

Lab 1.6: Simple Unpacking

Unpacking of executables

Fredrik Helgesson



DV2582 - Malware Analysis
Blekinge Institute of Technology
371 79 Karlskrona
October 2 2018

1 Introduction

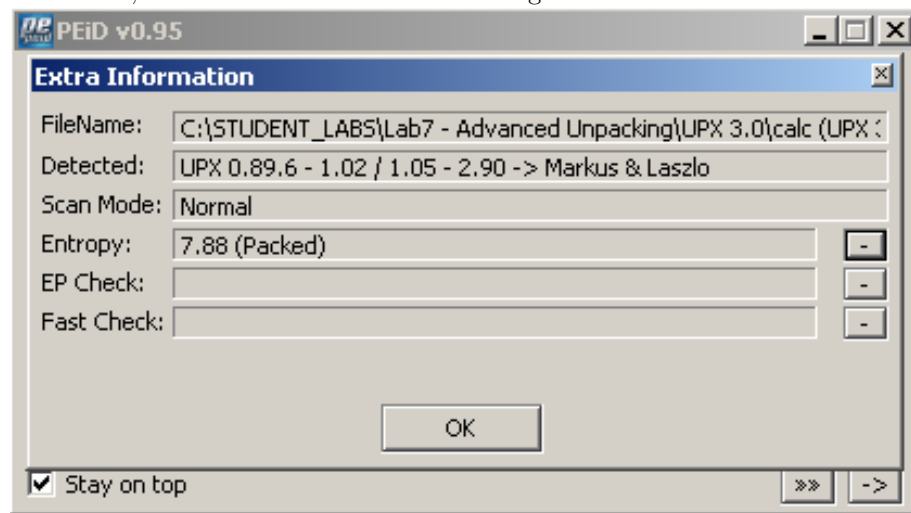
The objective of packing is to compress an executable by attaching an unpacked to the executable body which is used to decompress the executable during execution. Packing an executable is used for multiple reasons, simply reducing the file size, obstructing reverse engineering and making signature analysis more difficult. Unpacking a packed executable can be done either statically using external software or dynamically by running and analyzing the operation of the executable during the unpacking stage.

During this analysis four different packed executable files were unpacked using the dissassembler/debugger IDA Pro, PE Tools and ImpRec. The analyzed files are listed in the table below.

Filename	Packing Method	MD5 sum
calc (UPX 3.00).exe	UPX 3.0	a72e9f548fa99dc37817aae776afe42f
notepad (WinUpack 0.31).exe	WinUpack 0.31	d76dcb30c8eb9fed956ab506d62a73fe
calc (ASPack 2.12).exe	AsPack 2.12	6b1ab7c5b0b23d64615093d3e76a2777
calc (FSG 2.0).exe	FSG 2.0	5a373276245e8d182e67cb561ea292a5

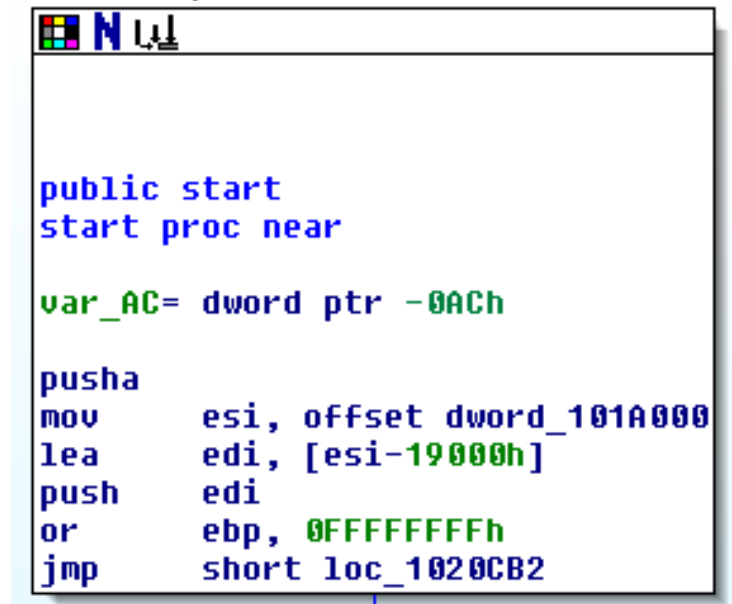
2 Unpacking UPX 3.0

To begin with the packing method was determined by loading the executable into PEID, the result can be seen in the image below.



The image shows that executable is packed using UPX and the entropy also implies that the executable is packed. The first task in the procedure to unpack "calc (UPX 3.00).exe" is to find the original entry point (OEP) of the executable. This is the point where the unpacking procedure of the executable is finished and the unpacked program is loaded into the primary memory. To

find the OEP the executable is loaded into the dissassembler and debugger IDA Pro. The first thing to notice in the assembler code is the "pusha"-instruction seen in the image below.



```

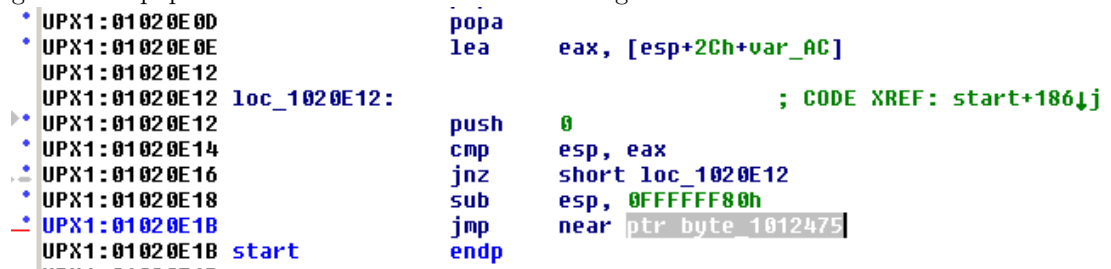
public start
start proc near

var_AC= dword ptr -0ACh

pusha
mov     esi, offset dword_101A000
lea     edi, [esi-19000h]
push    edi
or      ebp, 0FFFFFFFh
jmp     short loc_1020CB2

```

This instruction is often used before the unpacking process begins to make sure that no registers are changed after the procedure is completed. The instruction suggests that the instruction "popa" will be used after the unpacking procedure is finished to reset the registers as they were before the unpacking process began. The "popa" instruction can be seen in the image below.



```

UPX1:01020E0D      popa
UPX1:01020E0E      lea     eax, [esp+2Ch+var_AC]
UPX1:01020E12
UPX1:01020E12 loc_1020E12:                                ; CODE XREF: start+186↓j
UPX1:01020E12      push    0
UPX1:01020E14      cmp     esp, eax
UPX1:01020E16      jnz     short loc_1020E12
UPX1:01020E18      sub     esp, 0FFFFFF80h
UPX1:01020E1B      jmp     near ptr byte 1012475
UPX1:01020E1B      start  endp

```

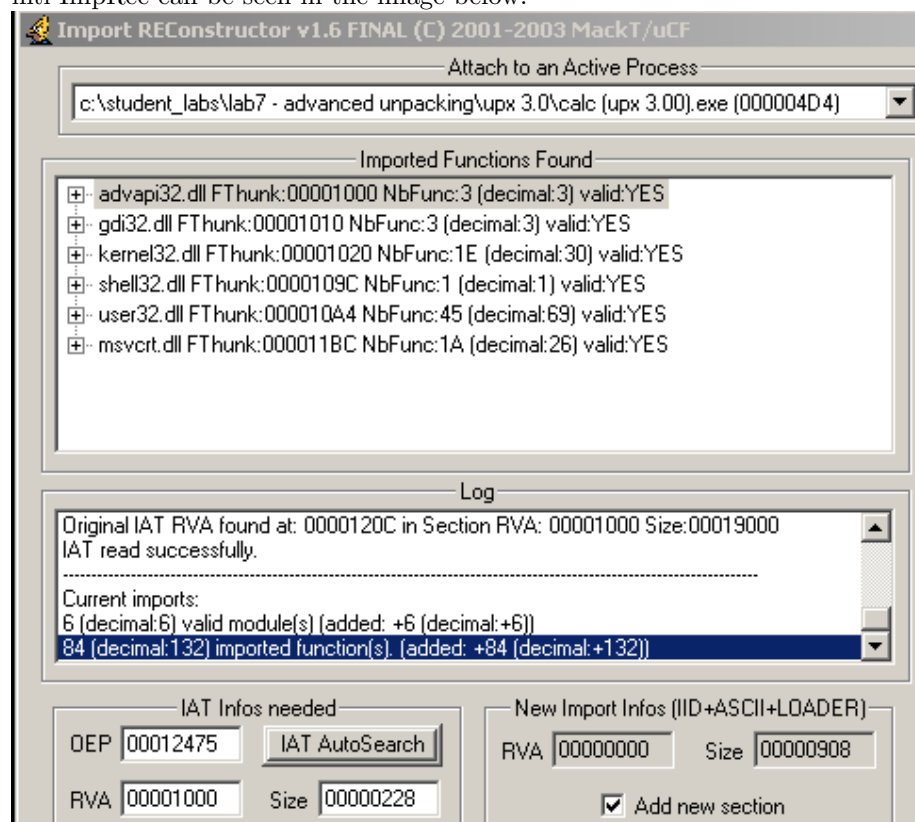
The highlighted text in the image shows a jump to a specified location. Since the unpacking is just finished according to the theory behind "popa", this jump suggests that the program is now jumping to the OEP, the starting point of the executable before it was packed. By setting a breakpoint at the location the jump is headed to and running the program through the debugger the program stops just after the unpacking is finished and right before the unpacked program which is now loaded into main memory is executed. The OEP can be seen in the image below at address 01012475.

```

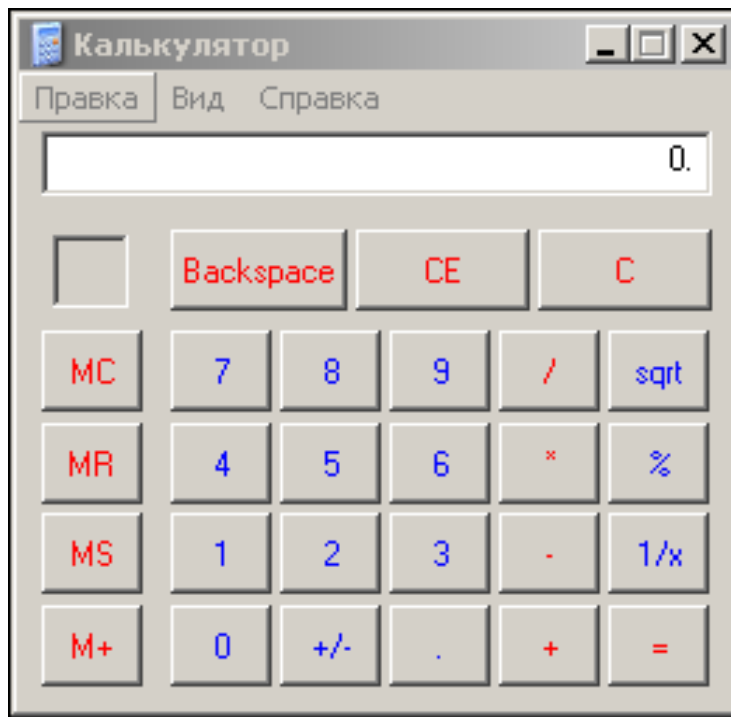
UPX0:01012475 loc_1012475:                                ; CODE XREF: start+18B↓j
UPX0:01012475 push 70h
UPX0:01012477 db 68h ; h
UPX0:01012478 dd 400h dup(10015E0h), 1AE2h dup(0)
UPX0:01012478 UPX0 ends

```

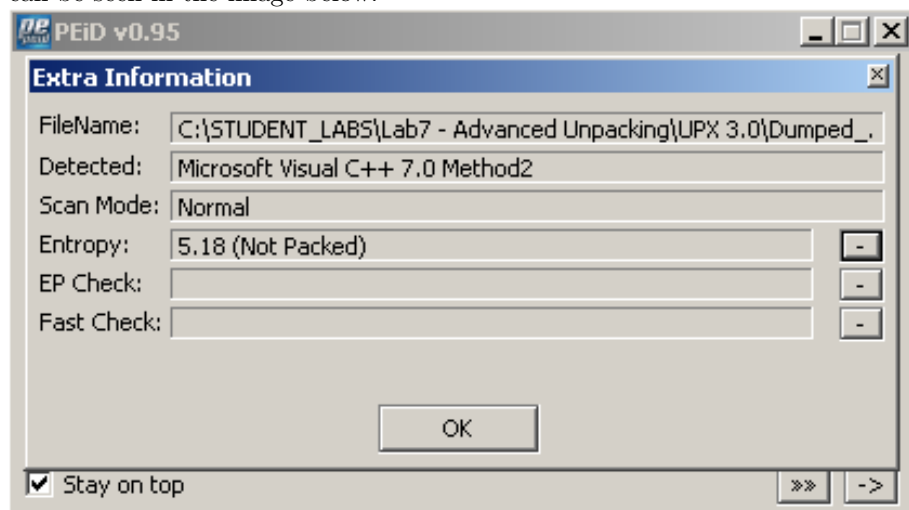
The unpacked program which at this point is loaded into main memory can now be dumped into a file named "Dumped.exe" using the software PEtools. The next step is to reconstruct the PE header of the dumped file. The reconstruction is done using the software ImpRec. By entering the discovered OEP into ImpRec the header was reconstructed. One sign that the OEP was correct is that ImpRec now identifies several imports done by the executable which were previously compressed and thereby hidden. The imports and the OEP inserted into ImpRec can be seen in the image below.



The reconstructed header was thereafter applied to the file "Dumped.exe" created by PEtools. The resulting file was named "Dumped.exe" and contains the unpacked executable. The success of the unpacking was tested in two ways, firstly by running "Dumped.exe" to confirm that the unpacked executable was still working as intended. The running program can be seen in the image below.



The second test was performed by loading "Dumped.exe" into PEID to confirm that no packing method was identified and that the entropy value was lower than 6 which suggests that the executable is not packed. The result from PEID can be seen in the image below.



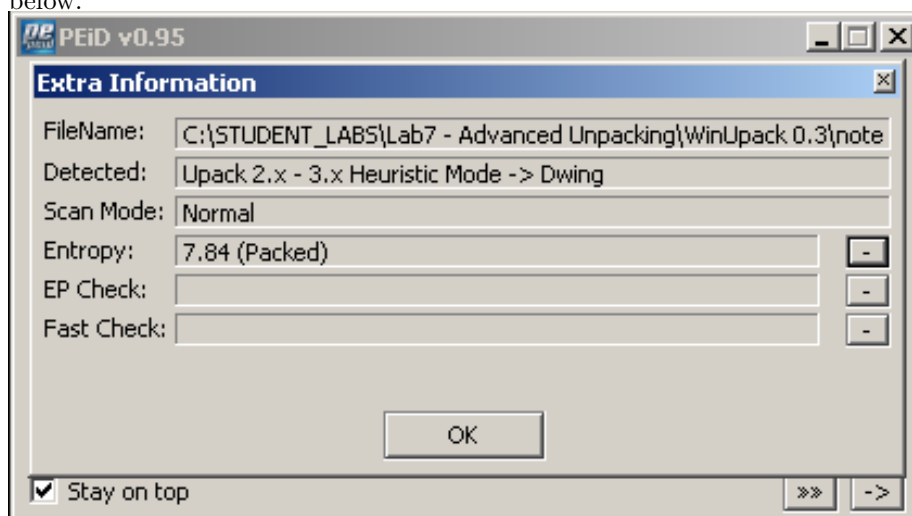
In the table below the sizes of the different files along the analysis process are listed.

Filename	File Size (K)	MD5 sum
"calc (UPX 3.00).exe"	56	a72e9f548fa99dc37817aae776afe42f
"Dumped.exe"	188	1dd40c404675f40f5508979fa7e7c6f3
"Dumped_.exe"	192	392773c23247fd46f6f45712bc9ee036

The reason of the different sizes is that when the file "calc (UPX 3.00).exe" is compressed and therefore has a much smaller size than "Dumped.exe" which is uncompressed. The file "Dumped_.exe" is slightly bigger than "Dumped.exe" because it contains the same data as "Dumped.exe" but with a reconstructed PE header appended to the start of the file.

3 Unpacking WinUPack

Firstly the executable was loaded into PEID to determine what packing method was used and the entropy value of the file. The result can be seen in the image below.



The image shows that PEID identifies the packing method as "Upack" which is short for "WinUPack" and calculates the entropy value as 7.84. The next step in the process of unpacking the executable is to locate the OEP. To do this the executable had to be ran through the debugger IDA Pro with a breakpoint at the starting function. The process of finding the OEP was to step through the instruction and identifying each function call. The goal behind the search is to find the function calls "LoadLibraryA" and "GetProcAddress" which are both imported from the Kernel32 library. These are the only identifiable imports when statically analyzing the packed executable. The calls to the mentioned functions can be seen in the image below where the top call on address 0101FE84 is for the function "LoadLibraryA" and the bottom one on

address 0101FE9C for "GetProcAddress".

```

notepad_(WinUpack_0.31).exe:0101FE78 loc_101FE78: ; CODE XREF: notepad_(WinUpack_0.31).exe:0101FE8F↓j
notepad_(WinUpack_0.31).exe:0101FE78 inc esi
notepad_(WinUpack_0.31).exe:0101FE79 lodsd
notepad_(WinUpack_0.31).exe:0101FE7A test eax, eax
notepad_(WinUpack_0.31).exe:0101FE7C jz near ptr dword_1001148+6255h
notepad_(WinUpack_0.31).exe:0101FE82 push esi
notepad_(WinUpack_0.31).exe:0101FE83 xchg eax, edi
notepad_(WinUpack_0.31).exe:0101FE84 call dword ptr [ebx-4]
notepad_(WinUpack_0.31).exe:0101FE87 xchg eax, ebp
notepad_(WinUpack_0.31).exe:0101FE88 loc_101FE88: ; CODE XREF: notepad_(WinUpack_0.31).exe:0101FE8B↓j
notepad_(WinUpack_0.31).exe:0101FE88 lodsb ; notepad_(WinUpack_0.31).exe:0101FE9F↓j
notepad_(WinUpack_0.31).exe:0101FE89 test al, al
notepad_(WinUpack_0.31).exe:0101FE8B jnz short loc_101FE88
notepad_(WinUpack_0.31).exe:0101FE8D cmp [esi], al
notepad_(WinUpack_0.31).exe:0101FE8F jz short loc_101FE78
notepad_(WinUpack_0.31).exe:0101FE91 mov eax, esi
notepad_(WinUpack_0.31).exe:0101FE93 jns short loc_101FE9A
notepad_(WinUpack_0.31).exe:0101FE95 inc esi
notepad_(WinUpack_0.31).exe:0101FE96 xor eax, eax
notepad_(WinUpack_0.31).exe:0101FE98 lodsw
notepad_(WinUpack_0.31).exe:0101FE9A loc_101FE9A: ; CODE XREF: notepad_(WinUpack_0.31).exe:0101FE93↑j
notepad_(WinUpack_0.31).exe:0101FE9A push eax
notepad_(WinUpack_0.31).exe:0101FE9B push ebp
notepad_(WinUpack_0.31).exe:0101FE9C call dword ptr [ebx]
notepad_(WinUpack_0.31).exe:0101FE9E stosd
notepad_(WinUpack_0.31).exe:0101FE9F jmp short loc_101FE88

```

The reason why these function calls are interesting for finding the OEP is that during the unpacking the program has to load all libraries that are imported in the executable. After this process is finished the program is usually loaded into main memory in an unpacked state and ready to be executable and thereby jumping into the start of the executable code. In the middle of the image this jump occurs, a zoomed in image of the jump can be seen below.

```

notepad_(WinUpack_0.31).exe:0101FE79 lodsd
notepad_(WinUpack_0.31).exe:0101FE7A test eax, eax
notepad_(WinUpack_0.31).exe:0101FE7C jz near ptr dword_1001148+6255h

```

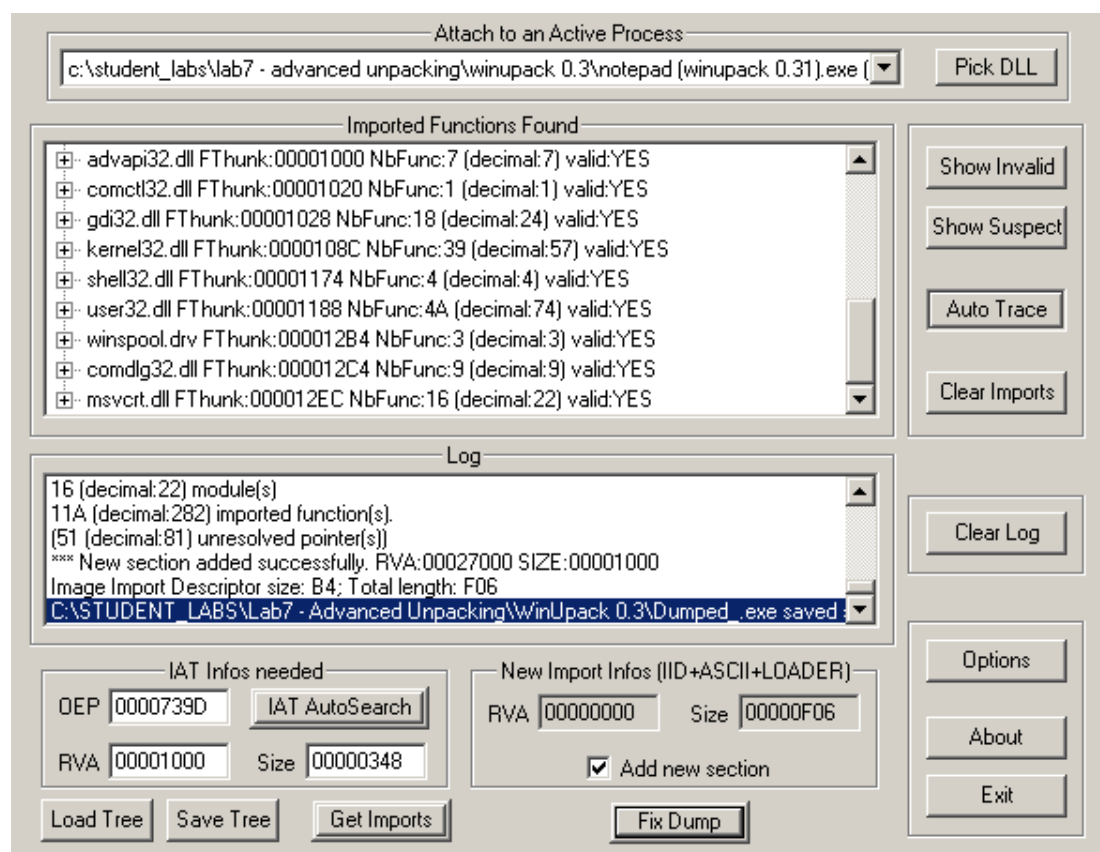
To be noted is that the three lines of instructions in the image above are standard in WinUPack when jumping to the OEP [3]. By stepping into the address that the jump points to the OEP can be seen. The start of the unpacked executable together with the address of the OEP, 0100739D, can be seen in the image below.

```

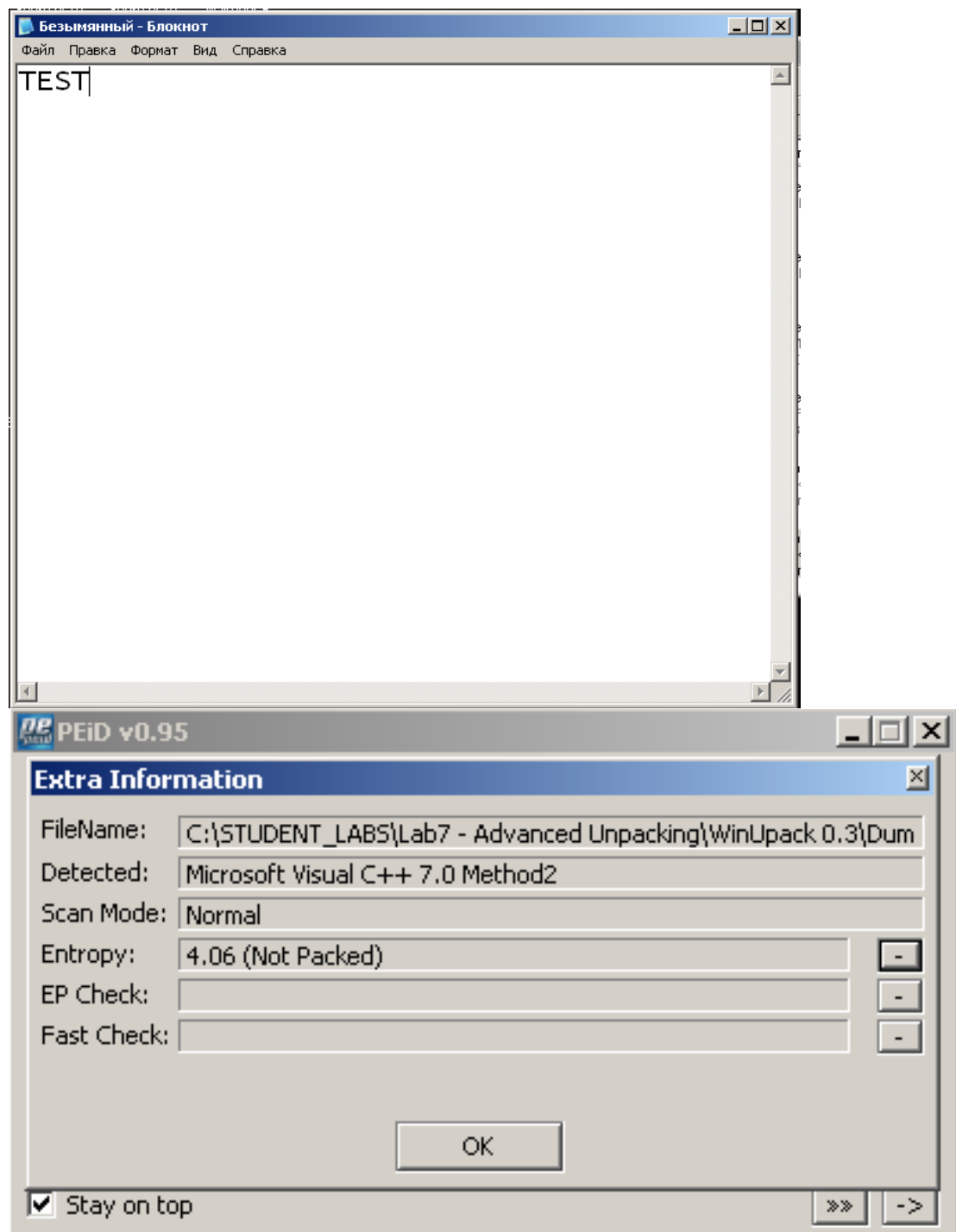
.Upack:0100739D push 70h ; CODE XREF: notepad_(WinUpack_0.31).exe:0101FE7C↓j
.Upack:0100739F push offset dword_1001898
.Upack:010073A4 call sub_1007568
.Upack:010073A9 xor ebx, ebx
.Upack:010073AB push ebx
.Upack:010073AC mov edi, dword ptr unk_10010CC
.Upack:010073B2 call edi ; kernel32_GetModuleHandleA
.Upack:010073B4 cmp word ptr [eax], 5A40h
.Upack:010073B9 jnz short loc_10073DA
.Upack:010073BB mov ecx, [eax+3Ch]
.Upack:010073BE add ecx, eax
.Upack:010073C0 cmp dword ptr [ecx], 4550h
.Upack:010073C6 jnz short loc_10073DA
.Upack:010073C8 movzx eax, word ptr [ecx+18h]
.Upack:010073CC cmp eax, 10Bh
.Upack:010073D1 jz short loc_10073F2
.Upack:010073D3 cmp eax, 20Bh
.Upack:010073D8 jz short loc_10073DF
.Upack:010073DA loc_10073DA: ; CODE XREF: .Upack:010073B9↑j

```

The OEP is tested in the same manner as during the unpacking of the UPX file above. The first step was to dump the main memory space of the process into the file "Dumped.exe". The second step was to load the process into ImpRec to reconstruct the PE-header. This required the discovered OEP to be manually inserted. One sign that the OEP was correct is that ImpRec identified several new imports which were hidden in the packed file. The output of ImpRec can be seen in the image below.



The reconstructed PE header was thereafter appended to the extracted "Dumped.exe" and the resulting file was named "Dumped.exe". To test if the unpacking was successful "Dumped.exe" was ran to make sure that it was running in the same way as the unpacked file, "Dumped.exe" was also loaded into PEID to confirm that PEID no longer could identify a packing method and that the entropy value was lower than 6. The results of the tests shows that the unpacking was successful and can be seen in the images below.



In the table below the sizes of the different files along the analysis process are

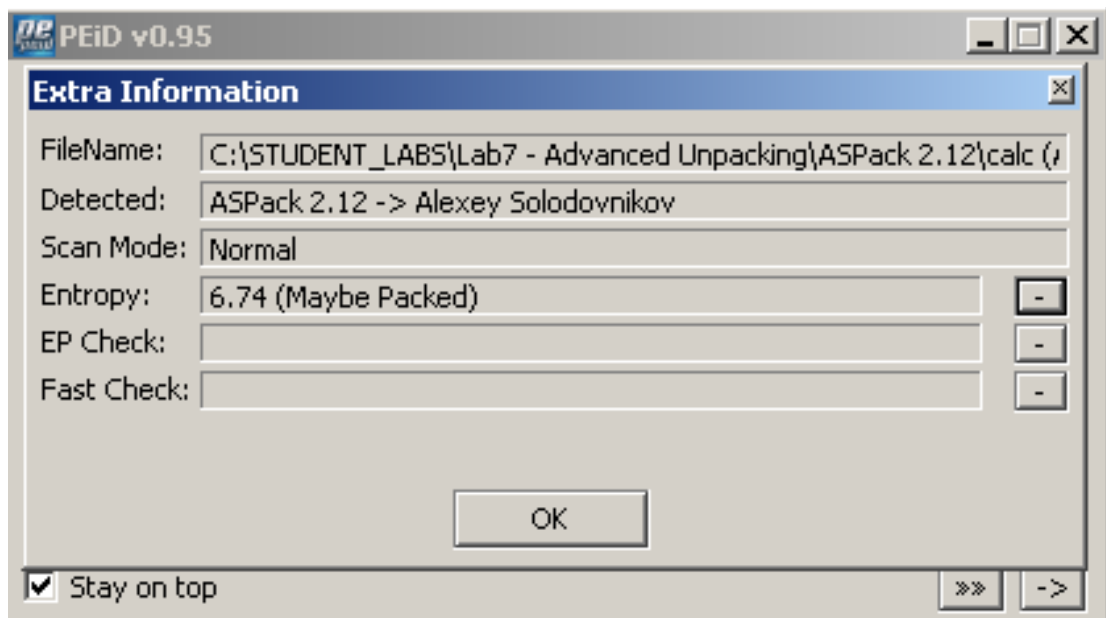
listed.

Filename	File Size (K)	MD5 sum
"notepad (WinUpack 0.31).exe"	45	d76dcb30c8eb9fed956ab506d62a73fe
"Dumped.exe"	156	077053ec5315d99162b90b225789bc00
"Dumped_.exe"	160	8678ee6af8ed9339930983ca6be55deb

The reasoning behind the file sizes in the table above are the same as discussed during the end of section "unpacking UPX" above.

4 Unpacking AsPack

Initially the packed executable was statically analyzed to determine which packing method was used and the entropy value of the file. This was determined by loading the file into PEID, the output can be seen in the image below.



The result shows that the packing method used is Aspack and the entropy value of the file is calculated to 6.74. Due to the error margin in PEID's algorithm for calculating the entropy value this file is considered "maybe packed" as seen in the output. The fact that the file is packed can be strengthened by simply loading the executable into IDA Pro and seeing that the two new sections ".aspack" and ".adata" are created which are known to be created by the aspack packer. The added sections can be seen along the left side in the image below.

```

* .aspack:010263FF          db      0
* .aspack:01026400          align 1000h
* .aspack:01026400 _aspack  ends
* .aspack:01026400
* .adata:01027000 ; Section 5. (virtual address 00027000)
* .adata:01027000 ; Virtual size           : 00008000 ( 32768.)
* .adata:01027000 ; Section size in file   : 00000000 ( 0.)
* .adata:01027000 ; Offset to raw data for section: 0000EC00

```

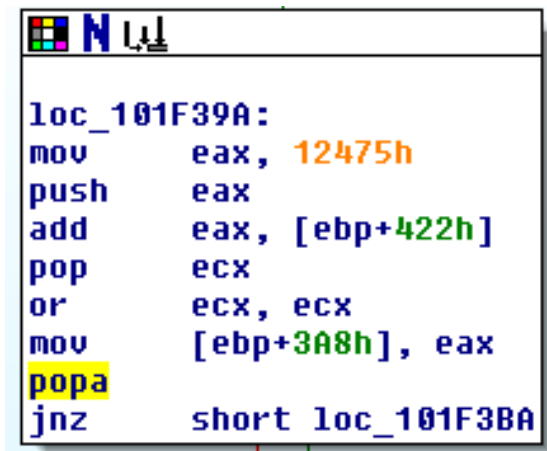
The procedure to unpack the executable is similar to the one used when unpacking the UPX executable above. By loading the executable into the debugger IDA Pro and placing a breakpoint in the start point of the execution the instructions could be stepped through. In the beginning of the program a "pusha"-instruction was discovered. This is a common way for packers to save all registry values on the stack during the unpacking procedure. The instruction can be seen in the image below.

```

* .aspack:0101F001          pusha
* .aspack:0101F002          call     loc_101F00A
* .aspack:0101F002 ; -----
* .aspack:0101F007          db      0E9h ; T
* .aspack:0101F008 ; -----
* .aspack:0101F008          jmp      short loc_101F00E
* .aspack:0101F00A ; -----

```

The occurrence of this instruction suggests that the opposite instruction "popa" is also used. "popa" is used to reset all registers as they were before the unpacking procedure started. By using a basic string search in IDA Pro the "popa"-instruction was localized and can be seen in the image below.



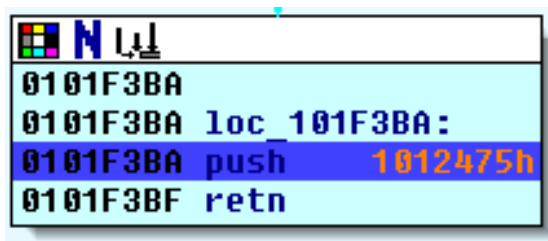
```

loc_101F39A:
mov     eax, 12475h
push    eax
add     eax, [ebp+422h]
pop     ecx
or      ecx, ecx
mov     [ebp+3A8h], eax
popa
jnz     short loc_101F3BA

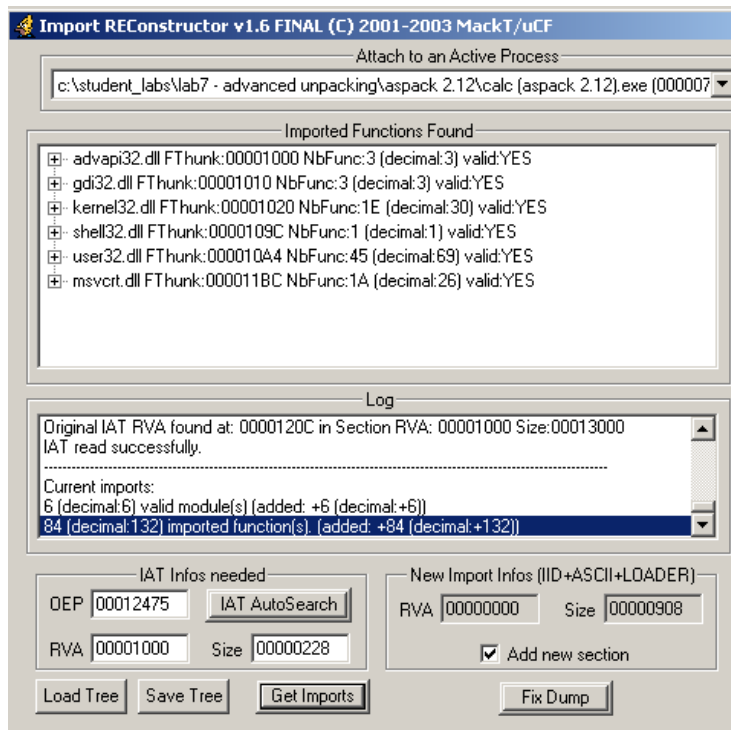
```

After the unpacking is finished and all the registers are reset by using "popa", the program is ready to execute the executable contained inside the packing.

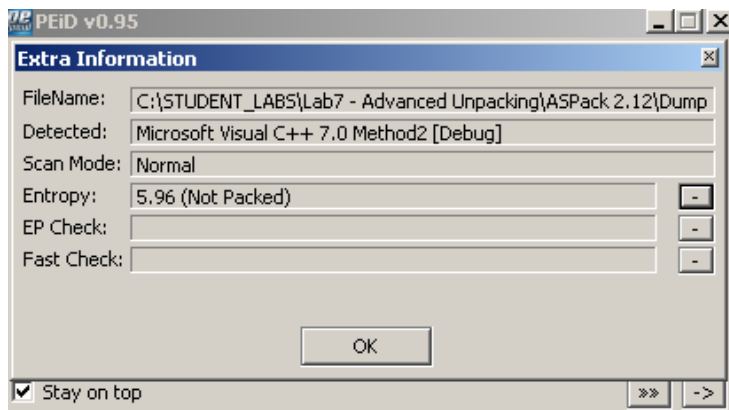
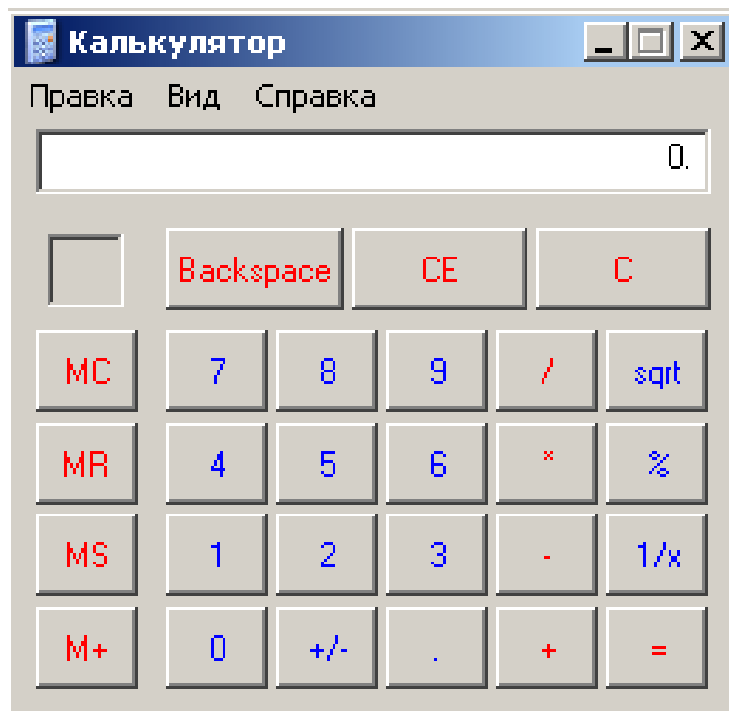
The start of this code, the OEP, is returned to straight after the use of `popa` by using the jump seen at the bottom of the image above. The destination of the short jump can be seen in the image below and the value 1012475 is hexadecimal format which is pushed to the stack and thereby returned to, is the OEP.



At this time in the execution of the program, when the instruction pointer is currently set on the OEP the formerly packed program is now unpacked and loaded into the main memory space of the process. The unpacked program can thereby be dumped to a file named "Dumped.exe" by using PEtools and choosing the running process of "calc (ASPack 2.12).exe". The next step is to reconstruct the PE header of "Dumped.exe". The reconstruction is done using the software ImpRec using the discovered OEP. By seeing that ImpRec has identified imports using the input OEP shows that the localization of the OEP address was successful. The output of ImpRec can be seen in the image below.



The PE header created by ImpRec is then appended to the file "Dumped" and the new file is named "Dumped.exe". At this point "Dumped.exe" is an unpacked version of the original file "calc (ASPack 2.12).exe" and should therefore work in the same way. The operation of "Dumped.exe" is tested by running the executable to confirm that it works in the same way as the original packed version. "Dumped." is thereafter statically analyzed in PEID to confirm that PEID can no longer detect any packing. The first image below shows the execution of "Dumped.exe" and the bottom image shows the output from PEID showing that no packing method is detected and thereby a low entropy value.



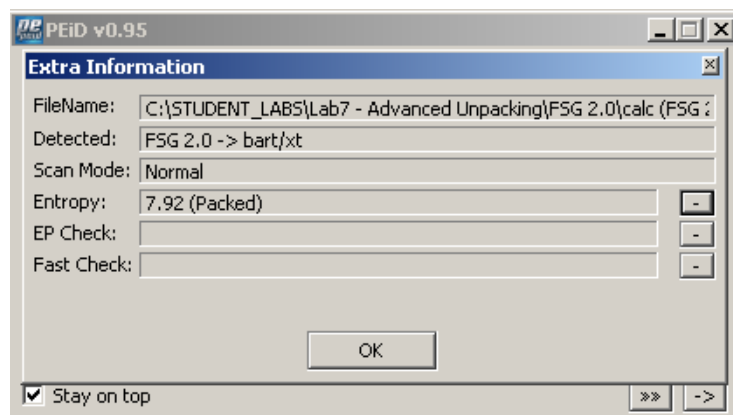
Note that PEID detects a compiler but has also outputted "[Debug]", the reason of this could be that the executable was compiled with a debug-flag set. In the table below the sizes of the different files along the analysis process are listed.

Filename	File Size (K)	MD5 sum
"calc (ASPack 2.12).exe"	59	6b1ab7c5b0b23d64615093d3e76a2777
"Dumped.exe"	188	46e272d4b089afba24a1244845a5c43b
"Dumped_.exe"	192	8d917c1f8f2e6a42719993ac6e26f4ec

The reasoning behind the file sizes in the table above are the same as discussed during the end of the unpacking of UPX seen in section 2.

5 Unpacking FSG

As during previous unpackings the first step was to identify the packing method. The result of loading the executable "calc (FSG 2.0).exe" is seen in the image below.



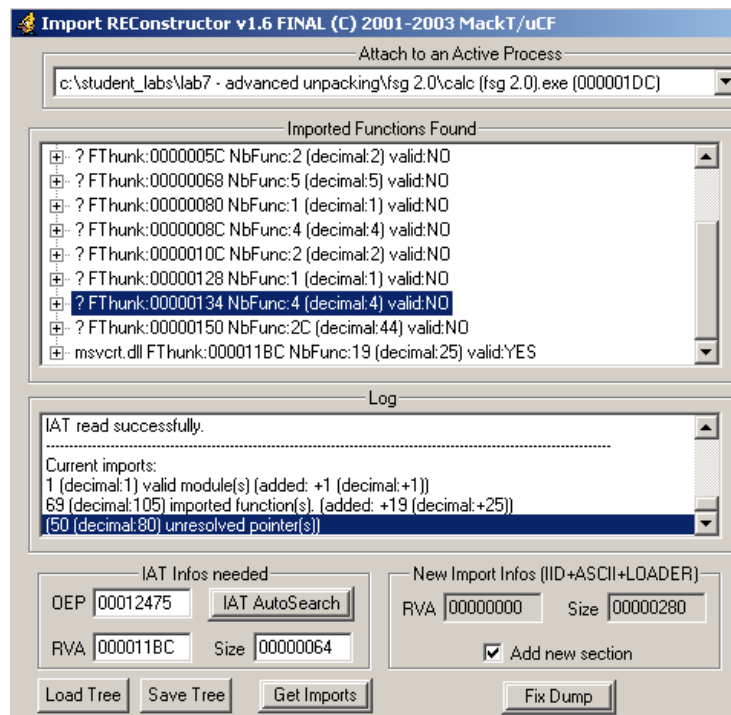
The result is that the packing method identified by PEID is FSG 2.0 and the calculated entropy is 7.92 which PEID considers as packed. The next step in unpacking the executable is to locate the OEP. The OEP was located in the same manner as in the unpacking of "WinUPack" seen in section 3. By loading the packed executable into the debugger Ida Pro and setting a breakpoint in the start of the executable the program was stepped through looking for function calls to either "GetProcAddress" or "LoadLibraryA". The reasoning behind searching for these function calls can be seen read during the unpacking of "WinUPack" seen in section 2. When locating the "GetProcAddress" and "LoadLibrary" it is probable that the jump from the unpacking procedure to the entry point of the unpacked executable (OEP) is nearby. The located function calls can be seen in the image below where the top call on address 010001C6 is for the function "LoadLibraryA" and the bottom one on address 010001D6 is for function "GetProcAddress".

```

calc_(FSG_2.0).exe:010001C2 ;
calc_(FSG_2.0).exe:010001C2
calc_(FSG_2.0).exe:010001C2 loc_10001C2: ; CODE
calc_(FSG_2.0).exe:010001C2 lodsd
calc_(FSG_2.0).exe:010001C3 xchg    eax, edi
calc_(FSG_2.0).exe:010001C4 lodsd
calc_(FSG_2.0).exe:010001C5 push    eax
calc_(FSG_2.0).exe:010001C6 call    dword ptr [ebx+10h]
calc_(FSG_2.0).exe:010001C9 xchg    eax, ebp
calc_(FSG_2.0).exe:010001CA
calc_(FSG_2.0).exe:010001CA loc_10001CA: ; CODE
calc_(FSG_2.0).exe:010001CA mov     eax, [edi]
calc_(FSG_2.0).exe:010001CC inc     eax
calc_(FSG_2.0).exe:010001CD js      short loc_10001C2
calc_(FSG_2.0).exe:010001CF jnz     short loc_10001D4
calc_(FSG_2.0).exe:010001D1 jmp     dword ptr [ebx+0Ch]
calc_(FSG_2.0).exe:010001D4 ;
calc_(FSG_2.0).exe:010001D4 loc_10001D4: ; CODE
calc_(FSG_2.0).exe:010001D4 push    eax
calc_(FSG_2.0).exe:010001D5 push    ebp
calc_(FSG_2.0).exe:010001D6 call    dword ptr [ebx+14h]
calc_(FSG_2.0).exe:010001D9 stosd
calc_(FSG_2.0).exe:010001DA jmp     short loc_10001CA
calc_(FSG_2.0).exe:010001DA ;

```

After executing the function calls a number of times to load the needed libraries the program executes the instruction on address 010001D1 which is a jumping instruction to the OEP. By executing the jump and thereby setting the instruction pointer at the OEP of the executable, the unpacked executable is now loaded into the running process's primary memory space. By using the software PEtools the unpacked executable was dumped from main memory into a file named "Dumped.exe". Running the unpacked executable requires a PE header containing the used imports of the program. The header was reconstructed using ImpRec by inputting the discovered OEP. The imports identified by ImpRec can be seen in the image below.



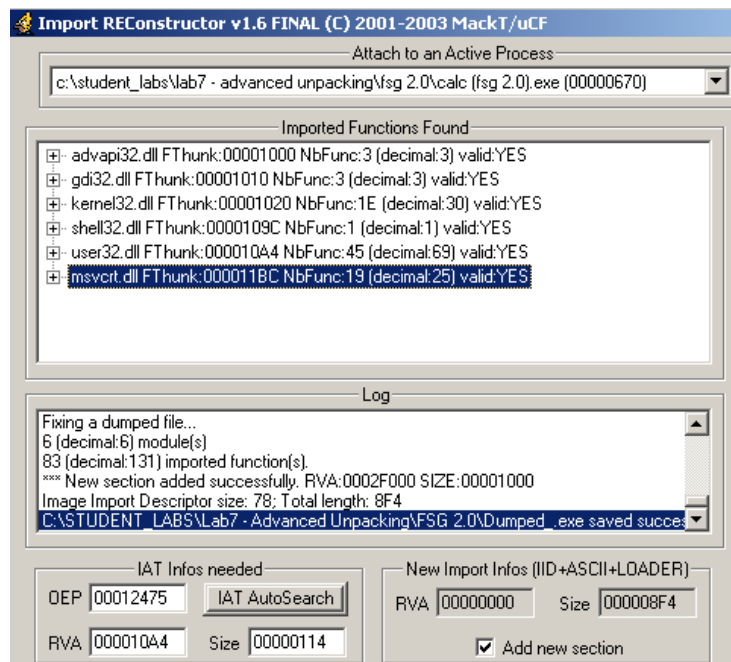
In the image it is clear that every import but one that were identified are invalid. This suggests that there are imports that could not be automatically identified and thereby inserted into a PE header by ImpRec. The solution to this is to locate the imported function from the executable manually in the IDA Pro disassembler. While the executable is packed the imports are obfuscated which means that the only way to manually identify the imports are when the program is unpacked, when the breakpoint mentioned above is triggered and the instruction pointer is pointing at the OEP. At this point a list of imported functions and their respective dll's can be found in the data segment of the code. A snippet of the list of imported functions can be seen in the image below.

```

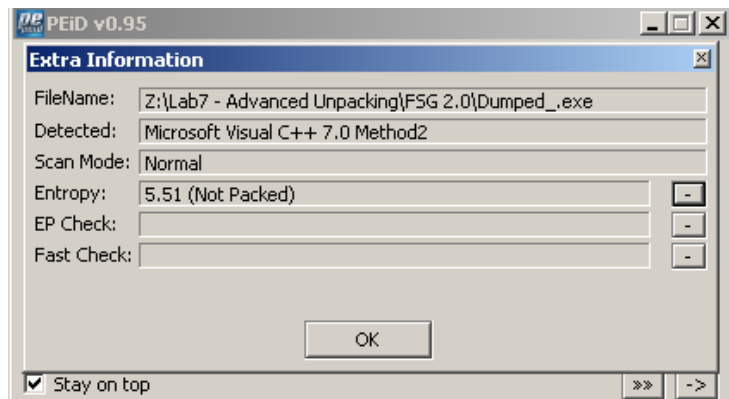
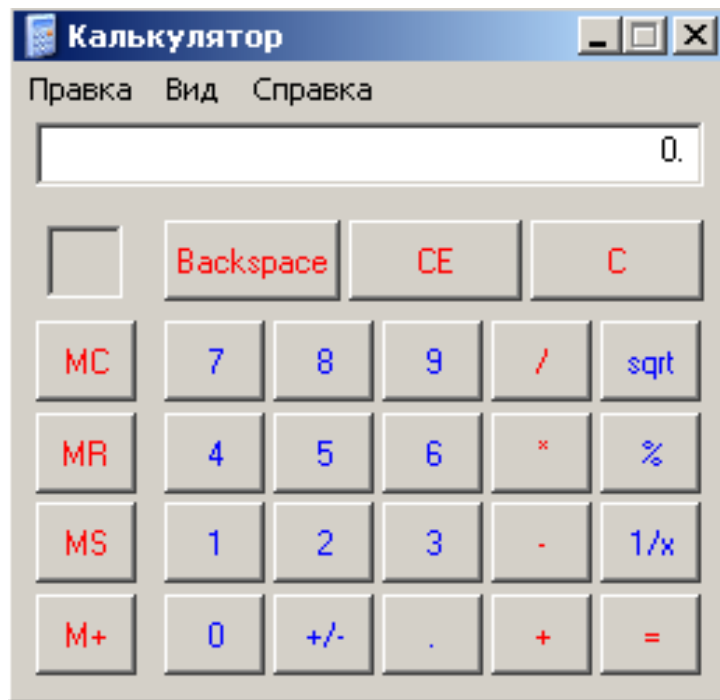
seg024:01001000 ; -----
seg024:01001000
seg024:01001000 ; Segment type: Pure code
seg024:01001000 ; Segment permissions: Read/Write
seg024:01001000 seg024 segment para public 'BSS' use32
seg024:01001000 assume cs:seg024
seg024:01001000 ;org 1001000h
seg024:01001000 assume es:nothing, ss:nothing, ds:seg024, fs:nothing, gs:nothing
* seg024:01001000 off_1001000 dd offset advapi32_RegOpenKeyExA ; DATA XREF:
* seg024:01001004 off_1001004 dd offset advapi32_RegQueryValueExA ; DATA XREF:
* seg024:01001008 off_1001008 dd offset advapi32_RegCloseKey ; DATA XREF:
* seg024:0100100C dd 7FFFFFFFh
* seg024:01001010 off_1001010 dd offset gdi32_SetBkColor ; DATA XREF: sub_1006118+3
* seg024:01001014 off_1001014 dd offset gdi32_SetTextColor ; DATA XREF: sub_1006118+3
* seg024:01001014 off_1001014 dd offset gdi32_SetTextColor ; DATA XREF: sub_1006118+3
* seg024:01001018 off_1001018 dd offset gdi32_SetBkMode ; DATA XREF: sub_1006118+3
* seg024:0100101C dd 7FFFFFFFh
* seg024:01001020 off_1001020 dd offset kernel32_GetModuleHandleA ; DATA XREF: sub_1006118+3
* seg024:01001020 off_1001020 dd offset kernel32_GetModuleHandleA ; DATA XREF: sub_1006118+3
* seg024:01001024 off_1001024 dd offset kernel32_LoadLibraryA ; DATA XREF: sub_1006118+3
* seg024:01001028 off_1001028 dd offset kernel32_GetProcAddress ; DATA XREF: sub_1006118+3
* seg024:0100102C off_100102C dd offset kernel32_GlobalCompact ; DATA XREF: sub_1006118+3
* seg024:01001030 off_1001030 dd offset kernel32_GlobalAlloc ; DATA XREF: sub_1006118+3
* seg024:01001030 off_1001030 dd offset kernel32_GlobalAlloc ; DATA XREF: sub_1006118+3
* seg024:01001034 off_1001034 dd offset kernel32_GlobalFree ; DATA XREF: sub_1006118+3
* seg024:01001034 off_1001034 dd offset kernel32_GlobalFree ; DATA XREF: sub_1006118+3
* seg024:01001038 off_1001038 dd offset kernel32_GlobalReAlloc ; DATA XREF: sub_1006118+3
* seg024:01001038 off_1001038 dd offset kernel32_GlobalReAlloc ; DATA XREF: sub_1006118+3
* seg024:0100103C off_100103C dd offset kernel32_IsStrcmpW ; DATA XREF: sub_1006118+3

```

These can then be manually entered into ImpRec by inputting the address and size of the imported dll. An example is the "advapi32.dll" import seen in the top of the image. The address of this import is 1001000 and the end of the import is 1001008 which means that the size is 8 in hex format. The recovered data is entered into ImpRec which adds the imports to the constructed PE header. All the missing imports was manually included into ImpRec using this method and the result is seen in the image below.



The next step was to append the constructed PE header onto "Dumped.exe" and thereby creating the new file "Dumped_.exe". "Dumped_.exe" is now a unpacked version of the packed executable "calc (FSG 2.0).exe". The last step of the procedure is to test if the unpacking was successful which is done by running the executable and confirming that the program behaves like the packed program and by loading the program into PEID to confirm that the tool no longer identifies any signs of packing. The output of the tests can be seen in the images below.



The tests confirms that the unpacking was successful, the program is working as intended and PEiD no longer identifies a packing method nor any signs of packing that would increase the calculated entropy value. In the table below the sizes of the different files along the analysis process are listed.

Filename	File Size (K)	MD5 sum
"calc (FSG 2.0).exe"	60	5a373276245e8d182e67cb561ea292a5
"Dumped.exe"	188	a28d2671e8cd061dea544fcadde789fe
"Dumped_.exe"	192	949ffb8205b390e6681dee671a206047

The reasoning behind the file sizes in the table above are the same as discussed during the end of the unpacking of UPX seen in section 2.

6 Questions & Answers

1. *What are the existing methods of executable code protection?*

Some of the methods are: Encryption, obfuscation, packing, polymorphism, hiding imported functions and implementing anti debug/dissassemble techniques.

2. *Describe the essence of the packing executable files.*

There are several reasons for packing an executable. The most obvious one is to compress the file which reduces the file size. Packing can also be used to add a layer of protection on the file data. The reason behind the protection is to obstruct analysis of the file which is often used in Malware.

3. *Describe the main ways to extract the executable files.*

Extracting the executable file from a package, unpacking, can either be done statically or dynamically. The static approach utilize external software which performs the unpacking process. The advantage of this approach is that the unpacking is automatic which means that a large number of files can be unpacked in a relative short amount of time, the unpacking does not require that the file is executed in the analyzing system and the unpacking does not require high knowledge. The dynamic unpacking is done by running the executable through a debugger. The procedure is to let the executable unpack itself as it intends to and as soon as the unpacked version of the executable is loaded into memory it is dumped into a new file which is thereby unpacked. The advantage of this approach is that with enough knowledge it can be implemented on almost every method of packing. The disadvantages is that it requires high knowledge, process can be time consuming and there is a possibility that "trash"-code (remaining parts of the packer) remains in the unpacked executable and automation of the process is hard to implement and impossible to make generic for every packing method.

4. *What packers do you know? Describe their features.*

- **UPX (Ultimate Packer for eXecutables):** UPX is a free packer which support many different file formats. The creators claims that the compression rates are higher than typically used zip compression tools. The used data compression algorithm UCL makes the unpacking function simple enough to only require a few hundred bytes of memory [1]. The source code of the packer is open source and the software is written in C++, this makes the program easy to expand to for example add new features or add new file formats able to the packer.
- **ASPack:** ASPack is a packer which is created to pack Win32 executables. ASPack creators claims that in addition to compression the packer

also provide protection from analysis, decompilers and debuggers [2]. The software also provides high compression rates (40-70%) and quick unpacking.

- **FSG (F [ast] S [mall] G [ood]):** FSG is a free packer created for Windows executables. The packer is written in assembler and use the compression algorithm aPLib. It is claimed that FSG is the ideal packer for packing small applications written in assembly.
- **WinUPack (Ultimate PE Packer):** WinUPack is packer used for Windows executables. Unlike the packers describes above the WinUPack project has stopped which means that the software does not receive any major updates but the author still patch bugs. The packer uses the compression algorithm LZMA which is used for example in the 7zip format.

7 References

1. <https://en.wikipedia.org/wiki/UPX>
2. <http://www.aspack.com/>
3. <https://www.virusbulletin.com/virusbulletin/2012/07/quick-reference-manual-unpacking-i>