# STOP/DJVU - Unpacking

## Summary

In this post, I will walk through the unpacking process of a **stop/djvu** sample with the sample hash below:

SHA256 `f02b45b579b65a1ea89f2d9443f2c1a1484dec0bc66591ff4d3ad6ce63d635aa`

I got this sample from Malware Bazaar:

https://bazaar.abuse.ch/sample/f02b45b579b65a1ea89f2d9443f2c1a1484dec0bc66591ff4d3ad6ce63d635aa/

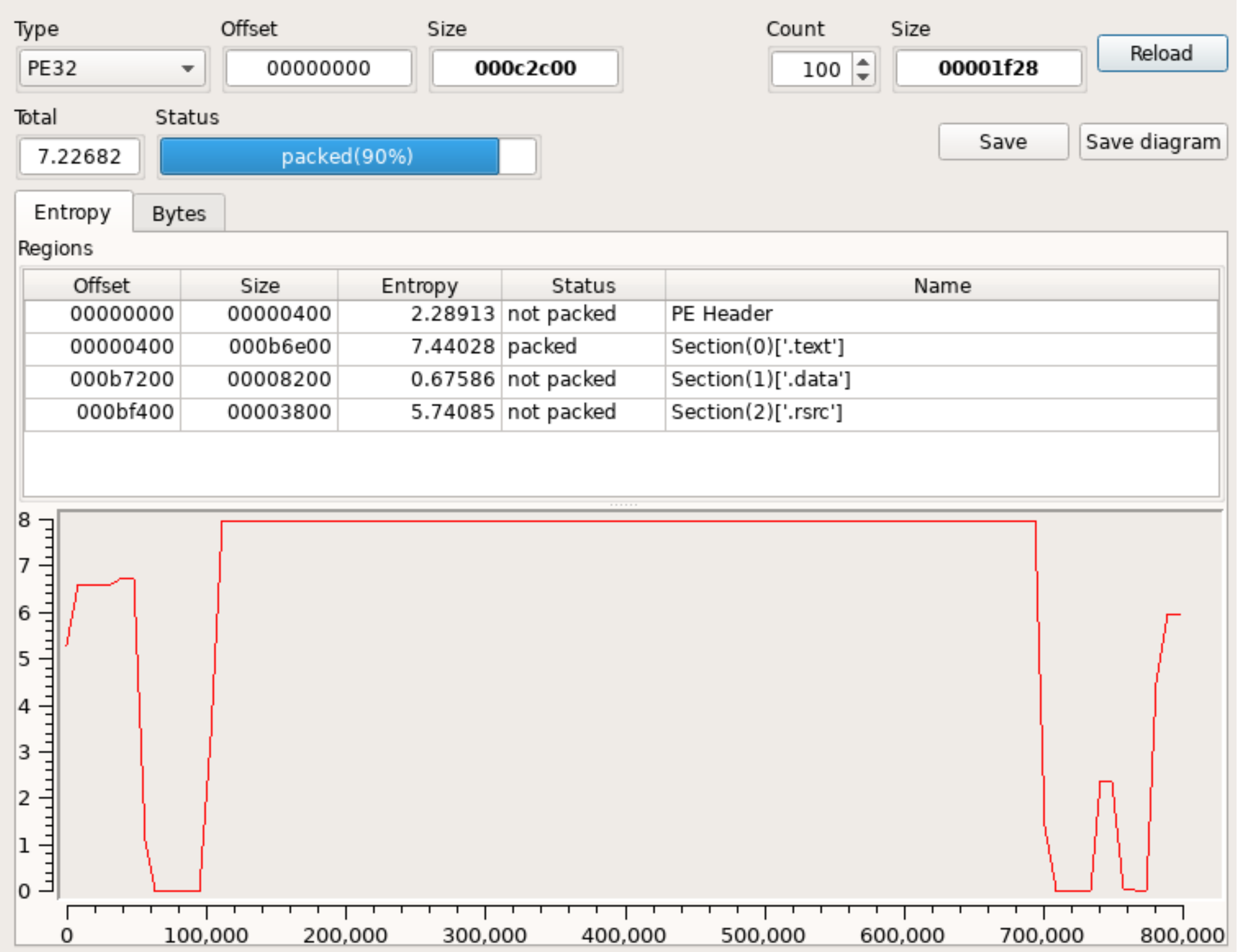
We will do:

- Static Analysis with Ghidra
- Dynamic Analysis with x32dbg, accompanied by static analysis in Cutter
- Dumping the payload with pe-sieve

This writeup was originally called "Teambot Unpacking", as Twitter user @gabbbarrr pointed out, this is, despite it's label, STOP/DJVU ransomware.

## Static Analysis

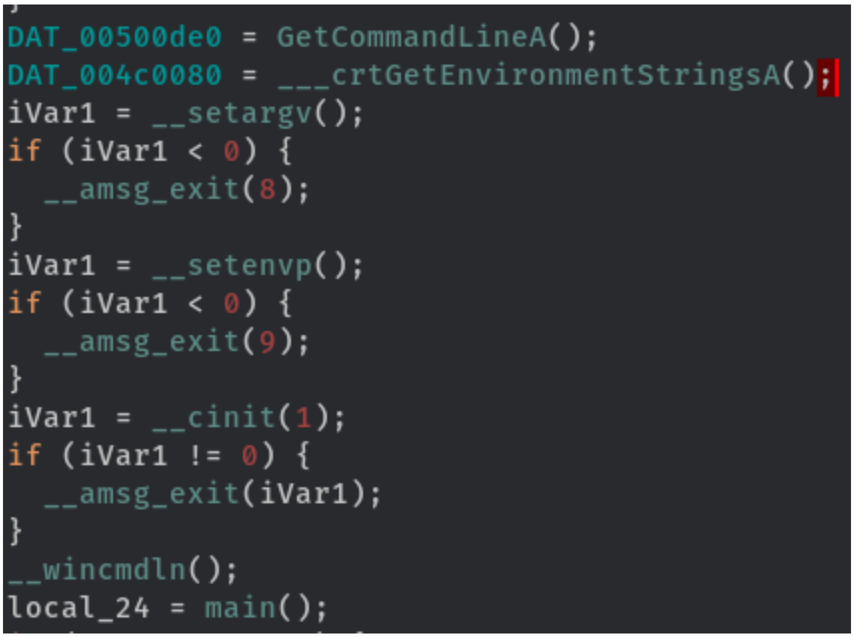
After a quick look with DetectItEasy to check the entropy, it's easy to see this sample is packed:



Type: PE32
Offset: 00000000
Size: 000c2c00
Count: 100
Size: 00001f28
Total: 7.22682
Status: packed(90%)

| Offset | Size | Entropy | Status | Name |
|---|---|---|---|---|
| 00000000 | 00000400 | 2.28913 | not packed | PE Header |
| 00000400 | 000b6e00 | 7.44028 | packed | Section(0)['.text'] |
| 000b7200 | 00008200 | 0.67586 | not packed | Section(1)['.data'] |
| 000bf400 | 00003800 | 5.74085 | not packed | Section(2)['.rsrc'] |

So, I loaded the sample into Ghidra for a closer examination. Starting from the `entry` function, it isn't very difficult to find main. After calls to `__setargv` and `__setenvp`, a call to `__wincmdln` is made. Immediately after this, the main function is called. This function is was not labled as main, so I renamed it as you can see in the image below:



```
DAT_00500de0 = GetCommandLineA();
DAT_004c0080 = ___crtGetEnvironmentStringsA();
iVar1 = __setargv();
if (iVar1 < 0) {
  __amsg_exit(8);
}
iVar1 = __setenvp();
if (iVar1 < 0) {
  __amsg_exit(9);
}
iVar1 = __cinit(1);
if (iVar1 != 0) {
  __amsg_exit(iVar1);
}
__wincmdln();
local_24 = main();
```

The main function contains various API calls. However, these seem to be noise. Except for the two functions before the return at the end, I found nothing significant here.

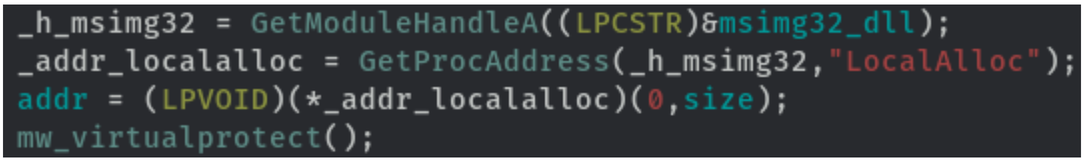
```
int iVar1;
DWORD local_244;
_RTL_CRITICAL_SECTION local_240;
_cpinfoexW local_228;

if (size == 0xc) {
  ReplaceFileA("veveleyajovuhelegivonilaxinevibehaferedijihofoludabohoni","sewijogudoyumevozepaj",
               "casihefugulajalifezabirapuzigijexosiyor",0,(LPVOID)0x0,(LPVOID)0x0);
  _feof((FILE *)0x0);
  FUN_00404286(&local_240.DebugInfo);
  ~exception(&local_240.DebugInfo);
  _atof((char *)0x0);
}
size = DAT_004bfebc;
iVar1 = 0x9ae37;
do {
  if (size == 0x15) {
    FindFirstChangeNotificationA("Kepigeyar bopugiharugopo likohep paruyaxovor tedic",0,0);
    GetConsoleAliasesLengthW((LPWSTR)0x0);
    EnterCriticalSection(&local_240);
    SetCalendarInfoA(0,0,0,"usodbfstef");
    UnregisterWait((HANDLE)0x0);
    GetCPInfoExW(0,0,&local_228);
    FillConsoleOutputCharacterA((HANDLE)0x0,'\0',0,(COORD)0x0,&local_244);
    WriteConsoleOutputCharacterW((HANDLE)0x0,(LPCWSTR)0x0,0,(COORD)0x0,(LPDWORD)0x0);
    GetStdHandle(0);
  }
  iVar1 = iVar1 + -1;
} while (iVar1 != 0);
mw_load_msimg32();
FUN_00403906();
return 0;
```

The first function, relabled by me to `mw_load_msimg32()` loads `msimg32.dll` via **LoadLibraryA**:

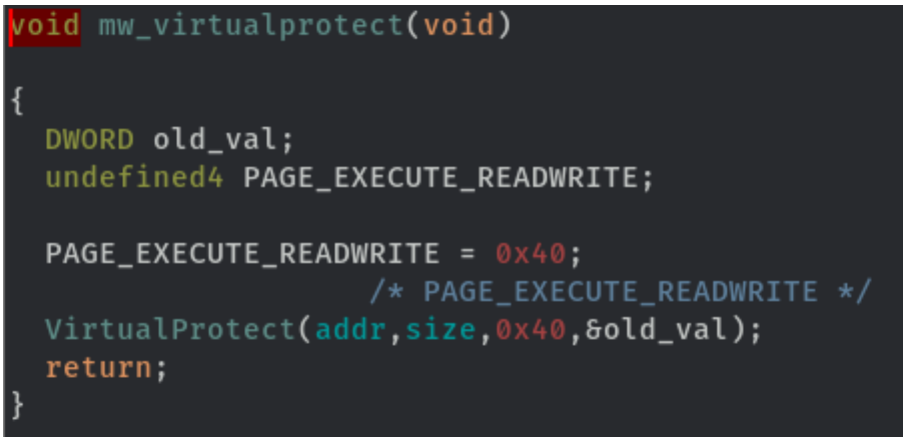
```
msimg32_dll._0_1_ = 'm';
msimg32_dll._1_1_ = 's';
msimg32_dll._2_1_ = 'i';
msimg32_dll._3_1_ = 'm';
DAT_004c0b2c = 'g';
DAT_004c0b2d = '3';
DAT_004c0b2e = '2';
DAT_004c0b2f = '.';
DAT_004c0b30 = 'd';
DAT_004c0b31 = 'l';
DAT_004c0b32 = 'l';
DAT_004c0b33 = 0;
LoadLibraryA((LPCSTR)&msimg32_dll);
return;
```

The second function, labled as `FUN_403906`, is where the unpacking takes place.

After some initialization, **LocalAlloc** is retrieved from `msimg32.dll` via **GetProcAddress**. The resulting function address of LocalAlloc is stored in **_addr_localalloc**. With this, some memory is allocated, pointed to by **addr**. I labled the next function `mw_virtualprotect`. This function changes the permissions of the allocated memory region to execute, read, write:

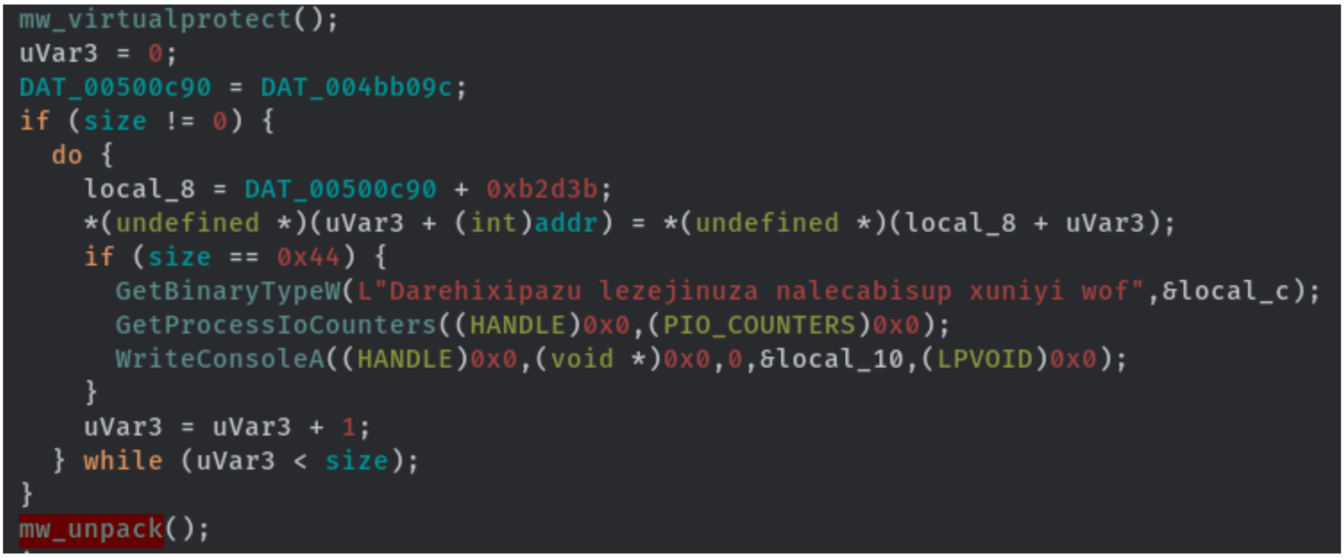


```
_h_msimg32 = GetModuleHandleA((LPCSTR)&msimg32_dll);
_addr_localalloc = GetProcAddress(_h_msimg32,"LocalAlloc");
addr = (LPVOID)(*_addr_localalloc)(0,size);
mw_virtualprotect();
```



```
void mw_virtualprotect(void)

{
  DWORD old_val;
  undefined4 PAGE_EXECUTE_READWRITE;

  PAGE_EXECUTE_READWRITE = 0x40;
                    /* PAGE_EXECUTE_READWRITE */
  VirtualProtect(addr,size,0x40,&old_val);
  return;
}
```

Continuing to analyze `FUN_403906` reveals the function which unpacks the malware, thus, I renamed this to `mw_unpack`:



```
mw_virtualprotect();
uVar3 = 0;
DAT_00500c90 = DAT_004bb09c;
if (size != 0) {
  do {
    local_8 = DAT_00500c90 + 0xb2d3b;
    *(undefined *)(uVar3 + (int)addr) = *(undefined *)(local_8 + uVar3);
    if (size == 0x44) {
      GetBinaryTypeW(L"Darehixipazu lezejinuza nalecabisup xuniyi wof",&local_c);
      GetProcessIoCounters((HANDLE)0x0,(PIO_COUNTERS)0x0);
      WriteConsoleA((HANDLE)0x0,(void *)0x0,0,&local_10,(LPVOID)0x0);
    }
    uVar3 = uVar3 + 1;
  } while (uVar3 < size);
}
mw_unpack();
```

In this function, the unpacking routine is applied. It begins with a large number of hard-coded 32-bit constants and ends with a loop containing the unpacking logic. The unpacked code is stored in the previously allocated memory.

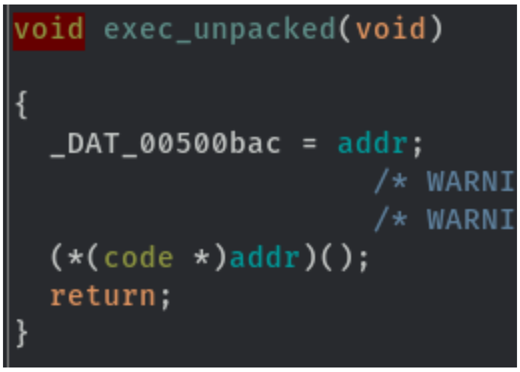
```
  local_1c0 = 0x5e24fb8f;
  local_11c = 0x13cc9676;
  local_128 = 0x145ce368;
  local_c4 = 0x53a4e4aa;
  local_178 = 0x180454cc;
  local_1b4 = 0x5abc23bd;
  local_74 = 0x7a98d7b;
  local_228 = 0x48a0c18f;
  local_f0 = 0x6e4572f2;
  local_170 = 0x924e26a9;
  local_8 = (local_2d4 >> 5) + local_2ec ^ local_2d8 ^ local_2d0 + uVar2;
  if (size == 0x1a3) {
    MoveFileW((LPCWSTR)0x0,(LPCWSTR)0x0);
  }
  uVar4 = uVar4 - local_8;
  local_10 = 4;
  uVar3 = uVar4 * 0x10 + local_2e4;
  local_8 = (uVar4 >> ((byte)local_c & 0x1f)) + local_2e8;
  uVar1 = local_2d0 + uVar4;
  if (size == 0xb3f) {
    GetConsoleAliasesA(local_6ec,0,(LPSTR)0x0);
    InterlockedPushEntrySList((PSLIST_HEADER)0x0,(PSINGLE_LIST_ENTRY)0x0);
  }
  _DAT_004c0b98 = 0;
  uVar2 = uVar2 - (uVar3 ^ uVar1 ^ local_8);
  local_2d4 = uVar2;
  FUN_00402fc3();
  local_2dc = local_2dc + -1;
} while (local_2dc != 0);
param_1[1] = uVar4;
*param_1 = uVar2;
return;
```

I didn't dig through the unpacking algorithm itself very much.

Eventually, before the return of `FUN_403906`, the unpacked code is executed in `exec_unpacked`.

```
mw_unpack();
iVar2 = 0;
do {
  GetLastError();
  if (iVar2 == 0x770e) {
    FUN_004038b5();
  }
  iVar2 = iVar2 + 1;
} while (iVar2 < 0x286b97d);
iVar2 = 0x7b;
do {
  if (size == 0xd) {
    CreateDirectoryA((LPCSTR)0x0,(LPSECURITY_ATTRIBUTES)0x0);
    lstrlenA("Yukanevakuriya duhifufacisubop");
    CloseEventLog((HANDLE)0x0);
  }
  iVar2 = iVar2 + -1;
} while (iVar2 != 0);
exec_unpacked();
return;
```

The `exec_unpacked` was again labled as such by me:

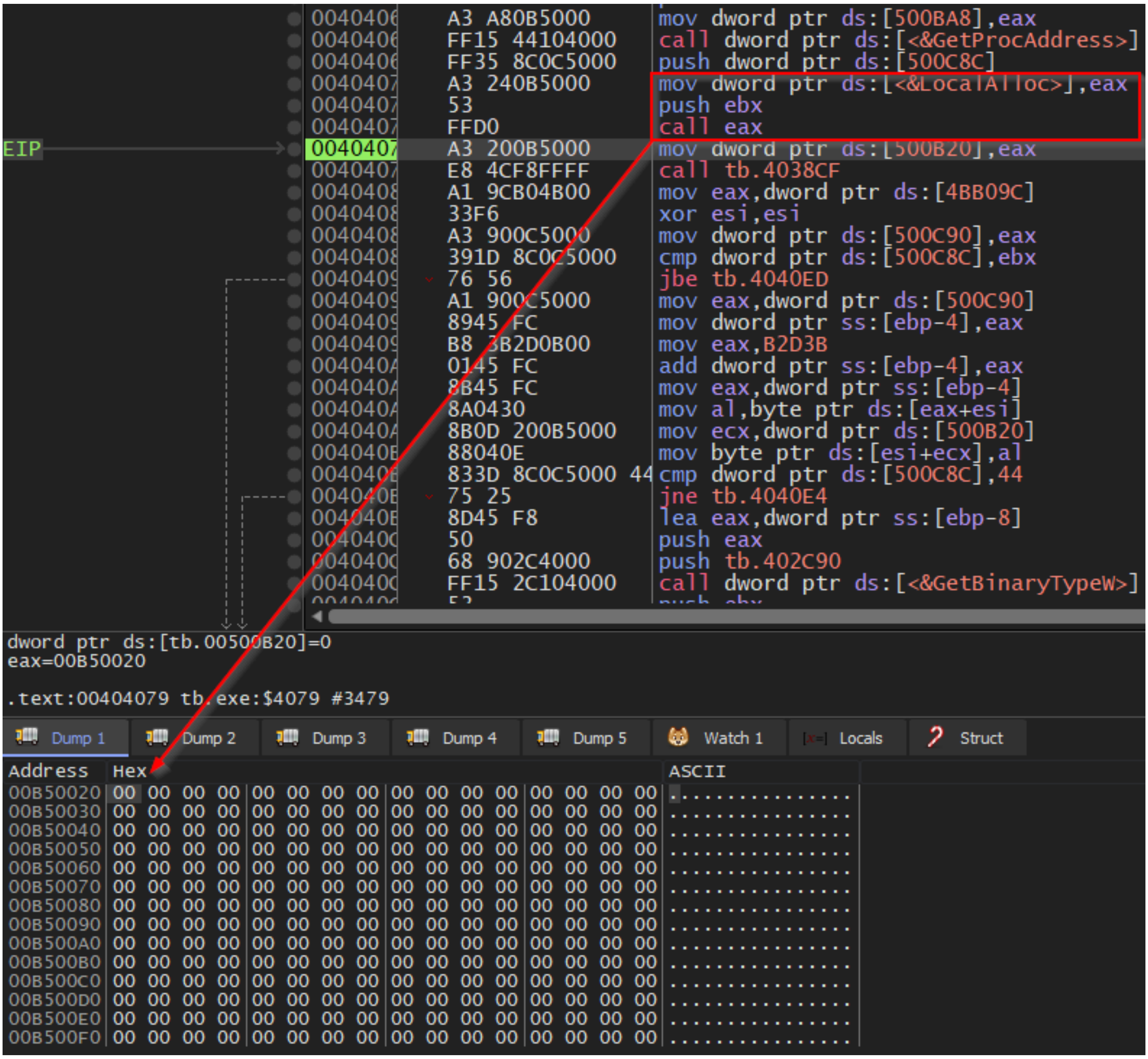


```
void exec_unpacked(void)

{
  _DAT_00500bac = addr;
                    /* WARNI
                    /* WARNI
  (*(code *)addr)();
  return;
}
```

---

What I pieced together above can be confirmed by debugging in x32dbg:

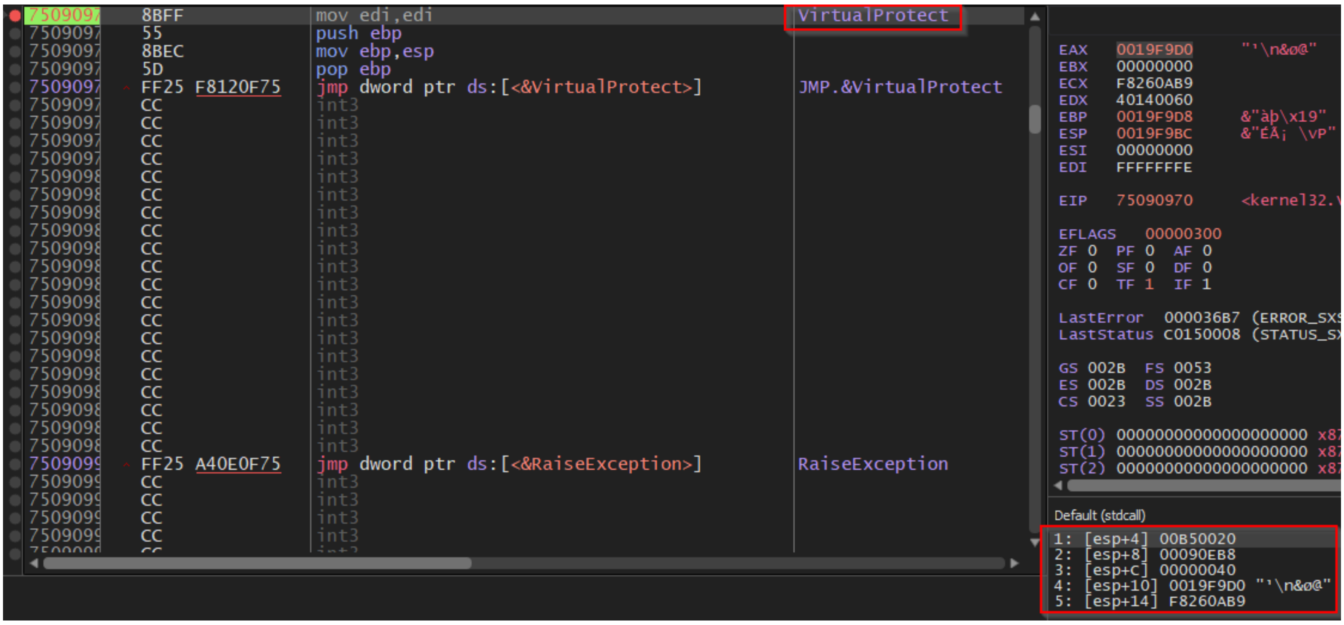
LocalAlloc - allocates memory at address 0xb50020



```
    0040406     A3 A80B5000            mov dword ptr ds:[500BA8],eax
    0040406     FF15 44104000          call dword ptr ds:[<&GetProcAddress>]
    0040406     FF35 8C0C5000          push dword ptr ds:[500C8C]
    0040407     A3 240B5000            mov dword ptr ds:[<&LocalAlloc>],eax
    0040407     53                     push ebx
    0040407     FFD0                   call eax
EIP 0040407     A3 200B5000            mov dword ptr ds:[500B20],eax
    0040407     E8 4CF8FFFF            call tb.4038CF
    0040408     A1 9CB04B00            mov eax,dword ptr ds:[4BB09C]
    0040408     33F6                   xor esi,esi
    0040408     A3 900C5000            mov dword ptr ds:[500C90],eax
    0040408     391D 8C0C5000          cmp dword ptr ds:[500C8C],ebx
    0040409     76 56                  jbe tb.4040ED
    0040409     A1 900C5000            mov eax,dword ptr ds:[500C90]
    0040409     8945 FC                mov dword ptr ss:[ebp-4],eax
    0040409     B8 3B2D0B00            mov eax,B2D3B
    004040A     0145 FC                add dword ptr ss:[ebp-4],eax
    004040A     8B45 FC                mov eax,dword ptr ss:[ebp-4]
    004040A     8A0430                 mov al,byte ptr ds:[eax+esi]
    004040A     8B0D 200B5000          mov ecx,dword ptr ds:[500B20]
    004040B     88040E                 mov byte ptr ds:[esi+ecx],al
    004040B     833D 8C0C5000 44       cmp dword ptr ds:[500C8C],44
    004040B     75 25                  jne tb.4040E4
    004040B     8D45 F8                lea eax,dword ptr ss:[ebp-8]
    004040C     50                     push eax
    004040C     68 902C4000            push tb.402C90
    004040C     FF15 2C104000          call dword ptr ds:[<&GetBinaryTypeW>]

dword ptr ds:[tb.00500B20]=0
eax=00B50020

.text:00404079 tb.exe:$4079 #3479
```

```
Address  Hex                                              ASCII
00B50020 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00B50030 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00B50040 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00B50050 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00B50060 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00B50070 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00B50080 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00B50090 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00B500A0 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00B500B0 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00B500C0 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00B500D0 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00B500E0 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
00B500F0 00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00 ................
```

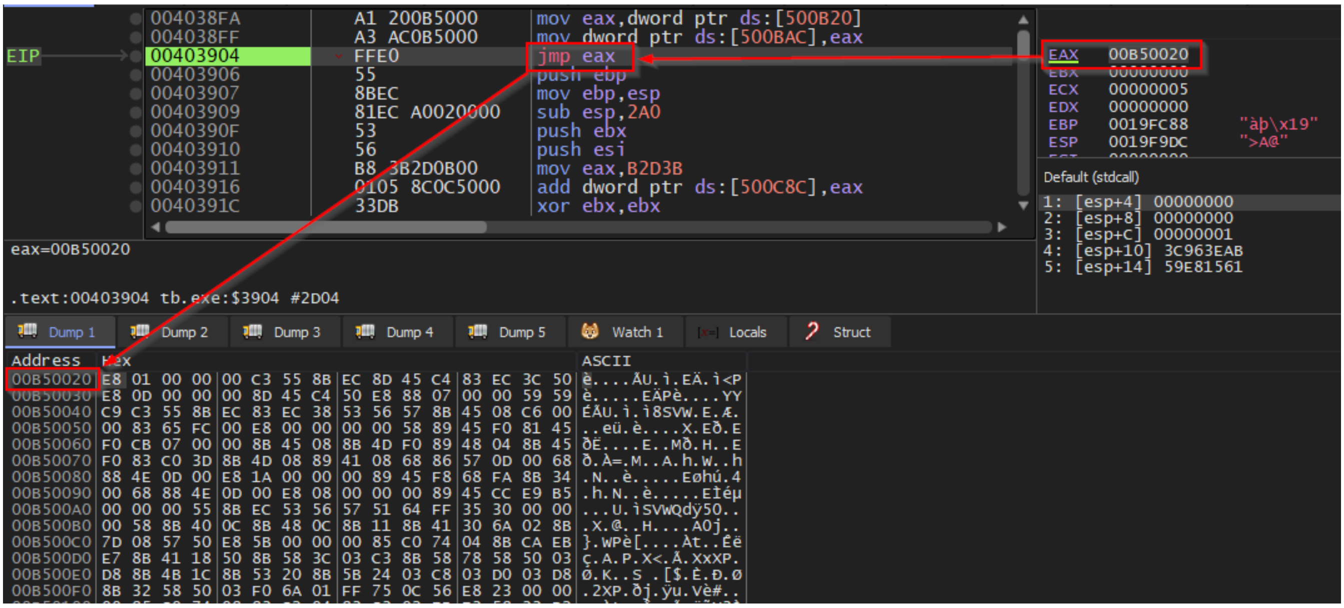
Virtual Protect - applied to the memory area 0xb50020, set permissions to ERW



```
7509097     8BFF            mov edi,edi                                VirtualProtect
7509097     55              push ebp
7509097     8BEC            mov ebp,esp
7509097     5D              pop ebp
7509097     FF25 F8120F75   jmp dword ptr ds:[<&VirtualProtect>]       JMP.&VirtualProtect
7509097     CC              int3
7509097     CC              int3
7509097     CC              int3
7509097     CC              int3
7509098     CC              int3
7509098     CC              int3
7509098     CC              int3
7509098     CC              int3
7509098     CC              int3
7509098     CC              int3
7509098     CC              int3
7509098     CC              int3
7509098     CC              int3
7509098     CC              int3
7509098     CC              int3
7509098     CC              int3
7509098     CC              int3
7509098     CC              int3
7509098     CC              int3
7509098     CC              int3
7509099     FF25 A40E0F75   jmp dword ptr ds:[<&RaiseException>]       RaiseException
7509099     CC              int3
7509099     CC              int3
7509099     CC              int3
7509099     CC              int3
```

```
EAX   0019F9D0     "'\n&ø@"
EBX   00000000
ECX   F8260AB9
EDX   40140060
EBP   0019F9D8     &"àþ\x19"
ESP   0019F9BC     &"ÉÄ¡ \vP"
ESI   00000000
EDI   FFFFFFFE

EIP   75090970     <kernel32.V

EFLAGS   00000300
ZF 0  PF 0  AF 0
OF 0  SF 0  DF 0
CF 0  TF 1  IF 1

LastError  000036B7 (ERROR_SXS
LastStatus C0150008 (STATUS_SX

GS 002B  FS 0053
ES 002B  DS 002B
CS 0023  SS 002B

ST(0) 00000000000000000000 x87
ST(1) 00000000000000000000 x87
ST(2) 00000000000000000000 x87

Default (stdcall)
1: [esp+4] 00B50020
2: [esp+8] 00090EB8
3: [esp+C] 00000040
4: [esp+10] 0019F9D0 "'\n&ø@"
5: [esp+14] F8260AB9
```

After unpacking, a jump to the unpacked payload occurs.



```
EIP
004038FA    A1 200B5000      mov eax,dword ptr ds:[500B20]
004038FF    A3 AC0B5000      mov dword ptr ds:[500BAC],eax
00403904    FFE0             jmp eax
00403906    55               push ebp
00403907    8BEC             mov ebp,esp
00403909    81EC A0020000    sub esp,2A0
0040390F    53               push ebx
00403910    56               push esi
00403911    B8 3B2D0B00      mov eax,B2D3B
00403916    0105 8C0C5000    add dword ptr ds:[500C8C],eax
0040391C    33DB             xor ebx,ebx

eax=00B50020

.text:00403904 tb.exe:$3904 #2D04
```

```
EAX   00B50020
EBX   00000000
ECX   00000005
EDX   00000000
EBP   0019FC88        "àþ\x19"
ESP   0019F9DC        ">A@"

Default (stdcall)
1: [esp+4] 00000000
2: [esp+8] 00000000
3: [esp+C] 00000001
4: [esp+10] 3C963EAB
5: [esp+14] 59E81561
```

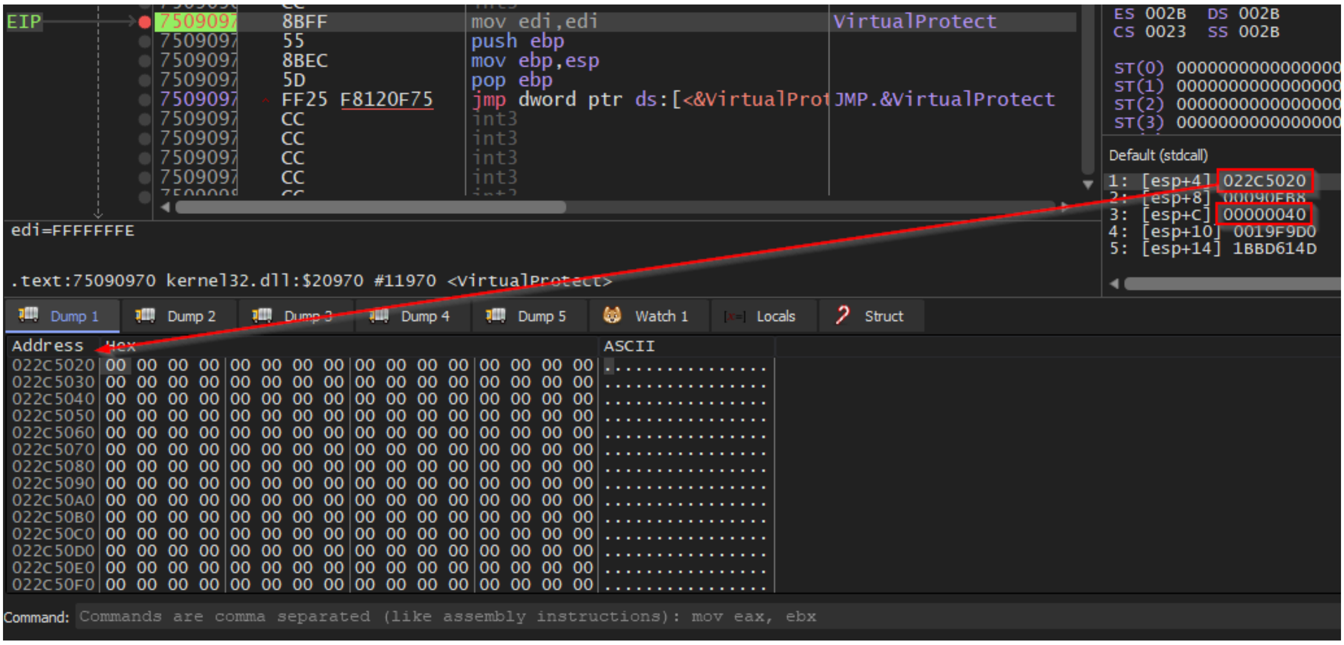
```
Address   Hex                                              ASCII
00B50020 E8 01 00 00 00 C3 55 8B EC 8D 45 C4 83 EC 3C 50 è....ÃU.ì.EÄ.ì<P
00B50030 E8 0D 00 00 00 8D 45 C4 50 E8 88 07 00 00 59 59 è.....EÄPè....YY
00B50040 C9 C3 55 8B EC 83 EC 38 53 56 57 8B 45 08 C6 00 ÉÃU.ì.ì8SVW.E.Æ.
00B50050 00 83 65 FC 00 E8 00 00 00 00 58 89 45 F0 81 45 ..eü.è....X.Eð.E
00B50060 F0 CB 07 00 00 8B 45 08 8B 4D F0 89 48 04 8B 45 ðË....E..Mð.H..E
00B50070 F0 83 C0 3D 8B 4D 08 89 41 08 68 86 57 0D 00 68 ð.À=.M..A.h.W..h
00B50080 88 4E 0D 00 E8 1A 00 00 00 89 45 F8 68 FA 8B 34 .N..è.....Eøhú.4
00B50090 00 68 88 4E 0D 00 E8 08 00 00 00 89 45 CC E9 B5 .h.N..è.....EÌéµ
00B500A0 00 00 00 55 8B EC 53 56 57 51 64 FF 35 30 00 00 ...U.ìSVWQdÿ50..
00B500B0 00 58 8B 40 0C 8B 48 0C 8B 11 8B 41 30 6A 02 8B .X.@..H....A0j..
00B500C0 7D 08 57 50 E8 5B 00 00 00 85 C0 74 04 8B CA EB }.WPè[....Àt..Êë
00B500D0 E7 8B 41 18 50 8B 58 3C 03 C3 8B 58 78 58 50 03 ç.A.P.X<.Ã.XxXP.
00B500E0 D8 8B 4B 1C 8B 53 20 8B 5B 24 03 C8 03 D0 03 D8 Ø.K..S .[$.È.Ð.Ø
00B500F0 8B 32 58 50 03 F0 6A 01 FF 75 0C 56 E8 23 00 00 .2XP.ðj.ÿu.Vè#..
```

That's as far as I went with the static analysis, next, let's fire up x32dbg and get that payload out.

## Dynamic Analysis

I will restart the debugging process here, the debugging in the "Static Analysis" section was only to link the static analysis findings with the actual execution flow of the program.

After I restarted the debugging process, I set a breakpoint at `VirtualProtect`. That way, I can get the address of the allocated memory and break before the unpacked code is executed.



```
EIP
7509097    8BFF            mov edi,edi                        VirtualProtect
7509097    55              push ebp
7509097    8BEC            mov ebp,esp
7509097    5D              pop ebp
7509097    FF25 F8120F75   jmp dword ptr ds:[<&VirtualProt   JMP.&VirtualProtect
7509097    CC              int3
7509097    CC              int3
7509097    CC              int3
7509097    CC              int3

edi=FFFFFFFE

.text:75090970 kernel32.dll:$20970 #11970 <VirtualProtect>
```

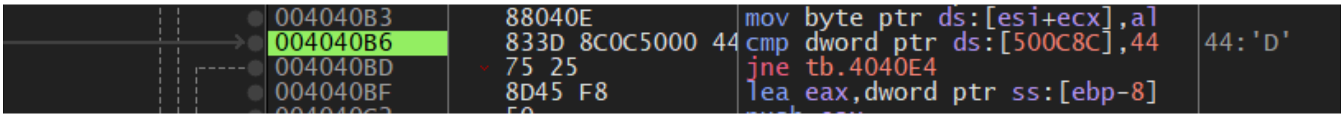
```
ES 002B   DS 002B
CS 0023   SS 002B

ST(0) 0000000000000000
ST(1) 0000000000000000
ST(2) 0000000000000000
ST(3) 0000000000000000

Default (stdcall)
1: [esp+4] 022C5020
2: [esp+8] 00090FB8
3: [esp+C] 00000040
4: [esp+10] 0019F9D0
5: [esp+14] 1BBD614D
```

```
Address   Hex                                              ASCII
022C5020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
022C5030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
022C5040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
022C5050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
022C5060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
022C5070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
022C5080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
022C5090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
022C50A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
022C50B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
022C50C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
022C50D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
022C50E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
022C50F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
Command: Commands are comma separated (like assembly instructions): mov eax, ebx
```

As you can see from the stack, the permissions are set to execute, read, write (0x40) and the allocated memory's address is the topmost argument.

I set a hardware breakpoint at the memory address in the dump.

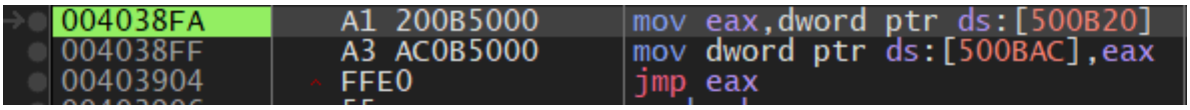
If you hit continue now, you will land in the routine before the part that executes the unpacked code:



```
004040B3   88040E             mov byte ptr ds:[esi+ecx],al
004040B6   833D 8C0C5000 44   cmp dword ptr ds:[500C8C],44    44:'D'
004040BD   75 25              jne tb.4040E4
004040BF   8D45 F8            lea eax,dword ptr ss:[ebp-8]
```

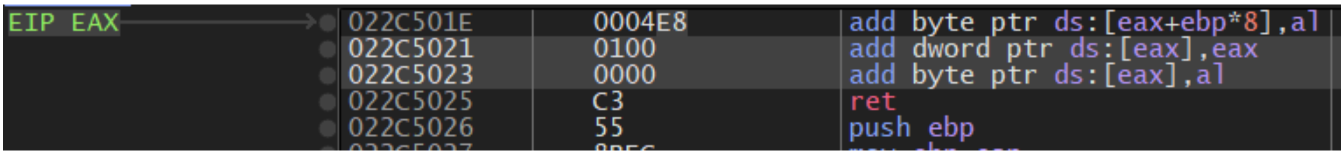
We're looking for the call at `0x404139`, this calls the unpacked code, remember `FUN_403906` from the static analysis section, this is the routine we are in, so we can find 0x404139 by scrolling down. I set a breakpoint here and removed the hardware breakpoint. After that, I hit continue and ended up at the call at 0x404139.

After stepping into this call, we want to follow the execution from `jmp eax`.



```
004038FA   A1 200B5000   mov eax,dword ptr ds:[500B20]
004038FF   A3 AC0B5000   mov dword ptr ds:[500BAC],eax
00403904   FFE0          jmp eax
```

After following this, I landed in the allocated memory, where the unpacked code resides.



```
EIP EAX   022C501E   0004E8   add byte ptr ds:[eax+ebp*8],al
          022C5021   0100     add dword ptr ds:[eax],eax
          022C5023   0000     add byte ptr ds:[eax],al
          022C5025   C3       ret
          022C5026   55       push ebp
```

From here, there are two possible ways to continue.

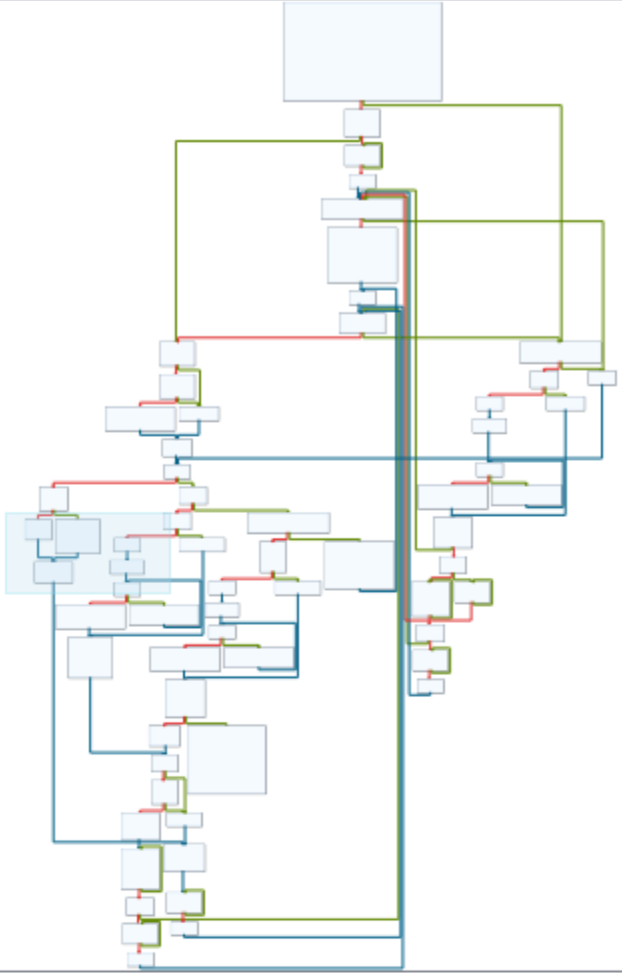
1. Step through the code until we find something interesting
2. dump the memory of the shellcode and check it out in a disassembler.

I prefer Option 1.

---

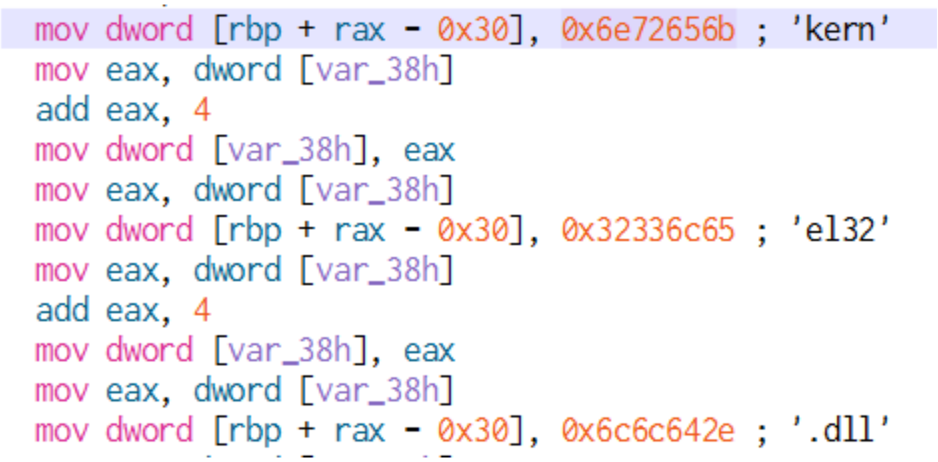
## Examine the Shellcode - 1

Dumping the memory region where the shellcode is stored and loading the dump into Cutter reveals several interesting routines. First this routine, which looks complicated judging by it's graph overview:

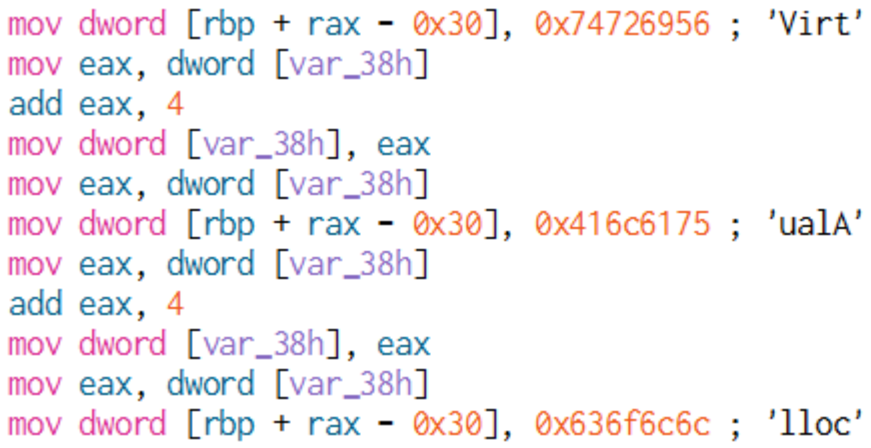


From my examination, it seems this performs further unpacking. This is confirmed later.

Also, there is a function containing stack strings:



```
mov dword [rbp + rax - 0x30], 0x6e72656b ; 'kern'
mov eax, dword [var_38h]
add eax, 4
mov dword [var_38h], eax
mov eax, dword [var_38h]
mov dword [rbp + rax - 0x30], 0x32336c65 ; 'el32'
mov eax, dword [var_38h]
add eax, 4
mov dword [var_38h], eax
mov eax, dword [var_38h]
mov dword [rbp + rax - 0x30], 0x6c6c642e ; '.dll'
```



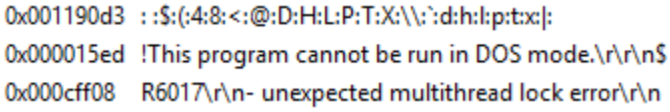
```
mov dword [rbp + rax - 0x30], 0x74726956 ; 'Virt'
mov eax, dword [var_38h]
add eax, 4
mov dword [var_38h], eax
mov eax, dword [var_38h]
mov dword [rbp + rax - 0x30], 0x416c6175 ; 'ualA'
mov eax, dword [var_38h]
add eax, 4
mov dword [var_38h], eax
mov eax, dword [var_38h]
mov dword [rbp + rax - 0x30], 0x636f6c6c ; 'lloc'
```

Continuing in the debugger, I put a breakpoint at VirtualAlloc and hit continue. Indeed, VirtualAlloc is called and some memory is allocated. I put a hardware breakpoint there and kept on stepping.

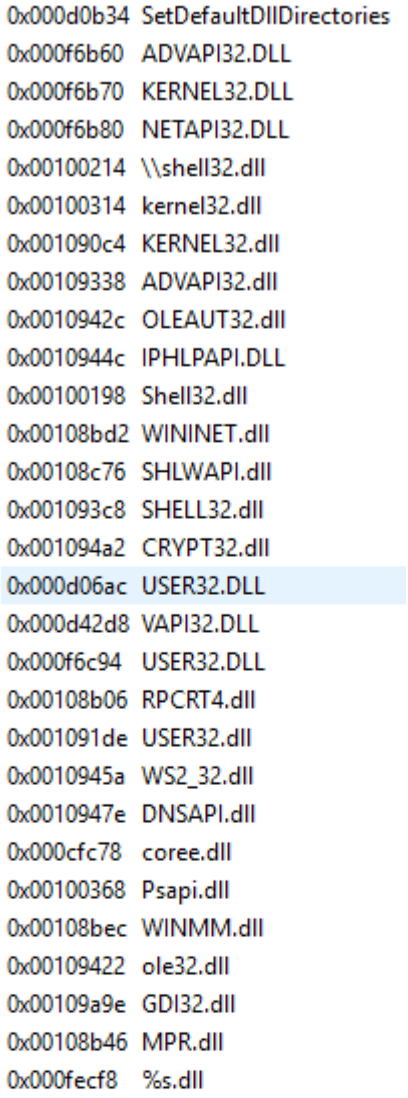
Once the hardware breakpoint was hit, I found myself in the unpacking routine shown in the graph overview above. I set a breakpoint at the return of this routine and watched the allocated memory being populated. Once the memory area was filled, I dumped that memory as well. As with the dump before, I loaded it into Cutter to examine it the code in detail.

## Examine the Shellcode - 2

Looking at the strings of this dump shows that it contains the DOS header and DLL names among other interesting things.



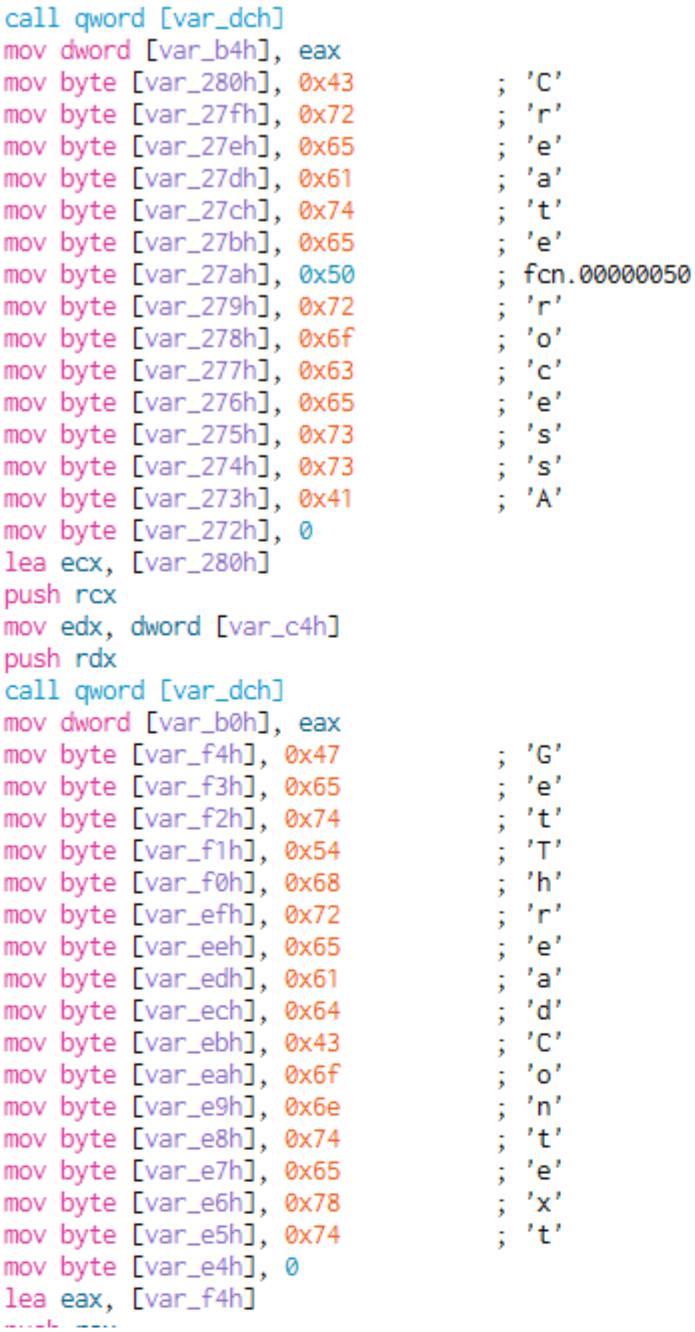
```
0x001190d3  : :$:(:4:8:<:@:D:H:L:P:T:X:\\:`:d:h:l:p:t:x:|:
0x000015ed  !This program cannot be run in DOS mode.\r\r\n$
0x000cff08  R6017\r\n- unexpected multithread lock error\r\n
```



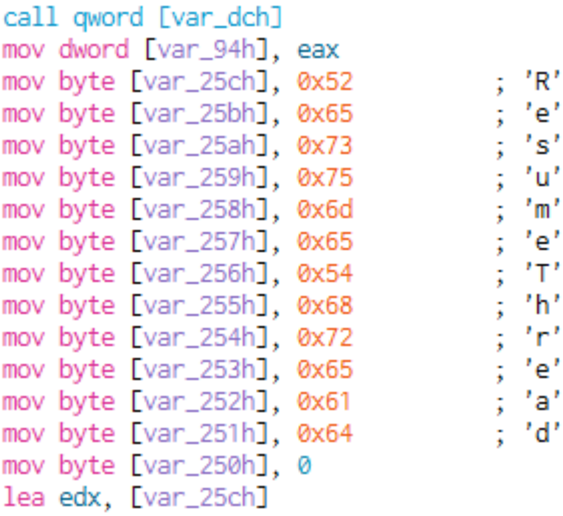
```
0x000d0b34  SetDefaultDllDirectories
0x000f6b60  ADVAPI32.DLL
0x000f6b70  KERNEL32.DLL
0x000f6b80  NETAPI32.DLL
0x00100214  \\shell32.dll
0x00100314  kernel32.dll
0x001090c4  KERNEL32.dll
0x00109338  ADVAPI32.dll
0x0010942c  OLEAUT32.dll
0x0010944c  IPHLPAPI.DLL
0x00100198  Shell32.dll
0x00108bd2  WININET.dll
0x00108c76  SHLWAPI.dll
0x001093c8  SHELL32.dll
0x001094a2  CRYPT32.dll
0x000d06ac  USER32.DLL
0x000d42d8  VAPI32.DLL
0x000f6c94  USER32.DLL
0x00108b06  RPCRT4.dll
0x001091de  USER32.dll
0x0010945a  WS2_32.dll
0x0010947e  DNSAPI.dll
0x000cfc78  coree.dll
0x00100368  Psapi.dll
0x00108bec  WINMM.dll
0x00109422  ole32.dll
0x00109a9e  GDI32.dll
0x00108b46  MPR.dll
0x000fecf8  %s.dll
```

Looking through the functions in Cutter, one of them stands out. It contains stack strings of various APIs, among them, in this order:

- CreateProcessA
- GetThreadContext
- VirtualAlloc
- VirtualAllocEx
- VirtualFree
- ReadProcessMemory
- WriteProcessMemory
- SetThreadContext
- ResumeThread



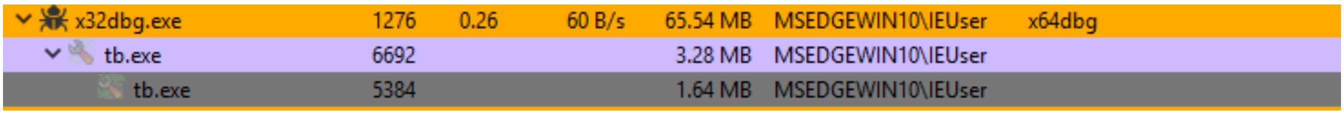
```
call qword [var_dch]
mov dword [var_b4h], eax
mov byte [var_280h], 0x43          ; 'C'
mov byte [var_27fh], 0x72          ; 'r'
mov byte [var_27eh], 0x65          ; 'e'
mov byte [var_27dh], 0x61          ; 'a'
mov byte [var_27ch], 0x74          ; 't'
mov byte [var_27bh], 0x65          ; 'e'
mov byte [var_27ah], 0x50          ; fcn.00000050
mov byte [var_279h], 0x72          ; 'r'
mov byte [var_278h], 0x6f          ; 'o'
mov byte [var_277h], 0x63          ; 'c'
mov byte [var_276h], 0x65          ; 'e'
mov byte [var_275h], 0x73          ; 's'
mov byte [var_274h], 0x73          ; 's'
mov byte [var_273h], 0x41          ; 'A'
mov byte [var_272h], 0
lea ecx, [var_280h]
push rcx
mov edx, dword [var_c4h]
push rdx
call qword [var_dch]
mov dword [var_b0h], eax
mov byte [var_f4h], 0x47           ; 'G'
mov byte [var_f3h], 0x65           ; 'e'
mov byte [var_f2h], 0x74           ; 't'
mov byte [var_f1h], 0x54           ; 'T'
mov byte [var_f0h], 0x68           ; 'h'
mov byte [var_efh], 0x72           ; 'r'
mov byte [var_eeh], 0x65           ; 'e'
mov byte [var_edh], 0x61           ; 'a'
mov byte [var_ech], 0x64           ; 'd'
mov byte [var_ebh], 0x43           ; 'C'
mov byte [var_eah], 0x6f           ; 'o'
mov byte [var_e9h], 0x6e           ; 'n'
mov byte [var_e8h], 0x74           ; 't'
mov byte [var_e7h], 0x65           ; 'e'
mov byte [var_e6h], 0x78           ; 'x'
mov byte [var_e5h], 0x74           ; 't'
mov byte [var_e4h], 0
lea eax, [var_f4h]
```



```
call qword [var_dch]
mov dword [var_94h], eax
mov byte [var_25ch], 0x52          ; 'R'
mov byte [var_25bh], 0x65          ; 'e'
mov byte [var_25ah], 0x73          ; 's'
mov byte [var_259h], 0x75          ; 'u'
mov byte [var_258h], 0x6d          ; 'm'
mov byte [var_257h], 0x65          ; 'e'
mov byte [var_256h], 0x54          ; 'T'
mov byte [var_255h], 0x68          ; 'h'
mov byte [var_254h], 0x72          ; 'r'
mov byte [var_253h], 0x65          ; 'e'
mov byte [var_252h], 0x61          ; 'a'
mov byte [var_251h], 0x64          ; 'd'
mov byte [var_250h], 0
lea edx, [var_25ch]
```

This looks very much like Process Hollowing, which prompted me to set a breakpoint at ResumeThread.

I continued execution, hitting the breakpoint at `ResumeThread`, after which I opened Process Hacker:



| | | | | | | |
|---|---|---|---|---|---|---|
| x32dbg.exe | 1276 | 0.26 | 60 B/s | 65.54 MB | MSEDGEWIN10\IEUser | x64dbg |
| tb.exe | 6692 | | | 3.28 MB | MSEDGEWIN10\IEUser | |
| tb.exe | 5384 | | | 1.64 MB | MSEDGEWIN10\IEUser | |

The PID 5384, child of the original executable, is what was created by CreateProcessA. After the processes memory was written it should contain the stop/djvu payload. As ResumeThread is the last call for the Process Hollowing dumping PID 5384 with pe-sieve yields the unpacked stop/djvu malware.

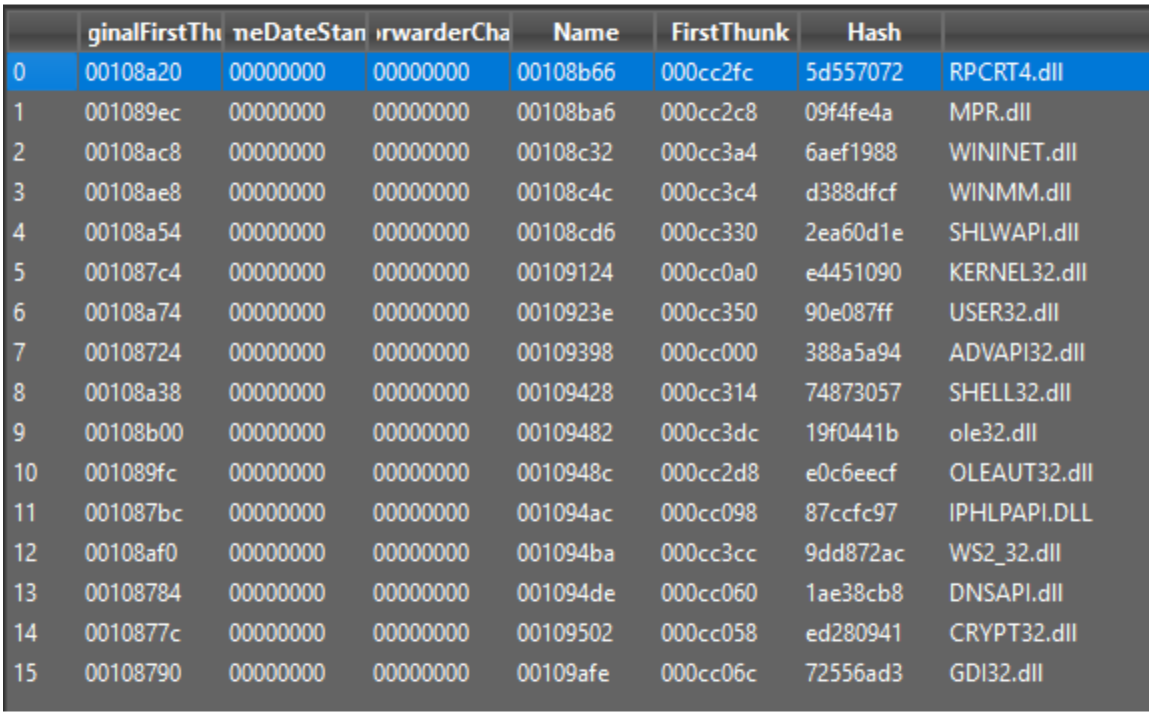
I loaded the dumped executable into DetectItEasy, the entropy looks better:

Type: PE32
Total: 6.65698
83%
Status: packed
Offset: 00000000
Size: 00119000

Regions

| Name | Offset | Size | Entropy | Status |
|---|---|---|---|---|
| PE Header | 00000000 | 00000400 | 3.00222 | not packed |
| Section(0)['.text'] | 00000400 | 000ca600 | 6.57013 | packed |
| Section(1)['.rdata'] | 000caa00 | 0003dc00 | 5.66760 | not packed |
| Section(2)['.data'] | 00108600 | 00006400 | 4.93968 | not packed |
| Section(3)['.rsrc'] | 0010ea00 | 00000200 | 4.72050 | not packed |

... and there are several interesting imports:



| | ginalFirstThı | neDateStan | rwarderCha | Name | FirstThunk | Hash | |
|---|---|---|---|---|---|---|---|
| 0 | 00108a20 | 00000000 | 00000000 | 00108b66 | 000cc2fc | 5d557072 | RPCRT4.dll |
| 1 | 001089ec | 00000000 | 00000000 | 00108ba6 | 000cc2c8 | 09f4fe4a | MPR.dll |
| 2 | 00108ac8 | 00000000 | 00000000 | 00108c32 | 000cc3a4 | 6aef1988 | WININET.dll |
| 3 | 00108ae8 | 00000000 | 00000000 | 00108c4c | 000cc3c4 | d388dfcf | WINMM.dll |
| 4 | 00108a54 | 00000000 | 00000000 | 00108cd6 | 000cc330 | 2ea60d1e | SHLWAPI.dll |
| 5 | 001087c4 | 00000000 | 00000000 | 00109124 | 000cc0a0 | e4451090 | KERNEL32.dll |
| 6 | 00108a74 | 00000000 | 00000000 | 0010923e | 000cc350 | 90e087ff | USER32.dll |
| 7 | 00108724 | 00000000 | 00000000 | 00109398 | 000cc000 | 388a5a94 | ADVAPI32.dll |
| 8 | 00108a38 | 00000000 | 00000000 | 00109428 | 000cc314 | 74873057 | SHELL32.dll |
| 9 | 00108b00 | 00000000 | 00000000 | 00109482 | 000cc3dc | 19f0441b | ole32.dll |
| 10 | 001089fc | 00000000 | 00000000 | 0010948c | 000cc2d8 | e0c6eecf | OLEAUT32.dll |
| 11 | 001087bc | 00000000 | 00000000 | 001094ac | 000cc098 | 87ccfc97 | IPHLPAPI.DLL |
| 12 | 00108af0 | 00000000 | 00000000 | 001094ba | 000cc3cc | 9dd872ac | WS2_32.dll |
| 13 | 00108784 | 00000000 | 00000000 | 001094de | 000cc060 | 1ae38cb8 | DNSAPI.dll |
| 14 | 0010877c | 00000000 | 00000000 | 00109502 | 000cc058 | ed280941 | CRYPT32.dll |
| 15 | 00108790 | 00000000 | 00000000 | 00109afe | 000cc06c | 72556ad3 | GDI32.dll |

Now on to analyzing the unpacked sample ^o^ - updates to follow ;-)

---

**0xca7**