Lab 1.7: Morphine Unpacking

Unpacking of an executable packed using Morphine

Fredrik Helgesson

DV2582 - Malware Analysis
Blekinge Institute of Technology
371 79 Karlskrona
October 3 2018

1 Introduction

Morphine is a advanced packer which objective is to protect the contents of an executable. Unlike other packers this is the main objective, high create compression rates and file size reduction is not prioritized. The documentation of Morphine even claims that it adds at least 50K of junk code to obstruct analysis. Morphine also contains a polymorphic engine which slightly changes the encryption and thereby the decryption code. This makes creation of a static Morphine unpacking software much more difficult. An other feature of Morphine is that it uses its own PE loader. This makes it able to put the whole unpacked executable into the .text section of the new packed PE file. This makes subsequent packing with for example UPX possible to greater the PE file protection further.

The objective of this lab is to dynamically unpack a PE file packed with Morphine. Information about the packed executable can be seen in the table below.

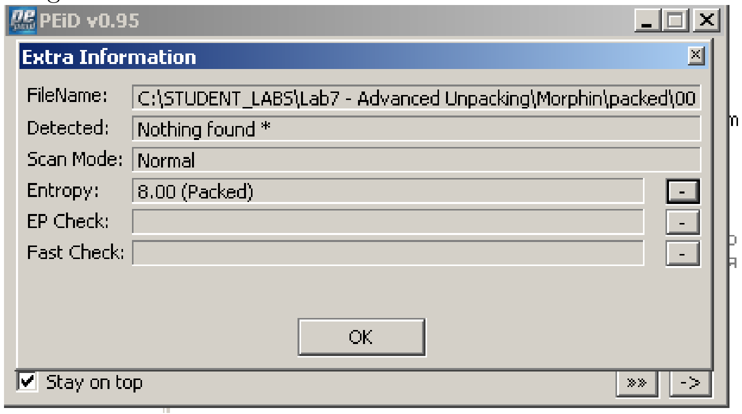
Filename	Packing Method	MD5 sum
"calc_m0.exe"	Morphine 3.0	e010abd7451328513d5a18b180c67635

2 Unpacking Morphine

The unpacking procedure contains of four steps. First of the packing method of the executable is identified. Secondly the OEP of the executable is located. The third step is to dump the unpacked from main memory to a file and reconstruct the PE header of the dumped file. The final step is focused on testing if the unpacked executable runs as intended.

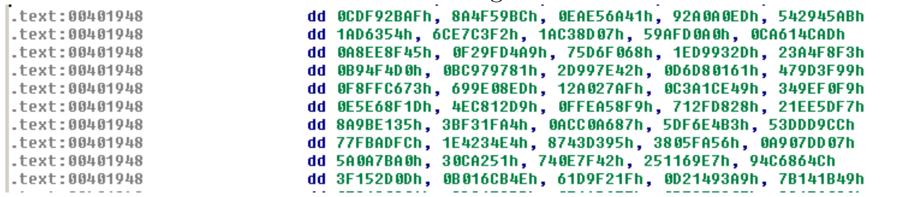
2.1 Step 1 - Packer Identification

To begin with the packing method can occasionally be automatically identified by loading the executable into PEID, the output of PEID can be seen in the image below.



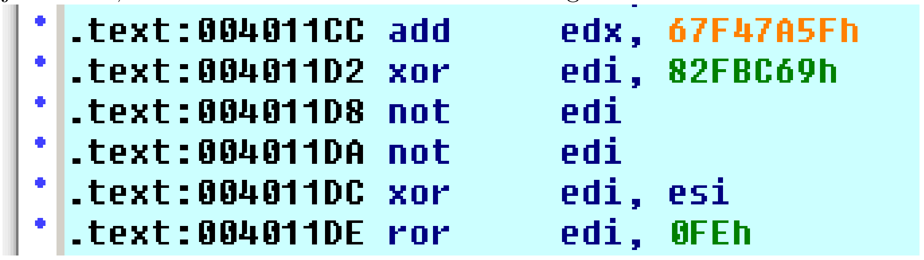
FileName: C:\STUDENT_LABS\Lab7 - Advanced Unpacking\Morphin\packed\00
Detected: Nothing found *
Scan Mode: Normal
Entropy: 8.00 (Packed)
EP Check:
Fast Check:

The image shows tha PEID does not identify which packing method is used but it does identify that the PE file is indeed packed by calculating a high entropy value which is based on signs of packing in the analyzed file. An entropy value greater than 6 signals that the analyzed file is packed. A second approach of identifying the packing method is to manually analyze the code in a dissasembler and search for signs of a certain packer. The fact that Morphine is polymorhpic obstructs this approach greatly since there is no code structure that characterizes this packer. By analyzing the dissasembled PE file in IDA Pro the only signs found that Morphine was used was that the whole code was located in the .text section which can be seen in the image below.



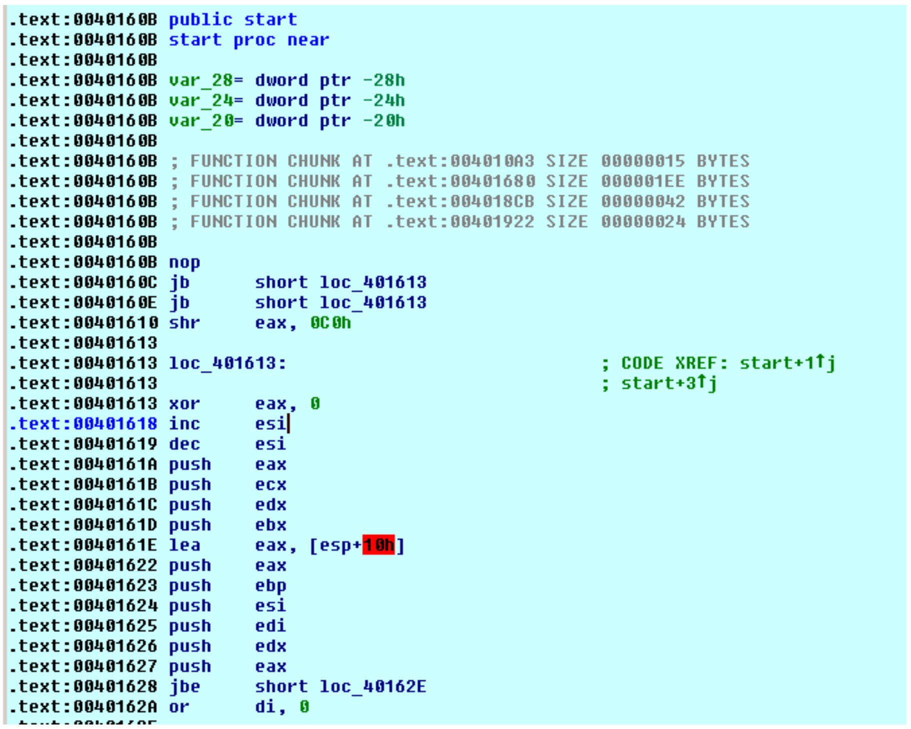
```
.text:00401948                 dd 0CDF92BAFh, 8A4F59BCh, 0EAE56A41h, 92A0A0EDh, 542945ABh
.text:00401948                 dd 1AD6354h, 6CE7C3F2h, 1AC38D07h, 59AFD0A0h, 0CA614CADh
.text:00401948                 dd 0A8EE8F45h, 0F29FD4A9h, 75D6F068h, 1ED9932Dh, 23A4F8F3h
.text:00401948                 dd 0B94F4D0h, 0BC979781h, 2D997E42h, 0D6D80161h, 479D3F99h
.text:00401948                 dd 0F8FFC673h, 699E08EDh, 12A027AFh, 0C3A1CE49h, 349EF0F9h
.text:00401948                 dd 0E5E68F1Dh, 4EC812D9h, 0FFEA58F9h, 712FD828h, 21EE5DF7h
.text:00401948                 dd 8A9BE135h, 3BF31FA4h, 0ACC0A687h, 5DF6E4B3h, 53DDD9CCh
.text:00401948                 dd 77FBADFCh, 1E4234E4h, 8743D395h, 3805FA56h, 0A907DD07h
.text:00401948                 dd 5A0A7BA0h, 30CA251h, 740E7F42h, 251169E7h, 94C6864Ch
.text:00401948                 dd 3F152D0Dh, 0B016CB4Eh, 61D9F21Fh, 0D21493A9h, 7B141B49h
```

Another sign of Morphine was that the executable contained great masses of junk code, some of which can be seen in the image below.

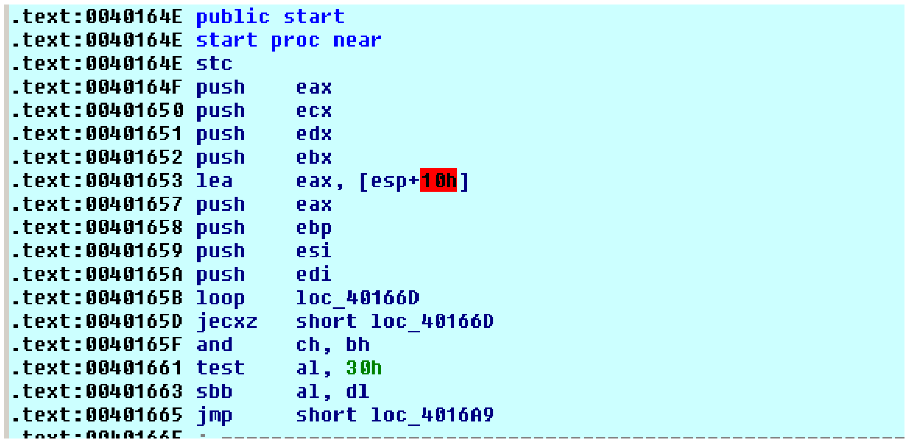


```
.text:004011CC add     edx, 67F47A5Fh
.text:004011D2 xor     edi, 82FBC69h
.text:004011D8 not     edi
.text:004011DA not     edi
.text:004011DC xor     edi, esi
.text:004011DE ror     edi, 0FEh
```

For example the "NOT" instruction reverse the bits in a byte, by doing this instruction twice in the changes are undone. The two "NOT"-instructions can be seen i the image above on address 004011D8. Having access to multiple packed instances of the same executable creates the possibility to crosscheck sections of the code to confirm that the packer is indeed using a polymorphic engine. This confirmation can be seen in the images below.



```
.text:0040160B public start
.text:0040160B start proc near
.text:0040160B
.text:0040160B var_28= dword ptr -28h
.text:0040160B var_24= dword ptr -24h
.text:0040160B var_20= dword ptr -20h
.text:0040160B
.text:0040160B ; FUNCTION CHUNK AT .text:004010A3 SIZE 00000015 BYTES
.text:0040160B ; FUNCTION CHUNK AT .text:00401680 SIZE 000001EE BYTES
.text:0040160B ; FUNCTION CHUNK AT .text:004018CB SIZE 00000042 BYTES
.text:0040160B ; FUNCTION CHUNK AT .text:00401922 SIZE 00000024 BYTES
.text:0040160B
.text:0040160B nop
.text:0040160C jb      short loc_401613
.text:0040160E jb      short loc_401613
.text:00401610 shr     eax, 0C0h
.text:00401613
.text:00401613 loc_401613:                             ; CODE XREF: start+1↑j
.text:00401613                                         ; start+3↑j
.text:00401613 xor     eax, 0
.text:00401618 inc     esi
.text:00401619 dec     esi
.text:0040161A push    eax
.text:0040161B push    ecx
.text:0040161C push    edx
.text:0040161D push    ebx
.text:0040161E lea     eax, [esp+10h]
.text:00401622 push    eax
.text:00401623 push    ebp
.text:00401624 push    esi
.text:00401625 push    edi
.text:00401626 push    edx
.text:00401627 push    eax
.text:00401628 jbe     short loc_40162E
.text:0040162A or      di, 0
```

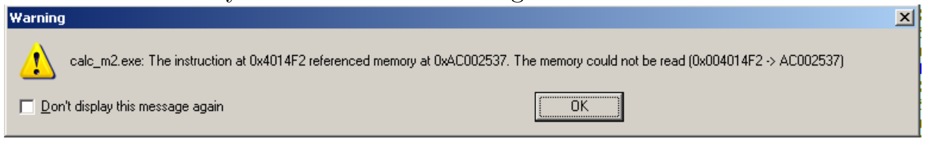


```
.text:0040164E public start
.text:0040164E start proc near
.text:0040164E stc
.text:0040164F push    eax
.text:00401650 push    ecx
.text:00401651 push    edx
.text:00401652 push    ebx
.text:00401653 lea     eax, [esp+10h]
.text:00401657 push    eax
.text:00401658 push    ebp
.text:00401659 push    esi
.text:0040165A push    edi
.text:0040165B loop    loc_40166D
.text:0040165D jecxz   short loc_40166D
.text:0040165F and     ch, bh
.text:00401661 test    al, 30h
.text:00401663 sbb     al, dl
.text:00401665 jmp     short loc_4016A9
```

The top image shows the "start"-function of the analyzed packed file "calc_m0.exe". The bottom image shows the same function in another packed instance of the same executable, "calc_m1.exe". If the packer was not using a polymorphic engine these two images would be identical. These differences are one of the reasons why static identification and unpacking of Morphine packed executables proves so difficult.

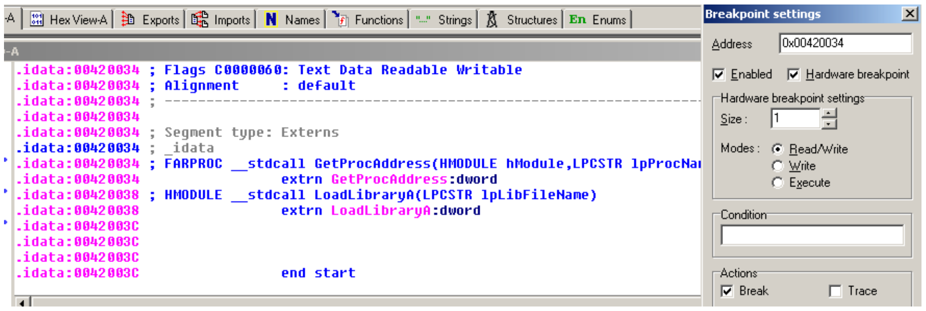
2.2 Step 2 - Locating OEP

The OEP was found by dynamically analyzing the PE file in the debugger IDA Pro. The first thing to notice when trying to debug the program is that the executable contains several protection features such as junk code and anti-debugging features. The anti-debugging features are apparent when stepping through the code and the execution are sticking in infinite loops or the program crashes because it cannot access memory space that is required for the execution to proceed. The crash error signaling that the program is trying to reach inaccessible memory can be seen in the image below.



```
Warning
calc_m2.exe: The instruction at 0x4014F2 referenced memory at 0xAC002537. The memory could not be read (0x004014F2 -> AC002537)
```

To bypass some of the anti-debugging features is by using hardware breakpoints instead of the software breakpoints used in the initial debugging of the executable. The hardware was placed on the imported function "GetProcAddress" with read/write permissions. The configuration and placement of the breakpoint can be seen in the image below.



```
.idata:00420034 ; Flags C0000060: Text Data Readable Writable
.idata:00420034 ; Alignment     : default
.idata:00420034 ; ---------------------------------------------------------------------------
.idata:00420034
.idata:00420034 ; Segment type: Externs
.idata:00420034 ; _idata
.idata:00420034 ; FARPROC __stdcall GetProcAddress(HMODULE hModule,LPCSTR lpProcNa
.idata:00420034                 extrn GetProcAddress:dword
.idata:00420038 ; HMODULE __stdcall LoadLibraryA(LPCSTR lpLibFileName)
.idata:00420038                 extrn LoadLibraryA:dword
.idata:0042003C
.idata:0042003C
.idata:0042003C
.idata:0042003C                 end start
```

Breakpoint settings

Address: 0x00420034

[✓] Enabled [✓] Hardware breakpoint

Hardware breakpoint settings

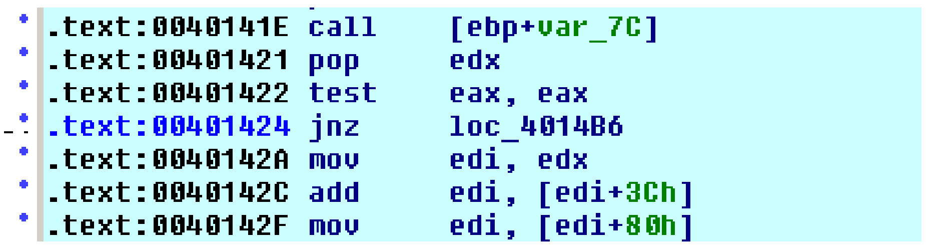
Size: 1

Modes: [● Read/Write] ○ Write ○ Execute

Condition:

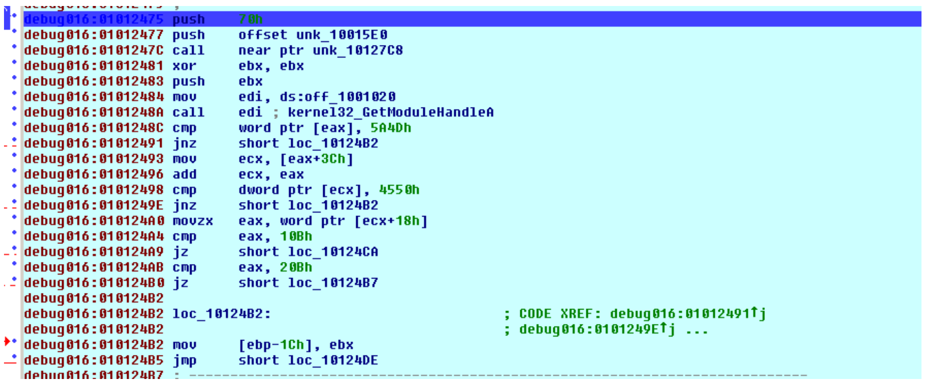
Actions: [✓] Break [ ] Trace

The reason behind the placement is that the unpacker usually calls "GetProcAddress" before the executable is unpacked to make sure that the executable has access to all imported libraries needed. The goal of using the breakpoint is to pass some of the junk code that Morphine is known to place in the beginning of the executable. The next step is to stop the program execution when "GetProcAddress" is called for the last time which means that all the libraries are imported. After this point the program is stepped through manually, the reason behind this is to be able to skip calls to functions in imported libraries and skip some of the loops. Before a new loop was skipped a hardware breakpoint was placed just before it to make sure that if the program finished its execution during the loop or the debugger crashed it would be a simple task to return to the section of the code just before the crash or finish. By performing this procedure multiple times until the OEP was found. The OEP was found during a loop on a jump to the label seen in the image below.



```
.text:0040141E call    [ebp+var_7C]
.text:00401421 pop     edx
.text:00401422 test    eax, eax
.text:00401424 jnz     loc_4014B6
.text:0040142A mov     edi, edx
.text:0040142C add     edi, [edi+3Ch]
.text:0040142F mov     edi, [edi+80h]
```

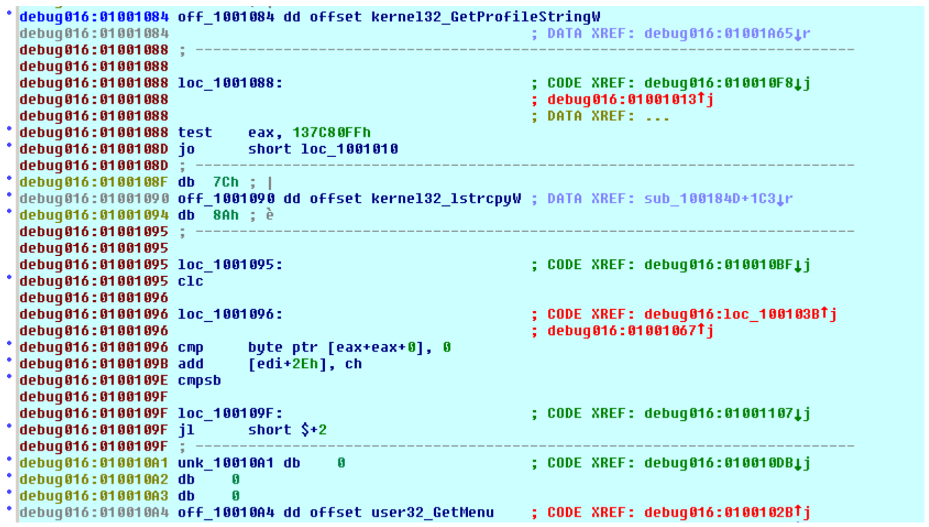
This jump lead to the OEP which can be seen in the image below on address 01012475.



```
debug016:01012475 push    70h
debug016:01012477 push    offset unk_10015E0
debug016:0101247C call    near ptr unk_10127C8
debug016:01012481 xor     ebx, ebx
debug016:01012483 push    ebx
debug016:01012484 mov     edi, ds:off_1001020
debug016:0101248A call    edi ; kernel32_GetModuleHandleA
debug016:0101248C cmp     word ptr [eax], 5A4Dh
debug016:01012491 jnz     short loc_10124B2
debug016:01012493 mov     ecx, [eax+3Ch]
debug016:01012496 add     ecx, eax
debug016:01012498 cmp     dword ptr [ecx], 4550h
debug016:0101249E jnz     short loc_10124B2
debug016:010124A0 movzx   eax, word ptr [ecx+18h]
debug016:010124A4 cmp     eax, 10Bh
debug016:010124A9 jz      short loc_10124CA
debug016:010124AB cmp     eax, 20Bh
debug016:010124B0 jz      short loc_10124B7
debug016:010124B2
debug016:010124B2 loc_10124B2:                            ; CODE XREF: debug016:01012491↑j
debug016:010124B2                                         ; debug016:0101249E↑j ...
debug016:010124B2 mov     [ebp-1Ch], ebx
debug016:010124B5 jmp     short loc_10124DE
```

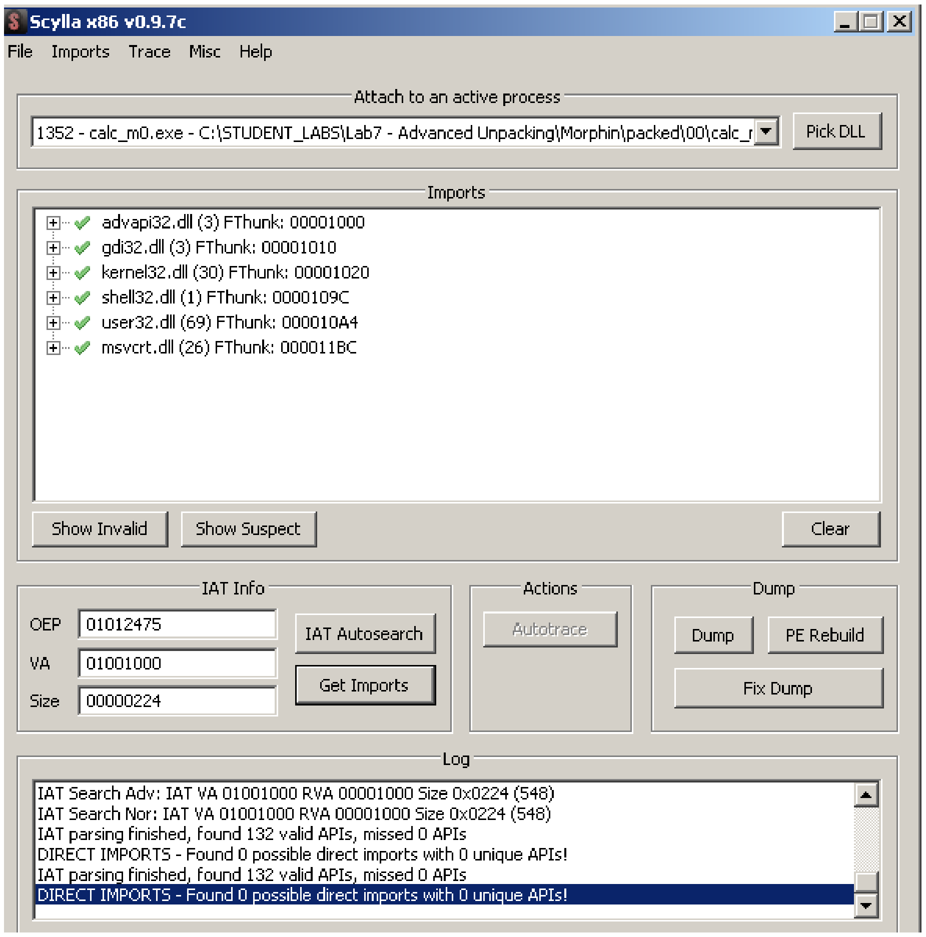
2.3 Step 3 - Memory dump & header reconstruction

At this point in the debugging of the program the value of the instruction pointer is equal to the address of the OEP. This means that at this point the unpacked executable is loaded into the process's main memory space. By using the software PEtools the memory of the process was dumped into a file named "Dumped.exe". This file now contains the complete unpacked executable but requires a correct PE header to be able to run. To recreate a valid PE header for "Dumped.exe" a list of all imports done by the executable are required. Since the executable is unpacked at this point the imports are listed as unobfuscated strings in the .text section of the code, this list is called the IAT. Three of the many imports used by the program can be seen in the image below.



```
debug016:01001084 off_1001084 dd offset kernel32_GetProfileStringW
debug016:01001084                                         ; DATA XREF: debug016:01001A65↓r
debug016:01001088 ; ---------------------------------------------------------------------------
debug016:01001088
debug016:01001088 loc_1001088:                            ; CODE XREF: debug016:010010F8↓j
debug016:01001088                                         ; debug016:01001013↑j
debug016:01001088                                         ; DATA XREF: ...
debug016:01001088 test    eax, 137C80FFh
debug016:0100108D jo      short loc_1001010
debug016:0100108D ; ---------------------------------------------------------------------------
debug016:0100108F db  7Ch ; |
debug016:01001090 off_1001090 dd offset kernel32_lstrcpyW ; DATA XREF: sub_100184D+1C3↓r
debug016:01001094 db  8Ah ; è
debug016:01001095 ; ---------------------------------------------------------------------------
debug016:01001095
debug016:01001095 loc_1001095:                            ; CODE XREF: debug016:010010BF↓j
debug016:01001095 clc
debug016:01001096
debug016:01001096 loc_1001096:                            ; CODE XREF: debug016:loc_100103B↑j
debug016:01001096                                         ; debug016:01001067↑j
debug016:01001096 cmp     byte ptr [eax+eax+0], 0
debug016:0100109B add     [edi+2Eh], ch
debug016:0100109E cmpsb
debug016:0100109F
debug016:0100109F loc_100109F:                            ; CODE XREF: debug016:01001107↓j
debug016:0100109F jl      short $+2
debug016:0100109F ; ---------------------------------------------------------------------------
debug016:010010A1 unk_10010A1 db    0                     ; CODE XREF: debug016:010010DB↓j
debug016:010010A2 db    0
debug016:010010A3 db    0
debug016:010010A4 off_10010A4 dd offset user32_GetMenu    ; CODE XREF: debug016:0100102B↑j
```

The header is reconstructed by the software Scylla. By inputting the OEP of the executable Scylla was able to automatically identify the start of the IAT and thereby all the imported libraries. Using this information Scylla was able to reconstruct the PE header. The constructed PE header is then appended to the file ”Dumped.exe” and the new file is named ”Dumped_SCY.exe”. The imports identified by Scylla can be seen in the image below.



```
1352 - calc_m0.exe - C:\STUDENT_LABS\Lab7 - Advanced Unpacking\Morphin\packed\00\calc_r
```

```
advapi32.dll (3) FThunk: 00001000
gdi32.dll (3) FThunk: 00001010
kernel32.dll (30) FThunk: 00001020
shell32.dll (1) FThunk: 0000109C
user32.dll (69) FThunk: 000010A4
msvcrt.dll (26) FThunk: 000011BC
```

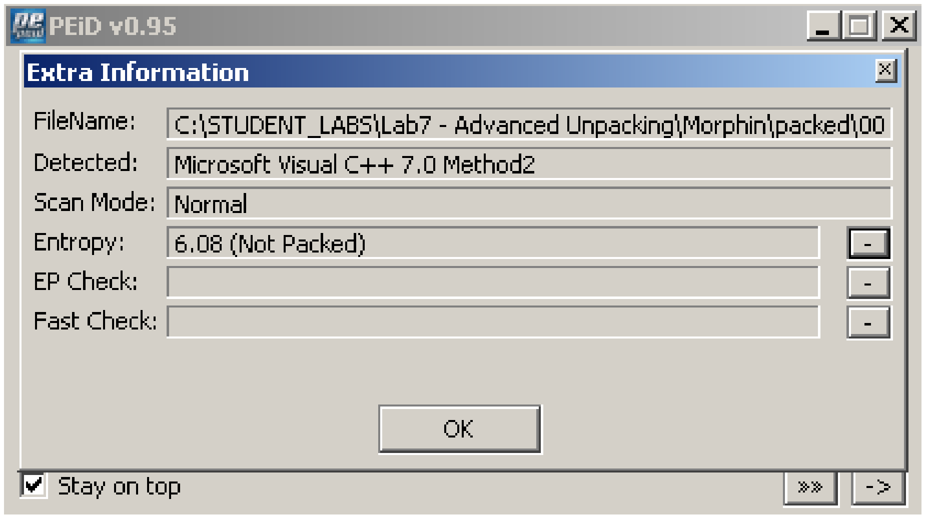
```
OEP  01012475
VA   01001000
Size 00000224
```

```
IAT Search Adv: IAT VA 01001000 RVA 00001000 Size 0x0224 (548)
IAT Search Nor: IAT VA 01001000 RVA 00001000 Size 0x0224 (548)
IAT parsing finished, found 132 valid APIs, missed 0 APIs
DIRECT IMPORTS - Found 0 possible direct imports with 0 unique APIs!
IAT parsing finished, found 132 valid APIs, missed 0 APIs
DIRECT IMPORTS - Found 0 possible direct imports with 0 unique APIs!
```

At this point "Dumped_SCY.exe" is a complete unpacked version of the executable "calc_m0.exe" with a reconstructed PE header.

2.4 Step 4 - Testing

The final step is to test if the unpacked file runs as intended. The first part of testing the unpacked executable is by loading it into PEID to confirm that the PE analyzing tool does not detect any packing method nor assigns a entropy value above 6 for the executable. The result from loading the "Dumped_SCY.exe" into PEID can be seen in the image below.


PEiD v0.95

Extra Information

FileName: C:\STUDENT_LABS\Lab7 - Advanced Unpacking\Morphin\packed\00
Detected: Microsoft Visual C++ 7.0 Method2
Scan Mode: Normal
Entropy: 6.08 (Not Packed)
EP Check:
Fast Check:

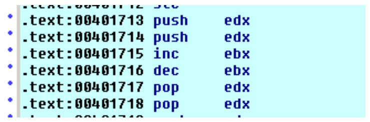
The result suggests that the executable is no longer packed since unlike the result for the packed file, the software now detects the compiler used for the program and assigns the program a lower entropy value. The second test is done by running the unpacked executable to confirm that the program runs and works as intended. The result of running the executable can be seen below and confirms the success of unpacking ”calc_m0.exe”. The table below shows the packed file ”calc_m0.exe”, the memory dump ”Dumped”, the unpacked file with a reconstructed header ”Dumped_SCY.exe” and their respective file sizes.

Filename	**File Size (K)**	**MD5 sum**
”calc_m0.exe”	141	e010abd7451328513d5a18b180c67635
”Dumped.exe”	124	a84ea38e1d3d391671e85baa989b1dda
”Dumped_SCY.exe”	116	d8da155e69fd0afee7d4e7d142ae019a

The reason that even though the file ”calc_.exe” is compressed it has much greater file size than the unpacked executable ”Dumper_SCY.exe” is that the packed file is extended by around 50K of junk code according to the creators of the Morphine packer.

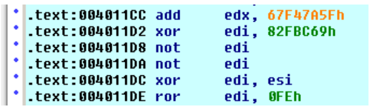
3 Questions & Answers

1. *What obvious junk code (meaningless) sequences do you observe in the polymorphic junk code?*
   According to the creators of Morphine the packer inserts around 50K of junk code, some of which can be seen in the imagee below where the code executes some instructions just to undo them after.



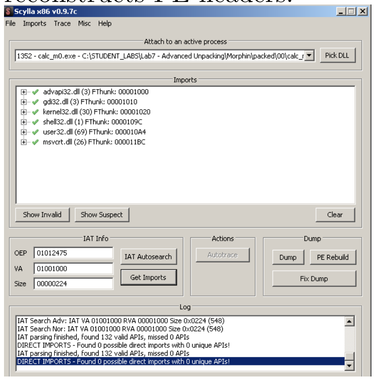
```
.text:00401713 push    edx
.text:00401714 push    edx
.text:00401715 inc     ebx
.text:00401716 dec     ebx
.text:00401717 pop     edx
.text:00401718 pop     edx
```

Another example is the pair of "NOT"-instructions seen in the image below where the second instruction reverts the actions done by the first.



```
.text:004011CC add     edx, 67F47A5Fh
.text:004011D2 xor     edi, 82FBC69h
.text:004011D8 not     edi
.text:004011DA not     edi
.text:004011DC xor     edi, esi
.text:004011DE ror     edi, 0FEh
```

2. *What is the aim of import restoration procedure? What modules are imported by the protected executable binary?*
   The objective of the import restoration procedure is to reconstruct a PE header which can be appended to the unpacked executable dumped from the process's main memory. This header must contain all imported functions which can be found in the protected executable binary during its unpacked state. The imported modules can be seen in the image below which is the output from the software "Scylla" which similar to "ImpRec" reconstructs PE headers.



1352 - calc_m0.exe - C:\STUDENT_LABS\Lab7 - Advanced Unpacking\Morphin\packed\00\calc_r

```
advapi32.dll (3) FThunk: 00001000
gdi32.dll (3) FThunk: 00001010
kernel32.dll (30) FThunk: 00001020
shell32.dll (1) FThunk: 0000109C
user32.dll (69) FThunk: 000010A4
msvcrt.dll (26) FThunk: 000011BC
```

OEP 01012475
VA 01001000
Size 00000224

```
IAT Search Adv: IAT VA 01001000 RVA 00001000 Size 0x0224 (548)
IAT Search Nor: IAT VA 01001000 RVA 00001000 Size 0x0224 (548)
IAT parsing finished, found 132 valid APIs, missed 0 APIs
DIRECT IMPORTS - Found 0 possible direct imports with 0 unique APIs!
IAT parsing finished, found 132 valid APIs, missed 0 APIs
DIRECT IMPORTS - Found 0 possible direct imports with 0 unique APIs!
```

3. *What concrete approach would you suggest to identify Original Entry Point for the protected binary?*
   I suggest that the entire debugging session is done using hardware breakpoints since the packed executable use multiple ways of anti-debugging. By using a breakpoint on the functioncall to "GetProcAddress" most of the junk code in the start of the binary can be skipped. After this call is made for the last time step through the code and try to skip loops and function calls. Before skipping a loop or chunks of instructions place a breakpoint on the instruction pointer. This is a safety measure to ensure that you can return to the sama location in the execution if the debugger crashes or finishes. Another way of saving the current state of execution is making a virtual machine snapshot before skipping chunks of code.