ASPROTECT 1.22 - 1.32 beta 21

UnAsProtecting AsProtect

{*Written by* AndreaGeddon}
{andreageddon@hotmail.com}
[www.reteam.org]

INTRO

Target program: WebPics 1.8
Url: http://www.express-soft.com/
Crypter: AsProtect 1.22 / 1.32 beta 21
Someone reported this program in a mailing list, i was interested in it because it is crypted with asprotect, we all know that this crypter is a bad bone! Alexey Solodovnikov has done a great job, i think this is a good PE crypter, however it is not unbreakable!
Let's go for it.

TOOLS

OllyDebug, Ida, PeID to recognize the crypter :) nothing else needed, I will do almost all the work by hand. Wondering about Softice? Well olly is a wonderful debugger, it is usermode but it is really powerful, i debug and solve almost everything with it. You can use softice of course, or WinDbg, or, if you are completely crazy, ntsd :)

DECRYPTING & DUMPING

As usual the purpose is to debug the crypter until it reaches the original entry point of the program. At that point the sections will be completely decrypted, so we will dump code and data, and eventually rebuild the PE. First of all let's have a look at the PE. Actual entry point is at 00401000 (in first section, just to trick automatic tracers), we see there a

```
00401000 PUSH webpics.0067C001
00401005 CALL webpics.0040100B
0040100A C3 RETN
0040100B C3 RETN
```

so it just is a jump to loader sections (last two). We check the Import Table and we see that it is not valid, that is, we have all the modules listed but only one api per module is imported, so we know that we will have to rebuild

imports. I knew aspr would have been funny! Time to trace now. The loader is long, so i will write just the main lines with some comment.

0067C001 - Starting of the aspr loader

There is some polimorphic code, that is things like

```
0067C00E CALL webpics.0067C014
0067C013 JMP webpics.0067C072  <- interference byte
0067C015 MOV EBX,-13
```

check always the calls you execute, you should "step into" them to avoid losing tracing control. When we see accesses to

```
0067C022 CMP DWORD PTR SS:[EBP+25],0
0067C026 MOV DWORD PTR SS:[EBP+25],EBX
```

where ebp is 0067C013, these are global data used by the aspr loader. Due to its relocability, the loader can't use fixed addresses, so a loader usually have to use address relative to itself, then calculate the delta offset to add to relative addresses to obtain absolute addresses. Aspr also relocates the dinamyc code it uses.

```
0067C0D4  this loop decrypts 0x750 dwords at the address 0067C160
...       there are several layers of decryption in the loader
0067C14A
```

when the loop ends it will jump to decrypted area and continue execution. The loader is full of crypted routines, this is good for the crypter.

```
0067C181  decrypt of 06C4 dwords at 0067C1EA
...
0067C1E4
```

there are other 2 decrypt layers, i am not pasting them, the code here simply runs loops to decrypt his next code. We trace other loops and we arrive at

```
0067C6A8 MOV EAX,DWORD PTR SS:[ESP+24]
0067C6AC AND EAX,FFFF0000
0067C6B1 ADD EAX,10000
0067C6B6 SUB EAX,10000
0067C6BB CMP WORD PTR DS:[EAX],5A4D   <- MZ
0067C6C0 JNZ SHORT webpics.0067C6B6
```

this code takes an address from the kernel32 and works on it to obtain its imagebase (that is module handle). Where the address come from? [Esp+24]? Yes, simply the kernel before arriving at entry point of the exe runs this code:

```
77E5EB56 PUSH 4
77E5EB58 LEA EAX,DWORD PTR SS:[EBP+8]
77E5EB5B PUSH EAX
77E5EB5C PUSH 9
77E5EB5E PUSH -2
77E5EB60 CALL DWORD PTR DS:[ZwSetInformationThread]
77E5EB66 CALL DWORD PTR SS:[EBP+8]    <- call to exe entry point (main thread)
77E5EB69 PUSH EAX
77E5EB6A CALL kernel32.ExitThread
```

the pe loader of win calls our exe main thread with a call (this is on nt, on 9x you have a jmp [entry point] but the trick works the same because first dword in the stack, when at entry point, is a return address to the kernel32) so on the stack we will have a return address to the kernel, that is an address inside the kernel32 module. Once it has the module handle we can see in the following lines he gets the MZ_Header->e_lfanew (PE offset) pointer and then it accesses the original first thunk array. We arrive here

```
0067C6E5 MOV ESI,DWORD PTR DS:[EBX]  -> ptr to crypted api identifiers
0067C6E7 MOV DWORD PTR SS:[EBP+325],ESI
0067C6ED CALL webpics.0067C6FD    -> enter here to see the calculus
0067C6F2 STOS DWORD PTR ES:[EDI]
0067C6F3 ADD EBX,4
0067C6F6 CMP DWORD PTR DS:[EBX],0
0067C6F9 SHORT webpics.0067C6E5
```

this code simply finds the entry point of some apis used by the loader (GetProcAddress, VirtualAlloc and so on). How are the apis found? There are some crypted identifiers for each api (ones in [ebx]), the aspr begins scanning the original first thunks of imported apis to build from api names their cripted identifiers (dwords), once they are built if they match the crypted identifiers in [ebx] then aspr has found the api he wants to import.

```
0067C46E REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
```

here aspr restores the bytes at 00401000. As we have seen the entry point of the application was 00401000, but that was a code of the aspr loader, that address should be part of the code section of the real program. Infact aspr overwrited it, now it has to restore original code, they are 12 bytes at

0067C476 address. They are crypted of course, we still have to see decription routines. Next we arrive at 0067C48A and we find two calls at VirtualAlloc. The whole code is used to load a dll by hand :). Infact we see this call:

```
0067C4AF PUSH EAX     <- address of first allocated area
0067C4B0 PUSH EBX     <- 67CE1C
0067C4B1 CALL webpics.0067C56B   <- decrypt dll
```

this decrypts the data at 67CE1C and maps it into the memory allocated by the first VirtualAlloc. So now we have a map of a dll (a PE file) in memory. What's gonna happen?

```
0067C4FB REP MOVS DWORD PTR ES:[EDI],DWORD PTR DS:[ESI]
```

there are six of this movement, each one for a section of the dll. ESI is the pointer to the section in the dll raw data (first allocated area), EDI is the pointer to the second allocated area, and will be the area where this dll will work in. If we check the first of these movement we see that esi = B10400 and edi = B31000 (of course on my pc the two allocated areas are B10000 and B30000 respectively). What we notice is that at xx0400 starts the first section of the dll in the raw file, xx1000 instead is the address in memory of the section (FileAlignment = 400, SectionAlignment = 1000), so the PE header (wich is under xx0400) is not mapped! If you were thinking about dumping this dll, well, it will be an hard task :). However we can avoid dumping this dll as a PE, we will see it later. After this works we can see

```
0067C557 CALL DWORD PTR SS:[EBP+39D] ; kernel32.VirtualFree
0067C55D PUSH 00B4B000
0067C562 RETN
```

the first allocated area is released, the aspr no longer needs it. Then we jump to the second allocated area (the dll). Now the tracing moves to this allocated area. Keep in mind that my allocated area address is B30000, so *all addresses i will paste are relative to this address*. We continue and we see that at B4B04D there are two calls at GetProcAddress to get entry for VirtualAlloc and VirtualFree. Then VirtualAlloc is called to alloc a new area, and we get here:

```
00B4B0D6 PUSH EAX    <- address of new allocated area
00B4B0D7 PUSH EBX    <- address of some crypted data (00B4B101 for me)
00B4B0D8 74050000 CALL 00B4B651
```

the call decrypts data and copies it in the new buffer, then we see

00B4B0DF LEA EDI,DWORD PTR SS:[EBP+442A45] <- B4B101 (crypted data)
00B4B0E5 MOV ESI,DWORD PTR SS:[EBP+442975] <- new allocated area (decrypted)
00B4B0EB REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]

so crypted data is overwritten with the decrypted data. The new allocated area is no longer needed so at B4B0FB you see a VirtualFree. The decrypted data is the code that follows this line. Again we continue tracing and we see another VirtualAlloc followed by

00B4B358 PUSH EAX   <- new allocated area
00B4B359 PUSH EBX   <- 00B31000 for me (crypted data)
00B4B35A CALL 00B4B651

again a new buffer is allocated and filled with decrypted data. The decryption function is always the same

int Decrypt(pCrypted, pDecrypted);

the size of decrypted data is returned. The decrypted data is executable code, so the aspr is gonna relocate some calls, you see a cycle at B4B390 that scans the new allocated code for E8 and E9 (call or jump opcode) and relocate them. Once relocation is complete we arrive to

00B4B3C7 REP MOVS DWORD PTR ES:[EDI],DWORD PTR DS:[ESI]

where esi = new allocated area and edi = crypted data (B31000). So as before the crypted is overwritten with decrypted and relocated data. VirtualFree releases the area used to decrypt and relocate the code. Again, we continue an see another VirtualAlloc followed by the Decrypt routine! This time the crypted area is 200h dwords long and is located at 00B45000. At line B4B3C7 there is the usual REP MOVS which fills B45000 with decrypted data. There are still other two decrypts to do at address B47000, B49000. The following code contains a loop:

00B4B43E
...   handle relocation of the mapped dll
00B4B484 LOOPD B4B43E

this code will search in the .reloc section of this "ghost" dll and will handle all the relocations (absoulte addresses are in the form of 0040xxxx for this ghost PE dll). We continue tracing and get an interesting code:

```
00B4B496 MOV EAX,DWORD PTR DS:[ESI+C]  Import_Descriptor.Name
00B4B499 TEST EAX,EAX
00B4B49B JE 00B4B5AB                   Walk through all imported descriptors
00B4B4A1 ADD EAX,EDX
00B4B4A3 MOV EBX,EAX
00B4B4A5 PUSH EAX
00B4B4A6 CALL DWORD PTR SS:[GetModuleHandle]
00B4B4AC TEST EAX,EAX
00B4B4AE JNZ SHORT 00B4B4B7
...
00B4B4B7 MOV DWORD PTR SS:[EBP+44294D],EAX
00B4B4BD MOV DWORD PTR SS:[EBP+442951],0
00B4B4C7 MOV EDX,DWORD PTR SS:[EBP+4430D8]
Import_Descriptor.OriginalFirstThunk
00B4B4CD MOV EAX,DWORD PTR DS:[ESI]
00B4B4CF TEST EAX,EAX
00B4B4D1 JNZ SHORT 00B4B4D6
00B4B4D3 MOV EAX,DWORD PTR DS:[ESI+10]
Import_Descriptor.FirstThunk
00B4B4D6 ADD EAX,EDX                   Gets current api
...
00B4B4F3 TEST EBX,80000000             Check for import by ordinal
00B4B4F9 JNZ SHORT 00B4B4FF
00B4B4FB ADD EBX,EDX
00B4B4FD INC EBX                       Avoid Import_By_Name.Hint
00B4B4FE INC EBX
00B4B4FF PUSH EBX
00B4B500 AND EBX,7FFFFFFF
00B4B506 PUSH EBX
00B4B507 PUSH DWORD PTR SS:[EBP+44294D]
00B4B50D CALL DWORD PTR SS:[GetProcAddress]
...
00B4B595 MOV DWORD PTR DS:[ESI],EAX    Set Resolved
OriginalFirstThunk
00B4B597 MOV DWORD PTR DS:[ESI+C],EAX  Name and
00B4B59A MOV DWORD PTR DS:[ESI+10],EAX  FirstThunk
00B4B59D ADD ESI,14
00B4B5A0 MOV EDX,DWORD PTR SS:[EBP+4430D8]
00B4B5A6 JMP 00B4B496
```

this code builds the import table for the mapped dll, the import api names/modules are at B47258, the thunks are at B470C4. All this work just to map an auxiliary dll, decrypt it, relocate it, build its import thunks. Now the code starts executing the code of this dll. Going on we will see a call

00B44917 CALL 00B35018 <- initializations
00B4491C CALL 00B33310 <- we enter here

you can avoid stepping in the first call, it makes some initializations like handling tls, working on registry and command line and so on. So we trace the second call. We enter there and again see a lot of code of no interest. You will see various calls to procedures that set seh handlers, then a set of calls to api such GetVersion, GetCurrentProcess, GetCommandLine.
One thing you should know to debug the code is the seh handler trick. That is tracing the second call you encounter a code like this
xor [eax], eax
where eax = 0. Nothing to worry about, this is a call to the seh, to see where this instruction will bring you just watch in the TIB. Now since ollydbg does not allow you to enter a segment identifier for the memory dump (or at least I don't know how to enter it ;-P) you can look at the TEB (TDB if you are on win9x) by watching segment register FS: you will see both segment base (linear) address and size, so you should look at that address. The first dword is the ExceptionList* for the thread you are tracing. Then you can see ExceptionList->Handler (that is [ExceptionList+4]) to find the address of the topmost seh. So, once you step the previous xor you will be transferred to the seh, that uses to do something like:

```
add [context.eip], 2
xor eax, eax
ret
```

this tells the system seh manager to restore the context and continue execution, so you will continue at address context.eip+2. Sometimes you will also see something like this in the exception handler

```
00B42E50 XOR ECX,ECX
00B42E52 MOV DWORD PTR DS:[EAX+4],ECX   ;context.DR0
00B42E55 MOV DWORD PTR DS:[EAX+8],ECX   ;context.DR1
00B42E58 MOV DWORD PTR DS:[EAX+C],ECX   ;context.DR2
00B42E5B MOV DWORD PTR DS:[EAX+10],ECX  ;context.DR3
00B42E5E MOV DWORD PTR DS:[EAX+18],155  ;context.DR7
```

where eax is a pointer to the thread context passed by the system to the exception handler by [esp+0C], do you remember the exception prototype?

int Seh(EXCEPTION_RECORD *Exc, void* Frame, CONTEXT *Context, void* DispatcherContext)

Return value = 0 means "restore context and resume execution", return value = 1 means "continue to the next seh handler in chain". So this code

fills with zeros the debug registers 0 to 3, that is if you had some BPM set this trick will delete them. Infact these DRx are responsible to hold addresses of hardware bpm you set. On DR7 is put the value 0x0155, that is all local breakpoint flags are set and other flags are cleared (but sometimes aspr will put 0 in DR7, this is terrible for debugging!). This interfers with the debugger, so you should avoid to execute these lines if you dont want troubles when tracing this code.
So going on we will find the section decrypt:

```
00B421AA CALL 00B3250C                          ;alloc a buffer
00B421AF MOV ESI,EAX
00B421B1 MOV EAX,DWORD PTR DS:[EBX]      ;rva of section virtual offset
00B421B3 ADD EAX,DWORD PTR SS:[EBP-14]  ;rva of section virtual offset
00B421B6 MOV DWORD PTR SS:[EBP-4],EAX
00B421B9 MOV ECX,DWORD PTR DS:[EBX+4]       ;virtual size of section
00B421BC MOV EDX,ESI                            ;edx = ptr to buffer
00B421BE MOV EAX,DWORD PTR SS:[EBP-4]
00B421C1 CALL 00B41C8C                ;copy decrypted section to buffer
00B421C6 MOV EDI,EAX
00B421C8 CMP EDI,DWORD PTR DS:[EBX+4]
00B421CB JE SHORT 00B421D7
00B421CD PUSH 0B42298
00B421D2 CALL 00B41A3C
00B421D7 CMP BYTE PTR SS:[EBP-5],0
00B421DB JNZ SHORT 00B421FB
00B421DD MOV BYTE PTR SS:[EBP-5],1
00B421E1 PUSH ESI
00B421E2 MOV ESI,DWORD PTR SS:[EBP-4]
00B421E5 ADD ESI,14
00B421E8 PUSH DWORD PTR DS:[ESI]
00B421EA MOV BYTE PTR DS:[ESI],0C3     ;trick to fool automatic tracers
00B421ED CALL ESI                               ;ESI = 00401014
00B421EF POP DWORD PTR DS:[ESI]
00B421F1 POP ESI
00B421F2 MOV EDX,EDI
00B421F4 MOV EAX,ESI
00B421F6 CALL 00B41CB4
00B421FB MOV ECX,EDI
00B421FD MOV EDX,ESI
00B421FF MOV EAX,DWORD PTR SS:[EBP-4]
00B42202 CALL 00B351D0 ;copy decrypted section from buffer to exe image
00B42207 MOV EDX,DWORD PTR DS:[EBX+4]
00B4220A MOV EAX,ESI
00B4220C CALL 00B32524                          ;free buffer
00B42211 ADD EBX,0C
```

```
00B42214 MOV EAX,DWORD PTR DS:[EBX+4]
00B42217 TEST EAX,EAX
00B42219 JA SHORT 00B421AA                    ;go to next section
```

you see there is a trick for automatic debuggers, infact if you trace until the eip goes in first section to find original entry point, you will break at 00B421ED CALL ESI but this is not oep, its just a CALL to a RET. However the interesting thing here is the section decrypt. As you can see its quite easy, if you are interested in the decrypt algorithm just dig into here: 00B421C1 CALL 00B41C8C. So once the sections are decrypted we have the image of the program. Now we have to step a little to arrive at the following point:

```
00B42892 LODS DWORD PTR DS:[ESI]          ;get address (RVA) of current
module iat
00B42893 OR EAX,EAX
00B42895 JE SHORT 00B428E1                ;if zero then iat construction is
complete
00B42897 MOV EDI,EAX
00B42899 ADD EDI,DWORD PTR DS:[B46978] ;add image base to obatin VA
00B4289F MOV DWORD PTR SS:[EBP-8],EDI
00B428A2 MOV EBX,ESI                  ;ebx = ptr to module name (string)
00B428A4 XOR ECX,ECX
00B428A6 DEC ECX
00B428A7 XCHG ESI,EDI
00B428A9 XOR AL,AL
00B428AB REPNE SCAS BYTE PTR ES:[EDI]  ;go to end of string
00B428AD XCHG ESI,EDI
00B428AF LODS BYTE PTR DS:[ESI]      ;load first byte of crypted api (a flag)
00B428B0 CMP AL,0
00B428B3 JE SHORT 00B42892            ;if flag is zero then goto next module
00B428B5 CMP AL,6
00B428B8 JNZ SHORT 00B428C0           ;if flag is 6 then
00B428BA ADD DWORD PTR SS:[EBP-8],4    ;api will not be resolved here,
see emulation
00B428BE JMP SHORT 00B428AF           ;so goto for next api
;here we process the api
;with the stolen byte method (redirection)
00B428C0 PUSH EBX                          ;module name
00B428C1 PUSH ESI                          ;crypted api
00B428C2 PUSH EBX                          ;module name
00B428C3 LEA EBX,DWORD PTR SS:[EBP-8]
00B428C6 PUSH EBX                          ;iat va for api to be resolved
00B428C7 CMP AL,2
00B428CA JE SHORT 00B428D2
```

```
00B428CC MOVZX ECX,BYTE PTR DS:[ESI]   ;get 2nd byte of crypted api (api
name length)
00B428CF INC ECX
00B428D0 JMP SHORT 00B428D7
00B428D2 MOV ECX,4
00B428D7 ADD ESI,ECX
00B428D9 CALL 00B425F0                 ;work is done here
00B428DE POP EBX
00B428DF JMP SHORT 00B428AF
```

this is a cycle where the import table is built. We now have the address of the original IAT of the program (first time you arrive at this cicle the va of the api iat va will be 005EC1E0, that is the base address of the iat) and the crypted data relative to imported api names. How is this data stored? You have this form:

```
1 byte   - NULL          \
n bytes  - Module name    |-> module descriptor
1 byte   - NULL          /
1 byte   - Api flag        \
1 byte   - Api length       |-> api descriptor
n bytes  - Crypted api name /
... array of api descriptors
```

there is the module descriptor and all its api descriptor. As you can imagine the two api (flag and length) bytes were the original HINT field, and the following were the original api name bytes. To see how real api address is gained we have to dig into the call. I will paste only the main lines of the routine:

```
00B42609 LEA EAX,DWORD PTR SS:[EBP-101]   ;pointer to buffer area
(stack)
00B4260F XOR ECX,ECX
00B42611 MOV EDX,100                      ;size of area
00B42616 CALL 00B32794                    ;clear buffer area
...
00B4263B MOV BL,BYTE PTR SS:[EBP-1]
00B4263E MOV ECX,EBX                      ;api string size
00B42640 LEA EAX,DWORD PTR SS:[EBP-101]   ;pointer to buffer area
00B42646 MOV EDX,ESI                  ;edx = api crypted bytes
00B42648 CALL 00B351D0                  ;fill buffer area with crypted bytes
...
00B4264F MOV ECX,0B46D5A                  ;ptr to some parameter
00B42654 MOV EDX,EBX                      ;api string size
00B42656 LEA EAX,DWORD PTR SS:[EBP-101]   ;ptr to buffer area
```

```
00B4265C CALL 00B40E54                    ;decrypt api name
...
00B42661 LEA ESI,DWORD PTR SS:[EBP-101]
00B42667 PUSH ESI                         ;ptr decrypted api
00B42668 MOV EAX,DWORD PTR SS:[EBP+C]
00B4266B PUSH EAX                         ;ptr to module name
00B4266C CALL 00B422C4                    ;get the api entry point
00B42671 CALL 00B42500                    ;get stolen bytes
00B42676 MOV EDX,DWORD PTR DS:[EDI]
00B42678 MOV DWORD PTR DS:[EDX],EAX  ;store redirection address in iat
```

a buffer is created and zeroed, then crypted api bytes are copied in it and decrypted. Once the name of the api is decrypted the aspr gets its address, then the stolen byte method is applied. How does it work? Well in the final executable the situation will be this:

executable	redirection bridge	dll
...	...	...
...	...	...
call [api] --->	api instruction1	api instruction1 (stolen!)
...	api instruction2	api instruction2
...	api instruction3	api instruction3
...	...	...
...	jmp api + n --->	api instruction n
...	...	...

the aspr scans first n instruction at api entry point (n is variable), copies it in the redirection bridge (a buffer in the process space, 0x00C30000 in my pc), then adds a jump to the real api to continue execution. So the import address in the iat will not be the one of the api but the one of the redirection bridge. This is a nice trick, it also prevents api breakpoints: infact you usually set a bpx on the entry point of the api, but such entry point is not executed so the debugger will not break. To avoid this just put the bpx some line after the api ep. Note that not all the api are redirected with stolen bytes, when api flag is 01 the real api address is put in the iat.
When all this cycle will be executed, the iat will be almost complete: some apis are not resolved, do you remember this line?

```
00B428BA ADD DWORD PTR SS:[EBP-8],4    ;api will not be resolved here,
see emulation
```

now let's see how they are built: we arrive at this loop

```
00C38658 LODS BYTE PTR DS:[ESI]        ;get api flag
00C38659 OR AL,AL
00C3865B JE SHORT 00C38679             ;if flag==0 then iat is finished
00C3865D DEC AL
;the flag is an index in an emulation array, so it is decremented
;(array 0 based) and the pointer is calculated
00C3865F SHL EAX,2                     ;multiply index * 4
00C38662 ADD EAX,DWORD PTR SS:[ESP+8]
00C38666 MOV EBX,DWORD PTR DS:[EAX]    ;get routine[index*4]
00C38668 LODS DWORD PTR DS:[ESI]
00C38669 ADD EAX,DWORD PTR SS:[ESP+4]  ;get va of iat address to be
filled
00C3866D MOV DWORD PTR DS:[EAX],EBX    ;store the thunk to emulated
api
00C3866F XOR EAX,EAX
00C38671 MOV DWORD PTR DS:[ESI-4],EAX
00C38674 MOV BYTE PTR DS:[ESI-5],AL
00C38677 JMP SHORT 00C38658
00C38679 RETN 8
```

the routines that emulate apis are stored in the 00B4xx bridge, so if you the [ESP+8] parameter you see *all the addresses of all possible emulations*. You can also list all corresponding index, this helps you if you want to make an unpacker :). Viewing all the redirection routines you can easly figure out what api do they emulate. We will see that you not always can just put the address of the emulated api to rebuild iat, you have to re add these emulation strips in the code.
We continue tracing and we arrive at this code:

```
00B439F3 50          PUSH EAX
00B439F4 A1 0C56B400 MOV EAX,DWORD PTR DS:[B4560C]
00B439F9 8B40 04     MOV EAX,DWORD PTR DS:[EAX+4]
00B439FC FFD0        CALL EAX    ; 005CE524 <- routine inside the
program!
```

Are we at oep??? No. If you trace this call you will see that a routine of the decrypted program is executed, then the execution flow will return to the instruction following the call. So the aspr loader calls some routines in the code before it arrives to the original entry point. How many calls are there? Well we can see in 00B4560C there is a pointer, that is 00b46988, which points to:

00B46988 00 00 00 00 24 E5 5C 00 00 00 00 00 00 00 00 00
00B46998 08 E5 5C 00 00 00 00 00 00 00 00 00 48 E5 5C 00
00B469A8 98 E5 5C 00 38 E5 5C 00 00 00 00 00 00 00 00 00

there are 5 pointers to program routines: 005CE524, 005CE508, 005CE548, 005CE598 and 005CE538. If you continue tracing you will see that all these routines are called in that order, except 005CE548. It seems that these routines can be avoided in the final unpacked exe. We will see it later. Going on you can find problems in the seh trick, the debugger will not run correctly and will end debugged process. To avoid this just locate the seh the debugger is not able to handle and avoid its call. For example:

```
00B42D49 XOR EAX,EAX
00B42D4B PUSH DWORD PTR FS:[EAX]
00B42D4E MOV DWORD PTR FS:[EAX],ESP
00B42D51 XOR DWORD PTR DS:[EAX],EAX   <- seh call
00B42D53 POP DWORD PTR FS:[0] ; 0012FFE0
00B42D5A POP EAX
00B42D5B CMP DWORD PTR DS:[B46D84],0
00B42D62 JE SHORT 00B42D78
```

you can nop the xor (seh call) instruction. Infact the seh trick simply makes a turn-around execution and then continues at the line after the xor. However, now all the work is done, sections are decrypted, iat is built, we are about to go to oep. We arrive at a code like this

```
00B42D81 CMP DWORD PTR DS:[EAX],0
00B42D84 JE SHORT 00B42D88
00B42D86 PUSH DWORD PTR DS:[EAX]
00B42D88 PUSH DWORD PTR SS:[EBP-10]
00B42D8B PUSH DWORD PTR SS:[EBP-14]  ;00C3B460
00B42D8E RETN
```

and the execution moves to the 00C3xxxx memory area. So now we are outside the 00B4xxxx (which is the self-mapped dll). So we can think we are near the oep. The base of this area is 0x00C30000 and its length is 0xC000, so you can translate the addresses i will paste to your address for this memory bridge. Stepping in this bridge we will see some decrypt cycles:

```
00C35617
...       1st decrypt loop
00C35987

00C34B99
...       2nd decrypt loop
```

00C34BF0

00C34C52
... 3rd
00C34CC3

00C34D3D
... 4th
00C34DB7

00C34E11
... 5th
00C34EDB

00C34F41
... 6th
00C34FDC

00C35029
... 7th
00C350D3

00C35153
... 9th
00c35247

00C352E0
... 10th
00C3535D

ten decypt loops, each one decrypts the code immediatly following. After all this decryption you arrive at

00C35395 JMP SHORT 00C35397
00C35397 55 PUSH EBP <- stolen eip bytes (in blue)!
00C35398 8BEC MOV EBP,ESP
00C3539A 83C4 F0 ADD ESP,-10
00C3539D 53 PUSH EBX
00C3539E B8 80E65C00 MOV EAX,5CE680
00C353A3 PUSH 5CED24
00C353A8 RETN

ok, we arrived at the entry point! The last push indicates the oep. It is built at runtime and is contained at memory location 00C353A4. Look at the stolen bytes (the blue ones): they were in the original exe image, so you have to

remember them when you will rebuild the exe. Now you can use any pe dumper and dump all pe image. Well, we have to dump redirection memory bridges too! That is, the one at 00B3xxxx (length 0x01D000) and the one at 00C3xxxx (length 0xC000), infact dumping these bridges will allow us to rebuild IT correctly.
NOTE! I assume after the dump you have FileAlignment = 0x1000, if you dumped and fixed it to 0x400 or if you have FileAlignment != SectionAlignment then all the addresses you will see from now on are different and you will have to recalculate them.

REBUILDING & FIXING

Ok the file is dumped. We have seen that the import table is crypted, part of the imported api are redirected and emulated. There are also the stolen bytes at the entry point. So we have a bit of work to do. First of all we fix the entry point. In the dumped exe we see this:

```
005CED18 0000 ADD BYTE PTR DS:[EAX],AL
005CED1A 0000 ADD BYTE PTR DS:[EAX],AL
005CED1C 0000 ADD BYTE PTR DS:[EAX],AL
005CED1E 0000 ADD BYTE PTR DS:[EAX],AL
005CED20 0000 ADD BYTE PTR DS:[EAX],AL
005CED22 0000 ADD BYTE PTR DS:[EAX],AL
005CED24 E8 D77AE3FF CALL webpics.00406800
```

do you remember the oep stolen bytes we've seen just before the jump to the oep? Time to put them to their place! So the fixed code will be

```
005CED18 55             push ebp
005CED19 8B EC            mov ebp, esp
005CED1B 83 C4 F0         add esp, 0FFFFFFF0h
005CED1E 53             push ebx
005CED1F B8 80 E6 5C 00   mov eax, offset dword_5CE680
005CED24 E8 D7 7A E3 FF   call sub_406800
```

the bytes are ok, now just use a peditor and change the EIP from 0x1000 to 0x001CED18. Ok now we run the program and... aren't you expecting it to run! We have to put back import table. This will be really funny. First of all, we dumped the two api redirection/emulation memory bridges. Now the one at 0x00C3xxxx is for api redirection and stolen bytes, the one at 0x00B4xxxx is for api emulation. Let's look at the import address table in the dumped exe. It starts at 0x005EC1E0. We see the dumped addresses in there, if you check at runtime you can resolve all the modules:

```
005EC1E0
...    kernel32   (rebuild)
005EC280
```

```
---------------
005EC288
...    user32
005EC294
---------------
005EC29C
...    advapi32
005EC2A4
---------------
005EC2AC
...    oleaut32
005EC2B4
---------------
005EC2BC
...    kernel32   (rebuild)
005EC2C8
---------------
005EC2D0
...    advapi32
005EC304
---------------
005EC30C
...    kernel32   (rebuild)
005EC4CC
---------------
005EC4D4   mpr

005EC4DC
...    version
005EC4E4
---------------
005EC4EC
...    gdi32
005EC640
---------------
005EC648
...    user32
005EC940
---------------
005EC948   kernel32   (rebuild)
---------------
005EC950
...    oleaut32
005EC96C
---------------
```

005EC974
... ole32
005EC9C4

005EC9CC
... oleaut32
005EC9E0

005EC9E8
... comctl32 (rebuild)
005ECA48

005ECA50
... winspool
005ECA5C

005ECA64
... shell32
005ECA7C

005ECA84
... wininet
005ECA90

005ECA98
... urlmon
005ECAA0

005ECAA8
... shell32
005ECABC

005ECAC4
... avifil32
005ECAD0

005ECAD8 winmm

the "rebuild" label indicates modules that have emulated or redirected apis.
We start from the redirected/stolen byte apis, that is the bridge at 0xC30000.
In the modules we have addresses like

dword_5EC1E0 dd 0C3948Ch

so to write an automatic rebuilder we have to:

- open the dumped bridge
- go to the offset at which the api is thunking
- read how many stolen bytes are there before the "jump api"
- once we get the real api address subtract the number of stolen bytes so we have the api entry.

Well this is not the best method, using the symbols api you can get, given an address, the api name + offset. For example if we look at some instruction after MessageBoxA we would have something like
user32.MessageBoxA + 0x10
yeah, the same way symbols are resolved in softice and olly :). However the method i've written works. We have to check the 0x00C3xxxx bridge, and we see that api redirection is just in two forms:

1- jmp version
```
00C39548 55            push ebp
00C39549 8B EC         mov ebp, esp
00C3954B FF 75 10      push dword ptr [ebp+10h]
00C3954E FF 75 0C      push dword ptr [ebp+0Ch]
00C39551 FF 75 08      push dword ptr [ebp+8]
00C39554 6A FF         push 0FFFFFFFFh
00C39556 E9 F7 5A 22 77 jmp near ptr 77E5F052h
```
2- push/ret version
```
00C394B8 55            push ebp
00C394B9 8B EC         mov ebp, esp
00C394BB FF 75 10      push dword ptr [ebp+10h]
00C394BE FF 75 0C      push dword ptr [ebp+0Ch]
00C394C1 FF 75 08      push dword ptr [ebp+8]
00C394C4 6A FF         push 0FFFFFFFFh
00C394C6 68 42 9E E5 77 push 77E59E42h
00C394CB C3            retn
```
so we know we have to count bytes and check for E9 or C3 opcode. Of course there could be some interference (for example a push 0xC3), however we will see there is only one byte interference in all api we are going to resolve, so no need to write a more complex analisys routine. Here is the code

```
-----8<-------------------------------------------------
#include <windows.h>

//params
#define T_IAT_START startaddress
#define T_IAT_END endaddress
#define LOAD_NAME "name of foreign dll"

#define BRIDGE_BASE 0xC30000 //for me its C30000
#define PROGNAME "name of dumped exe"
```

```
#define BRIDGENAME "name of dumped bridge"

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPreInst, LPSTR
CmdLine, int CmdShow)
{
   HANDLE hTarget, hBridge;
   void *tBuffer, *bBuffer;
   DWORD temp, tSize, bSize;
   DWORD *Base, TempAddr, TempApi, DeltaIat;
   BYTE OpCode;
   int i;

   //open program file
   hTarget = CreateFile(PROGNAME, GENERIC_READ + GENERIC_WRITE,
NULL, NULL,
   OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
   tSize = GetFileSize(hTarget, &temp);
   tBuffer = malloc(tSize);
   ReadFile(hTarget, tBuffer, tSize, &temp, NULL);

   //open redirect bridge dump
   hBridge = CreateFile(BRIDGENAME, GENERIC_READ + GENERIC_WRITE,
NULL, NULL,
   OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
   bSize = GetFileSize(hBridge, &temp);
   bBuffer = malloc(bSize);
   ReadFile(hBridge, bBuffer, bSize, &temp, NULL);
   CloseHandle(hBridge); //bridge handle no longer needed

   LoadLibrary(LOAD_NAME); //modules that are not loaded from this src

   DeltaIat = 0;
   while(T_IAT_START + DeltaIat <= T_IAT_END)
   {
      Base = ((DWORD*)tBuffer + ((T_IAT_START + DeltaIat) / 4));
      TempAddr = *Base;
      if(TempAddr > BRIDGE_BASE)
      { //process only first redirection method
         //second redirect method has 0x00B30000 base
         //so the check should be adjusted according to that base (dynamic)
         TempAddr -= BRIDGE_BASE;
         i = 0;
         while(true)
         {
            OpCode = ((BYTE*)bBuffer)[TempAddr + i];
```

```
      if(OpCode == 0xC3) //is it a return?
      {
        if( (((BYTE*)bBuffer)[TempAddr + i - 5]) == 0x68)
        {       //was previous instruction a push?
          temp = TempAddr + i - 4;
          __asm mov eax, bBuffer
          __asm add eax, temp
          __asm mov eax, [eax]
          __asm mov [TempApi], eax
          i -= 6;
          break; //then we have found the push/ret to va
        }
      }
      if(OpCode == 0xE9) //is it a jump?
      {
        temp = TempAddr + i + 1; //next dword is va
        __asm mov eax, bBuffer
        __asm add eax, temp
        __asm mov eax, [eax]
        __asm mov [TempApi], eax
        //E9 is relative jump so we must add the calling va
        TempApi += TempAddr + BRIDGE_BASE + i + 5;
        i--;
        break; //then next bytes are the va
      }
      i++;
    }
    //now find the ep of api
    while(i>=0)
    {
      TempApi--;
      if( ((BYTE*)bBuffer)[TempAddr + i] != ((BYTE*)(TempApi))[0] )
      {
        //if some opcode is different then there is code ignjction
        MessageBox(NULL, "Different opcodes", "Error", NULL);
        CloseHandle(hTarget);
        return 0;
      }
      i--;
    } //at the end TempApi = EP of api
    //write ep of api
    SetFilePointer(hTarget, T_IAT_START + DeltaIat, NULL, FILE_BEGIN);
    WriteFile(hTarget, &TempApi, 4, &temp, NULL);
  }
  DeltaIat += 4;
```

```
  }
  CloseHandle(hTarget);
  return 0;
}
-----8<------------------------------------------------
```

doh! looks like a monkey wrote this code! However you just have to set the parameters in the defines. In particular:

T_IAT_START
T_IAT_END

these define the first and the last C3xx thunked apis you want to resolve. Note that all thunks that are in the middle of this interval must have a C3xx address, not a B4xx or other. So for example for first module you should set start=005EC1E0 and end=005EC204, because at 005EC208 there is a B4xx address. So you resolve the first group. To continue you start from 005EC20C and end at 005EC238, again after this address there are two B4xx addresses, and so on. The foreign dll is a dll that is not mapped in memory (in our case kernel32 is present, comctl32 no), so you should map it when resolving its relative apis.
For this program i have only a byte interference, that is at 005ECA38, infact we see it calls this code in the bridge

```
00C3A00C 55            push ebp
00C3A00D 8B EC           mov ebp, esp
00C3A00F 51            push ecx
00C3A010 68 C7 E9 96 71 push 7196E9C7h
00C3A015 C3 retn
```

that E9 is an interference. You can fix it just watching the asprotected program and tracing manually this thunk (or coding a better routine!). Ok, now all the 0x00C3xx work is done. Now in the iat array we have all api addresses (working FirstThunks). Note! Pay attention to relocation! Infact you could have a module relocated in the asprotected program process, but when trying to rebuild IT, we could have the same module not relocated. So the FirstThunks could not work. To avoid this you can dump the relocated dll, or you can look at the Executable Modules window in ollydebug, then you see the path of relocated dlls
Path=D:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.0.0_x-ww_1382d70a\comctl32.dll
and you can copy that dll and use it in your own directory (or if you still have problems you can dump such relocated dll). You only need this to correctly resolve api names, you will not need relocated dll for the final unpacked exe. Once 0x00C3xx is done we need to resolve the 0x00B4xx (emulated) trick.

Luckily there are few (11) apis using this trick. Some are duplicates, so let's see every single emulated api.

1 (005EC208 dword_5EC208 dd 0B41388h)

```
00B41388 push 0
00B4138A call sub_B35158
00B4138F push dword ptr ds:0B46CE8h
00B41395 pop eax
00B41396 retn
```

this is a wrapper at GetVersion function. As you can see there is a call to GetModuleHandle, but it's not useful because the last value returned in eax is in 0B46CE8, that is a previously stored return value of GetVersion, this is a trick to fool automatic tracers that would try to determine imported api by tracing the intermodule call. We could substitute this api just with the address of the api, but it is not always possible. Ok no problem, we just have to add some emulated apis in some cave and then fix the iat at runtime. We will see it later.

2 (005EC23C dword_5EC23C dd 0B40EF0h)

```
00B40EF0 push ebp
00B40EF1 mov ebp, esp
00B40EF3 mov edx, [ebp+0Ch]
00B40EF6 mov eax, [ebp+8]
00B40EF9 mov ecx, ds:0B4543Ch
00B40EFF mov ecx, [ecx]
00B40F01 cmp ecx, eax
00B40F03 jnz short loc_B40F0E
00B40F05 mov eax, ds:0B45350h[edx*4]
00B40F0C jmp short loc_B40F15
00B40F0E push edx
00B40F0F push eax
00B40F10 call sub_B35160
00B40F15 pop ebp
00B40F16 retn 8
```

this wrapper calls GetProcAddress in case of a standard call. Insted of the api name the parameter can be a number, so the aspr does not call GetProcAddress but an internal function corresponding to the array in 0B45350. I've run the program but it seems this second case never happens (in this program).

3 (005EC240 dword_5EC240 dd 0B41360h)

```
00B41360 push ebp
00B41361 mov ebp, esp
00B41363 mov eax, [ebp+8]
00B41366 test eax, eax
00B41368 jnz short loc_B4137D
00B4136A cmp dword ptr ds:0B46978h, 400000h
00B41374 jnz short loc_B4137D
00B41376 mov eax, ds:0B46978h
00B4137B jmp short loc_B41383
00B4137D push eax
00B4137E call sub_B35158
00B41383 pop ebp
00B41384 retn 4
```

a wrap to GetModuleHandle. If the parameter (ebp+8) is NULL, then the program avoids calling the function and returns 0x00400000 (standard hInstance of a standard exe), else calls the function to find the real module imagebase and returns it.

4 (005EC254 dword_5EC254 dd 0B413D0h)

```
00B413D0 push 0
00B413D2 call sub_B35158
00B413D7 push dword ptr ds:0B46CE8h
00B413DD pop eax
00B413DE mov eax, ds:0B46CF8h
00B413E4 retn
```

The call is to GetModuleHandleA function, but as you can see at the end the B46CF8 address is returned, that is the pointer to the command line for the program, so this is the emulator for GetCommandLineA.

5 (005EC390 dword_5EC390 dd 0B413E8h)

```
00B413E8 push ebp
00B413E9 mov ebp, esp
00B413EB mov eax, ds:0B46CF8h
00B413F1 mov eax, [ebp+8]
00B413F4 pop ebp
00B413F5 retn 4
```

this just returns the only parameter this function takes ([ebp+8]) Can't be fixed with an api, it must be replicated!

6 (005EC454 dword_5EC454 dd 0B413C0h)

00B413C0 push ebp
00B413C1 mov ebp, esp
00B413C3 call sub_B35170
00B413C8 mov eax, ds:0B46CF4h
00B413CD pop ebp
00B413CE retn
The call is at GetVersion, but the return value is always the one at 0B46CF4, that is a value that chagnes at runtime, we will see it later.
7 (005EC464 dword_5EC464 dd 0B413F8)

00B413F8 push ebp
00B413F9 mov ebp, esp
00B413FB pop ebp
00B413FC retn 4
this is a null call, but it has a parameter so pay attention to the stack, you can't nop it.

Ok there are not other calls. The problem now is: we can't just put the address of emulated api in the IAT for some of them. When buiding the IT the OriginalFirstThunks and FirstThunks of these api *must* be valid (or we have to split the modules in more descriptors), so the easiest thing is to:
- make a working IT so the api address will be written in the IAT for these apis
- add the needed emulation routines somewhere
- change the entry point so we fix at runtime the FirstThunk of the emulated apis (so every FirstThunk points to emulated code)
So for now we can replace all 0x00B4xx calls with some valid api address, we can choose the address of the api they refer to, so the fixed thunks will be:

005EC208 dd 0B41388 -> 77E5C486 (GetVersion)
005EC23C dd 0B40EF0 -> 77E5A5FD (GetProcAddress)
005EC240 dd 0B41360 -> 77E59F93 (GetModuleHandleA)
005EC254 dd 0B413D0 -> 77E5C938 (GetCommandLineA)
005EC2C8 dd 0B41360 -> 77E59F93 (GetModuleHandleA)
005EC390 dd 0B413E8 -> 77E5751A (return parameter? let's make a GetTickCount)
005EC3DC dd 0B41388 -> 77E5C486 (GetVersion)
005EC414 dd 0B40EF0 -> 77E5A5FD (GetProcAddress)
005EC41C dd 0B41360 -> 77E59F93 (GetModuleHandleA)
005EC454 dd 0B413C0 -> 77E5C486 (changing value? lets make GetTickCount)
005EC464 dd 0B413F8 -> 77E5751A (null, so lets make another GetTickCount)

now all our IAT is filled with valid addresses of api (*remember that we will change them later,* so for now we just need some valid api to be in the IT). Why we did this? Because now we can build a new import table with the 23 imported modules. Once all the descriptor are ok, we have each descriptor in this form:

IMAGE_IMPORT_DESCRIPTOR.OriginalFirstThunk: we have to build it
IMAGE_IMPORT_DESCRIPTOR.TimeDateStamp: 0
IMAGE_IMPORT_DESCRIPTOR.ForwarderChain: -1
IMAGE_IMPORT_DESCRIPTOR.Name: ptr to dll name
IMAGE_IMPORT_DESCRIPTOR.FirstThunk: ptr to IAT data (we've just built it!)

so it is easy to rebuild the IT, just look into imported module, find the api that has export address == first thunk, then copy the name of such api. We can put the new IT at offset 0x0028EE00, it is the zero padding near the end of file. The it will be 24*5 = 120 bytes long (0x78, 23 modules + 1 null descriptor). Here is an example of the it:

... other dll names
75 73 65 72 33 32 2E 64 6C 6C 00 00 00 00 00 00 user32.dll
6B 65 72 6E 65 6C 33 32 2E 64 6C 6C 00 00 00 00 kernel32.dll
0C 13 29 00 00 00 00 00 FF FF FF FF F0 ED 28 00
E0 C1 1E 00 B4 13 29 00 00 00 00 00 FF FF FF FF
E0 ED 28 00 88 C2 1E 00
... other import descriptors

i've written the name of each imported module, then all the 5-dwords dscriptors. As you can see OriginalFirstThunk = FirstThunk = IAT array of api addresses. The dll names are ok. So if you now open the program with a pe editor you will see in the import table all 23 dll modules loaded, but no api names. However, having the import api addresses will resolve this problem, we can just use an import rebuilder: infact this step is mechanical, we have to search all import address in the imported modules and write their corresponding names. I used Wark (www.pmode.cjb.net), a tool of some friends of mine, you can use any rebuilder you want, it just must have this rebuilding feature (that is, translation of FirstThunk to OriginalFirstThunk). Still pay attention to relocations, here comctl32.dll is relocated, so if a rebuilder simply uses LoadLibrary to load the SYSTEM dll, the rebuild will fail (infact I had to write all the 25 OriginalFirstThunks for comctl32 by hand!). You can avoid this writing your own IAT-IT translator, i think i will write mine one day :). Back to us, now the import table is done. We have all the 23 import modules, each one with all imported api names resolved correctly. Perfect. Is this the end? No. We have to re-add the emulated apis! If you remember the emulated apis were:

	iat	import address	
1	005EC208	00B41388	GetVersion
2	005EC23C	00B40EF0	GetProcAddress
3	005EC240	00B41360	GetModuleHandle
4	005EC254	00B413D0	GetCommandLine
5	005EC2C8	00B41360	GetModuleHandle
6	005EC390	00B413E8	return parameter
7	005EC3DC	00B41388	GetVersion
8	005EC414	00B40EF0	GetProcAddress
9	005EC41C	00B41360	GetModuleHandle
A	005EC454	00B413C0	some changing value
B	005EC464	00B413F8	null + stack

i have already listed the code of emulation routines. The 1 and 7 can be fixed just by putting in the iat the address of the GetVersion api. Infact the return value is the one provided by GetVersion. You should always check the caller to be sure that the params are ok (otherwise stack will be corrupted). If you follow an xref to 5EC3DC you see

0048593C jnb short loc_48599C
0048593E call sub_406D00
00485943 and eax, 0FFh
00485948 cmp ax, 4
the call is perfect, so no problems of stack, we can fix it with GetVersion api address. Time to fix 2 and 8. We have seen this emulation is for GetProcAddress. So as before let's find some xref to check out the caller code:

0040E72D push offset aGetdiskfreespa ; "GetDiskFreeSpaceExA"
0040E732 push ebx
0040E733 call sub_406C90
0040E738 mov ds:dword_5CF13C, eax
this too is a valid code. The problem with this emulation could be the fact that instead of an api name the program could pass a number, however i monitored the emulation routine and it seems that a number is never passed. So again as before, we can put in the iat the address of GetProcAddress api. Next we have 3, 5 and 9. Again you can go and see some xref to check the caller code. You will find it's all ok, so we can fix it with the GetModuleHandle api. Let's continue with emulation number 4, we can fix it with GetCommandLine api. We are now at 6th emulated api. We cant fix it with a real api, however the emulation code is simple because it just returns the parameter passed to the function. So we have to fix it with a similiar code, we must make the fix at runtime (i explained before that because the

IT have to be correct the win pe loader will overwrite the fix we would make to the iat). Now we have the A emulated api. We see it always return a value, we just have to understand what this value is! It changes at runime, so probably it is a handle? To find what it is we can put a breakpoint on memory access on 0x00B46CF4 and run the asprotected exe. We will find this code in the aspr loader:

```
00B411EE push 4111FCh
00B411F3 mov eax, ds:B3512Ah
00B411F9 jmp dword ptr [eax]
00B411FC push 411204h
00B41201 jmp short loc_B41222
00B41222 pop edx
00B41223 pop ebx
00B41224 push 41122Bh
00B41229 retn
...
00B4122B mov [ebx-0Ah], eax    <- bpm lands here
00B4122E jmp edx
```

if you see B3512A at runtime you will find it is the address of GetCurrentProcessId. So this emulation snippet calls GetCurrentProcessId, we can fix it easily. The last emulation routine is the B, it is a null call, it just use a stack parameter and removes it (stdcall), so don't just nop the call, you have to adjust stack.
Ok, all emulated apis can be resolved just putting their OriginalFirstThunk in the IT, only the 7 and B emulation routines must be replicated. So we can write the following code:

```
 B8 D01F6900 MOV EAX,dump7_ia.00691FD0
 A3 64C45E00 MOV DWORD PTR DS:[005EC464],EAX
 E01F6900    MOV EAX,dump7_ia.00691FE0
 90C35E00    MOV DWORD PTR DS:[005EC390],EAX
 E9 7FCDF3FF JMP 005CED18
```

where 00691FD0 and 00691FE0 are the two emulation routines:

```
00691FD0 55      PUSH EBP
00691FD1 8BEC    MOV EBP,ESP
00691FD3 5D      POP EBP
00691FD4 C2 0400 RETN 4

00691FE0 55      PUSH EBP
00691FE1 8BEC    MOV EBP,ESP
00691FE3 8B45 08 MOV EAX,DWORD PTR SS:[EBP+8]
```

00691FE6 5D   POP EBP
00691FE7 C2 0400 RETN 4
Of course you can put this code where you want. We fix the two IAT emulated APIs and put there our emulation code, then we jump at original entry point. Remember to change AddressOfEntryPoint in the PE. Okay, APIs are fixed, now you run the program and... It works! We have the working exe and we have completely removed the ASPR layer. Wasn't it funny?

FINAL CONSIDERATIONS

Really good crypter. It has a lot of nice tricks, the API emulation is really a good idea. Note that the code that solves the emulated APIs is deleted from the memory after execution, so you will not have it in the dumped 0x00B4xxxx memory area. Making an unpacker for this ASPR would be a good challenge, I think it could be done by using debug APIs and synthesizing the approach I used for this essay (so you don't have to study description algorithms). That is, I think that making an "offline" unpacker would be really difficult (but not impossible)! Hope you will like this essay :)

GREETS AND THANKS

Thanks to all RET bros! Thanks to Kathras who is helping me really a lot with the VB decompiler, hope we will release a good version soon!
Thanks to Devine9 who always reads and corrects grammar of my writings!
Greets to all UIC members and to all #crack-it people.
A particular greet to Giulia, the craziest binary coder in the world!
GoodBye!

[AndreaGeddon]

	andreageddon@hotmail.com	my mail
	www.andreageddon.com	my lame italian site
[RET]	www.reteam.org	RET's great site
[UIC]	www.quequero.org	italian university of cracking