



ASProtect SKE 2.3 unpacking approach

deroko/ARTeam
Version 1.01 - July 2006

1.	Abstract	2
2.	Examine ASProtect 1 st layer	3
3.	Debugging ASPR with symbols	7
4.	Poly OEP story.....	9
4.1.	Locating Poly OEP one way	9
4.2.	Locating Poly OEP other way	10
4.3.	How does Poly OEP work?	10
4.4.	One way to Fix Poly OEP	19
4.5.	Poly OEP when it comes to Delphi application	21
5.	Advanced Import Protection	23
5.1.	Normal and Advanced Import protection	23
5.2.	Fixing Advanced Import Protection.....	24
5.3.	Fixing Normal Import Protection.....	26
6.	Conclusion	28
7.	Greetings	29

Keywords

ASProtect, SKE, Advanced OEP Protection, Advanced Import Protection, unpacking, reversing



1. Abstract

ASProtect SKE, very good protection system, but not unbreakable (hello unpack.cn©). There are no many tuts about unpacking ASProtect SKE with Advanced OEP Protection, nor there are tuts that cover AIP fixing in the easiest way, nor there is document that covers analyze of virtual.dll (crusader did great job for asprotect 1.x).

Also be aware that this is not tutorial that will let any newbie to fix ASProtect fast, nope, it will give you hints how to fix it, and what you should consider during your unpacking and how to speedup process of unpacking, also I'm mostly dealing with Advanced OEP protection, because, as usual I'm fixing it on my specific way, much different from something that you got used to.

I'm using ASProtect 2.3 DEMO because I was sure that it will have latest ASProtect, so we will deal with it. You may download demo from www.aspack.com.

Tools that I'm using here are:

- SoftICE
- IDA
- idasym or i2s by mostek
- LordPE
- ImportRec
- tasm32 (w/o it this tut wouldn't be possible)

S verom u Boga, deroko/ARTeam



2. Examine ASProtect 1st layer

If we want too examine protection of ASPR we have to first locate it. As all of us know whole protection of ASPR is located in virtual.dll which is loaded in memory from 1st layer. Pain with this dll is that it is Delphi dll so tracing it in debugger is really hard. One option we have is to load dll in IDA, apply signatures and see what is really going on in it.

Oki first we will examine 1st layer and DLL loading in memory, and to see how to dump this dll.

We start from entry of ASPR code here:

```
00401000 >/$ 68 01006100    PUSH ASProtec.00610001
00401005 |. E8 01000000    CALL ASProtec.0040100B
0040100A \. C3          RETN
0040100B $ C3          RETN
```

Well you have seen this dozen of times so lets advance to interesting part, where ASPR decompresses virtual.dll, 1st layer is only loader for virtual.dll so if you really plan to reverse ASPR you have to start with dll, but we will come to dll as soon as we learn how to dump it ☺

After a little bit of tracing 4 small decryption layers you will get to the part where ASPR is using custom implementation of getkernelbase (get base of kernel32.dll) and GetProcAddress to find some APIs that are going to be used by ASPR dll loader.

First that get used is VirtualAlloc to allocate memory where ASPR virtual.dll will be decompressed and memory where it will copy decompressed dll:

```
001B:006104D6  PUSH      00001000
001B:006104DB  PUSH      DWORD PTR [EBP+00000408]
001B:006104E1  PUSH      00
001B:006104E3  CALL      [EBP+000003F0]          <--- VirtualAlloc 1st time
001B:006104E9  MOV       [EBP+000001CC],EAX      for decompression of dll
001B:006104EF  MOV       EBX, [EBP+00000400]
001B:006104F5  ADD       EBX, [EBP+0000040D]
001B:006104FB  PUSH      EAX
001B:006104FC  PUSH      EBX
001B:006104FD  CALL      00610606                <--- decompress
001B:00610502  PUSH      40
001B:00610504  PUSH      00001000
001B:00610509  PUSH      DWORD PTR [EBP+00000408]
001B:0061050F  PUSH      00
001B:00610511  CALL      [EBP+000003F0]          <--- VirtualAlloc 2nd, memory
001B:00610517  MOV       [EBP+00000431],EAX      buffer for Virtual.dll
001B:0061051D  MOV       [EBP+000001D0],EAX
001B:00610523  MOV       EAX, FS: [0000]
```

Aspr loader will copy section to right places:



```
001B:0061056B MOV     EDI,[EBX+EDX+0C]
001B:0061056F OR      EDI,EDI
001B:00610571 JZ      _00610591
001B:00610573 MOV     ECX,[EBX+EDX+10]
001B:00610577 OR      ECX,ECX
001B:00610579 JZ      _0061058C
001B:0061057B ADD     EDI,[EBP+000001D0]
001B:00610581 MOV     ESI,[EBX+EDX+14]
001B:00610585 ADD     ESI,EDX
001B:00610587 SAR     ECX,02
001B:0061058A REPZ MOVSD
001B:0061058C ADD     EBX,28      <--- size of section header
001B:0061058F JMP     _0061056B
```

After all section are copied, ASPR will jump to part of virtual.dll which is responsible for loading APIs needed for dll to work, and also will apply relocations to dll because this is needed for dll, and especially because aspr must work on all machines and when it is allocated with VirtualAlloc it is always on dynamic memory location ☺

Oki we will focus immediately at relocs handling and iat preserving here:

```
001B:00D02403 MOV     EDX,[EBP+004430D8]
001B:00D02409 MOV     EAX,[EBP+00442A51]
001B:00D0240F SUB     EDX,EAX      <--- calculate reloc displacement
001B:00D02411 JZ      _00D02488      <--- if zero no relocations are
001B:00D02413 MOV     EAX,EDX      needed
001B:00D02415 SHR     EAX,10
001B:00D02418 XOR     EBX,EBX
001B:00D0241A MOV     ESI,[EBP+00442A5D]      <-- HERE RELOC RVA
001B:00D02420 ADD     ESI,[EBP+004430D8]      <-- + DLL base
001B:00D02426 CMP     DWORD PTR [ESI],00
001B:00D02429 JZ      _00D02488      <-- set brake at D02488h
001B:00D0242B MOV     ECX,[ESI+04]      or whatever adderes you
001B:00D0242E SUB     ECX,08      get, to skip relocations
001B:00D02431 SHR     ECX,1
001B:00D02433 MOV     EDI,[ESI]
001B:00D02435 ADD     EDI,[EBP+004430D8]
001B:00D0243B ADD     ESI,08
001B:00D0243E MOV     BX,[ESI]
```

After we reach our brake we will see how imports are processed here:

```
001B:00D02488 MOV     ESI,[EBP+00442A61]      <--- RVA of IAT
001B:00D0248E MOV     EDX,[EBP+004430D8]      <--- DLL base
001B:00D02494 ADD     ESI,EDX      <--- IAT in virtual.dll
001B:00D02496 MOV     EAX,[ESI+0C]
001B:00D02499 TEST    EAX,EAX
001B:00D0249B JZ      _00D025AB
001B:00D024A1 ADD     EAX,EDX
001B:00D024A3 MOV     EBX,EAX
001B:00D024A5 PUSH    EAX
```



If you examine address pointed by ESI you will see that there is valid IAT in ASPR virtual.dll:

0010:00CFC000	00000000	00000000	00000000	0004C424\$...
0010:00CFC010	0004C118	00000000	00000000	00000000
0010:00CFC020	0004C6F0	0004C1C0	00000000	00000000
0010:00CFC030	00000000	0004C736	0004C1D4	000000006.....
0010:00CFC040	00000000	00000000	0004C776	0004C1E4v.....
0010:00CFC050	00000000	00000000	00000000	0004C806
...					
0010:00CFC420	00000000	6E72656B	32336C65	6C6C642Ekernel32.dll
0010:00CFC430	00000000	43746547	65727275	6854746EGetCurrentTh
0010:00CFC440	64616572	00006449	65440000	6574656C	readId....Delete
0010:00CFC450	74697243	6C616369	74636553	006E6F69	CriticalSection.
0010:00CFC460	654C0000	43657661	69746972	536C6163	..LeaveCriticals
0010:00CFC470	69746365	00006E6F	6E450000	43726574	ection....EnterC

Good, write down RVA of import table and relocs we will need it for our dumped.dll to work nicely, okay you now may skip IAT filling process and focus on good part of ASPR here:

001B:00D025C1	POPAD		
001B:00D025C2	JNZ	00D025CC	
001B:00D025C4	MOV	EAX,00000001	
001B:00D025C9	RET	000C	
001B:00D025CC	PUSH	00CF0BC4	<--- was PUSH 0
001B:00D025D1	RET		

Push/ret combination will take us to entry point of virtual dll, but place where we want to dump virtual.dll is here:

001B:00D02488	MOV	ESI,[EBP+00442A61]	<--- RVA of IAT
001B:00D0248E	MOV	EDX,[EBP+004430D8]	<--- DLL base
001B:00D02494	ADD	ESI,EDX	<--- here dump

The reason why I dump virtual.dll at this instruction is nothing more then the way I like to work, I hate to assemble jmp \$ when instruction is bigger then 2 bytes. Also we are dumping here because of 2 reasons:

1. we need dll reallocated to new base + relocs
2. we need IAT intact

We also need to know some values before we move on:

RVA_relocs	:	4F000h
RVA_import	:	4C000h
RVA_entry	:	40BC4h
DLL_base	:	CB0000h



ASProtect SKE 2.3 unpacking approach

You will get all these values if you examine logic of aspr loader code, plus I gave you spot where to look for them.

Once we have dumped virtual.dll we see that there is a huge problem, it doesn't have PE header so we are going to reconstruct it, chose some .dll from your system and read 1024 bytes from it (pe header) and copy it to dumped region. Good now you have PE header in dumped.dll but still you have to fix it so dll will work, here are all modifications:

```
+-----+
| Count of sections          1 | Machine          Intel386 |
| Symbol table 00000000[00000000] | Thu Jan 01 01:00:00 1970 |
| Size of optional header    00E0 | Magic optional header 010B |
| Linker version             2.25 | OS version         1.00 |
| Image version              0.00 | Subsystem version   4.00 |
| Entry point                00000500 | Size of code        0003F400 |
| Size of init data          0004CC00 | Size of uninit data 00000000 |
| Size of image              00054000 | Size of header       00001000 |
| Base of code               00001000 | Base of data         00041000 |
| Image base                 00CB0000 | Subsystem            GUI |
| Section alignment          00001000 | File alignment       00000200 |
| Stack 00000000/00000000 | Heap 00100000/00001000 |
| Checksum 00000000 | Number of dirs      16 |
+-----+
```

Only importan here are NumberOfSection which I set to 1, imagebase set to CB0000h, entrypoint set to 4B0C4h (don't worry about entry at 500h will come to that later), header size to 1000h and imagesize to size of dumped.dll. Then I make my own section like this:

```
+--Number Name      VirtSize  RVA      PhysSize  Offset    Flag---+
|      1 CODE      00053000 00001000 00053000 00001000 E0000020 |
|-----|
+- Cursors out of sections -----+
```

After that we have to fix import and relocs in DataDirectories:

```
+--      Name      RVA      Size  --+
| Export          00000000 00000000 |
| Import          0004C000 00000028 |
| Resource        00000000 000047C0 |
| Exception       00000000 00000000 |
| Security        00000000 00000000 |
| Fixups          0004F000 00002D7C |
| Debug          00000000 00000000 |
| Description     00000000 00000000 |
| MIPS GP        00000000 00000000 |
| TLS            00000000 00000000 |
| Load config    00000000 00000000 |
| Bound Import   00000000 00000000 |
```



```
| Import Table 00000000 00000000 |  
| Delay Import 00000000 00000000 |  
| COM Runtime  00000000 00000000 |  
| (reserved)   00000000 00000000 |  
+-----+-----+
```

If you are wondering how I got size of relocs it is very simple, just go to relocs and find where they are ended with a lots of zeros, subtract end of relocs with start and you will get size of relocs.

Nice, after all this steps you should get working dll which you may load from any application (why?☺) or analyze it in IDA (clap, clap). Let's load it in IDA and apply Borland signatures to it. Wait a little bit and get ready for mind blow tech ☺

3. Debugging ASPR with symbols

Now once you got your dll analyzed in IDA you may now perform live debugging of it but down fail is that you have to look in your disassemble every time you need to see if some procedure is Delphi library or ASPR code (especially annoying when you are using SoftICE for debugging). Well this little inconvenience of Delphi debugging is over.

We are going to produce .map file from analyzed dump or we may directly produce .nms file using IDA 2 SoftICE plug-in for IDA. I prefer to use idasym and produce .sym out of .map file, but i2s plug-in which takes more time to produce .nms file has better recognition of all labels, so if you are sure that you have commented everything use i2s to produce .nms or if you are still tracing and experimenting use idasym, it is much faster ☺

We have to force our debugger to recognize symbols but we can't do it by simply loading them into debugger, we have to force also dumped.dll to be loaded as well, so all loaded symbols become valid and then we will be able also to set BPXes fast using symbols instead of typing addresses. This is also reason why we need working and fixed dll so it can be loaded and make our symbols valid.

Now we come to entry point of 500h that you saw in above chapter. When virtual.dll is loaded it is not loaded via LoadLibraryA nor will it be "registered" in PEB_LDR_DATA, which also means that dll entry point won't be called on THREAD_ATTACH, THREAD_DETACH, PROCESS_DETACH. That's why we have to set entry point somewhere where it will not be overwritten by aspr .dll loader and will be called by system whenever new thread starts, otherwise, my best guess is UnhandledException ☺ and termination of application.

Once we have fixed dll we may load it following this simple procedure, break on 2nd VirtualAlloc (remember where decompression and copying of virtual.dll takes place?) once you break at 2nd VirtualAlloc simply change its arguments so 1st argument will point to string "dumped.dll" and redirect execution to LoadLibraryA. Yes, ASPR will overwrite your dll later on, except your PE header but you got what you were looking for. Symbols are now visible in your favorite debugger.



Stack at entry of VirtualAlloc:

```
:dd esp
0010:0013FF90 00610517 00000000 00054000 00001000 ..a.....@.....
0010:0013FFA0 00000040 00150178 7C9012D6 0013FFF0 @...x.....|....
```

Then we store “dumped.dll” somewhere in memory:

```
0010:00400500 706D7564 642E6465 00006C6C 00000000 dumped.dll.....
0010:00400510 00000000 00000000 00000000 00000000 .....
```

And we change argument on stack to point to our “dumped.dll”:

```
0010:0013FF90 00610517 00400500 00054000 00001000 ..a...@...@.....
0010:0013FFA0 00000040 00150178 7C9012D6 0013FFF0 @...x.....|....
```

Now type:

```
:r eip kernel32!LoadLibraryA
```

In normal conditions we would simply type `r eip LoadLibraryA` but this time in `.sym` or `.nms` there is symbol named `LoadLibraryA` (comes from IDA), and symbols have “priority” over exports in SoftICE so redirecting to `LoadLibraryA` will redirect you to wrong code ☺

Bingo, once you reach your entry of ASPR dll you may see labels and it is much easier to debug now, isn't it? Oki I will show you one picture, just to show you power of symbols loaded in such way, although I will avoid pictures from now on, and mainly use disassembly from IDA:

```
001B:00CD5044 LEA EAX,ES:[EAX+00]
001B:00CD5048 PUSH EBP
001B:00CD5049 MOV EBP,ESP
001B:00CD504B ADD ESP,-08
001B:00CD504E PUSH EBX
001B:00CD504F PUSH ESI
001B:00CD5050 PUSH EDI
001B:00CD5051 MOV BYTE PTR [EBP-01],01
001B:00CD5055 MOV EBX,[EBP+0C]
001B:00CD5058 XOR ECX,ECX
001B:00CD505A PUSH EBP
001B:00CD505B PUSH 00CD5093
001B:00CD505D PUSH DWORD PTR FS:[ECX]
001B:00CD5063 MOV FS:[ECX],ESP
001B:00CD5066 JMP 00CD5083
001B:00CD5068 ADD EAX,[EBP+08]
001B:00CD506B LEA EDX,[EBX+04]
001B:00CD506E MOV EDX,[EDX]
001B:00CD5070 ADD EDX,[EBP+10]
001B:00CD5073 LEA ECX,[EBP-08]
001B:00CD5076 PUSH ECX
001B:00CD5077 MOV ECX,EDX
001B:00CD5079 MOV EDX,EAX
001B:00CD507B CALL redirect_jump
001B:00CD5080 ADD EBX,08
001B:00CD5083 MOV EAX,[EBX]
001B:00CD5086 TEST EAX,EAX
001B:00CD5087 JNZ 00CD5068
001B:00CD5089 XOR EAX,EAX
001B:00CD508B POP EDX
001B:00CD508C POP ECX
001B:00CD508D POP ECX
001B:00CD508E MOV FS:[EAX],EDX
001B:00CD5091 JMP loc_CD50A1
```




4. Poly OEP story

4.1. Locating Poly OEP one way

To locate poly OEP we have to first locate some stolen procedure so we can know where to break in virtual.dll, simply you have to search in dumped file for procedure that will take you out of .code section starting with `jmp __outofcode` (there are 2-3 more ways of locating poly oep, some easier some more advanced but I'll describe only one way of locating it). When you find such procedure, set hardware breakpoint on it and wait until aspr starts writing address there. This trick is simple, stolen procedures are part of poly oep, and those might be called from other parts of code.

Wait for your break to hit and you will break here in virtual.dll:

```
CODE:00CD3A33      mov     byte ptr [ebx], 0E9h
CODE:00CD3A36      lea     edx, [ebx+1]
CODE:00CD3A39      mov     [edx], eax
CODE:00CD3A3B      mov     eax, [ebp+arg_0]
```

Instruction at CD3A39h is responsible for making `jmp __outofcode`, ok trace a little bit (till `retn`) and you will be in this beast:

```
CODE:00CD5066      jmp     short jmp_modify_loop_start
CODE:00CD5068  jmp_modify_loop:
CODE:00CD5068      add     eax, [ebp+imagebase]
CODE:00CD506B      lea     edx, [ebx+4]
CODE:00CD506E      mov     edx, [edx]
CODE:00CD5070      add     edx, [ebp+poly_oep_va]
CODE:00CD5073      lea     ecx, [ebp+var_8]
CODE:00CD5076      push    ecx
CODE:00CD5077      mov     ecx, edx
CODE:00CD5079      mov     edx, eax
CODE:00CD507B      call    redirect_jmp
CODE:00CD5080      add     ebx, 8
CODE:00CD5083  jmp_modify_loop_start:
CODE:00CD5083      mov     eax, [ebx]
CODE:00CD5085      test    eax, eax
CODE:00CD5087      jnz     short jmp_modify_loop
```

By looking at this code it seems like `ebx` is pointing to some kind of struct if we dump that struct from memory we might see how it is organized and figure how to locate `poly_oep` start:

```
seg000:00002070      dd 3BD8h
seg000:00002074      dd 0DFh
seg000:00002078      dd 3C50h
seg000:0000207C      dd 103h
seg000:00002080      dd 3C57h
seg000:00002084      dd 10Ah
```



seg000:00002088	dd 6B68h
seg000:0000208C	dd 2CAh
seg000:00002090	dd 52EA8h
seg000:00002094	dd 95Fh
seg000:00002098	dd 52EC0h
seg000:0000209C	dd 643h
seg000:000020A0	dd 52F14h
seg000:000020A4	dd 5A0h
seg000:000020A8	dd 13810Ch
seg000:000020AC	dd 0h

If you trace this part of code trough debugger you may see how this structure has only 2 members:

1. RVA of stolen procedure
2. RVA of real procedure relative to VA of poly_oeop

So here is place where you might catch poly_oeop address. Find this code once, and latter on use byte search to find this code and find poly_oeop VA fast. That's how you find poly_oeop, one way, more then enough. You still may use exceptions to locate poly oeop (what if there are no exceptions?). Most protectors use Exceptions to clear debug registers, which is apparently used against them to reach OEP, but what if they would use only NtContinue to clear debug registers, lame exception counting will fail. Never, never relay on exceptions during unpacking it can be changed and your way of "automated" unpacking will fail. Learn to analyze code, well Alexey I hope you got idea? Ehh...

When you dump your application, you should fix those jmp to rihgt places because those are procedures that might be called from code itself and if those are not rebased, well, you dump will fail.

4.2. Locating Poly OEP other way

This method is rather simple, but we will have to examine poly OEP and how it works to know where and how to catch poly oeop VA. Sorry, section is small because without knowing how poly oeop dispatcher works it is pointless to show data. So keep in mind that I will show you where to catch poly oeop va, so read carefully ☺

4.3. How does Poly OEP work?

Well if you trace from start of poly oeop at one point you will see this code:

001B:01E607FE	PUSH	01E608B9
001B:01E60803	CALL	02050000

Call will call procedure responsible for saving registers, saving last branch, eflags and calling poly_oeop_dispatcher, if you dump region from memory you can get nice and clean disassembly by looking at it (I removed jmps and do nothing instructions):



ASProtect SKE 2.3 unpacking approach

```
push    eax                ;save dispatch register
pushf                   ;save eflags
sub     esp, 20h           ;make some space on stack
mov     eax, esp
mov     [eax+14h], ebp     ;save ebp
mov     [eax+18h], esi     ;save esi
mov     [eax+0Ch], ebx     ;save ebx
mov     [eax+4], ecx       ;save ecx
mov     [eax+8], edx       ;save edx
mov     [eax+1Ch], edi     ;save edi
mov     ebp, esp          ;ebp = eax (eax = esp)
add     ebp, 2Ch           ;ebp = stack pointer
push    ebp               ;save esp
push    eax               ;save delta register
mov     ebp, eax
add     ebp, 20h           ;ebp points to eflags now
mov     ebp, [ebp]         ;ebp = eflags
push    ebp               ;save eflags
mov     ebp, eax
lea     ebp, [ebp+28h]     ;ebp points to return value
mov     ebp, [ebp]         ;grab return address of VM enter loop
sub     ebp, 5
add     ebp, 400h
push    ebp               ;save instr_bfr_VM + PID on stack
mov     ebp, 0D20600h
push    ebp               ;save jmp dispatcher on stack
mov     eax, 0CD4D38h    ;poly_oep_dispatcher
call    eax
```

I have commented it so it isn't problem for you to figure how it works, it is rather simple.

After that we are entering into `poly_oep_dispatcher`, somewhat obfuscated but we can follow it without a problem if we are in disassembler. Be sure to understand what is passed on stack to `poly_oep_dispatcher`, otherwise you will be lost, note also how I bolded some parts of `call_poly_oep_dispatcher`, we will also need dump of data stored at `D20600h` to understand what is going on in `poly_oep_dispatcher` (here is other place where you may fish poly oep va):

```
seg000:00010600      dd 0CD3E4Ch
seg000:00010604      dd 0
seg000:00010608      dd 1E60DE5h
seg000:0001060C      dd 0
seg000:00010610      dd 400000h          ; image base
seg000:00010614      dd 25h             ; number_of_poly_entries
seg000:00010618      dd 1E602F1h         ; poly oep va
seg000:0001061C      dd 2050000h        ; call_poly_oep
seg000:00010620      dd 2060000h        ; exit_poly_oep
seg000:00010624      db 1               ; indexes for procedures
seg000:00010625      db 0               ; starting at 106040
seg000:00010626      db 9
seg000:00010627      db 6
seg000:00010628      db 2
seg000:00010629      db 5
seg000:0001062A      db 8
seg000:0001062B      db 3
```



ASProtect SKE 2.3 unpacking approach

```
seg000:0001062C      db  4
seg000:0001062D      db  7
seg000:0001062E      db  0
seg000:0001062F      db  0
seg000:00010630      dd  1E6180Ah          ;VM opcodes start addr
seg000:00010634      dd  6090001h
seg000:00010638      dd  3080502h
seg000:0001063C      dd  704h
seg000:00010640      dd  1E60F9Eh          ;procedures responsible
seg000:00010644      dd  1E6139Ch          ;for manipulating VM
seg000:00010648      dd  1E61034h          ;entries
seg000:0001064C      dd  1E612D5h
seg000:00010650      dd  1E60ECCh
seg000:00010654      dd  1E6118Ch
seg000:00010658      dd  1E60DEAh
seg000:0001065C      dd  1E61432h
seg000:00010660      dd  1E610DDh
seg000:00010664      dd  1E61233h
seg000:00010668      dd  0CEB527A6h
seg000:0001066C      dd  3Ah          ;VM opcode size
seg000:00010670      dd  0E733E243h
```

We enter into poly_oeplib_dispatcher here:

```
CODE:00CD4D38 poly_oeplib_dispatcher proc near
CODE:00CD4D38 hash_indexes      = dword ptr -8
CODE:00CD4D38 var 4            = dword ptr -4
CODE:00CD4D38 jmp_dispatcher_struct = dword ptr 8
CODE:00CD4D38 last_branch_plus_pid = dword ptr 0Ch
CODE:00CD4D38 saved_eflags      = dword ptr 10h
CODE:00CD4D38 saved_registers_ptr = dword ptr 14h
CODE:00CD4D38 saved_esp        = dword ptr 18h
CODE:00CD4D38
CODE:00CD4D38      push      ebp
CODE:00CD4D39      mov       ebp, esp
CODE:00CD4D3B      add       esp, 0FFFFFFF8h
CODE:00CD4D3E      push      ebx
CODE:00CD4D3F      push      esi
CODE:00CD4D40      push      edi
```

Note the names of arguments passed to poly oep dispatcher, using this we may easily understand what is going on, and with a little bit of creating our own structures in IDA we may trace execution very easily, so let's focus on some important parts:

```
CODE:00CD4D79      mov       eax, ds:aspr_saved_apis
CODE:00CD4D7E      mov       eax, [eax+34h]
CODE:00CD4D81      call      eax          ; GetCurrentProcessId
CODE:00CD4D83      sub       [ebp+last_branch_plus_pid], eax
CODE:00CD4D86      mov       eax, [ebp+last_branch_plus_pid]
```



ASProtect SKE 2.3 unpacking approach

Now look how pid is subtracted from last_bracnh_plus_pid, scroll back to call_poly_oeop_dispatcher and you will see that to return address is added PID so here ASPR is subtracting PID to get position of call __call_poly_oeop_dispatcher. (code that saves regs, eflags, remember? Ehh)

After that some calculations are performed, then we have code that actually seeks for right VM opcode, if none of them is right one, you will get “Protection Error” message.

So let’s see how VM opcode is located:

```
CODE:00CD4D73      mov     esi, [ebx+ jmp_dispatcher_table.VM_opcodes_array]
CODE:00CD4D76      mov     edi, [ebx+_jmp_dispatcher_table._poly_entries_num]
```

EBX is pointing to data buffer dumped above, and now ESI holds start of 1st VM opcode, while EDI has number of VM opcodes, and then searching for right opcode occurs:

```
CODE:00CD4D92      lea     eax, [ebx+_jmp_dispatcher_table._VM_proc_index_array]
CODE:00CD4DA2      movzx  eax, byte ptr [eax]
CODE:00CD4DA5      mov     edx, [ebx+eax*4+40h]
CODE:00CD4DA9      mov     eax, esi
CODE:00CD4DAB      call   edx
CODE:00CD4DB0      jnz     short cycle_poly_oeop
...
CODE:00CD4DF7      dec     edi
CODE:00CD4DF8      add     esi, [ebx+6Ch] <-- size of VM opcode (3A)
CODE:00CD4DFB      test   edi, edi
CODE:00CD4DFD      ja     short cont_poly_oeop2
```

Now we have to locate right VM using call edx but that procedure is heavily obfuscated so I suggest you to dump it using index and VA in data table and then simple deobfuscate it. Here is how you do it by analyze of jmp_dispatcher (data buffer at D20600h):

```
seg000:00010624      db 1                ; indexes for procedures
seg000:00010625      db 0                ; starting at 10640
seg000:00010626      db 9
```

Indexes, well 10 of them are used in this poly_oeop_dispatcher, and procedures are here:

```
seg000:00010640      dd 1E60F9Eh         ;procedures responsible
seg000:00010644      dd 1E6139Ch         ;for manipulating VM
seg000:00010648      dd 1E61034h         ;entries
seg000:0001064C      dd 1E612D5h
```

So by looking at this we may already make a conclusion, eax is pointing to indexes and then 1st index is loaded into eax(1) and then procedure at eax*4+40 is used, and that is 44h, so edx is calling procedure at 1E6139Ch to analyze VM opcode, it is rather simple procedure when is deobfuscated:



ASProtect SKE 2.3 unpacking approach

```
push    edx
mov     edx, eax
add     edx, 2Dh
mov     eax, [edx]
pop     edx
retn
```

At offset 2Dh of VM opcode we will have it's ID, it is used to know later on which VM opcode is responsible for this particular entry, once right opcode is located we continue execution here:

```
CODE:00CD4DBE      mov     eax, [ebp+saved_eflags]
CODE:00CD4DC1      push    eax
CODE:00CD4DC2      mov     eax, [ebp+saved_registers_ptr]
CODE:00CD4DC5      push    eax
CODE:00CD4DC6      call    save_old_SEH
CODE:00CD4DCB      push    eax
CODE:00CD4DCC      mov     ecx, esi
CODE:00CD4DCE      mov     edx, [ebp+saved_esp]
CODE:00CD4DD1      mov     eax, ebx
CODE:00CD4DD3      call    call_next_poly_OEP
```

This code will actually call procedure which will decide where to go, decide if we have jmp or jcc or cmp/jcc combination. So we advance into call_next_poly_OEP, also comment what is passed on stack, otherwise you will be lost ☺ :

```
CODE:00CD4648      call_next_poly_OEP proc near
CODE:00CD4648
CODE:00CD4648      var_10      = dword ptr -10h
CODE:00CD4648      var_C       = dword ptr -0Ch
CODE:00CD4648      jmp_dispatcher = dword ptr -8
CODE:00CD4648      saved_esp    = dword ptr -4
CODE:00CD4648      ptr_to_exception_record= dword ptr 8
CODE:00CD4648      register_pointer= dword ptr 0Ch
CODE:00CD4648      saved_eflags = dword ptr 10h
CODE:00CD4648
CODE:00CD4648      push     ebp
CODE:00CD4649      mov      ebp, esp
CODE:00CD464B      add      esp, 0FFFFFFF0h
CODE:00CD464E      push     ebx
CODE:00CD464F      push     esi
CODE:00CD4650      push     edi
CODE:00CD4651      mov      esi, ecx
CODE:00CD4653      mov      [ebp+saved_esp], edx
CODE:00CD4656      mov      [ebp+jmp_dispatcher], eax
CODE:00CD4659      mov      eax, [ebp+jmp_dispatcher]
CODE:00CD465C      lea      edi, [eax+jmp_dispatcher_table._VM_proc_index_array]
CODE:00CD465F      xor      eax, eax
CODE:00CD4661      mov      al, [edi+VM_proc_index.VM_opcode_type_proc]
CODE:00CD4664      mov      edx, [ebp+jmp_dispatcher]
CODE:00CD4667      mov      ebx, [edx+eax*4+40h]
CODE:00CD466B      mov      eax, esi
CODE:00CD466D      call     ebx
```



```
CODE:00CD466F      mov     ebx, eax
...
CODE:00CD4697      sub     bl, 2
CODE:00CD469A      jb     jmp_or_call
CODE:00CD46A0      jz     short only_jcc
CODE:00CD46A2      dec     bl
CODE:00CD46A4      jnz    error_in_vm
```

Between “...” there are 2 more procedure called to extract data from VM opcode and those are used depending if opcode is jcc or jmp, if jcc one will give us where we go if jmp is taken, other were to go if jmp isn’t taken. When it is jmp it is used to determine if jmp goes back to poly_oeop or it is going to .code section, but we may skip those because jmp/jcc are not really problem to fix in aspr. Let us focus at call ebx, this time it is extracting from VM opcode type of operation, looking at procedure that is called is a little bit nasty until we deobsfucate it:

```
push     edx
mov      edx, eax
add      edx, 11h
mov      edx, [edx]
and      edx, 0FFh
mov      eax, edx
pop      edx
retn
```

What it does is simple, it extracts type of opcode, well this can be rewritten as one line movzx eax, byte ptr [eax+11h] but still it is simple to understand isn’t it? ☺

Now take a look at cmp at CD4697h, you may see what is what, by looking at my comments and what type of opcode we are looking for:

- 1 – jmp or call (well call is recognized by push/call __call_poly_oeop_dispatcher) so this is actually changing EIP instruction only
- 2 – it is jcc
- 3 – it is emulation of cmp and jcc and this is place where I gave up in wiring unpacker. I simply couldn’t brute force this. Jcc are simple to brute force, simple change eflags and watch when execution flow is changed and assemble good jcc, but this emulation, there was no point in exploring it because Alexey is or will change vm so making static analyzer for one target makes it fail on next version ☹ Although I have this procedure, emulation, fully commented but there is no point in showing it, I will make ASPR code to do all hard work for me.

Once type of operation is determined we may proceed to jcc or jmp or emulate_cmp procedures. Characteristic for emulate_cmp is that it will decide what kind of cmp is this particular VM and then will decide if jcc that follows it, is taken or not.

Now let’s focus on jcc and how it is organized in this particular target:



ASProtect SKE 2.3 unpacking approach

```
CODE:00CD4717      xor     eax, eax
CODE:00CD4719      mov     al, [edi+VM_proc_index.VM_jcc_type_proc]
CODE:00CD471C      mov     edx, [ebp+jmp_dispatcher]
CODE:00CD471F      mov     ebx, [edx+eax*4+40h]
CODE:00CD4723      mov     eax, esi
CODE:00CD4725      call    ebx
CODE:00CD4727      mov     ebx, eax
CODE:00CD4729      mov     eax, [ebp+jmp_dispatcher]
CODE:00CD472C      xor     bl, [eax+70h]
CODE:00CD472F      mov     ecx, [ebp+saved_eflags]
CODE:00CD4732      mov     edx, ebx
CODE:00CD4734      mov     eax, [ebp+jmp_dispatcher]
CODE:00CD4737      call    jcc_type
```

First ASPR will call ebx with right procedure to decide what kind of jcc is used here (again deobsfucated):

```
push    ebp
mov     ebp, eax
add     ebp, 15h
mov     ebp, [ebp]
and     ebp, 0FFh
mov     eax, ebp
pop     ebp
retn
```

Once extracted jcc type is xored with value stored at jmp_dispatcher+70, this is used to get right jcc type in range from 0 - 0Fh:

```
CODE:00CD4A84  jcc_type      proc near
CODE:00CD4A84      push     ebx
CODE:00CD4A85      push     esi
CODE:00CD4A86      mov     esi, ecx          ; eflags esi
CODE:00CD4A88      xor     ebx, ebx
CODE:00CD4A8D      xor     eax, eax
CODE:00CD4A8F      mov     al, dl            ; jcc type
CODE:00CD4A91      cmp     eax, 0Fh
CODE:00CD4A94      ja     error_wrong_jmp
CODE:00CD4A9A      jmp     dword ptr ds:jmp_case_table[eax*4] ; switch jump
CODE:00CD4AA1  jmp_case_table:
CODE:00CD4AA1      dd offset jcc_jo
CODE:00CD4AA1      dd offset jcc_jno
CODE:00CD4AA1      dd offset jcc_jb
CODE:00CD4AA1      dd offset jcc_jnb
CODE:00CD4AA1      dd offset jcc_jz
CODE:00CD4AA1      dd offset jcc_jnz
```

Well I didn't past whole table for switch but I'll show you how jcc conditions are tested, let's look at jcc_jz label:



```
CODE:00CD4B3C      mov     edx, 6
CODE:00CD4B41      mov     eax, esi
CODE:00CD4B43      call   bit_test
CODE:00CD4B48      mov     ebx, eax
CODE:00CD4B4A      jmp     loc_CD4CE9
```

And bit_test:

```
CODE:00CD3A60 bit_test      proc near
CODE:00CD3A60      bt      eax, edx
CODE:00CD3A63      sbb     eax, eax
CODE:00CD3A65      and     eax, 1
CODE:00CD3A68      retn
CODE:00CD3A68 bit_test      endp
```

In eax we pass state of saved eflags and in edx we have bit that we are testing, by looking at IA manual we see that bit 6 in eflags is Z flag (you didn't know this? ☺) , so what it does here is simple, it tests if bit 6 is set, if so on exit from here ebx will have 1 otherwise it will be zero, so we decide if jmp is taken/not taken depending on bit test. That's how jcc emulation is performed, but we know this. Other method to easily recover jcc is brute forcing them. Brute force jcc? Yes, bruteforce!!! While I was working on my poly oep recovery tool I used to bruteforce type of jcc by simply playing with eflags of traced process and determining what magic combination of jcc resulted in different EIP so I just assembled that jcc that was right for certain combination of eflags. So not only crypto keys can be brute forced ☺ also jcc can ☺ I gave up on this tool when finally I was sure how poly oep works and that I'm not able to bruteforce emulated cmp/jcc. Too bad, it was damn fast tool ☹

So lets step out of jcc emulation proc and see what is then going on:

```
CODE:00CD473C      test    al, al
CODE:00CD473E      jz      short jcc_not_taken
CODE:00CD4740      mov     eax, [ebp+jmp_dispatcher]
CODE:00CD4743      mov     eax, [eax+ jmp_dispatcher_table._poly_oep_va]
CODE:00CD4746      add     eax, [ebp+var_10]
CODE:00CD4749      mov     edx, [ebp+jmp_dispatcher]
CODE:00CD474C      add     eax, [edx+68h]
CODE:00CD474F      mov     edx, [ebp+jmp_dispatcher]
CODE:00CD4752      add     eax, [edx+70h] <-- next entry VA in eax
CODE:00CD4755      jmp     short go_to_real_code
CODE:00CD4757 jcc_not_taken:
CODE:00CD4757      mov     eax, [ebp+jmp_dispatcher]
CODE:00CD475A      mov     eax, [eax+ jmp_dispatcher_table._poly_oep_va]
CODE:00CD475D      add     eax, [ebp+var_C]
CODE:00CD4760      mov     edx, [ebp+jmp_dispatcher]
CODE:00CD4763      add     eax, [edx+68h]
CODE:00CD4766      mov     edx, [ebp+jmp_dispatcher]
CODE:00CD4769      add     eax, [edx+70h] <-- next entry VA in eax
CODE:00CD476C      jmp     short go_to_real_code
```

If you remember I told you to read carefully and you will see where to fish poly_oep_va, so here is one place to catch it without a problem – CD475Ah or CD4743h for example.



I don't know if this is done to make our tracing a little bit harder or this is how code is being generated by Delphi:

```
CODE:00CD4760      mov     edx, [ebp+jmp_dispatcher]
CODE:00CD4763      add     eax, [edx+68h]
CODE:00CD4766      mov     edx, [ebp+jmp_dispatcher]
```

Just side note, has no impact on this small essay.

Oki we reach our label `go_to_real_code`:

```
CODE:00CD47A2      mov     edx, [ebp+saved_esp]
CODE:00CD47A5      sub     edx, 4
CODE:00CD47A8      mov     [edx], eax
CODE:00CD47AA      mov     eax, [ebp+ptr_to_exception_record]
CODE:00CD47AD      call   restore_old_SEH
CODE:00CD47B2      push   [ebp+saved_esp]
CODE:00CD47B5      push   [ebp+saved_eflags]
CODE:00CD47B8      push   [ebp+register_pointer]
CODE:00CD47BB      mov     eax, [ebp+jmp_dispatcher]
CODE:00CD47BE      jmp     [eax+_jmp_dispatcher_table._exit_poly_oep_va]
```

As you may see address of next executed instruction is saved on stack-4 so `jmp [esp-4]` can redirect us to right place, also old seh is restored, old esp, structure that describes state of registers is passed on stack, eflags too, and now we `jmp` to `poly_oep_out`, yet another obfuscated procedure located at buffer 2600000h for me, but if we deobfuscate it we get nice and simple code:

```
pop     ebx
mov     ecx, [ebx+4]
mov     edx, [ebx+8]
mov     edi, [ebx+1Ch]
mov     eax, [ebx]
mov     ebp, [ebx+14h]
mov     esi, [ebx+18h]
mov     ebx, [ebx+0Ch]
popf
pop     esp
jmp     dword ptr [esp-4]
```

Not much to say, restore registers, eflags, esp and `jmp` to next code in obfuscated poly oep. That's how it is, it is simple when you analyze it. You may easily emulate all procedures and extract poly oep without a problem if you have time to analyze it in detail ☺ Situation with redirected `jmp` is even simpler it will only decide if `jmp` goes to code or to poly oep and call it, but logic is similar as is for `jcc` so you may investigate it on your own.



4.4. One way to Fix Poly OEP

Fixing poly oep can be done if you know what is what VM opcode, but method that I'm using here is to use my own dll and to handle poly oep from it. What we need is to reach entry of poly oep, and dump needed buffers so we may make our dll. We need 5 things here:

1. poly oep buffer
2. call_poly_oep_dispatcher (for me it was 2500000h)
3. get_out_of_poly (for me it was 260000h)
4. disassembled dll and it's dump as asm file, make sure to make nice comments of procedures so you can easily erase not needed parts
5. buffer at D20600h (I call it jmp_dispatcher, you name it as you want)

Now we are going to build our dll that will have all needed parts in it, I will only copy/past some important parts of it because you will get src so analyze it on your own:

Dll has structure like this:

1. Dll has 1 export named poly_oep_start, which should be called and will start executing poly_oep
2. DllEntry point is responsible for fixing AIP and reallocating calls in poly_oep dumped region, all calls are redirected to my deobsfucated call_poly_oep_dispatcher (proc that saves registers, eflags)
3. Here comes whole ripped code that is actually decoding what is what VM opcode, also note this little change here:

```
mov     eax, 114h           ;mov eax, ds:aspr_saved_apis
                          ;mov eax, [eax+34h]
                          ;call     eax           ;GetCurrentProcessId
```

We are using PID of process from which we dumped call_poly_oep_dispatcher so we can get right last_branch position

4. Then we have get_out_of_poly, procedure that is restoring regs, eflags and jmps to next code that should be executed
5. dumped region from D20600, which is rebased to point to right offsets in dumped poly_oep
6. dump of poly oep region

Now we will see some important parts located inside of my dll:

```
VM_DATA:
include      vm_dataz.asm
old_poly_base equ      1e60000h
poly_oep_base:
include      poly_oep.inc
```



ASProtect SKE 2.3 unpacking approach

and in vm_dataz.asm I'm rebasing important data structures to point to my right places in dumped poly oep region, here is how I did it (not nuclear physics, just simple rebasing ☺):

```
poly_oep:  dd 1E602F1h - old_poly_base + poly_oep_base    ;poly oep va
           dd 2050000h                                     ;enter_poly_oep_va
           dd 2060000h                                     ;exit_poly_oep_va
           dd 9080102h
           dd 3070504h
           dd 6
vm_opcodes: dd 1E6180Ah - old_poly_base + poly_oep_base    ;VM opcodes
           dd 9080102h
           dd 3070504h
           dd 6
           dd 1E61432h - old_poly_base + poly_oep_base    ;procedures
           dd 1E60F9Eh - old_poly_base + poly_oep_base
           dd 1E6139Ch - old_poly_base + poly_oep_base
           dd 1E612D5h - old_poly_base + poly_oep_base
           dd 1E61034h - old_poly_base + poly_oep_base
           dd 1E6118Ch - old_poly_base + poly_oep_base
           dd 1E60ECCh - old_poly_base + poly_oep_base
           dd 1E610DDh - old_poly_base + poly_oep_base
           dd 1E61233h - old_poly_base + poly_oep_base
           dd 1E60DEAh - old_poly_base + poly_oep_base
```

Once compiled, dll should be loaded from traced process and it's export "poly_oep_start" should be called:

```
delta:      call    delta
            pop     ebp
            sub     ebp, offset delta
            call    getkernelbase

            mov     [ebp+kernel32], eax
            gethash <GetProcAddress>
            push    hash
            push    eax
            call    getprocaddress
            mov     [ebp+GetProcAddress], eax

            gethash <LoadLibraryA>
            push    hash
            push    [ebp+kernel32]
            call    getprocaddress
            mov     [ebp+LoadLibraryA], eax

            x_push  ebx, <poly_oep.dll~>
            push    esp
            calle   LoadLibraryA
            x_pop

            x_push  ebx, <poly_oep_start~>
            push    esp
            push    eax
            call    [ebp+GetProcAddress]
            x_pop
            call    eax
```



x_push/x_pop is z0mbie's tasm32 macros to generate strings on stack, and those are not visible in disassembly until you finally generate string on stack ☺

That's it about fixing poly oep. Short but I'm not going to teach you how to rip data from IDA and make binary out of it. Heh, if you wrote any simple keygen for any crackme from www.crackmes.de then you know how to perform this, otherwise stop reading and go to www.crackmes.de, there is so much resources about this issue.

4.5. Poly OEP when it comes to Delphi application

When you fix poly oep in such way, you will still need to fix something that is specific for Delphi application:

```
CODE:00403B98      mov     eax, [edi+ebx*8]
CODE:00403B9B      inc     ebx
CODE:00403B9C      mov     ds:dword_46F640, ebx
CODE:00403BA2      test    eax, eax
CODE:00403BA4      jz      short loc_403BA8
CODE:00403BA6      call    eax
CODE:00403BA8  loc_403BA8:
CODE:00403BA8      cmp     esi, ebx
CODE:00403BAA      jg      short loc_403B98
```

It will call lots of procedures responsible for initialization of Delphi application but with ASPR we will have only one call to ASPR virtual.dll which will execute all of these pointers:

```
CODE:00CEDE28  InitExeStolenPart proc near
CODE:00CEDE28      push    ebx
CODE:00CEDE29      mov     bl, 1
CODE:00CEDE2B      cmp     ds:byte_CF29D0, 0
CODE:00CEDE32      jz      short loc_CED4E
CODE:00CEDE34      mov     eax, ds:off_CF2BF8
CODE:00CEDE39      mov     eax, [eax]
CODE:00CEDE3B      push    eax
CODE:00CEDE3C      mov     eax, ds:dword_CF29C8
CODE:00CEDE41      push    eax
CODE:00CEDE42      mov     eax, ds:dword_CF29C4
CODE:00CEDE47      push    eax
CODE:00CEDE48      call    ds:dword_CF29CC
CODE:00CEDE4E
CODE:00CEDE4E  loc_CED4E:
CODE:00CEDE4E      mov     eax, ebx
CODE:00CEDE50      pop     ebx
CODE:00CEDE51      retn
CODE:00CEDE51  InitExeStolenPart endp
```

call ds:dword_CF29CC will take you to yet another obfuscated procedure which is responsible for calling stolen "pointers" common for this is that after some calculations it will call edx, where edx



is ptr to valid code in code section and will actually loop until all procedures are not called. For this you will use debug loader, but this time you may hardcode value of call edx and log it. Make sure that you use at this point hardware breakpoints, because there is checksum check on this part of code. For me it is located at 20400C6h, but only find above procedure in virtual.dll break on it and after a little bit of tracing you will find call edx. When you get all values you may fix your dump:

```
seg000:00000000      dd 406BCCh
seg000:00000004      dd 406968h
seg000:00000008      dd 407A78h
seg000:0000000C      dd 424138h
seg000:00000010      dd 407AB0h
seg000:00000014      dd 40E9E0h
seg000:00000018      dd 4239ACh
seg000:0000001C      dd 407DB8h
seg000:00000020      dd 40E8C8h
seg000:00000024      dd 40EE00h
seg000:00000028      dd 40F804h
```

I show only a first 10 entries while there is like 100 of them, and all of them are logged via debug loader, now we have to fix our dump:

```

      xor     ebx, ebx
      mov     esi, offset ptrs
      mov     edi, 537FE4h
      sub     edi, 400000h
      add     edi, mhandle
__cycle_ptr:
      lodsd
      test    eax, eax
      jz      __nomoreptrs
      stosd
      add     edi, 4
      inc     ebx
      jmp     __cycle_ptr
__nomoreptrs:
      mov     esi, 5378b4h
      sub     esi, 400000h
      add     esi, mhandle
      mov     [esi], ebx
```

Here you see how I fix pointers with hardcoded values in my ptrfix tool, address 537FE4h is actually address where EDI in InitExe part is pointing so I simply store new pointers to that address. Remember it is `mov eax, [edi+ebx*8]` so after storing pointer you have also to add to edi + 8 and store new pointer there and do that until you finish with fixing. Of course, we have to store number of pointers in dump and that is actually what 2nd part of code is doing [`mov [esi], ebx`]. This step is only required for Delphi applications.

That's pretty much all about fixing poly oep, now is your turn to write debug loader that will automate process of fixing ☺ or do it by hand if you are crazy ☺



5. Advanced Import Protection

5.1. Normal and Advanced Import protection

I will very briefly explain how to fix this, because there are already some txts for AIP, and with a little imagination you may fix it in, let's say less, then 10min. I won't provide you with tool for fixing AIP, I will give you guidelines how to do it, as I said at the beginning, this is not tutorial for complete newbies. It is more about my approach on unpacking and reversing ASProtect.

ASProtect uses two ways of obfuscating IAT; one is infamous (why infamous? have no idea) AIP and second is normal, week, protection of IAT which consist of stealing a couple of bytes from entry of API and then redirecting jmp [] or call [] to separate buffer where you will see bytes from API and then push/ret to API.

AIP on other hand is similar to poly_oep_dispatcher, all jmp [], call [] that are being redirected are written as:

call __aip_dispatcher

and this proc looks similar to poly_oep_dispatcher except its purpose is to call API. First we will take a look at some interesting parts of virtual.dll to figure how this is done in ASPR code itself:

```
CODE:00CED6C0 import_protection proc near
CODE:00CED6C0          push    ebx
CODE:00CED6C1          push    esi
CODE:00CED6C2          push    edi
CODE:00CED6C3          push    ebp
CODE:00CED6C4          add     esp, 0FFFFFFDCh
CODE:00CED6C7          mov     ebx, eax
CODE:00CED6C9          xor     eax, eax
CODE:00CED6CB          mov     edx, [ebx+54h]
CODE:00CED6CE          test    edx, edx
CODE:00CED6D0          jz      loc_CED8E3
```

This procedure is responsible for foobaring APIs, it will use AIP or normal IAT protection depending on protection settings during application packing. We are here interested in writing call instead of jmp [] or call [] in code section itself:

```
CODE:00CED877          call     steal_bytes
CODE:00CED87C          mov     ecx, eax
CODE:00CED87E          mov     dl, byte ptr [esp+28h+var_10]
CODE:00CED882          mov     eax, ebx
CODE:00CED884          call    sub_CEDA78
CODE:00CED889          sub     eax, ebp
CODE:00CED88B          sub     eax, 5
CODE:00CED88E          inc     ebp
CODE:00CED88F          mov     [ebp+0], eax
CODE:00CED892          mov     eax, [esp+28h+var_18]
CODE:00CED896          mov     eax, [eax]
CODE:00CED898          mov     [esp+28h+var_14], eax
CODE:00CED89C          jmp     short loc_CED8AA
```



ASProtect SKE 2.3 unpacking approach

```
CODE:00CED89E AIP_iat:
CODE:00CED89E      mov     eax, [ebx+2Ch]
CODE:00CED8A1      sub     eax, ebp
CODE:00CED8A3      sub     eax, 5
CODE:00CED8A6      inc     ebp
CODE:00CED8A7      mov     [ebp+0], eax
```

At 00CED88F is for rewriting jmp [] or call [] if normal IAT protection is used and at 00CED8A7 is place where redirection to AIP is written, when we have AIP there is not much to talk, ASPR will only write call __aip_dispatcher there w/o resolving API, so you can't fish it here.

I have also bolded steal_bytes procedure which is responsible for stealing bytes from entry of original API so you might fish addresses of normal IAT protection here:

```
CODE:00CE4220      mov     eax, edi
CODE:00CE4222      call   call_lde
```

And then:

```
CODE:00CD37EC      push    eax
CODE:00CD37ED      push    offset dword_CF8FA4
CODE:00CD37F2      call   aspr_lde
CODE:00CD37F7      add     esp, 8
CODE:00CD37FA      retn
```

And you enter into aspr_lde (Length Decoding Engine):

```
CODE:00CD37FB aspr_lde      proc near
CODE:00CD37FB      pusha
CODE:00CD37FC      mov     esi, [esp+20h+arg_0]
CODE:00CD3800      mov     ecx, [esp+20h+arg_4]
CODE:00CD3804      xor     edx, edx
CODE:00CD3806      xor     eax, eax
```

Here you may fish APIs for normal IAT protection if you want too, but there is simple way of fixing Normal IAT protection.

5.2. Fixing Advanced Import Protection

In above disassembly you might see where are AIP redirections written, (you may log em), and you will know where are all call __aip_dispatcher located, also take care that with AIP is also present normal IAT protection so you will have to fix some parts of IAT that belong to normal IAT protection.



call `__aip_dispatcher` takes us to similar code as in `poly_oep` that will save registers and call `aip_dispatcher` here:

```
CODE:00CED1A0 aip_dispatcher proc near
CODE:00CED1A0      push     ebp
CODE:00CED1A1      mov      ebp, esp
CODE:00CED1A3      add      esp, 0FFFFFFD8h
CODE:00CED1A6      push     ebx
CODE:00CED1A7      push     esi
CODE:00CED1A8      push     edi
```

AIP dispatcher is responsible for loading dll, locating API in it and then simply calling API, to load dll it will use, `hmmm`, `LoadLibraryA` and API locating is implemented through custom `GetProcAddress`:

```
CODE:00CEBE28 getprocaddress_by_name proc near
CODE:00CEBE28      push     ebp
CODE:00CEBE29      mov      ebp, esp
CODE:00CEBE2B      add      esp, 0FFFFFFE4h
CODE:00CEBE2E      push     ebx
CODE:00CEBE2F      push     esi
CODE:00CEBE30      push     edi
CODE:00CEBE31      mov      [ebp+ptr_to_api_name], ecx
CODE:00CEBE34      mov      ebx, edx
CODE:00CEBE36      mov      [ebp+var_4], eax
CODE:00CEBE39      xor      eax, eax
CODE:00CEBE3B      mov      [ebp+var_C], eax
CODE:00CEBE3E      mov      eax, ebx
CODE:00CEBE40      mov      edx, [eax+3Ch]
CODE:00CEBE43      add      edx, eax
CODE:00CEBE45      mov      eax, edx
CODE:00CEBE47      mov      edx, [eax+78h]
CODE:00CEBE4A      mov      [ebp+export_table_rva], edx
CODE:00CEBE4D      mov      eax, [eax+7Ch]
CODE:00CEBE50      mov      [ebp+var_14], eax
CODE:00CEBE53      mov      eax, [ebp+export_table_rva]
CODE:00CEBE56      add      eax, ebx
CODE:00CEBE58      mov      [ebp+export_table_va], eax
CODE:00CEBE5B      mov      eax, [ebp+export_table_va]
CODE:00CEBE5E      mov      edi, [eax+20h]
CODE:00CEBE61      add      edi, ebx
CODE:00CEBE63      xor      esi, esi          ; counter
CODE:00CEBE65      jmp      loc_CEBF0D
CODE:00CEBE6A scan_names:
CODE:00CEBE6A      mov      eax, [edi]        ; rva of apiname
CODE:00CEBE6C      add      eax, ebx          ; ebx dll base
CODE:00CEBE6E      mov      [ebp+name_VA], eax
CODE:00CEBE71      xor      ecx, ecx
CODE:00CEBE73      push     ebp
CODE:00CEBE74      push     offset exception_handler
CODE:00CEBE79      push     dword ptr fs:[ecx]
CODE:00CEBE7C      mov      fs:[ecx], esp
```



```
CODE:00CEBE7F      mov     edx, [ebp+name_VA]
CODE:00CEBE82      mov     eax, [ebp+ptr_to_api_name]
CODE:00CEBE85      call    strcmp_AIP      ; compare em
CODE:00CEBE8A      test    eax, eax
CODE:00CEBE8C      jnz     short next_name
CODE:00CEBE8E      mov     eax, [ebp+arg_0]
CODE:00CEBE91      mov     eax, [eax]
CODE:00CEBE93      push    eax
CODE:00CEBE94      push    esi
CODE:00CEBE95      mov     eax, [ebp+export_table_va]
CODE:00CEBE98      push    eax
CODE:00CEBE99      push    ebx
CODE:00CEBE9A      call    ordinal_api
```

Here you find strcmp_AIP, it is Delphi procedure, but to make it nicer in my loaded dll I renamed it to strcmp_AIP, actually here you see custom implementation of GetProcAddress where it will search in export table of DLL for API name so you may fish API name here.

In my debug loader I break at LoadLibraryA to find DLL name, then I break at strcmp_AIP and grab API name, with debug loader you may easily recover whole AIP.

All my data are organized in binary file like this:

```
dd      407664h
db      "user32.dll",0
db      "MessageBoxA",0
dd      4076b4h
db      "user32.dll",0
db      "RegisterClassA", 0
```

Now I know from where is called which API and I'm able to FIX AIP very fast by injecting API loader in my dump. Fixing AIP in such way is very fast.

5.3. Fixing Normal Import Protection

For normal IAT protection fixing is even easier, let's get back to place from where aspr_lde is called:

```
CODE:00CE4220      mov     eax, edi
CODE:00CE4222      call    call_lde
CODE:00CE4227      mov     ebx, eax
CODE:00CE4229      cmp     bl, 0FFh
CODE:00CE422C      jnz     short bad_opcode
```

Heh, now take a closer look, lde will return size of instruction if size = 0FFh that is bad opcode, well the one not recognized by aspr_lde so at this point aspr will assemble:



ASProtect SKE 2.3 unpacking approach

```
push    current_pointer_in_API  
retn
```

Where `current_pointer_in_API` will take you to part of API that is not dissembled, so you solution is simple, patch LDE to return -1 and ASPR will assemble:

```
push    API_address  
retn
```

And now if you have logged where normal IAT is redirected simple fire up importrec or your debug loader can do this and trace till push/ret combo and find API, and produce some output that can be used later on with your apiloader that you will inject into last section. That's what I do.

Remember to load dumped.dll symbols in SoftICE or Olly, and your analyze of aspr will become much, much easier.



6. Conclusion

Well, good protector, but when you find your way through it, it isn't that hard. I'm not providing here my tools, the reason for that is simple, you have to do it for yourself, and you will learn a lot. Again not common methods are being used to defeat some protector.

S verom u Boga, deroko/ARTeam

All the code provided with this tutorial is free for public use, just make a greetz to the authors and the ARTeam if you find it useful to use. Don't use these concepts for making illegal operation, all the info here reported are only meant for studying and to help having a better knowledge of application code security techniques.



7. Greetings

I wish to tank all the ARTeam members for sharing their knowledge, to 29a virus writing group for one of the best e-zines, to my friends from phearless e-zine, to all my friends from Reversing Labs, to great unpackers from unpack.cn for their great work, and to all great unpackers, coders all over the world and you for reading this article ☺



<http://cracking.accessroot.com>

© ARTeam 2006