVA: 0A568  DataSize: 00668  CodePage: 4E4
```

Asprotect has changed the RVA!. Hmmm... let's load the program and let's go to that RVA. If you do so, you'll easily find out that the resources are unpacked in memory and are identical to the original ones. So we only need to take them as they are in memory, we don't even need to copy them to the old .rsrc section. You can dump the datas for the different resources to examine which ones have been moved and which haven't. The ellegant way of dealing with the resources is to move them back, this way you will be able to manipulate them all with your favourite resource editor and to enable all those buttons you've found not to work.

For example, if we dump from memory for the icon above we find this:

```
(notepad, just the first lines of the data for this icon)

0100A568  28 00 00 00 30 00 00 00 60 00 00 00 01 00 04 00
0100A578  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0100A588  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0100A598  00 80 00 00 00 80 80 00 80 00 00 00 80 00 80 00
0100A5A8  80 80 00 00 C0 C0 C0 00 80 80 80 00 00 00 FF 00
0100A5B8  00 FF 00 00 00 FF FF 00 FF 00 00 00 FF 00 FF 00
0100A5C8  FF FF 00 00 FF FF FF 00 11 11 11 11 11 11 11 11
0100A5D8  11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
0100A5E8  11 11 11 11 11 11 11 11 10 00 00 00 00 00 00 00
0100A5F8  00 00 00 00 00 00 00 11 11 11 11 11 11 11 11 11
0100A608  88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 01
```

(asprotect, just the first lines of the data for this icon. MIND THE RVA!)

```
0100A568  28 00 00 00 30 00 00 00 60 00 00 00 01 00 04 00
0100A578  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0100A588  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0100A598  00 80 00 00 00 80 80 00 80 00 00 00 80 00 80 00
0100A5A8  80 80 00 00 C0 C0 C0 00 80 80 80 00 00 00 FF 00
0100A5B8  00 FF 00 00 00 FF FF 00 FF 00 00 00 FF 00 FF 00
0100A5C8  FF FF 00 00 FF FF FF 00 11 11 11 11 11 11 11 11
0100A5D8  11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
0100A5E8  11 11 11 11 11 11 11 11 10 00 00 00 00 00 00 00
0100A5F8  00 00 00 00 00 00 00 11 11 11 11 11 11 11 11 11
0100A608  88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 01
0100A618  11 11 11 11 11 11 11 18 FF FF FF FF FF FF FF FF
```

As you see, the data HAS SIMPLY BEEN MOVED. So... move it back :-) If you compare the not-protected notepad with the protected one you can see the exact moment when the resources are set up in memory.

BTW1: Asprotect keeps the string and accelerator resources as well, it just changes the icons

BTW2: in olly, ALT+M to see the memory map, select the resource section, press-on the right button of the mouse and choose ”view all resources” (you can also see the string resources and check that they’re still the same). Indeed, if yo do so you will see the following (excluding the language info):

```
Resources of module pnotepad, item 2
 Address=01013C5C
 Type=18
 Name=1
 Language=0C0A Espaol (alfabetizacin internacional)
 Size=0000029E (670.)
 Information=(Resource crosses section limits) = THE RESOURCES DATAS HAVE BEEN MOVED!
```

BTW3: If you don’t need to manipulate the resources with a res-editor then you can simply not to remove the sections added by the protector, since the resources will be decrypted in memory it will work perfectly (but the file will be bigger and will have some ugly empty sections, the choice is up to you).

XVI. CONCLUSIONS

I've (intentionally) omitted the following:

- When the application has been succesfully dumped the problem is just a standard reverse-me. Go to the resources to correct them - the same you do with the ones of REA - and also look at the strings references, hook the access to the registry. I guess you don't need to be explained this stuff, right?.
- Asprotect allocates different buffers to unpack or unencrypt different pieces of itself, you only have to hook VirtualAlloc,VirtualFree,GlobalAlloc,... to see how it works. The best is to look at the buffers and observe where and how they're filled. For example, you can see the exact moment when the resources are set up.
- Time-limits and other limitations: You don't have to worry too much about them if you know how to unpack and dump, but it's interesting to see how it works. Also hooking the APIs will give you interesting information about how it computes the Hardware-ID and so on.
- Different levels of protection: I don't know if it's a good idea to let users to choose if they want to add anti-debug or not, they MUST add it. If you give too much choice to them you're gonna end up having targets with all possible levels of difficulty, this will make the cracker's job a "learning" process that will help him to get it. Anti-debug, resources protection and checksums should always be there.
-
  ```
  0084246C    3B7B 04         CMP EDI,DWORD PTR DS:[EBX+4]
  0084246F    74 0A           JE SHORT 0084247B
  00842471    68 3C258400     PUSH 84253C     ; ASCII: "1", HEX: 31h, DCh 0Ah
  ```
  This also corresponds to different validations, defeated as usual. This the very same comment holds for HEX: 32h, DCh 0Ah and the rest of them.

I'd like to have included more information into this tutorial, but it's already long enough... [Zero will hate me for having to read so much].

XVII. FINAL WORDS

Well, my friend. It's time to say goodbye. I hope you enjoyed this tutor at least a little bit. Of course, any kind of [legal] comment/critic/question... you know where to find me...

Nice reversing, Havok.

Acknowledments:

To all REA staff, for obvious reasons
To Merlin, who provoked me to crack this one :D (some way... this is your fault, he,he)
To A.Solodovnikov, for all this sleepless nights
To P. Cerven , for the ones to come (i dunno if you should have written that book, man).

That's all.