Code obfuscation and Lighty Compressor unpacking

A few words from the author...

When I first started this article I had no idea what "*Lighty Compressor*" is. After a little research I found out that it's a code compressor mostly used in the malware developing scene, which means it's not freely downloadable.

The text below does not pretend to be professionally written, and I don't pretend to be a reverse engineering expert. However, this is my approach of defeating code obfuscation and Lighty's compression.

The application I unpack in the lines below is an old malware sample, probably from the end of 2008, and it's called "*buritos.exe*".

So, get yourself a beer and continue reading!

Loaded in *OllyDbg*, the code starts here:

```
00401001 $ 2BC1          SUB EAX,ECX
00401003 . 90            NOP
00401004 . 33C0          XOR EAX,EAX
00401006 . 90            NOP
```

The main obfuscation code contains a lot of bulk *NOP* instructions that I removed in some of the images below to save space.

```
00401025 . 90            NOP
00401026 . E8 00000000   CALL buritos.0040102B
0040102B $ 90            NOP
0040102C . 90            NOP
0040102D . 90            NOP
0040102E . 5B            POP EBX
0040102F . 90            NOP
00401030 . 90            NOP
```

The first *CALL* is a simple implementation of a code taking the *EIP* value. Entering that call will put the return address in the stack:

```
0012FFB0  0040102B RETURN to buritos.<ModuleEntryPoint>+2A from buritos.0040102B
```

Then by calling the next address, the stack will contain the returning address, that is then *POP*-ed to the *EBX* register at address 0040102E.

```
00401034 . 66:33DB       XOR BX,BX
00401037 . 90            NOP
00401038 . 90            NOP
00401039 . 8BC3          MOV EAX,EBX
```

Since *EBX* holds the current *EIP*, its low word will get *XOR*-ed by *XOR BX,BX* to null it and voilà, *EBX* is pointing to the *ImageBase* address.

```
EAX 00000000
ECX FEBD050F
EDX 3E4875CA
EBX 00400000 buritos.00400000
ESP 0012FFB4
EBP 0012FFF0
ESI FFFFFFFF
EDI 7C910738 ntdll.7C910738
EIP 00401037 buritos.00401037
```

```
00401039 . 8BC3          MOV EAX,EBX
0040103B . 0340 3C       ADD EAX,DWORD PTR DS:[EAX+3C]
0040103E . 90            NOP
0040103F . 90            NOP
00401040 . 90            NOP
00401041 . 0FB750 14     MOVZX EDX,WORD PTR DS:[EAX+14]
00401045 . 90            NOP
00401046 . 90            NOP
00401047 . 90            NOP
00401048 . 8D4410 18     LEA EAX,DWORD PTR DS:[EAX+EDX+18]
```

Moving down, the pointer to the PE header is moved to *EAX* by adding 0x3C to our image base offset. Adding 0x3C to the Image base will point to MZ's *DOS_PEOffset* value. It's always at 0x3C offset, so it's hardcoded into the application code. However, the *SizeOfOptionalHeader* may vary, so it takes its value *EDX* by adding 0x14 to the *EAX* pointer.
Then using *LEA*, a sum of *DOS_PEOffset* (*EAX*), plus *SizeOfOptionalHeader* (*EDX*), plus 0x18 (size of the PE header which is also a constant value) we get the beginning of the PE sections.

```
0040104F . 8B40 34       MOV EAX,DWORD PTR DS:[EAX+34]
```

By adding 0x34 to the current pointer in *EAX* (pointing at the beginning of the PE sections), the program skips the PE header, the first "*.text*" section, and takes the *VirtualAddress* of the second "*.data*" section.

```
00401052 . 03C3          ADD EAX,EBX
00401054 . 05 F6070000   ADD EAX,7F6
```

To the "*.data*" section *VirtualAddress* is then added the *ImageBase* in *EBX*, and *EAX* will hold a

pointer to the beginning of "*.data*" section where the encryptor Import Table begins.
Then, by adding 0x7F6 to *EAX*, the *Import Table* will be skipped, so *EAX* will point to the beginning of the next code chunk.

```
0040108A  . 91            XCHG EAX,ECX
0040108B  . 90            NOP
0040108C  . 90            NOP
0040108D  . 51            PUSH ECX
0040108E  . 33CB          XOR ECX,EBX
00401090  . 8BCC          MOV ECX,ESP
00401092  . 90            NOP
00401093  . 83C4 04       ADD ESP,4
00401096  . 90            NOP
00401097  . 90            NOP
00401098  . FF7424 FC     PUSH DWORD PTR SS:[ESP-4]
0040109C  . 90            NOP
0040109D  . 90            NOP
0040109E  . 90            NOP
0040109F  . C3            RETN
```

Few lines before the *RETN* instruction, *EAX* and *ECX* values get exchanged, then *ECX* is pushed to the stack.
At the end, *PUSH [ESP-4]* store the return address that *ECX* point at, to the stack.

Here, the code saves the register values by *PUSHAD*, and jumps to 00405809.

```
004057F6  60              PUSHAD
004057F7  EB 10           JMP SHORT buritos.00405809
```

At 00405809 a *CALL*, followed by most probably encrypted code is executed.

```
00405809  E8 8C000000     CALL buritos.0040589A
0040580E  95              XCHG EAX,EBP
0040580F  D9A8 81C3ACCF   FLDCW WORD PTR DS:[EAX+CFACC381]
00405815  9C              PUSHFD
00405816  C8 61846A       ENTER 8461,6A
0040581A  B5 42           MOV CH,42
0040581C  B3 30           MOV BL,30
0040581E  4B              DEC EBX
0040581F  1F              POP DS
00405820  48              DEC EAX
00405821  322A            XOR CH,BYTE PTR DS:[EDX]
00405823  93              XCHG EAX,EBX
00405824  AD              LODS DWORD PTR DS:[ESI]
00405825  E2 37           LOOPD SHORT buritos.0040585E
00405827  80A7 8A8A3213 6 AND BYTE PTR DS:[EDI+13328A8A],6A
0040582E  DFD9            FISTP ECX
```

I'll step into that call to see what's going on there.

```
0040589A  90              NOP
0040589B  5A              POP EDX
0040589D  53              PUSH EBX
0040589F  BB 34000000     MOV EBX,34
004058A5  8DB49A BE000000 LEA ESI,DWORD PTR DS:[EDX+EBX*4+BE]
004058AE  33FF            XOR EDI,EDI
004058B3  33FE            XOR EDI,ESI
004058B9  AD              LODS DWORD PTR DS:[ESI]
004058BB  40              INC EAX
004058BD  83E8 01         SUB EAX,1
004058C0  05 3A8D5A23     ADD EAX,235A8D3A
004058C7  AB              STOS DWORD PTR ES:[EDI]
004058CA  4B              DEC EBX
004058CC  8BC3            MOV EAX,EBX
004058CE  75 E9           JNZ SHORT buritos.004058B9
004058D0  53              PUSH EBX
004058D1  C1A9 6CC203D1 D SHR DWORD PTR DS:[ECX+D103C26C],0D7
```

(BULK NOP INSTRUCTIONS REMOVED FROM THE IMAGE!)

First, the return address is *POP*-ed into *EDX*. *EBX* is replaced with 0x34 and finally *EDX+EBX*4+0xBE* goes to *ESI*.
A two *XOR* instructions *EDI,EDI* and *EDI,ESI*, can be described as *MOV EDI,ESI*

implementation, so both *EDI* and *ESI* contain the address 0040599C, which point to the end of the encrypted code.

Now I know the end of the encrypted code, and the next *JNZ* loop could give me an idea for the beginning. There are *LODS* and *STOS* instructions, that will move data “TO” and then “FROM” *EAX*.
EBX still contains 0x34, gets decreased and then checked by *JNZ* (that's why *EBX* gets moved to EAX before the *JNZ* instruction), and since *LODS* and *STOS* use *DWORD*s, I should multiply 0x34 by 4 (1 *DWORD* = 4 *BYTE*) to get the encrypted code chunk size in bytes.

Then, 0x34 * 0x04 = 0xD0, and since the end of the encrypted code is 0x0040599C, by subtracting 0xD0 from 0x0040599C, the result 0x004058CC will be the beginning address. But, because 0040599C is actually the beginning of the final encoded *DWORD*, I should add 4 to it, and also add 4 to the beginning address 004058CC too.

Finally, that code will decrypt everything between 0x004058D0 and 0x004059A0.

Now after I know both the encrypted code beginning and ending, I’ll take a look at the "decryption" routine between *LODS* and *STOS*.
As I said, the instruction *LODS* loads a *DWORD* into the *EAX* register from the operand. Then *EAX* gets increased by one, then 1 gets subtracted from it. Obviously that's a bulk code, put there probably to fool the Antivirus fingerprinting technique.
Next a value of 0x235A8D3A gets added to *EAX* and finally *EAX* is put back where it was by *STOS*.
So the "encryption" is a simple add obfuscation, again designed to fool the AV.
Because it will decrypt the instructions after the *JNZ* at 004058CE, setting a regular breakpoint won’t work, so I'll set "*Hardware on execution*” at the next address 004058D0, and run the program.

And the code get decrypted:

```
004058D0  8D4E 04          LEA ECX,DWORD PTR DS:[ESI+4]
004058D4  FC               CLD
004058D6  2BFB             SUB EDI,EBX
004058DA  8BF9             MOV EDI,ECX
004058DD  F7D3             NOT EBX
004058E0  5B               POP EBX
004058E2  41               INC ECX
004058E5  81EF 36000000    SUB EDI,36
004058EC  33C9             XOR ECX,ECX
004058EF  B9 23000000      MOV ECX,23
004058F6  B8 C05244E1      MOV EAX,E14452C0
004058FB  FD               STD
004058FE  AF               SCAS DWORD PTR ES:[EDI]
00405901  3107             XOR DWORD PTR DS:[EDI],EAX
00405905  6A 04            PUSH 4
00405908  032424           ADD ESP,DWORD PTR SS:[ESP]
0040590B  FC               CLD
0040590E ^E2 EB            LOOPD SHORT buritos.004058FB
00405910  8BEF             MOV EBP,EDI
00405912  B9 A8140000      MOV ECX,14A8
```

First *LEA* instruction here will take a pointer at *ESI+4*.
ESI is the beginning of the decrypted code - 0x004058CC. So it adds 4(what the *DWORD* value length is), and finally *ECX* gets the current address - 004058D0.

Then subtract 0x36 from *EDI*, (holding *ECX*'s value) and *EDI* is now 0040589A. That's a line before the beginning address, where the first decryptor was.
The rest of the code is another decryption loop, this time using *XOR*.

Instruction *SCAS* at 004058FE first takes a *DWORD* from 0040589A then it *XOR* it with 0xE14452C0 (moved few lines above into *EAX*).
This decryption loop ends with *LOOPD* instruction, that use the default assembler counter *ECX*, then every time *LOOPD* gets executed, *ECX*'s value get decreased by 1.

At 004058EF 0x23 is moved to *ECX* register, and that's how much times the loop will be executed. Therefore, I'll take the end address 0040589A, that will get decoded, subtract 0x23 *DWORDS* (or 0x23*4), and the result 0040580E will be the beginning address of the code that will be decrypted here.

Again I'll place a hardware breakpoint after the *LOOPD* instruction and execute the code.

Right after the *LOOPD*, the value of *EDI* (0040580E, starting address of the decrypted code) gets moved into *EBP*, and 0x14A8 is moved to *ECX*.

```
00405952  90              NOP
00405953  FFD5            CALL EBP
00405955  90              NOP
```

Few lines down, at address 00405953 a *CALL EBP* will execute the code that was just decrypted.

But lets see what we have before that *CALL EBP*.

```
00405912  B9 A8140000     MOV ECX,14A8
0040591A  8DBC39 51010000 LEA EDI,DWORD PTR DS:[ECX+EDI+151]
00405923  C1E9 03         SHR ECX,3
00405927  FD              STD
00405928  83EF 08         SUB EDI,8
0040592D  68 31FE6119     PUSH 1961FE31
00405934  68 53063F61     PUSH 613F0653
0040593B  68 FB744861     PUSH 614874FB
00405942  68 A832AB81     PUSH 81AB32A8
00405949  54              PUSH ESP
0040594B  6A 08           PUSH 8
00405950  57              PUSH EDI
```

As I mentioned, *ECX* gets 0x14A8. Then at 0040591A, *EDI* gets a pointer from *EDI+ECX+0x151*.
EDI holds 0040580E (beginning address of the decoded code), so adding 0x14A8 (in *ECX*) plus 0x151 to it, will produce the value 00406E07.

Next *ECX* gets shifted left by 3 to become 0x295, and 8 get subtracted from *EDI*, so it's 00406DFF now.
The lines below are a few PUSH instructions, that will fill the stack with 0x1961FE31, 0x613F0653, 0x614874FB, 0x81AB32A8, *ESP*'s value, 8 and *EDI*'s value - in that order.
Then it's the *CALL EBP*, so lets trace into it.

It start here:

```
0040580E  55              PUSH EBP
0040580F  8BEC            MOV EBP,ESP
00405811  60              PUSHAD
```

and ends here:

```
00405895  61              POPAD
00405896  C9              LEAVE
00405897  C2 0C00         RETN 0C
```

There are two loops in here, that again decrypt part of the code, this time using *XOR*, *SUB*, *SHR/L*, etc. and I really don't want to get in details about them, so I'll just "*Execute till return*" this code to get out of the *CALL*.

And back to 0012FF78...

```
00405953  FFD5              CALL EBP
00405955  90                NOP
00405956  90                NOP
00405957  83C4 10           ADD ESP,10
0040595A  90                NOP
0040595B  90                NOP
0040595C  49                DEC ECX
0040595D ^75 C9             JNZ SHORT buritos.00405928
0040595F ^73 E5             JNB SHORT buritos.00405946
00405961  1D 15652072       SBB EAX,72206515
00405966  95                XCHG EAX,EBP
00405967  A2 2CA4DCA2       MOV BYTE PTR DS:[A2DCA42C],AL
0040596C  3E:FC             CLD
0040596E  8F                ???
```

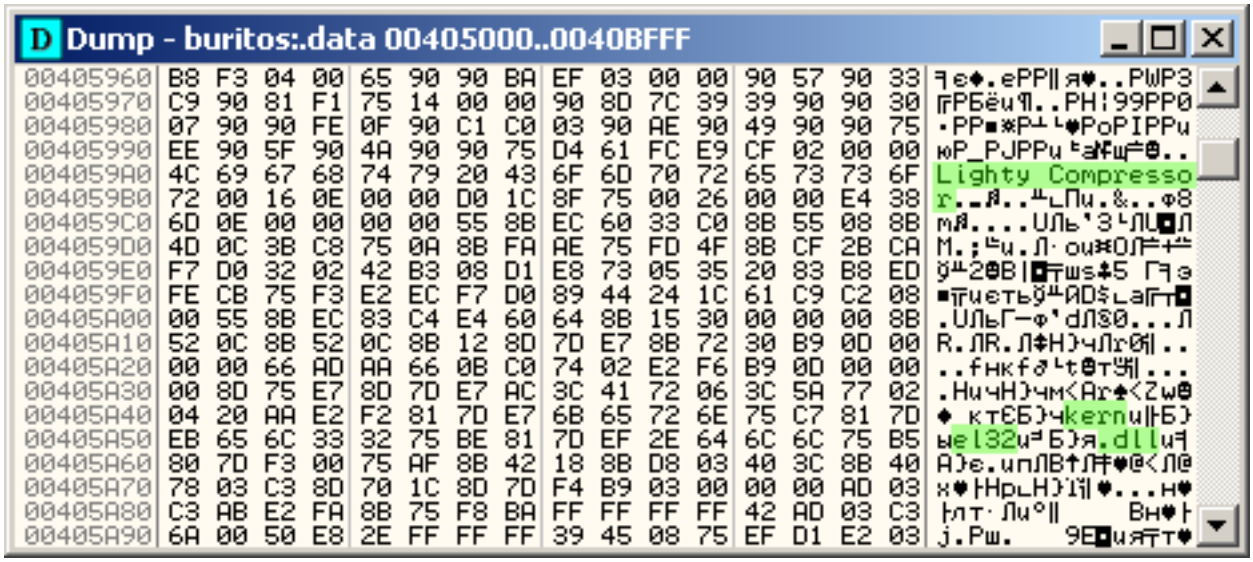
It seems like this is another loop, executing *CALL EBP* and using *ECX* as counter. Before the loop beginning, *ECX* was 0x295, so that's how much times *CALL EBP* will be run. I'll again put a hardware breakpoint, this time at 0040595F, where the loop ends and executes the code.

```
0040595F  90                NOP
00405960  B8 F3040065       MOV EAX,650004F3
00405967  BA EF030000       MOV EDX,3EF
0040596D  57                PUSH EDI
0040596F  33C9              XOR ECX,ECX
00405972  81F1 75140000     XOR ECX,1475
00405979  8D7C39 39         LEA EDI,DWORD PTR DS:[ECX+EDI+39]
0040597F  3007              XOR BYTE PTR DS:[EDI],AL
00405983  FE0F              DEC BYTE PTR DS:[EDI]
00405986  C1C0 03           ROL EAX,3
0040598A  AE                SCAS BYTE PTR ES:[EDI]
0040598C  49                DEC ECX
0040598F ^75 EE             JNZ SHORT buritos.0040597F
00405991  90                NOP
00405992  5F                POP EDI
00405993  90                NOP
00405994  4A                DEC EDX
00405995  90                NOP
00405996  90                NOP
00405997 ^75 D4             JNZ SHORT buritos.0040596D
00405999 v76 62             JBE SHORT buritos.004059FD
0040599B  B1 4B             MOV CL,4B
0040599D  DCCA              FMUL ST(2),ST
```

Another decrypt routine was decoded. It has two loops, inner - between 0040597F and 0040598F and outer - between 0040596D and 00405997. It uses hardcoded values like 0x650004F3, 0x3EF and 0x1475 to decode the next part, and I'll again put a hardware breakpoint after the outer loop, at address 00405999 and execute the decryptor.

```
00405999  61                POPAD
0040599A  FC                CLD
0040599B v E9 CF020000      JMP buritos.00405C6F
```

It looks like this was the last decoding routine, so let's take a quick look at the decoded "*.data*" section in the memory.



Looks like I'm getting closer to Lighty's code, so I'll follow the *JMP* to 00405C6F.

```
00405C6F  55              PUSH EBP
00405C70  8BEC            MOV EBP,ESP
00405C72  83C4 F4         ADD ESP,-0C
00405C75  60              PUSHAD
00405C76  33C0            XOR EAX,EAX
00405C78  50              PUSH EAX
00405C79  68 A8EDF2CE     PUSH CEF2EDA8
00405C7E  E8 7EFDFFFF     CALL buritos.00405A01
00405C83  FFD0            CALL EAX
```

Most probably that's the original "*Lighty compressor*" entry point.

The first *CALL* here is at address 00405C7E leading to 00405A01, probably takes two parameters - 0xCEF2EDA8 and *EAX* (previously *XOR*-ed), so I'll follow it.

```
00405A01  55              PUSH EBP
00405A02  8BEC            MOV EBP,ESP
00405A04  83C4 E4         ADD ESP,-1C
00405A07  60              PUSHAD
00405A08  64:8B15 3000000 MOV EDX,DWORD PTR FS:[30]
00405A0F  8B52 0C         MOV EDX,DWORD PTR DS:[EDX+C]
00405A12  8B52 0C         MOV EDX,DWORD PTR DS:[EDX+C]
00405A15  8B12            MOV EDX,DWORD PTR DS:[EDX]
00405A17  8D7D E7         LEA EDI,DWORD PTR SS:[EBP-19]
00405A1A  8B72 30         MOV ESI,DWORD PTR DS:[EDX+30]
00405A1D  B9 0D000000     MOV ECX,0D
00405A22  66:AD           LODS WORD PTR DS:[ESI]
00405A24  AA              STOS BYTE PTR ES:[EDI]
00405A25  66:0BC0         OR AX,AX
00405A28  74 02           JE SHORT buritos.00405A2C
00405A2A  E2 F6           LOOPD SHORT buritos.00405A22
```

At address 00405A08, a pointer to the Windows *PEB* structure is moved to *EAX*. This structure is officially undocumented, but lots of malware authors (and lately, not only...) use it.
You may learn more about PEB and other undocumented functions, methods and structures from http://undocumented.ntinternals.net/.

So, *FS[30]* points to the *PEB* structure. By using the ntinternals.net documentation, if I count right, *EDX+C* is the pointer to the *PEB*'s *LoaderData* which is a *PPEB_LDR_DATA* structure. Then there is another *EDX+C* that points to the *PPEB_LDR_DATA*'s *InLoadOrderModuleList* - a *LDR_MODULE* structure.
A pointer to *LDR_MODULE*'s *InLoadOrderModuleList* entry is taken in *EDX*. It's again a *LDR_MODULE* structure and finally from that structure *HashTableEntry* is taken by *EDX+30* at address 00405A1A.

The things are getting clear now. In *ESI* is the pointer to *HashTableEntry* to the modules loaded by our application.
A *LODS-STOS* routine in a loop by *LOOPD* gets executed, and *ECX* is 0x0D. So the code here will get a loaded library name from the *HashTableEntry* and store it to the buffer pointed by *EDI*.

```
00405A1A  8B72 30          MOV ESI,DWORD PTR DS:[EDX+30]    ntdll.7C9226A4
```

```
DS:[00241F78]=7C9226A4 (ntdll.7C9226A4), UNICODE "ntdll.dll"
ESI=FFFFFFFF
```

In my case, the first loaded module is "*ntdll.dll*".

```
00405A2C  B9 0D000000      MOV ECX,0D
00405A31  8D75 E7          LEA ESI,DWORD PTR SS:[EBP-19]
00405A34  8D7D E7          LEA EDI,DWORD PTR SS:[EBP-19]
00405A37  AC               LODS BYTE PTR DS:[ESI]
00405A38  3C 41            CMP AL,41
00405A3A  72 06            JB SHORT buritos.00405A42
00405A3C  3C 5A            CMP AL,5A
00405A3E  77 02            JA SHORT buritos.00405A42
00405A40  04 20            ADD AL,20
00405A42  AA               STOS BYTE PTR ES:[EDI]
00405A43  E2 F2            LOOPD SHORT buritos.00405A37
```

Another *LOOPD* loop is executed. This one looks like a string-to-lower implementation.
It takes a letter from the "*ntdll.dll*" string in *ESI* and compare it with 0x41 and 0x5A. If the letter's ASCII value is between these two, it will add 0x20, to convert it to lowercase letter.

```
00405A45  817D E7 6B65726E CMP DWORD PTR SS:[EBP-19],6E72656B
00405A4C  75 C7            JNZ SHORT buritos.00405A15
00405A4E  817D EB 656C3332 CMP DWORD PTR SS:[EBP-15],32336C65
00405A55  75 BE            JNZ SHORT buritos.00405A15
00405A57  817D EF 2E646C6C CMP DWORD PTR SS:[EBP-11],6C6C642E
00405A5E  75 B5            JNZ SHORT buritos.00405A15
00405A60  807D F3 00       CMP BYTE PTR SS:[EBP-D],0
00405A64  75 AF            JNZ SHORT buritos.00405A15
```

Here, *EBP-19* holds a pointer to the loaded library name, that just gets converted to lowercase - in my case "*ntdll.dll*". The comparison get its first *DWORD* value, or "*ntdl*" and compare it with 0x6E72656B. Then the second *DWORD* "*l.dl*" is compared with 0x32336C65, the third, which for "ntdll.dll" is "*l*" is compared with 0x6C6C642E, and finally the last byte is compared with 0.

So let's convert these *DWORD* values to letters.
0x6E72656B is "*nrek*", 0x32336C65 - "*23le*" and 0x6C6C642E is "*lld.*". Obviously this checks if the current loaded library (taken from *PEB*) is "*kernel32.dll*".
Well right now it's not so I'll continue by jumping back to address 00405A15, to take the next loaded library.

When the "*kernel32.dll*" is found the code continues here:

```
00405A66  8B42 18          MOV EAX,DWORD PTR DS:[EDX+18]
00405A69  8BD8             MOV EBX,EAX
00405A6B  0340 3C          ADD EAX,DWORD PTR DS:[EAX+3C]
00405A6E  8B40 78          MOV EAX,DWORD PTR DS:[EAX+78]
00405A71  03C3             ADD EAX,EBX
00405A73  8D70 1C          LEA ESI,DWORD PTR DS:[EAX+1C]
00405A76  8D7D F4          LEA EDI,DWORD PTR SS:[EBP-C]
```

EDX contains the *InLoadOrderModuleList*'s *LDR_MODULE* structure. On address 00405A66, *EAX* takes the module's *BaseAddress*, pointed by *EDX+18*, then moves it to *EBX* on the next line.
The rest few lines between 00405A6B and 00405A76 will "parse" the loaded module.

First, at 00405A6B the pointer is moved to the PE signature offset.
Then the Export address is taken by *EAX+78*, and added to *EBX* (pointing to the module's *BaseAddress*).

```
00405A84  8B75 F8          MOV ESI,DWORD PTR SS:[EBP-8]
00405A87  BA FFFFFFFF      MOV EDX,-1
00405A8C  42               INC EDX
00405A8D  AD               LODS DWORD PTR DS:[ESI]
00405A8E  03C3             ADD EAX,EBX
00405A90  6A 00            PUSH 0
00405A92  50               PUSH EAX
00405A93  E8 2EFFFFFF      CALL buritos.004059C6
00405A98  3945 08          CMP DWORD PTR SS:[EBP+8],EAX
00405A9B ^75 EF            JNZ SHORT buritos.00405A8C
```

At address 00405A8D, a *DWORD* value from *ESI* is taken. Now *ESI* points to the export table of kernel32. By adding the *BaseAddress* to it, a pointer to the name of export function from kernel32.dll is moved in *EAX*.

The loop between address 00405A8C and 00405A9B will pass the function name as first argument to *CALL 004059C6*, and compare the result with the current *CALL* Argument *EBP+8*, that was 0xCEF2EDA8. Depending on the result, it will continue through the rest of the exports of kernel32 until it finds the needed one.

Let's take a look at *CALL 004059C6* contents.

```
004059C6  55               PUSH EBP
004059C7  8BEC             MOV EBP,ESP
004059C9  60               PUSHAD
004059CA  33C0             XOR EAX,EAX
004059CC  8B55 08          MOV EDX,DWORD PTR SS:[EBP+8]
004059CF  8B4D 0C          MOV ECX,DWORD PTR SS:[EBP+C]
004059D2  3BC8             CMP ECX,EAX
004059D4 v75 0A            JNZ SHORT buritos.004059E0
004059D6  8BFA             MOV EDI,EDX
004059D8  AE               SCAS BYTE PTR ES:[EDI]
004059D9 ^75 FD            JNZ SHORT buritos.004059D8
004059DB  4F               DEC EDI
004059DC  8BCF             MOV ECX,EDI
004059DE  2BCA             SUB ECX,EDX
004059E0  F7D0             NOT EAX
004059E2  3202             XOR AL,BYTE PTR DS:[EDX]
004059E4  42               INC EDX
004059E5  B3 08            MOV BL,8
004059E7  D1E8             SHR EAX,1
004059E9 v73 05            JNB SHORT buritos.004059F0
004059EB  35 2083B8ED      XOR EAX,EDB88320
004059F0  FECB             DEC BL
004059F2 ^75 F3            JNZ SHORT buritos.004059E7
004059F4 ^E2 EC            LOOPD SHORT buritos.004059E2
004059F6  F7D0             NOT EAX
004059F8  894424 1C        MOV DWORD PTR SS:[ESP+1C],EAX
004059FC  61               POPAD
004059FD  C9               LEAVE
004059FE  C2 0800          RETN 8
```

Looks like a hashing algorithm to me.
As this is the first time I enter this *CALL*, it's first argument is pointing to "*ActivateActCtx*" export function of kernel32.

First at address 004059CC, *EDX* takes the first argument or pointer to "*ActivateActCtx*". On the next line, *ECX* takes the second one which was 0.
Then if *ECX* is different from 0, it takes the jump, but since it's not, it will continue.
The *JNZ* loop between 004059D8 and 004059D9 will take every character from "*ActivateActCtx*", then by *DEC EDI*, *MOV ECX,EDI* and *SUB ECX,EDX* the string's length is taken. So knowing that I could say the second argument of *CALL 004059C6* is the length of the first argument passed, in that case "*ActivateActCtx*". If the length passed is 0, the loop will determine it by searching for a string termination 0.
Then if binary data is passed to that function, its length should be previously specified because it can contain zeros.

Next *EAX* gets *NOT*-ed and it's lower byte gets *XOR*-ed with the first symbol of "*ActivateActCtx*". The result of *EAX* is then shifted right with 1 and checked by *JNB* at 004059E9.

This looks like a "*is integer signed*" check. Depending on the result, *EAX* may get *XOR*-ed with 0xEDB88320 if its content is negative number (bigger than 0x7FFFFFFF).
Then a inner loop between 004059E7 and 004059F2 will get executed 8 times (*BL* is the counter here), and keep shifting left *EAX*.
When the inner loop ends, the outer one will continue the same procedure for the rest of the "*ActivateActCtx*" letters.
Finally at address 004059F6, *EAX* get *NOT*-ed and stored into *ESP+1C*. So, the hashed value of "*ActivateActCtx*" will be 0x6AA0C20C.

So *CALL 004059C6* is indeed a hashing algorithm so I'll note that.

```
00405A9D  D1E2          SHL EDX,1
00405A9F  0355 FC       ADD EDX,DWORD PTR SS:[EBP-4]
00405AA2  0FB702        MOVZX EAX,WORD PTR DS:[EDX]
00405AA5  C1E0 02       SHL EAX,2
00405AA8  0345 F4       ADD EAX,DWORD PTR SS:[EBP-C]
00405AAB  8B00          MOV EAX,DWORD PTR DS:[EAX]
00405AAD  03C3          ADD EAX,EBX
00405AAF  894424 1C     MOV DWORD PTR SS:[ESP+1C],EAX   kernel32.Sleep
00405AB3  61            POPAD
00405AB4  C9            LEAVE
00405AB5  C2 0400       RETN 4
```

The rest of these instructions will get the entry point of the requested API function from kernel32 and at 00405AAF, the call will be obviously to the kernel's Sleep function.
As this is the end of *CALL 00405A01*, I can call it a *GetProcAddress* implementation, that will return a handle to kernel32's API function, defined by the first given argument.

```
00405C76  33C0          XOR EAX,EAX
00405C78  50            PUSH EAX
00405C79  68 A8EDF2CE   PUSH CEF2EDA8
00405C7E  E8 7EFDFFFF   CALL buritos.00405A01
00405C83  FFD0          CALL EAX                        kernel32.Sleep
00405C85  0BC0          OR EAX,EAX
00405C87  0F85 5F030000 JNZ buritos.00405FEC
00405C8D  E8 E7FEFFFF   CALL buritos.00405B79
```

Back to address 00405C83, *Sleep(0)* is executed. The zero actually comes from that *PUSH EAX* (previously *XOR*-ed) at 00405C78, so that *GetProcAddress* implementation has only one parameter.

I found that *Sleep(0)* quite strange. According to the official MSDN documentation, *Sleep* does not return a value, but here its result gets *OR*-ed, and depending on the result, the code will jump or continue.
In my case, when I execute the *Sleep(0)*, the result in *EAX* is 0, and the code continues without jumping to 00405FEC. However, if in any case the program jumps to 00405FEC, an *ExitProcess* will be executed and the program will be terminated, so I guess this is some kind of a debugging protection. I'll be happy to learn more about this, so anyone knowing more can contact me and give me a quick reference. ;)

The next thing that I will take a notice on is this *CALL* at address 00405C8D.

```
00405B79  60            PUSHAD
00405B7A  E8 00000000   CALL buritos.00405B7F
00405B7F  5B            POP EBX
00405B80  81EB EF194000 SUB EBX,buritos.004019EF
00405B86  E8 4D000000   CALL buritos.00405BD8
00405B8B  8B4424 0C     MOV EAX,DWORD PTR SS:[ESP+C]
00405B8F  FF80 B8000000 INC DWORD PTR DS:[EAX+B8]
00405B95  FF80 B8000000 INC DWORD PTR DS:[EAX+B8]
00405B9B  33C0          XOR EAX,EAX
00405B9D  C2 1000       RETN 10
```

There, the first *CALL* is referring to the next address 00405B7F and it's there to take the current *EIP* into *EBX* (by *POP*-ing it to *EBX*). Then from *EBX* is subtracted 0x004019EF and *EBX* now

holds 0x4190.

Let's now take a look at the second *CALL*.

```
00405BD8  64:FF35 00000000  PUSH DWORD PTR FS:[0]
00405BDF  64:8925 00000000  MOV DWORD PTR FS:[0],ESP
00405BE6  83EC 08           SUB ESP,8
00405BE9  8DBB 101A4000     LEA EDI,DWORD PTR DS:[EBX+401A10]
00405BEF  8907              MOV DWORD PTR DS:[EDI],EAX
00405BF1  CC                INT3
00405BF2  90                NOP
00405BF3  B9 04000000       MOV ECX,4
00405BF8  0F31              RDTSC
00405BFA  890424            MOV DWORD PTR SS:[ESP],EAX
00405BFD  895424 04         MOV DWORD PTR SS:[ESP+4],EDX
00405C01  CC                INT3
00405C02  90                NOP
00405C03  0F31              RDTSC
00405C05  2B0424            SUB EAX,DWORD PTR SS:[ESP]
00405C08  1B5424 04         SBB EDX,DWORD PTR SS:[ESP+4]
00405C0C  8907              MOV DWORD PTR DS:[EDI],EAX
00405C0E  8957 04           MOV DWORD PTR DS:[EDI+4],EDX
00405C11  AF                SCAS DWORD PTR ES:[EDI]
00405C12  AF                SCAS DWORD PTR ES:[EDI]
00405C13 ^E2 EE             LOOPD SHORT buritos.00405C03
00405C15  83C4 08           ADD ESP,8
```

These two *INT 3* at address 00405BF1 and 00405C01 will break my debugger, so I should definitely *NOP* them.
The *RDTSC* instruction will take a *Time Stamp Counter* into *EDX* and *EAX*, and the results then get moved to the stack. All this instructions are probably part of a debugging protection (by counting the time needed to execute a portion of code, and then decide if it was too slow - "*debugged*" or not - "*not debugged*").
So to pass them up, I will try to *NOP* the *INT 3*, put a hardware breakpoint at address 00405C15 where *LOOPD* ends, and execute the code.
But first, let's check the rest of the code.

```
00405C15  83C4 08           ADD ESP,8
00405C18  64:8F05 00000000  POP DWORD PTR FS:[0]
00405C1F  83C4 04           ADD ESP,4
00405C22  9B                WAIT
00405C23  DBE3              FINIT
00405C25  B9 04000000       MOV ECX,4
00405C2A  8DB3 101A4000     LEA ESI,DWORD PTR DS:[EBX+401A10]
00405C30  DD83 301A4000     FLD QWORD PTR DS:[EBX+401A30]
00405C36  DF2E              FILD QWORD PTR DS:[ESI]
00405C38  D8D1              FCOM ST(1)
00405C3A  9B                WAIT
00405C3B  DFE0              FSTSW AX
00405C3D  66:A9 0001        TEST AX,100
00405C41 v74 02             JE SHORT buritos.00405C45
00405C43  D9C9              FXCH ST(1)
00405C45  DDD8              FSTP ST
00405C47  83C6 08           ADD ESI,8
00405C4A ^E2 EA             LOOPD SHORT buritos.00405C36
00405C4C  DC93 401A4000     FCOM QWORD PTR DS:[EBX+401A40]
00405C52  9B                WAIT
00405C53  DFE0              FSTSW AX
00405C55  66:A9 0001        TEST AX,100
00405C59 v75 0F             JNZ SHORT buritos.00405C6A
00405C5B  DC93 381A4000     FCOM QWORD PTR DS:[EBX+401A38]
00405C61  9B                WAIT
00405C62  DFE0              FSTSW AX
00405C64  66:A9 0001        TEST AX,100
00405C68 v75 01             JNZ SHORT buritos.00405C6B
00405C6A  F4                HLT
00405C6B  DDD8              FSTP ST
00405C6D  61                POPAD
00405C6E  C3                RETN
```

There is another *LOOPD*, using the *FPU* instructions, and at the end of the code a *HLT* command at address 00405C6A. Executing that *HLT* means game over, and I don't want to start from the beginning again, so instead of putting a breakpoint after the first *LOOPD*, I will put one

after the *HLT* at 00405C6B, hoping that the program will execute fast enough without detecting the debugger.

Fortunately, my plan worked and I can leave the debug protection successfully through the *RETN* at 00405C6E.

```
00405C94  8B45 08          MOV EAX,DWORD PTR SS:[EBP+8]
00405C97  48               DEC EAX
00405C98  25 00F0FFFF      AND EAX,FFFFF000
00405C9D  66:8138 4D5A     CMP WORD PTR DS:[EAX],5A4D
00405CA2 ^75 F3            JNZ SHORT buritos.00405C97
00405CA4  8945 08          MOV DWORD PTR SS:[EBP+8],EAX
```

A loop is searching for the current process "*MZ*" (0x5A4D) header, so at address 00405CA4, *EAX* will move its pointer to the *ImageBase*, to *EBP+8*.

```
00405CA7  E8 00000000      CALL buritos.00405CAC
00405CAC  812C24 1C1B4000  SUB DWORD PTR SS:[ESP],buritos.00401B1C
00405CB3  5B               POP EBX
00405CB4  B9 58060000      MOV ECX,658
00405CB9  038B 22184000    ADD ECX,DWORD PTR DS:[EBX+401822]
00405CBF  894D F8          MOV DWORD PTR SS:[EBP-8],ECX
```

At 00405CA7, the *CALL* is used to take the current *EIP*. Then from the stack's last entry (where the current *EIP* is) is subtracted 0x00401B1C. The result - 0x4190 is *POP*-ed to *EBX* and later used as offset value with the address 00401822.

Adding 0x4190 to 00401822 will point here:

```
Address   Hex dump                                        ASCII
00405992  5F 90 4A 90 90 75 D4 61 FC E9 CF 02 00 00 4C 69  _PJPPu╘a№Ω☻..Li
004059A2  67 68 74 79 20 43 6F 6D 70 72 65 73 73 6F 72 00  ghty Compressor.
004059B2  16 0E 00 00 D0 1C 8F 75 00 26 00 00 E4 38 6D 0E  ▬♫..╨∟Пu.&..ф8m♫
004059C2  00 00 00 00 55 8B EC 60 33 C0 8B 55 08 8B 4D 0C  ....UЛь`3└ЛU◘ЛM.
004059D2  3B C8 75 0A 8B FA AE 75 FD 4F 8B CF 2B CA F7 D0  ;╚u.Л·ou▀OЛ╧+╩╢╨
004059E2  32 02 42 B3 08 D1 E8 73 05 35 20 83 B8 ED FE CB  2☻B│◘╤Φs♣5 Гў∍╦
```

That's the *DWORD* right after the "*Lighty compressor*" string.
So 0xE16 is added to *ECX*, which is 0x658 and the result is 0x146E - moved to *EBP-8*.

```
00405CC2  6A 40            PUSH 40
00405CC4  68 00300000      PUSH 3000
00405CC9  51               PUSH ECX
00405CCA  6A 00            PUSH 0
00405CCC  68 4A0DCE09      PUSH 9CE0D4A
00405CD1  E8 E2FDFFFF      CALL buritos.00405AB8
```

The next *CALL* to 00405AB8 is a wrapper over *CALL 00405A01* (which was a *GetProcAddress* analogue). Using *CALL 00405AB8*, will be like actually executing a kernel API function.
It has 5 arguments pushed to the stack. The last one is the hashed name of the API function taken by 00405A01, and the rest 4 arguments are the paramaters that will be passed to that API function.

To sort things up, if I have the following code:

```
PUSH 1000
PUSH CEF2EDA8
CALL buritos.00405AB8
```

it will execute *Sleep(0x1000)*, because as I already found out, 0xCEF2EDA8 equals the *Sleep* function, and 1000 will be the argument passed to this particular function.

Now let's take a look into that wrapper at 00405AB8

```
00405AB8  55              PUSH EBP
00405AB9  8BEC            MOV EBP,ESP
00405ABB  FF75 08         PUSH DWORD PTR SS:[EBP+8]
00405ABE  E8 3EFFFFFF     CALL buritos.00405A01
00405AC3  C9              LEAVE
00405AC4  870424          XCHG DWORD PTR SS:[ESP],EAX
00405AC7  874424 04       XCHG DWORD PTR SS:[ESP+4],EAX
00405ACB  C3              RETN
```

After the process address is taken, a *LEAVE* instruction follows, and two *XCHG*, that will exchange *EAX* value with the first and the second stack *DWORD*s.
By doing so, the *RETN* at 00405ACB will drop me into the current instruction entry point, so I'll step over the *CALL* to 00405A01 and look what *EAX* holds.

```
EAX 7C809A81 kernel32.VirtualAlloc
```

A *VirtualAlloc* handle, and by knowing the parameters 0x00, *ECX*, 0x3000 and 0x40 I can figure out what will be allocated, using the MSDN documentation:

```
LPVOID WINAPI VirtualAlloc(
  __in_opt  LPVOID lpAddress,
  __in      SIZE_T dwSize,
  __in      DWORD flAllocationType,
  __in      DWORD flProtect
);
```

Then, *ECX* (0x0000146E) is the size that will be allocated, *flAllocationType* will be *MEM_COMMIT|MEM_RESERVE* (because 0x3000 means exactly that) and *flProtect* will be *PAGE_EXECUTE_READWRITE* (0x40), and *EAX* will hold the pointer to the newly allocated memory region.
As I mentioned before, the last two *XCHG* instructions are pointers to *kernel32.VirtualAlloc* function, and the current return address from this *CALL*.

```
0012FF74  7C809A81  kernel32.VirtualAlloc
0012FF78  00405CD6  RETURN to buritos.00405CD6 from buritos.00405AB8
```

Everything seems right, so I will put a breakpoint at the return address 00405CD6 and execute the *VirtualAlloc* function.

```
00405CD1  E8 E2FDFFFF     CALL buritos.00405AB8
00405CD6  8945 FC         MOV DWORD PTR SS:[EBP-4],EAX
```

The allocation was successful, and *EAX* now holds a pointer to the allocated space (in my case 00830000), that is saved to *ESP-4*.

```
00405CD9  8DB3 10184000   LEA ESI,DWORD PTR DS:[EBX+401810]
00405CDF  8B7D FC         MOV EDI,DWORD PTR SS:[EBP-4]
00405CE2  8B4D F8         MOV ECX,DWORD PTR SS:[EBP-8]
00405CE5  03DF            ADD EBX,EDI
00405CE7  2BDE            SUB EBX,ESI
00405CE9  F3:A4           REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
00405CEB  8D83 631B4000   LEA EAX,DWORD PTR DS:[EBX+401B63]
00405CF1  FFE0            JMP EAX
```

At address 00405CD9, in *ESI* is moved the address of *EBX* (containing 0x4190) plus 00401810, that is pointing here:

```
Address   Hex dump                                        ASCII
004059A0  4C 69 67 68 74 79 20 43 6F 6D 70 72 65 73 73 6F  Lighty Compresso
004059B0  72 00 16 0E 00 00 D0 1C 8F 75 00 26 00 00 E4 38  r.▬♫..╨∟Пu.&..ф8
004059C0  6D 0E 00 00 00 00 55 8B EC 60 33 C0 8B 55 08 8B  m♫....UЛь`3└ЛU◘Л
```

Then *EDI* takes the pointer of our newly allocated space 00830000, and *ECX* gets the value of 0x146E from *EBP-8*.
Few lines below a *REP MOVS* instruction, will move 0x146E bytes from where *ESI* points to the allocated space pointed by *EDI*.

Finally, *EAX* gets a pointer to 00830353 (*EBX*, now containing 0x0042E7F0 plus 0x00401B63) and jumps on that address by *JMP EAX* at 00405CF1.

```
00830353  8DB3 681E4000   LEA ESI,DWORD PTR DS:[EBX+401E68]
00830359  FFB3 22184000   PUSH DWORD PTR DS:[EBX+401822]
0083035F  56              PUSH ESI
00830360  E8 C1FCFFFF     CALL 00830026
00830365  3B83 26184000   CMP EAX,DWORD PTR DS:[EBX+401826]
0083036B  0F85 DB020000   JNZ 0083064C
00830371  FFB3 2A184000   PUSH DWORD PTR DS:[EBX+40182A]
```

Here, the first *CALL* to 00830026 does exactly what this on 004059C6 was doing - it hashes a string, so lets check the arguments.

```
0012FF84  00830658  RETURN to 00830658 from 00830118
0012FF88  00000E16
```

The first argument is *ESI*, and it holds a pointer to 00830658:

```
Address   Hex dump
00830658  4D 38 5A 90 38 03 66 02 04 09 71 FF 81 B8 C2 91
00830668  01 40 C2 15 C7 D0 08 5D 70 50 45 C2 08 4C 01 03
00830678  9E 14 70 E0 1C 02 01 0B 90 08 2B 1E E9 A1 E0 11
00830688  20 22 10 57 30 EE 40 0C 02 AE 1F 8E 08 D0 A2 0D
```

The highlighted bytes are actually the encrypted "*MZ*" header.

The second argument is 0xE16 and that was a length parameter. So, the function at 00830026 will create a checksum of this data.

I'll step over that call and lets see the next comparison at 00830365.

```
DS:[00830016]=758F1CD0
EAX=758F1CD0
```

Both value equals, but in case they don't match, the jump will be taken to *ExitProcess* function. The checksum is compared against a value stored at 00830016 (*EBX+401826*), pointing here:

```
Address   Hex dump                                          ASCII
00830000  4C 69 67 68 74 79 20 43 6F 6D 70 72 65 73 73 6F  Lighty Compresso
00830010  72 00 16 0E 00 00 D0 1C 8F 75 00 26 00 00 E4 38
00830020  6D 0E 00 00 00 00 55 8B EC 60 33 C0 8B 55 08 8B
00830030  4D 0C 3B C8 75 0A 8B FA AE 75 FD 4F 8B CF 2B CA
```

```
00830371  FFB3 2A184000   PUSH DWORD PTR DS:[EBX+40182A]
00830377  6A 40           PUSH 40
00830379  68 3174BC7F     PUSH 7FBC7431
0083037E  E8 95FDFFFF     CALL 00830118
00830383  8945 F4         MOV DWORD PTR SS:[EBP-C],EAX
00830386  0BC0            OR EAX,EAX
00830388  0F84 BE020000   JE 0083064C
```

The next call is again an API function call, this time with two parameters. Again like before, I'll step into the call to see what's the API function executed here, and this time it's a *GlobalAlloc* with parameters *uFlags* = *GPTR* and *dwBytes* = 0x2600. The *dwBytes* parameter is taken by *EBX+40182A* (0083001A) here:

```
Address   Hex dump                                          ASCII
00830000  4C 69 67 68 74 79 20 43 6F 6D 70 72 65 73 73 6F  Lighty Compresso
00830010  72 00 16 0E 00 00 D0 1C 8F 75 00 26 00 00 E4 38
00830020  6D 0E 00 00 00 00 55 8B EC 60 33 C0 8B 55 08 8B
00830030  4D 0C 3B C8 75 0A 8B FA AE 75 FD 4F 8B CF 2B CA
```

In my case, the memory gets allocated at 001429E0 and stored in *EBP-C*. Then the *GlobalAlloc* result is checked and if the memory cannot be allocated, the *JE* jumps to *ExitProcess*.

```
0083038E  8BF8            MOV EDI,EAX
00830390  56              PUSH ESI
00830391  57              PUSH EDI
00830392  E8 95FDFFFF     CALL 0083012C
00830397  3B83 2A184000   CMP EAX,DWORD PTR DS:[EBX+40182A]
0083039D  0F85 A9020000   JNZ 0083064C
```

The next *CALL* is to address 0083012C and it gets two parameters. The first one is the newly allocated by *GlobalAllocate* buffer, and the second one is the the beginning of the compressed "*MZ*" header at 00830658.

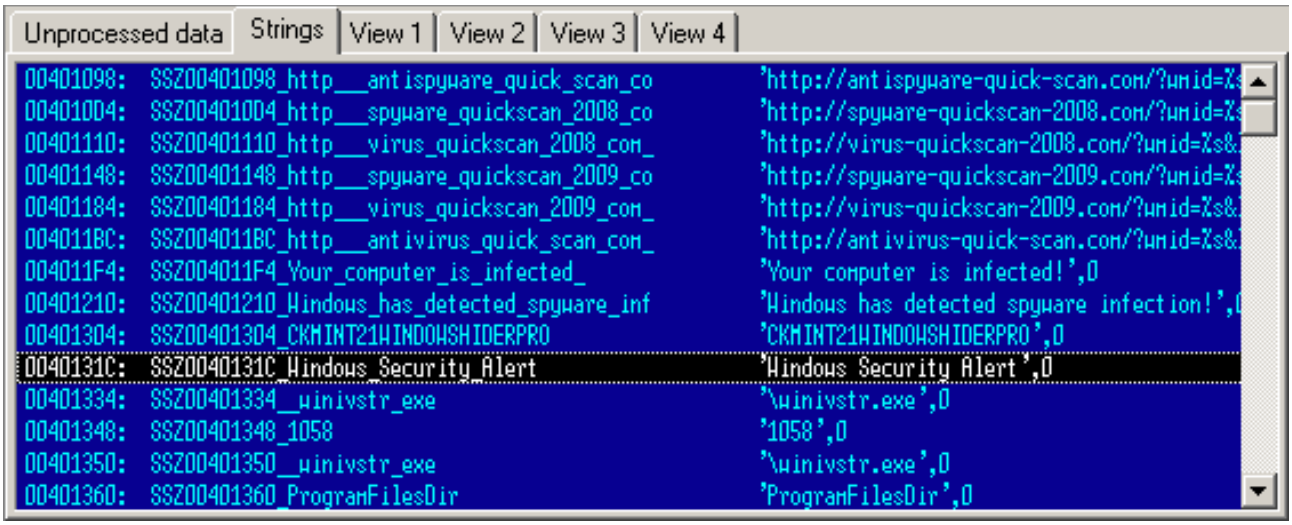
Looks like that *CALL* will uncompress the packed program, so I'll take a better look.

```
0083012C  55              PUSH EBP
0083012D  8BEC            MOV EBP,ESP
0083012F  60              PUSHAD
00830130  8B75 0C         MOV ESI,DWORD PTR SS:[EBP+C]
00830133  8B7D 08         MOV EDI,DWORD PTR SS:[EBP+8]
00830136  FC              CLD
00830137  B2 80           MOV DL,80
00830139  33DB            XOR EBX,EBX
0083013B  A4              MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
0083013C  B3 02           MOV BL,2
0083013E  E8 6D000000     CALL 008301B0
00830143 ^73 F6           JNB SHORT 0083013B
00830145  33C9            XOR ECX,ECX
00830147  E8 64000000     CALL 008301B0
0083014C  73 1C           JNB SHORT 0083016A
0083014E  33C0            XOR EAX,EAX
008301CC  2B7C24 28       SUB EDI,DWORD PTR SS:[ESP+28]
008301D0  897C24 1C       MOV DWORD PTR SS:[ESP+1C],EDI
008301D4  61              POPAD
008301D5  C9              LEAVE
008301D6  C2 0800         RETN 8
```

The function is not that long, and I can use it to create an unpacker, so I'll save a copy of the code for later, but for now I'll just execute the decoding call, to get the unpacked data.
After the unpacker *CALL*, a check compares the result of that call, with the size of the global allocated buffer, and if they match the code will continue down. However, if they don't, *JNZ* 0083064C will terminate the program with *ExitProcess*. Well, they do match, so let's take a look at whats just got decoded.

```
Address   Hex dump                                          ASCII
001429E0  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZP.♥...♦...  ..
001429F0  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ╕.......@.......
00142A00  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

The data is now unpacked in to the allocated by *GlobalAlloc* buffer, then I'll take a memory dump and view it in *PE Explorer*'s disassembler.



Looks promising, but I don't want to be a party killer and spoil my own fun, so let's leave that for later and continue with the unpacking process.

```
008303A3  50              PUSH EAX
008303A4  57              PUSH EDI
008303A5  E8 7CFCFFFF     CALL 00830026
008303AA  3B83 2E184000   CMP EAX,DWORD PTR DS:[EBX+40182E]
008303B0  0F85 96020000   JNZ 0083064C
```

What's next? Another checksum, and this time for the uncompressed data. If the two checked values don't match, the program will be terminated again by *ExitProccess*.

The sum of *EBX+40182E* point to 0083001E:

```
Address  Hex dump                                          ASCII
00830000 4C 69 67 68 74 79 20 43 6F 6D 70 72 65 73 73 6F  Lighty Compresso
00830010 72 00 16 0E 00 00 D0 1C 8F 75 00 26 00 00 E4 38  r.._Я..Ы_Пu.&..ф8
00830020 6D 0E 00 00 00 00 55 8B EC 60 33 C0 8B 55 08 8B  m♫....UЛь`3└ЛU◘Л
00830030 4D 0C 3B C8 75 0A 8B FA AE 75 FD 4F 8B CF 2B CA  M.;╚u.Л·ou▬OЛ╧+╩
```

And yet another *DWORD* has been decoded from *Lighty's* stub...

```
008303B6  8B4D F4          MOV ECX,DWORD PTR SS:[EBP-C]
008303B9  0349 3C          ADD ECX,DWORD PTR DS:[ECX+3C]
008303BC  6A 00            PUSH 0
008303BE  54               PUSH ESP
008303BF  6A 40            PUSH 40
008303C1  FF71 50          PUSH DWORD PTR DS:[ECX+50]
008303C4  FF75 08          PUSH DWORD PTR SS:[EBP+8]
008303C7  68 2F6F0610      PUSH 10066F2F
008303CC  E8 47FDFFFF      CALL 00830118
008303D1  58               POP EAX
008303D2  EB 77            JMP SHORT 0083044B
```

Here, a API call with 5 parameters (six, if we include the 10066F2F, that define the API function).
At address 008303B6, *ECX* get a pointer to the global allocated memory – 001429E0. Adding 0x3C to it, will point to the uncompressed *DOS_PEOffset* value, and adding its value to *ECX* itself will make *ECX* pointing to the beginning of the *PE* header. Later, *ECX* is used at 008303C1, by adding 0x50 to it, that will point to the *PE*'s *SizeOfImage* value, which is 0x5000. Used as first parameter to the API function, *ESP+8* holds the current *ImageBase* address - 0x00400000, and finally *ESP* used as forth parameter pointing at 0012FF74.

So, what's that API function?

```
EAX 7C801AD0 kernel32.VirtualProtect
```

```
BOOL WINAPI VirtualProtect(
  __in   LPVOID lpAddress      : current Image Base, 00400000
  __in   SIZE_T dwSize         : Unpacked's SizeOfImage 0x5000;
  __in   DWORD flNewProtect    : 0x40, or PAGE_EXECUTE_READWRITE
  __out  PDWORD lpflOldProtect : buffer DWORD, pointed by ESP
);
```

So that 0 pushed at address 008303BC is popped back to *EAX* at 008303D1 and it's not part of that API call.

```
0083044B  8B7D 08          MOV EDI,DWORD PTR SS:[EBP+8]
0083044E  8B75 F4          MOV ESI,DWORD PTR SS:[EBP-C]
00830451  8BCE             MOV ECX,ESI
00830453  034E 3C          ADD ECX,DWORD PTR DS:[ESI+3C]
00830456  8B49 54          MOV ECX,DWORD PTR DS:[ECX+54]
00830459  F3:A4            REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
0083045B  8B5D 08          MOV EBX,DWORD PTR SS:[EBP+8]
0083045E  035B 3C          ADD EBX,DWORD PTR DS:[EBX+3C]
00830461  0FB74B 06        MOVZX ECX,WORD PTR DS:[EBX+6]
00830465  0FB773 14        MOVZX ESI,WORD PTR DS:[EBX+14]
00830469  8D741E 18        LEA ESI,DWORD PTR DS:[ESI+EBX+18]
```

I already know that, *EBP+8* moved to *ESI*, holds the current *ImageBase* 00400000 and *EBP-C* moved to *ESI*, is the global allocated buffer, containing the uncompressed data of the original malware.

Then a *REP MOVS* at address 0083459, will move the data from the uncompressed malware, to the current image base address, overwriting the current process into the memory.
The instruction *REP* uses *ECX* as counter, so to determine how many bytes will be moved, I'll have to see what *ECX* will contain there.
First, *ECX* gets *ESI*'s value - 001429E0. Then, *ESI+3C*, pointing at the *DOS_PEOffset* is added to *ECX*, and at 00830453, *ECX* will hold the beginning of the *PE* header.

Finally, *ECX* is replaced by *ECX+54* (pointing to *PE*'s *SizeOfHeaders* value - in this case 0x400), and I now know that 0x400 bytes will be moved from the decoded data to the current image base. So, that *REP MOVS* will move all the *MZ* & *PE* data, plus the *PE Sections*, and basically everything between the beginning of the file, and the beginning of its first - in this case "*.data*" section.

Then, the current *ImageBase* address is moved to *EBX*, and agin using 0x3C as offset and adding it to *EBX*, at 0083045E will point to the current process *PE* header.
A *MOVZX* to *ECX* will take the *PE*'s *NumberOfSections*, and another *MOVZX*, this time to *ESI* takes the *PE*'s *SizeOfOptionalHeader* value.

At the end, *ESI* takes a pointer to *SizeOfOptionalHeader+EBX+0x18*, that will point to the beginning of the *PE* Sections - "*.text*", "*.data*" and "*.reloc*".

```
0083046D  56              PUSH ESI
0083046E  51              PUSH ECX
0083046F  8B7D 08         MOV EDI,DWORD PTR SS:[EBP+8]
00830472  037E 0C         ADD EDI,DWORD PTR DS:[ESI+C]
00830475  8B4E 08         MOV ECX,DWORD PTR DS:[ESI+8]
00830478  8B43 38         MOV EAX,DWORD PTR DS:[EBX+38]
0083047B  48              DEC EAX
0083047C  03C8            ADD ECX,EAX
0083047E  F7D0            NOT EAX
00830480  23C8            AND ECX,EAX
00830482  C1E9 02         SHR ECX,2
00830485  33C0            XOR EAX,EAX
00830487  F3:AB           REP STOS DWORD PTR ES:[EDI]
00830489  8B4E 10         MOV ECX,DWORD PTR DS:[ESI+10]
0083048C  8B7E 0C         MOV EDI,DWORD PTR DS:[ESI+C]
0083048F  8B76 14         MOV ESI,DWORD PTR DS:[ESI+14]
00830492  0375 F4         ADD ESI,DWORD PTR SS:[EBP-C]
00830495  037D 08         ADD EDI,DWORD PTR SS:[EBP+8]
00830498  F3:A4           REP MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
0083049A  59              POP ECX
0083049B  5E              POP ESI
0083049C  83C6 28         ADD ESI,28
0083049F ^E2 CC           LOOPD SHORT 0083046D
```

A loop using *REP MOVS*, and a *ECX* counter (containing the *NumberOfSections*) will run three times, to transfer all three sections "*.text*", "*.data*", "*.reloc*" to the current process memory.

```
008304A1  FF75 F4         PUSH DWORD PTR SS:[EBP-C]
008304A4  68 9D1E6B63     PUSH 636B1E9D
008304A9  E8 6AFCFFFF     CALL 00830118
008304AE  8B8B 84000000   MOV ECX,DWORD PTR DS:[EBX+84]
008304B4 vE3 66           JECXZ SHORT 0083051C
```

Another API call - this time *GlobalFree* will free the previously allocated memory at 001429E0. (I'll dump that buffer, before it gets freed because it contains the uncompressed malware and I can use it later to dump it in it's original state.)
A *MOV* is taking the *IAT* size from the current process, stores it in *ECX* and then *JECXZ* is jumping depending on the *ECX* value.
In this case, the *IAT* size is 0x8C, so the jump will not be taken.

```
008304B6  8B9B 80000000   MOV EBX,DWORD PTR DS:[EBX+80]
008304BC  035D 08         ADD EBX,DWORD PTR SS:[EBP+8]
```

EBX takes the *IAT* address, then the *ImageBase* is added to it, so *EBX* will point at the beginning of the *IAT*.

```
008304BF  8B73 0C          MOV ESI,DWORD PTR DS:[EBX+C]
008304C2  0BF6             OR ESI,ESI
008304C4  74 56            JE SHORT 0083051C
008304C6  0375 08          ADD ESI,DWORD PTR SS:[EBP+8]
008304C9  56               PUSH ESI
008304CA  68 706586B1      PUSH B1866570
008304CF  E8 44FCFFFF      CALL 00830118
008304D4  0BC0             OR EAX,EAX
008304D6  75 0B            JNZ SHORT 008304E3
008304D8  56               PUSH ESI
008304D9  68 8DBDC13F      PUSH 3FC1BD8D
008304DE  E8 35FCFFFF      CALL 00830118
008304E3  8BD0             MOV EDX,EAX
008304E5  8B7B 10          MOV EDI,DWORD PTR DS:[EBX+10]
008304E8  8B33             MOV ESI,DWORD PTR DS:[EBX]
008304EA  0BF6             OR ESI,ESI
008304EC  75 02            JNZ SHORT 008304F0
008304EE  8BF7             MOV ESI,EDI
008304F0  0375 08          ADD ESI,DWORD PTR SS:[EBP+8]
008304F3  037D 08          ADD EDI,DWORD PTR SS:[EBP+8]
008304F6  AD               LODS DWORD PTR DS:[ESI]
008304F7  0BC0             OR EAX,EAX
008304F9  74 1C            JE SHORT 00830517
008304FB  0FBAF0 1F        BTR EAX,1F
008304FF  72 05            JB SHORT 00830506
00830501  0345 08          ADD EAX,DWORD PTR SS:[EBP+8]
00830504  40               INC EAX
00830505  40               INC EAX
00830506  52               PUSH EDX
00830507  50               PUSH EAX
00830508  52               PUSH EDX
00830509  68 FF1F7CC9      PUSH C97C1FFF
0083050E  E8 05FCFFFF      CALL 00830118
00830513  5A               POP EDX
00830514  AB               STOS DWORD PTR ES:[EDI]
00830515  EB DF            JMP SHORT 008304F6
00830517  83C3 14          ADD EBX,14
0083051A  EB A3            JMP SHORT 008304BF
0083051C  8B55 08          MOV EDX,DWORD PTR SS:[EBP+8]
```

Two nested loops are executed here - inner from 008304F6 to 00830515 and outer from 008304BF to 0083051A.
Three API calls are executed here - at 0083050E (with parameter 0xC97C1FFF), 008304CF (taking 0xB1866570 as parameter) and 008304DE (with 0x3FC1BD8D as parameter) so I'll have to check them.

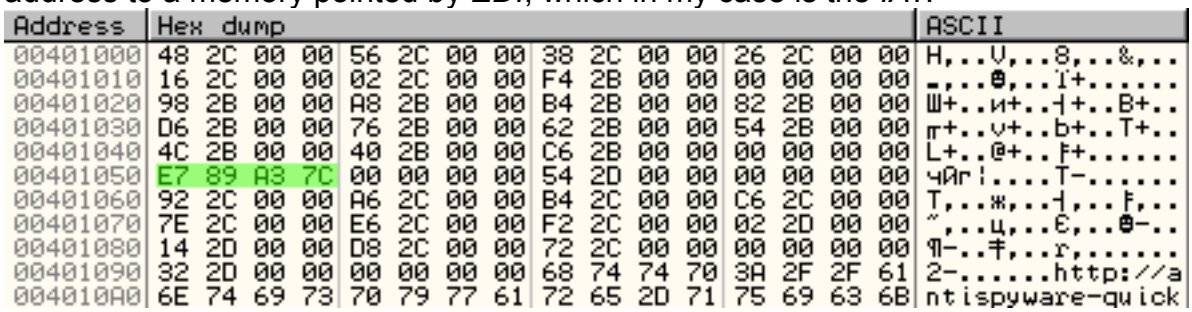
That first jump at address 008304C4 will take me right after the inner loop, so that's the loop terminator.
A *LODS-STOS* is also present, so there's some data moved here, but first things first.

Let's see all these calls and what will they do.
The first one at 008304CF, gets only one parameter and in the stack it says "*SHELL32.dll*".
`EAX 7C80B529 kernel32.GetModuleHandleA`
It's a *kernel32.GetModuleHandleA* call which will get the handle of "*SHELL32.dll*".

Then the second at 008304DE is executed only if the first one's result is *FALSE*. So if *GetModuleHandeA* fails:
`EAX 7C801D77 kernel32.LoadLibraryA`
a *LoadLibrary* is trying to get the library handle.

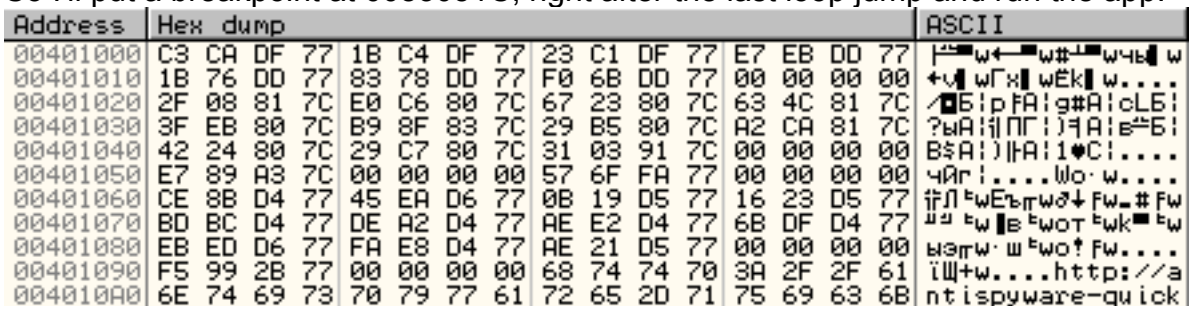
The last call at address 0083050E is a *kernel32.GetProcAddress*, on the first run taking the process address of *shell32*'s "*Shell_NotifyIconA*" function.

After the last *CALL*, the *STOS DWORD PTR ES:[EDI]* at 00830514 will save the process address to a memory pointed by *EDI*, which in my case is the *IAT*:



Seems like, the whole idea of those loops is to fill the current *IAT* with the API function handles. This way the unpacked code can be run from the memory, without needing to reinitialize these handles.

So I'll put a breakpoint at 0083051C, right after the last loop jump and run the app.



All the handles are filled now.

```
0083051C  8B55 08          MOV EDX,DWORD PTR SS:[EBP+8]
0083051F  8B5D 08          MOV EBX,DWORD PTR SS:[EBP+8]
00830522  035B 3C          ADD EBX,DWORD PTR DS:[EBX+3C]
00830525  2B53 34          SUB EDX,DWORD PTR DS:[EBX+34]
00830528  74 4D            JE SHORT 00830577
```

Here, both *EDX* and *EBX* take a pointer to the current *ImageBase* - 00400000. *EBX* is then added to the the *PE*'s *DOS_PEOffset* value, so *EBX* will point to *PE* header.
Then, from *EDX* (current *ImageBase*) is subtracted the overwritten *ImageBase*. Since both are 00400000, the *JE* to 00830577 will be taken. Most probably this is some compatibility check for compressed *DLL*'s or applications with different than the standard 400000 *ImageBase*.

```
00830577  8B8B C4000000    MOV ECX,DWORD PTR DS:[EBX+C4]
0083057D  E3 60            JECXZ SHORT 008305DF
```

EBX+C4 points to the *PE*'s *TLS_Table_size*, and then gets checked if equals zero. This application doesn't have a *TLS Table*, so the size is 0, and the jump to 008305DF is taken.

```
008305DF  64:A1 30000000   MOV EAX,DWORD PTR FS:[30]
008305E5  8B40 0C          MOV EAX,DWORD PTR DS:[EAX+C]
008305E8  8B40 0C          MOV EAX,DWORD PTR DS:[EAX+C]
008305EB  8B48 18          MOV ECX,DWORD PTR DS:[EAX+18]
```

Another *PEB* data reading is between 008305DF and 008305EB. The *PEB* pointer is taken in *EAX*. Then *EAX* gets a pointer to *PEB->LoaderData*, which was a *PPEB_LDR_DATA* structure. From that structure, in *EAX* is taken a pointer to *InLoadOrderModuleList* (a *LDR_MODULE* structure). And finally *ECX* stores the *BaseAddress* of the current image – 00400000.
So *ECX* is the *BaseAddress* 00400000 and *EAX* is the *InLoadOrderModuleList->LDR_MODULE* pointer.

```
008305F1  3950 18          CMP DWORD PTR DS:[EAX+18],EDX
008305F4 v74 07            JE SHORT 008305FD
008305F6  8B00             MOV EAX,DWORD PTR DS:[EAX]
008305F8  3948 18          CMP DWORD PTR DS:[EAX+18],ECX
008305FB ^75 F4            JNZ SHORT 008305F1
008305FD  8B5B 28          MOV EBX,DWORD PTR DS:[EBX+28]
00830600  035D 08          ADD EBX,DWORD PTR SS:[EBP+8]
00830603  8958 1C          MOV DWORD PTR DS:[EAX+1C],EBX
00830606  895C24 1C        MOV DWORD PTR SS:[ESP+1C],EBX
0083060A  8B5D F8          MOV EBX,DWORD PTR SS:[EBP-8]
0083060D  895C24 18        MOV DWORD PTR SS:[ESP+18],EBX
00830611  8B5D FC          MOV EBX,DWORD PTR SS:[EBP-4]
00830614  895C24 14        MOV DWORD PTR SS:[ESP+14],EBX
00830618  61               POPAD
00830619  C9               LEAVE
0083061A  890424           MOV DWORD PTR SS:[ESP],EAX
0083061D  33C0             XOR EAX,EAX
0083061F  C2 0400          RETN 4
```

I'm getting closer to the *RETN*, but what do we have here? A small loop between 008305F1 and 008305FB.
EAX is still holding the *PEB*'s *LDR_MODULE* structure, so *EAX+18* will point to *LDR_MODULE*'s *BaseAddress*, which is 00400000, just like the one *EDX* holds, so the jump will be taken to 008305FD.
Then at 008305FD, *EBX* is replaced by *EBX+28*. *EBX* is pointing at the current *PE* header, so adding 0x28 to it, will be the *AddressOfEntryPoint* value - in this case 0x2040.
EBP+8 is added to *EBX* and that builds the address 00402040 into *EBX*.
The *PEB*'s *LDR_MODULE EntryPoint* is replaced then at 00830603, by *EBX*. Basically, The current process *EntryPoint* is set to 00402040.
Next, stack entries 8, 7 and 6, are replaced at 00830606, 0083060D and 00830614 with 00402040 (the new entry point), 0x146E (from *EBP-8*) and pointer to 00830000 (from *EBP-4*), where the virtual allocated space is.

```
0012FF8C  7C910738 ntdll.7C910738
0012FF90  FFFFFFFF
0012FF94  0012FFB8
0012FF98  0012FFAC
0012FF9C  7FFDF000
0012FFA0  00830000 ASCII "Lighty Compressor"
0012FFA4  0000146E
0012FFA8  00402040 buritos.00402040
0012FFAC  001429E0
0012FFB0  0000146E
```

A *POPAD* pops back the stack values to the registers *EAX*, *ECX* and *EDX*.

```
Registers (MMX)
EAX 00402040 buritos.00402040
ECX 0000146E
EDX 00830000 ASCII "Lighty Compressor"
```

Finally, the first stack value is replaced with *EAX* (the new entry point 00402040) and *RETN* sends me here:

```
00402040    55             DB 55
00402041    8B             DB 8B
00402042    EC             DB EC
00402043    81             DB 81
00402044    EC             DB EC
00402045    10             DB 10
00402046    01             DB 01
00402047    00             DB 00
00402048    00             DB 00
00402049    E8             DB E8
0040204A    F2             DB F2
0040204B    05             DB 05
0040204C    00             DB 00
```

Oh... *Ctrl+A*

```
00402040  .  55              PUSH EBP
00402041  .  8BEC            MOV EBP,ESP
00402043  .  81EC 10010000   SUB ESP,110
00402049  .  E8 F2050000     CALL buritos.00402640
0040204E  .  83F8 02         CMP EAX,2
00402051  .v74 0A            JE SHORT buritos.0040205D
00402053  .  E8 68080000     CALL buritos.004028C0
00402058  .  83F8 01         CMP EAX,1
0040205B  .v75 08            JNZ SHORT buritos.00402065
0040205D  >  6A 00           PUSH 0                          ExitCode = 0
0040205F  .  FF15 3C104000   CALL DWORD PTR DS:[40103C]      ExitProcess
00402065  >  E8 26070000     CALL buritos.00402790
```

And thats the original malware entry point. Now I can make a dump and fix the *IAT* with *ImportRec*, and I did so, but

Size:	68,0 KB (69 632 bytes)
Size on disk:	68,0 KB (69 632 bytes)

that size, 68KB looks way too much. The file is working and can be analyzed as it is, but still, 68K?
I'll get rid off that fat dump, and rebuilt it myself from the memory dump, saved before *GlobalFree*-ing the memory holding it.

Because OllyDbg doesn't have a very flexible memory dumping functionality, the *MEM* file contain bulk data at the beginning and the end that I'll have to cut off.

The best way to determine where is the beginning of the program in that memory dump is to search for the "*MZ*" header like this:

```
OFFSET     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0000:29E0  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  M  Z  ╡     □           □           ¤  ¤
0000:29F0  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ë                       @
0000:2A00  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

and everything before offset 0x29E0 should be removed.

As I said, there is bulk data at the end too, that is in fact not necessary to remove, but I'll do it anyway.
To get to the programs end I'll use the *GlobalAlloc* size parameter as file size, and it was 0x2600, so 0x29E0+0x2600 = offset 0x4FE0.

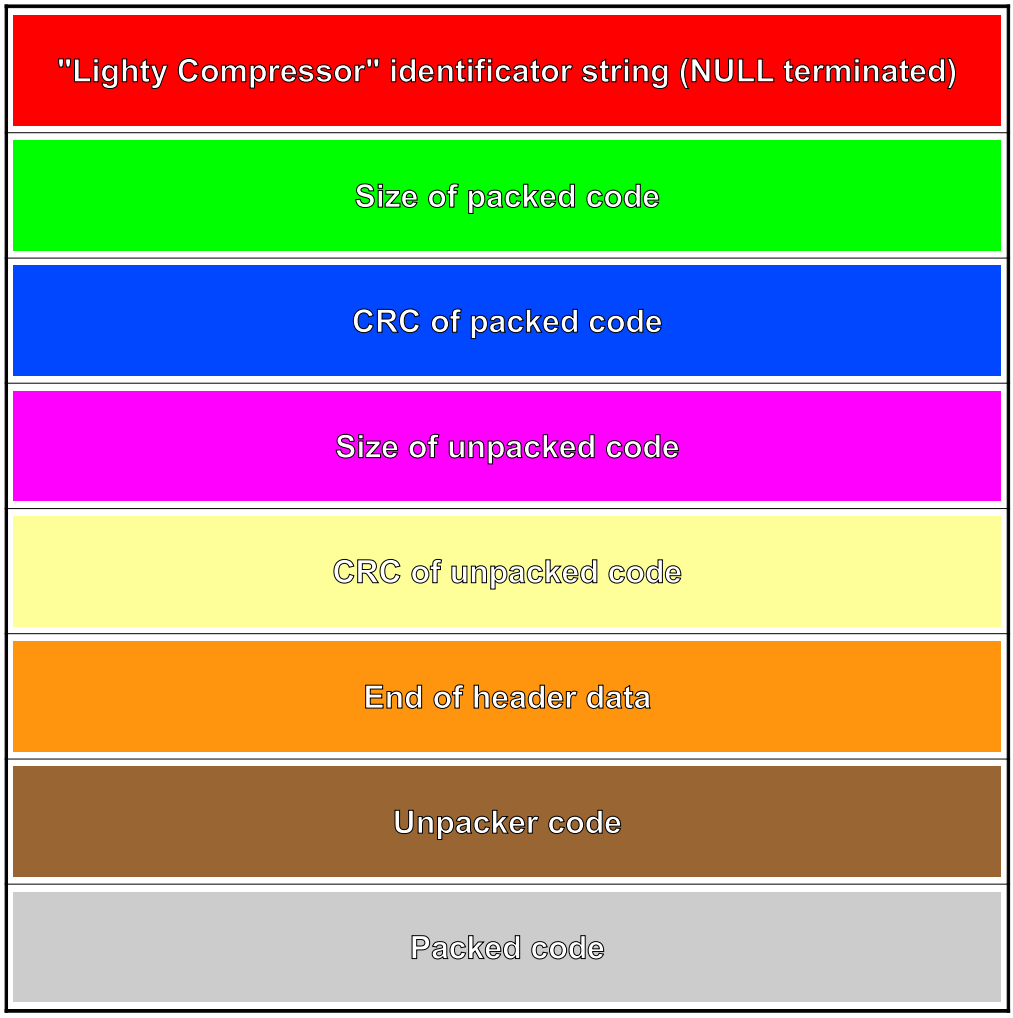
```
OFFSET     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0000:4FE0  AB AB AB AB AB AB AB AB 00 00 00 00 00 00 00 00  «  «  «  «  «  «  «  «
0000:4FF0  02 02 C3 04 EE 14 EE 00 78 01 14 00 78 01 14 00  □  □  Γ  □  o  □  o     x  □  □     x  □  □
0000:5000  EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE  o  ■  o  ■  o  ■  o  ■  o  ■  o  ■  o  ■  o  ■
```

So removing the bulk data will produce a 9728 bytes long EXE – much less than the dumped one.

And that's all!
Unpacking "*Lighty Compressed*" files is not hard at all.
The laziest way is to set a breakpoint at *GlobalAlloc*, and *GlobalFree*. Then, when Olly breaks on *GlobalAlloc*, write down the Size argument. Run the application until *GlobalFree* breakpoint is triggered.
Save the memory buffer that will be freed, then strip the bulk data from the beginning and the end of the memory dump, and you will get the original, unpacked executable!

Appendix A: "Lighty compressor" structure

```
0000:0000  4C 69 67 68 74 79 20 43 6F 6D 70 72 65 73 73 6F  L i g h t y   C o m p r e s s o
0000:0010  72 00 16 0E 00 00 D0 1C 8F 75 00 26 00 00 E4 38  r □ □ □ □ □ □ □ □ □ □ □ □ □ □ □
0000:0020  6D 0E 00 00 00 00 55 8B EC 60 33 C0 8B 55 08 8B  m □ □ □ □ □ U < м ` 3 A < U □ <
0000:0030  4D 0C 3B C8 75 0A 8B FA AE 75 FD 4F 8B CF 2B CA  M □ ; И u □ < ъ ® u э O < П + К
0000:0650  97 10 25 E8 C0 FA FF FF 4D 38 5A 90 38 03 66 02  — □ % и А ъ я я M 8 Z ђ 8 □ f □
0000:0660  04 09 71 FF 81 B8 C2 91 01 40 C2 15 C7 D0 08 5D  □ □ q я Ѓ ё В ` □ @ В □ З Р □ ]
0000:0670  70 50 45 C2 08 4C 01 03 9E 14 70 E0 1C 02 01 0B  p P E В □ L □ □ ћ □ p а □ □ □ □
```



Appendix B: "Lighty compressor" CRC algorithm, C implementation

```
#include <windows.h>
#include <stdio.h>

int main() {

	char data[] = "ActivateActCtx";	// data to get CRC'ed
	int i, j, t, data_length;
	unsigned int result = -1;

	data_length = strlen(data);		// for binary data replace strlen with its size
	for (i = 0; i < data_length; i++) {
		result ^= data[i];
		for(j = 8; j > 0; j--) {
			t = result & 1;
			result >>= 1;
			if (t) {
				result ^= 0xEDB88320;
			}
		}
	}
	printf("%08X", ~result);		// the result is NOT-ed
	return 0;
}
```

Appendix C: "Lighty compressor" Uncompressor, FASM implementation

```
format PE GUI 4.0
entry start

include 'win32ax.inc'

section '.data' data readable writeable

file_input              db "_00830000.mem",0    ; compressed data is here
file_output             db "unpacked.bin",0     ; the unpacked binary will be here
hwnd_file_input         dd ?
hwnd_file_output        dd ?
size_file_input         dd ?
size_file_output        dd 0x2600               ; unpacked binary size (see appendix A)
hmem_file_input         dd ?
hmem_file_output        dd ?
NnumberOfBytesToRW      dd ?

section '.code' code readable executable

start:

  invoke  CreateFile,file_input,GENERIC_READ,0,NULL,\
        OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL
  mov     [hwnd_file_input],eax
  invoke  GetFileSize,[hwnd_file_input],NULL
  mov     [size_file_input],eax
  invoke  GlobalAlloc,GPTR,[size_file_input]
  mov     [hmem_file_input],eax
  invoke  ReadFile,[hwnd_file_input],[hmem_file_input],\
        [size_file_input],nNumberOfBytesToRW,NULL
  invoke  CloseHandle,[hwnd_file_input]

  invoke  GlobalAlloc,GPTR,[size_file_output]
  mov     [hmem_file_output],eax

; Begin of uncompress routine
  pushad
  mov     esi,[hmem_file_input]
  mov     edi,[hmem_file_output]
  cld
  mov     dl,0x80
  xor     ebx,ebx
label_0083013B:
  movs    byte[edi],byte[esi]
  mov     bl,0x02
label_0083013E:
  call    func_008301B0
  jnb     label_0083013B
  xor     ecx,ecx
```

```
  call    func_008301B0
  jnb     label_0083016A
  xor     eax,eax
  call    func_008301B0
  jnb     label_0083017A
  mov     bl,0x02
  inc     ecx
  mov     al,0x10
label_0083015C:
  call    func_008301B0
  adc     al,al
  jnb     label_0083015C
  jnz     label_008301A6
  stos    byte[edi]
  jmp     label_0083013E
label_0083016A:
  call    func_008301BC
  sub     ecx,ebx
  jnz     lable_00830183
  call    func_008301BA
  jmp     label_008301A2
label_0083017A:
  lods    byte[esi]
  shr     eax,0x01
  je      label_008301CC
  adc     ecx,ecx
  jmp     label_0083019F
lable_00830183:
  xchg    eax,ecx
  dec     eax
  shl     eax,0x08
  lods    byte[esi]
  call    func_008301BA
  cmp     eax,0x7D00
  jnb     label_0083019F
  cmp     ah,0x05
  jnb     label_008301A0
  cmp     eax,0x7F
  ja      label_008301A1
label_0083019F:
  inc     ECX
label_008301A0:
  inc     ECX
label_008301A1:
  xchg    eax,ebp
label_008301A2:
  mov     eax,ebp
  mov     bl,0x01
label_008301A6:
  push    esi
  mov     esi,edi
  sub     esi,eax
```

```
  rep movs byte[edi],byte[esi]
  pop     esi
  jmp     label_0083013E
label_008301CC:
  popad
; end

  invoke  CreateFile,file_output,GENERIC_WRITE,FILE_SHARE_WRITE,NULL,\
          CREATE_ALWAYS,FILE_ATTRIBUTE_NORMAL,NULL
  mov     [hwnd_file_output],eax
  invoke  WriteFile,[hwnd_file_output],[hmem_file_output],\
          [size_file_output],nNumberOfBytesToRW,NULL
  invoke  CloseHandle,[hwnd_file_output]

  invoke  GlobalFree,[hmem_file_output]
  invoke  GlobalFree,[hmem_file_input]
  invoke  ExitProcess,0

func_008301B0:
  add     dl,dl
  jnz     label_008301B9
  mov     dl,byte[esi]
  inc     esi
  adc     dl,dl
label_008301B9:
  retn

func_008301BA:
  xor     ecx,ecx
func_008301BC:
  inc     ecx
label_008301BD:
  call    func_008301B0
  adc     ecx,ecx
  call    func_008301B0
  jb      label_008301BD
  retn

data import

 library kernel32,'KERNEL32.DLL',\
         user32,'USER32.DLL'

 include 'api\kernel32.inc'
 include 'api\user32.inc'

end data
```