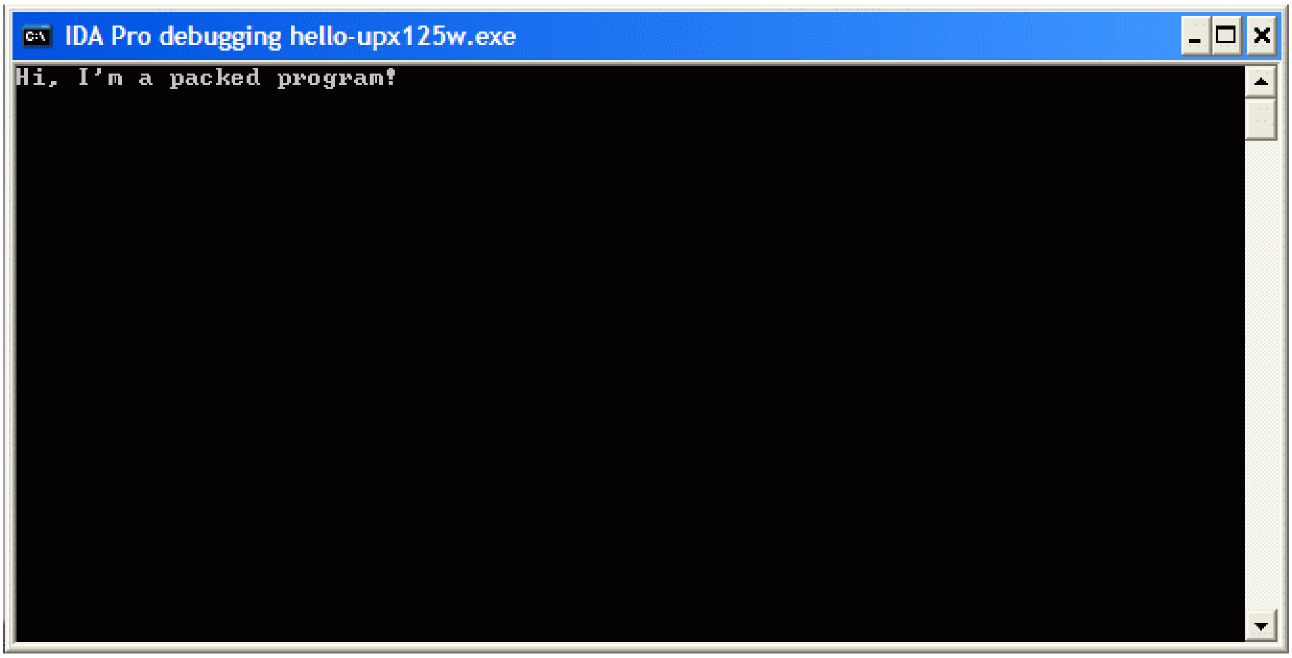
Using the Universal PE Unpacker Plug-in included in IDA Pro 4.9 to unpack compressed executables.

Since version 4.9, IDA Pro comes with an Universal PE Unpacker plug-in, whose source code is available in the IDA Pro SDK. This tutorial will show how to use this plug-in in practice and will briefly describe how it works internally.

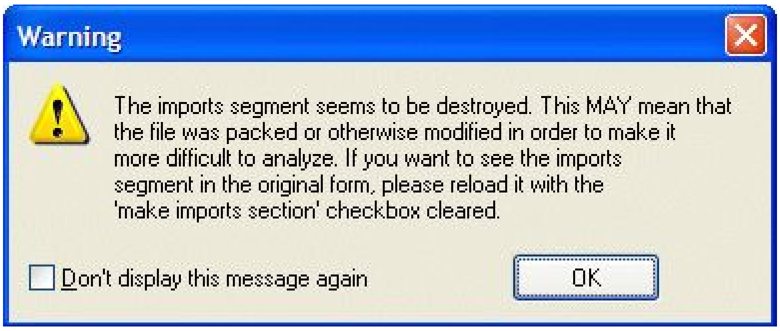
The compressed application.

Here is what appear on our screen if we execute the sample program:



```
Hi, I'm a packed program!
```

Quite innocent! However, if we open this executable in IDA Pro, the following warning appears:

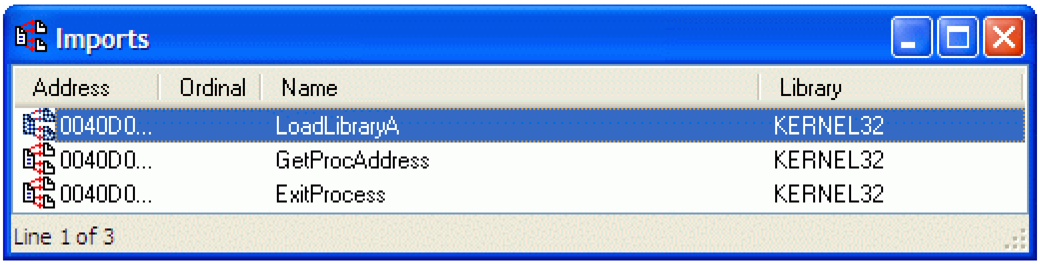


The imports segment seems to be destroyed. This MAY mean that the file was packed or otherwise modified in order to make it more difficult to analyze. If you want to see the imports segment in the original form, please reload it with the 'make imports section' checkbox cleared.

☐ Don't display this message again

IDA detects an unusual imports segment, and tells us the file might be packed...

Here is what we observe if we have a look at the Imports window:

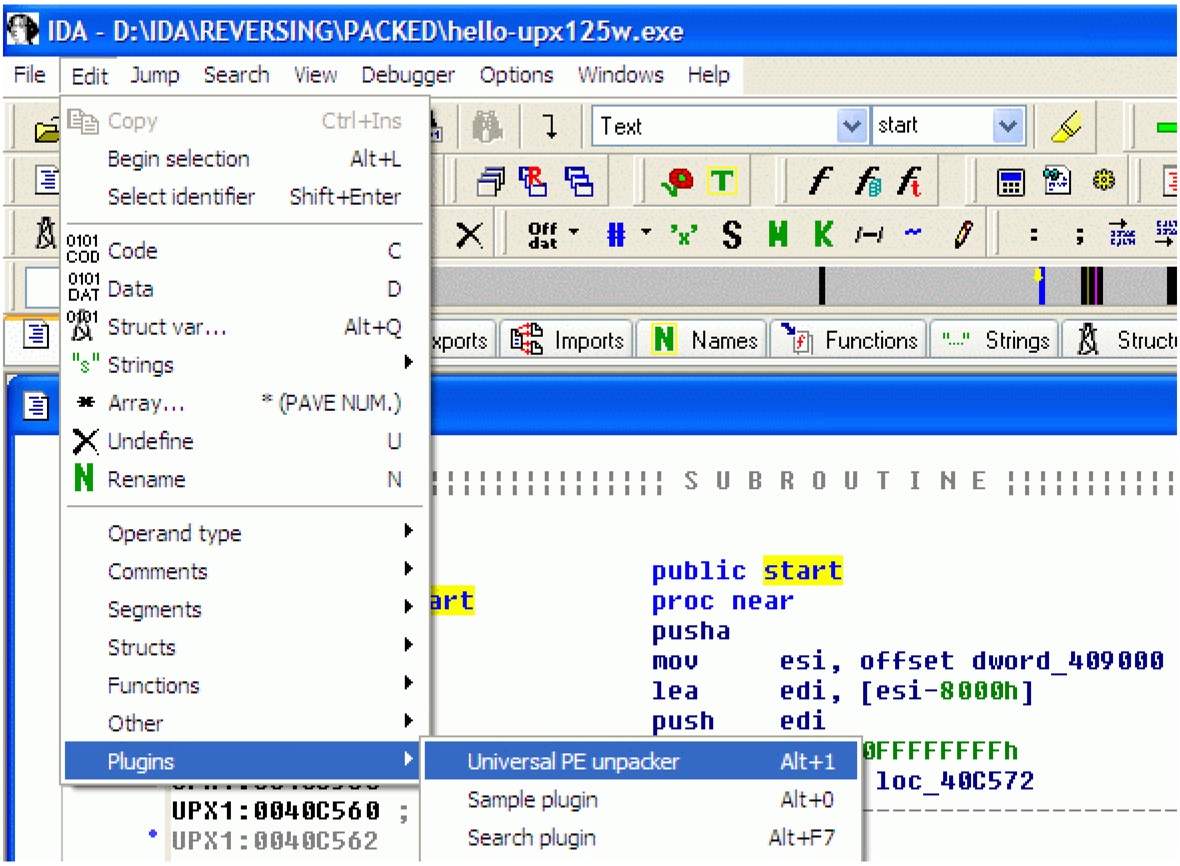


Address	Ordinal	Name	Library
0040D0...		LoadLibraryA	KERNEL32
0040D0...		GetProcAddress	KERNEL32
0040D0...		ExitProcess	KERNEL32

Our program only imports three functions from KERNEL32.DLL. We can recognize the usual *LoadLibrary()* and *GetProcAddress()* dynamic-link library functions, which will be more than probably used by the unpacker engine to restore the original executable's imports.

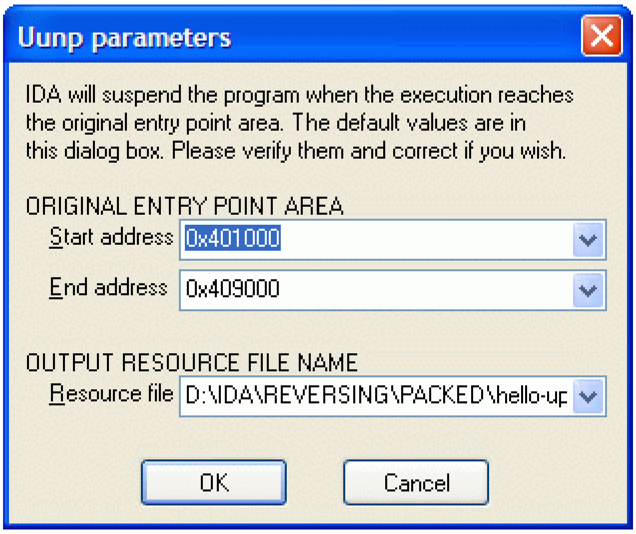
Using the Universal PE Unpacker plugin.

Let's now start the unpacker through the Plug-ins sub-menu:



```
                 public start
art              proc near
                 pusha
                 mov     esi, offset dword_409000
                 lea     edi, [esi-8000h]
                 push    edi
                 0FFFFFFFFh
                  loc_40C572
UPX1:0040C560 ;
UPX1:0040C562
```

The plug-in options dialog appears:



IDA will suspend the program when the execution reaches the original entry point area. The default values are in this dialog box. Please verify them and correct if you wish.

ORIGINAL ENTRY POINT AREA

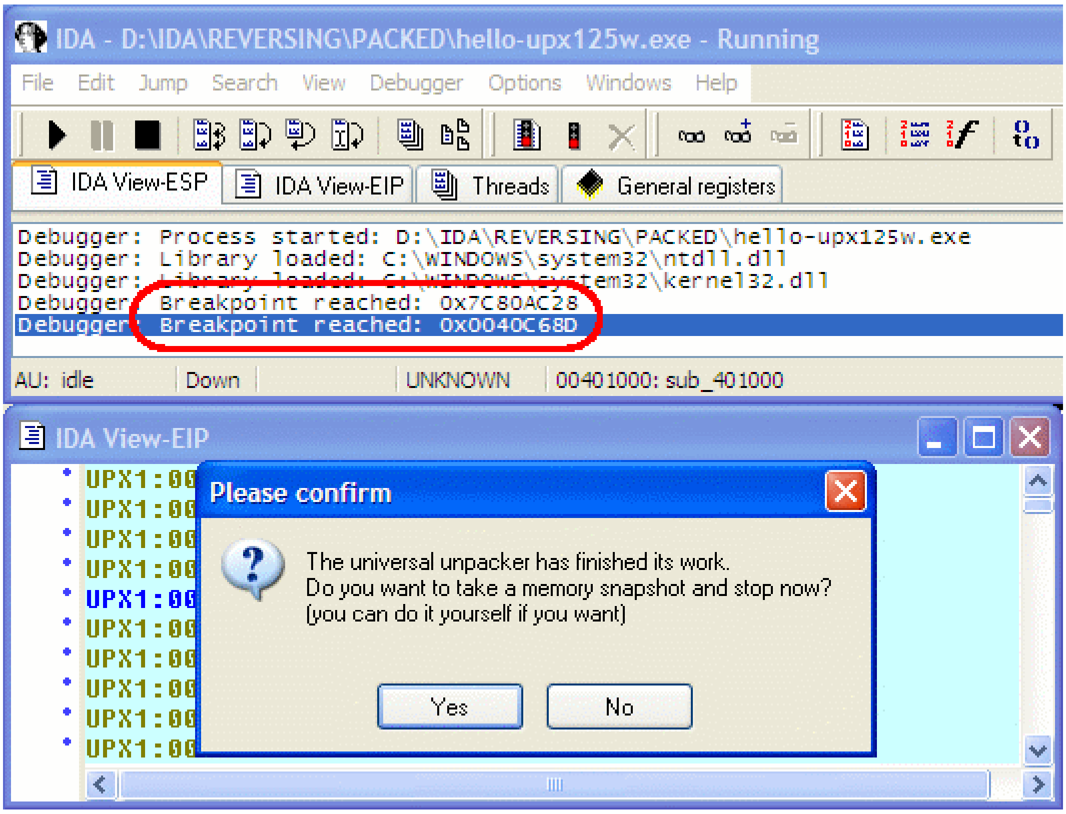
Start address: 0x401000

End address: 0x409000

OUTPUT RESOURCE FILE NAME

Resource file: D:\IDA\REVERSING\PACKED\hello-u

In this dialog, we can adjust the address range which, once reached, will cause the debugger to suspend the program's execution. It is also possible to specify a file where unpacked resources will be saved. After pressing OK, the plug-in starts our program, which will unpack itself until an address inside the previously defined range is reached. This indicates the unpacking is terminated, and the following dialog box appears, offering to take a memory snapshot of the result:

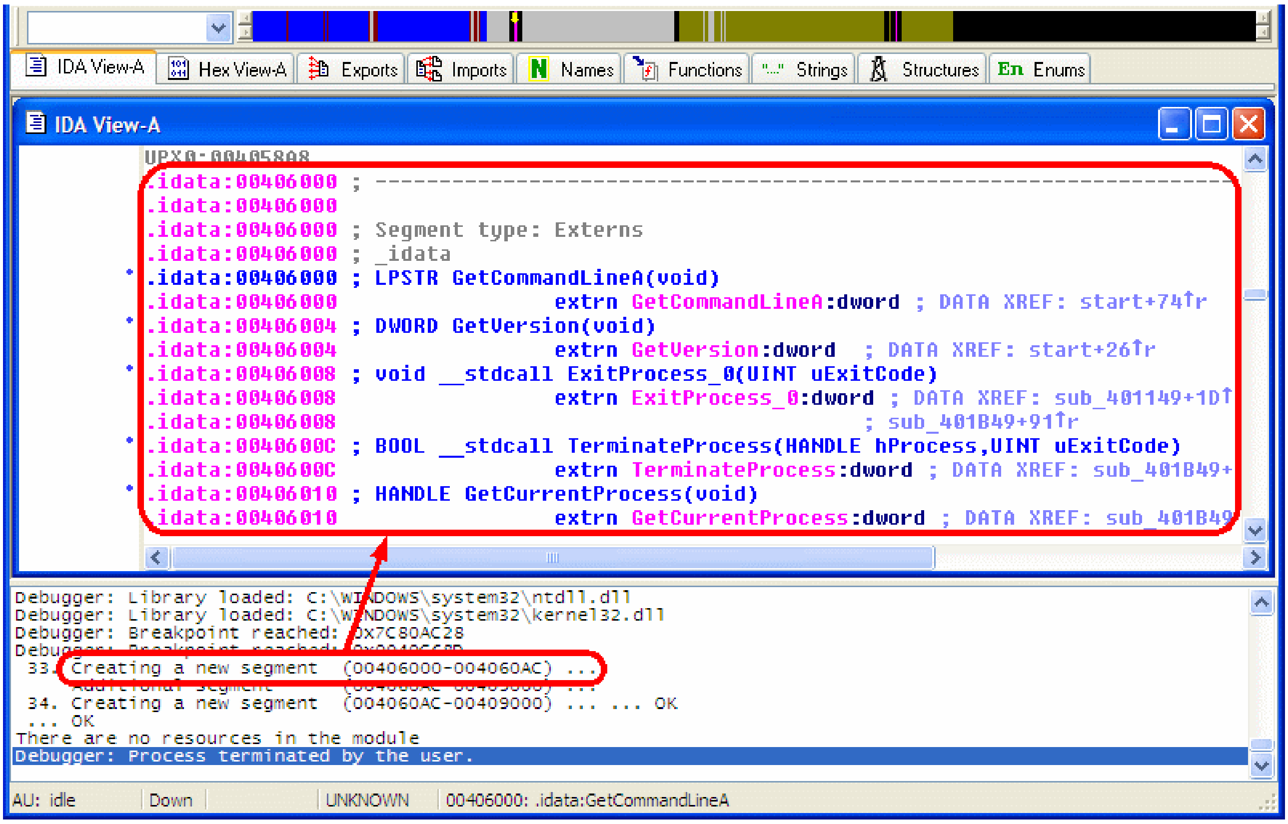


```
Debugger: Process started: D:\IDA\REVERSING\PACKED\hello-upx125w.exe
Debugger: Library loaded: C:\WINDOWS\system32\ntdll.dll
Debugger: Library loaded: C:\WINDOWS\system32\kernel32.dll
Debugger: Breakpoint reached: 0x7C80AC28
Debugger: Breakpoint reached: 0x0040C68D
```

The universal unpacker has finished its work.
Do you want to take a memory snapshot and stop now?
(you can do it yourself if you want)

Note that two breakpoints were reached during the unpacking: we'll say more on these later.

In order to rebuild original import section of the program, the plug-in created a new segment.

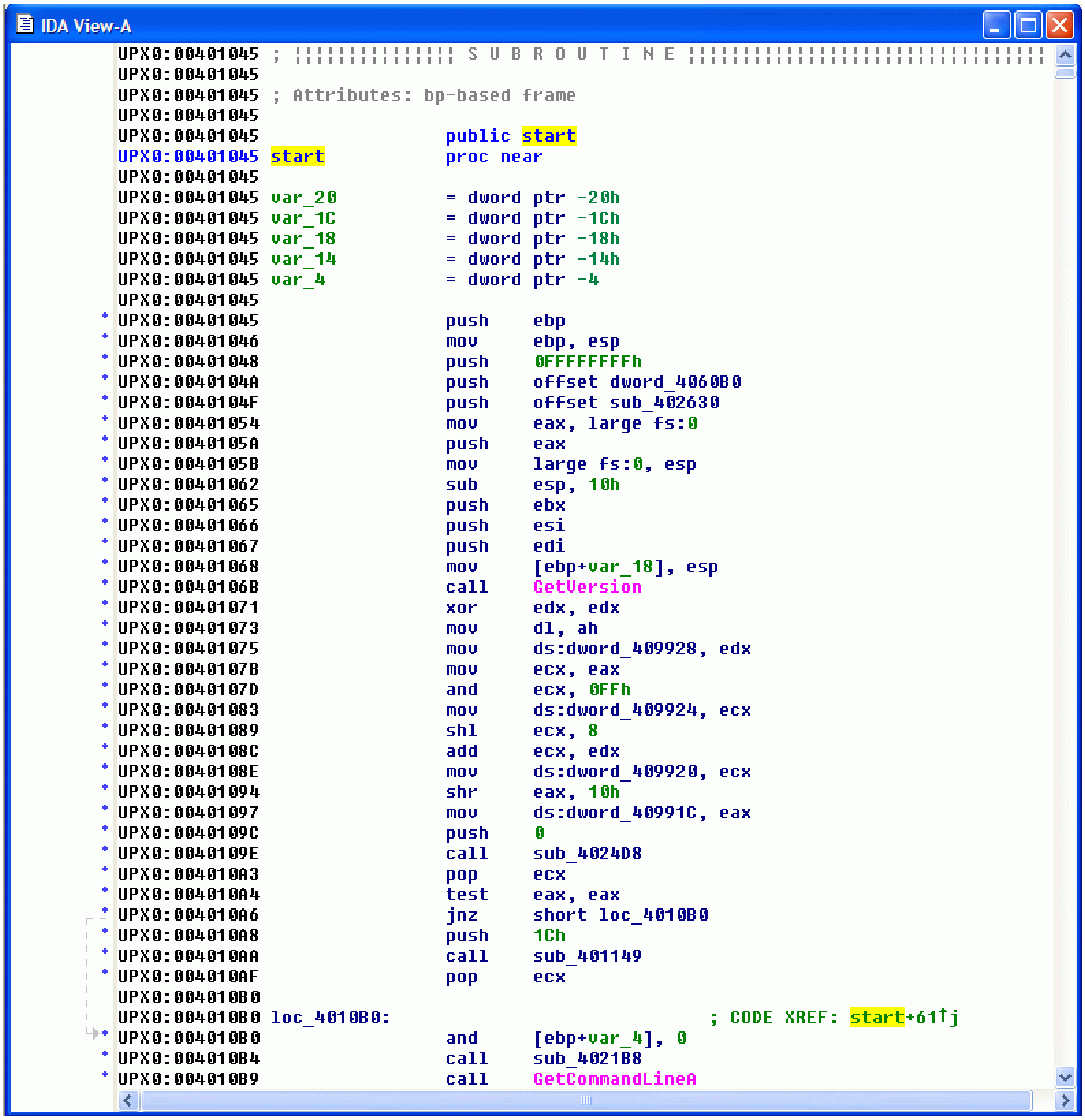


```
.idata:00406000 ; ---------------------------------------------------------------
.idata:00406000
.idata:00406000 ; Segment type: Externs
.idata:00406000 ; _idata
.idata:00406000 ; LPSTR GetCommandLineA(void)
.idata:00406000                 extrn GetCommandLineA:dword ; DATA XREF: start+74↑r
.idata:00406004 ; DWORD GetVersion(void)
.idata:00406004                 extrn GetVersion:dword  ; DATA XREF: start+26↑r
.idata:00406008 ; void __stdcall ExitProcess_0(UINT uExitCode)
.idata:00406008                 extrn ExitProcess_0:dword ; DATA XREF: sub_401149+1D↑
.idata:00406008                                         ; sub_401B49+91↑r
.idata:0040600C ; BOOL __stdcall TerminateProcess(HANDLE hProcess,UINT uExitCode)
.idata:0040600C                 extrn TerminateProcess:dword ; DATA XREF: sub_401B49+
.idata:00406010 ; HANDLE GetCurrentProcess(void)
.idata:00406010                 extrn GetCurrentProcess:dword ; DATA XREF: sub_401B49
```

```
Debugger: Library loaded: C:\WINDOWS\system32\ntdll.dll
Debugger: Library loaded: C:\WINDOWS\system32\kernel32.dll
Debugger: Breakpoint reached: 0x7C80AC28
Debugger: Breakpoint reached: [illegible]
 33. Creating a new segment  (00406000-004060AC) ...
     Additional segment      (004060AC-00409000) ...
 34. Creating a new segment  (004060AC-00409000) ... ... OK
 ... OK
There are no resources in the module
Debugger: Process terminated by the user.
```

```
AU: idle    Down        UNKNOWN    00406000: .idata:GetCommandLineA
```

Once unpacked, we now recognize the more typical structure of the start() function:

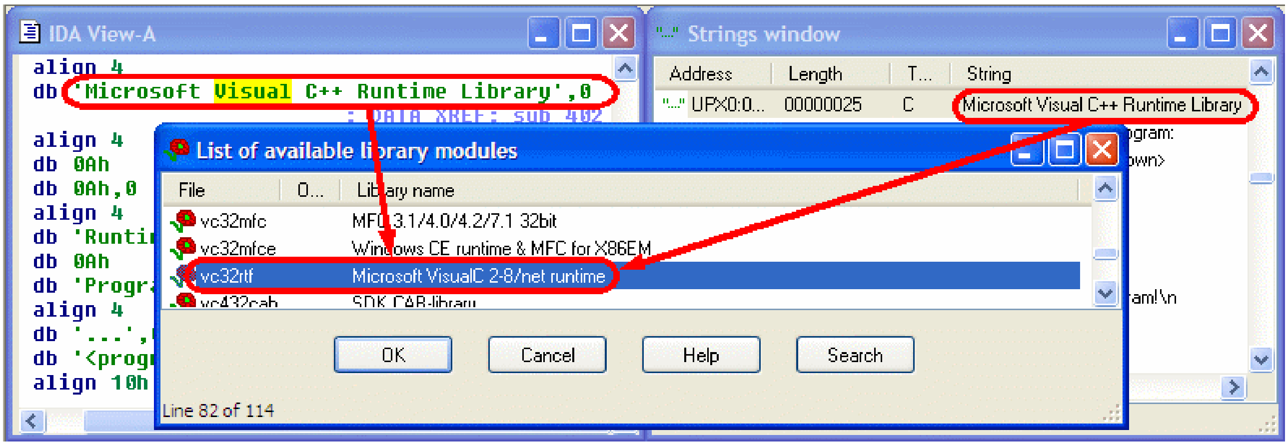


```
UPX0:00401045 ; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦
UPX0:00401045
UPX0:00401045 ; Attributes: bp-based frame
UPX0:00401045
UPX0:00401045                 public start
UPX0:00401045 start           proc near
UPX0:00401045
UPX0:00401045 var_20          = dword ptr -20h
UPX0:00401045 var_1C          = dword ptr -1Ch
UPX0:00401045 var_18          = dword ptr -18h
UPX0:00401045 var_14          = dword ptr -14h
UPX0:00401045 var_4           = dword ptr -4
UPX0:00401045
UPX0:00401045                 push    ebp
UPX0:00401046                 mov     ebp, esp
UPX0:00401048                 push    0FFFFFFFFh
UPX0:0040104A                 push    offset dword_4060B0
UPX0:0040104F                 push    offset sub_402630
UPX0:00401054                 mov     eax, large fs:0
UPX0:0040105A                 push    eax
UPX0:0040105B                 mov     large fs:0, esp
UPX0:00401062                 sub     esp, 10h
UPX0:00401065                 push    ebx
UPX0:00401066                 push    esi
UPX0:00401067                 push    edi
UPX0:00401068                 mov     [ebp+var_18], esp
UPX0:0040106B                 call    GetVersion
UPX0:00401071                 xor     edx, edx
UPX0:00401073                 mov     dl, ah
UPX0:00401075                 mov     ds:dword_409928, edx
UPX0:0040107B                 mov     ecx, eax
UPX0:0040107D                 and     ecx, 0FFh
UPX0:00401083                 mov     ds:dword_409924, ecx
UPX0:00401089                 shl     ecx, 8
UPX0:0040108C                 add     ecx, edx
UPX0:0040108E                 mov     ds:dword_409920, ecx
UPX0:00401094                 shr     eax, 10h
UPX0:00401097                 mov     ds:dword_40991C, eax
UPX0:0040109C                 push    0
UPX0:0040109E                 call    sub_4024D8
UPX0:004010A3                 pop     ecx
UPX0:004010A4                 test    eax, eax
UPX0:004010A6                 jnz     short loc_4010B0
UPX0:004010A8                 push    1Ch
UPX0:004010AA                 call    sub_401149
UPX0:004010AF                 pop     ecx
UPX0:004010B0
UPX0:004010B0 loc_4010B0:                             ; CODE XREF: start+61↑j
UPX0:004010B0                 and     [ebp+var_4], 0
UPX0:004010B4                 call    sub_4021B8
UPX0:004010B9                 call    GetCommandLineA
```

However, let's try to improve this again in order to obtain the nicest disassembly possible!

Applying signatures.

If we look at the strings found after the program was unpacked, we see that our program was compiled by the Visual C++ compiler. Let's apply the associated FLIRT library signatures:



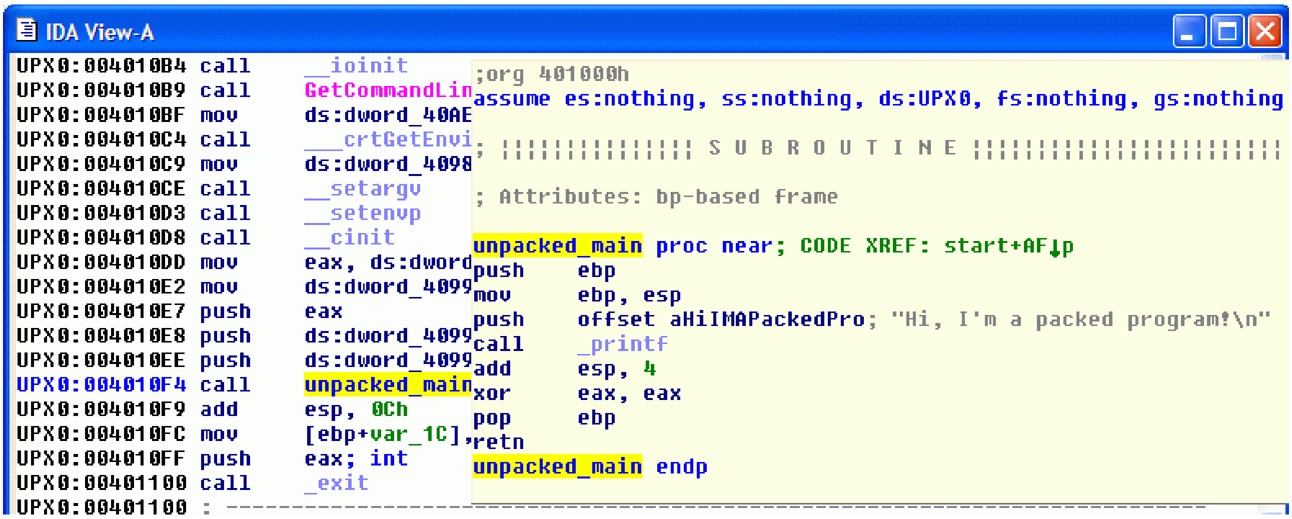
```
align 4
db 'Microsoft Visual C++ Runtime Library',0
                            ; DATA XREF: sub_402
align 4
db 0Ah
db 0Ah,0
align 4
db 'Runti
db 0Ah
db 'Progr
align 4
db '...',
db '<progr
align 10h
```

Address	Length	T...	String
UPX0:0...	00000025	C	Microsoft Visual C++ Runtime Library

File	O...	Library name
vc32mfc		MFC 3.1/4.0/4.2/7.1 32bit
vc32mfce		Windows CE runtime & MFC for X86EM
vc32rtf		Microsoft VisualC 2-8/net runtime
vc432cab		SDK CAB-library

Line 82 of 114

The final disassembly listing looks like this:



```
UPX0:004010B4 call    __ioinit
UPX0:004010B9 call    GetCommandLin
UPX0:004010BF mov     ds:dword_40AE
UPX0:004010C4 call    ___crtGetEnvi
UPX0:004010C9 mov     ds:dword_4098
UPX0:004010CE call    __setargv
UPX0:004010D3 call    __setenvp
UPX0:004010D8 call    __cinit
UPX0:004010DD mov     eax, ds:dword
UPX0:004010E2 mov     ds:dword_4099
UPX0:004010E7 push    eax
UPX0:004010E8 push    ds:dword_4099
UPX0:004010EE push    ds:dword_4099
UPX0:004010F4 call    unpacked_main
UPX0:004010F9 add     esp, 0Ch
UPX0:004010FC mov     [ebp+var_1C],
UPX0:004010FF push    eax; int
UPX0:00401100 call    _exit
UPX0:00401100 ; ---------------------------------------------------------------------------------
```

```
;org 401000h
assume es:nothing, ss:nothing, ds:UPX0, fs:nothing, gs:nothing

; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦

; Attributes: bp-based frame

unpacked_main proc near; CODE XREF: start+AF↓p
push    ebp
mov     ebp, esp
push    offset aHiIMAPackedPro; "Hi, I'm a packed program!\n"
call    _printf
add     esp, 4
xor     eax, eax
pop     ebp
retn
unpacked_main endp
```

Much better, isn't it?

Let's now have some deeper look at how the SDK debugger API functions were used to implement this unpacking in practice.

The main idea is to start the process, then react in an adequate way to various events caught by the debugger, until we determine the program was properly unpacked. So we first setup a handler to receive debugger events and to start the process until its entry point:

```
if ( !hook_to_notification_point(HT_DBG, callback, NULL) )
{
  warning("Could not hook to notification point\n");
  return;
}

// Let's start the debugger
if ( !run_to(inf.beginEA) )
{
  warning("Sorry, could not start the process");
  unhook_from_notification_point(HT_DBG, callback, NULL);
}
```

Events will be sent to our notification handler, defined as follow:

```
static int idaapi callback(void * /*user_data*/,
                           int notification_code,
                           va_list va)
{
  switch ( notification_code )
  {
    case dbg_process_start:
      ...
    case dbg_library_load:
      ...
    case dbg_run_to:
      ...
    case dbg_bpt:
      ...
    case dbg_trace:
      ...
    case dbg_process_exit:
      ...
    ...
  }
  return 0;
}
```

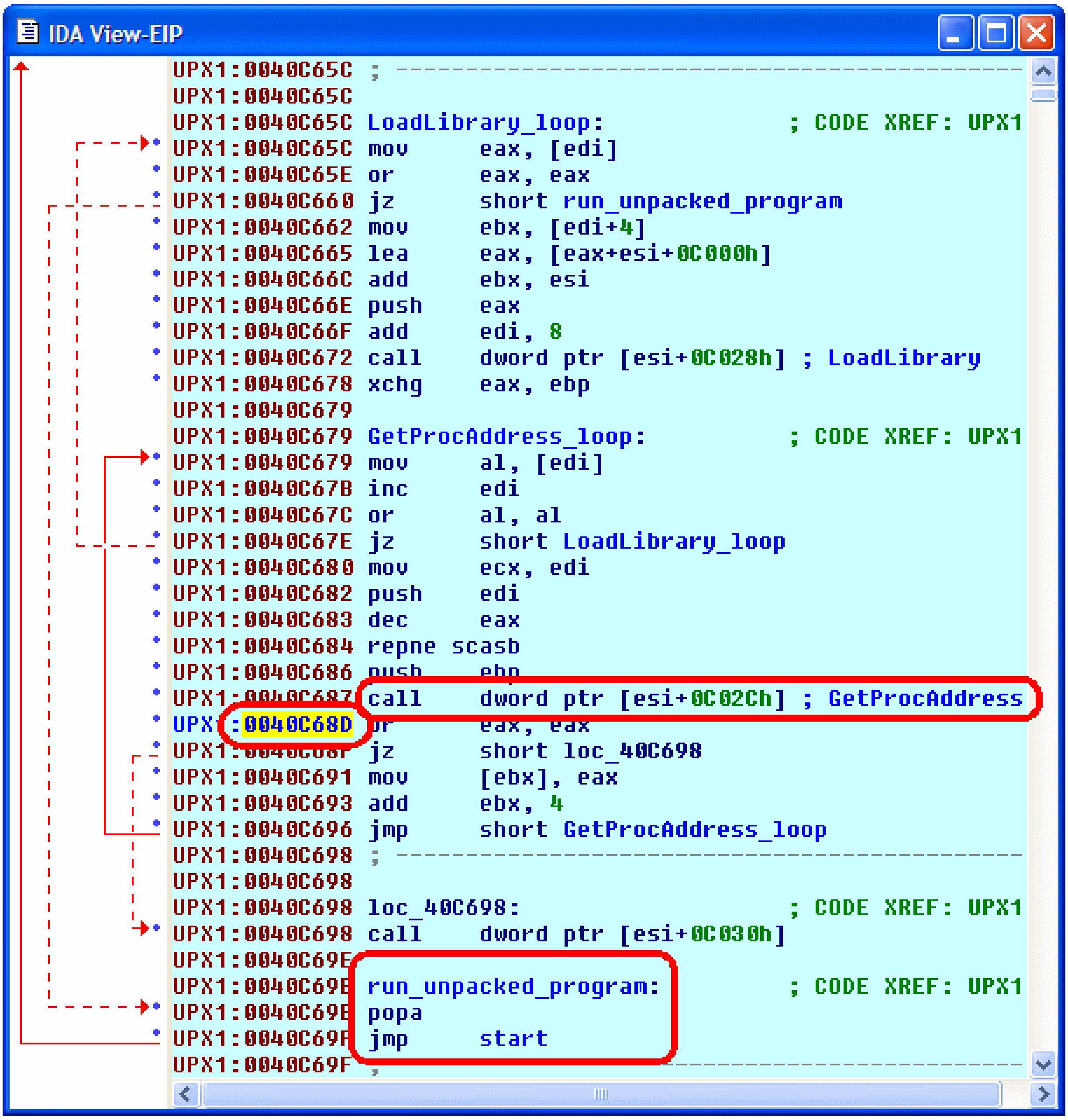
When we start our process through a call to *run_to()*, we will receive a corresponding *dbg_run_to* event, indicating the *run_to()* command was properly executed. We are now at the entry point of the packed program, and setup a breakpoint on the *GetProcAddress()* dynamic-link library function (assuming the unpacking engine has terminated its work before recreating the original import table of the original application):

```
case dbg_run_to:   // Parameters: thread_id_t tid
  dbg->stopped_at_debug_event(true);
  gpa = get_name_ea(BADADDR, "kernel32_GetProcAddress");
  ...
  else if( !add_bpt(gpa) )
  {
    bring_debugger_to_front();
    warning("Sorry, can not set bpt to kernel32.GetProcAddress");
    goto FORCE_STOP;
  }
  else
  {
    ++stage;
    set_wait_box("Waiting for a call to GetProcAddress()");
  }
  continue_process();
  break;
```

When our *GetProcAddress()* breakpoint is reached, we receive a *dbg_bpt* event. We can extract the return address from the stack, delete this first breakpoint, and setup a second breakpoint at the return address in order to get notified as soon as the *GetProcAddress()* function returns:

```
case dbg_bpt:      // A user defined breakpoint was reached.
                   // Parameters: thread_id_t tid
                   //             ea_t        breakpoint_ea
  {
    /*tid_t tid =*/ va_arg(va, tid_t);
    ea_t ea    = va_arg(va, ea_t);
    ...
    if ( ea == gpa )
    {
      regval_t rv;
      if ( get_reg_val("esp", &rv) )
      {
        ea_t esp = rv.ival;
        invalidate_dbgmem_contents(esp, 1024);
        ea_t ret = get_long(esp);
        ...
        if ( !del_bpt(gpa) || !add_bpt(ret) )
          error("Can not modify breakpoint");
```

Do you remember the two breakpoint messages we saw during the unpacking, occurring at addresses 0x7C80AC28 and 0x00040C68D? The first one was our *GetProcAddress()* breakpoint, while the second one was our breakpoint at the return address. In the following disassembly, you can see the call leading to our *GetProcAddress()* breakpoint. We now only have to execute instructions until the unpacking engine restores the program's original register contents and then jumps to the real entry point of the unpacked program:



```
UPX1:0040C65C ; ---------------------------------------------
UPX1:0040C65C
UPX1:0040C65C LoadLibrary_loop:                ; CODE XREF: UPX1
UPX1:0040C65C mov     eax, [edi]
UPX1:0040C65E or      eax, eax
UPX1:0040C660 jz      short run_unpacked_program
UPX1:0040C662 mov     ebx, [edi+4]
UPX1:0040C665 lea     eax, [eax+esi+0C000h]
UPX1:0040C66C add     ebx, esi
UPX1:0040C66E push    eax
UPX1:0040C66F add     edi, 8
UPX1:0040C672 call    dword ptr [esi+0C028h] ; LoadLibrary
UPX1:0040C678 xchg    eax, ebp
UPX1:0040C679
UPX1:0040C679 GetProcAddress_loop:             ; CODE XREF: UPX1
UPX1:0040C679 mov     al, [edi]
UPX1:0040C67B inc     edi
UPX1:0040C67C or      al, al
UPX1:0040C67E jz      short LoadLibrary_loop
UPX1:0040C680 mov     ecx, edi
UPX1:0040C682 push    edi
UPX1:0040C683 dec     eax
UPX1:0040C684 repne scasb
UPX1:0040C686 push    ebp
UPX1:0040C687 call    dword ptr [esi+0C02Ch] ; GetProcAddress
UPX1:0040C68D or      eax, eax
UPX1:0040C68F jz      short loc_40C698
UPX1:0040C691 mov     [ebx], eax
UPX1:0040C693 add     ebx, 4
UPX1:0040C696 jmp     short GetProcAddress_loop
UPX1:0040C698 ; ---------------------------------------------
UPX1:0040C698
UPX1:0040C698 loc_40C698:                      ; CODE XREF: UPX1
UPX1:0040C698 call    dword ptr [esi+0C030h]
UPX1:0040C69E
UPX1:0040C69E run_unpacked_program:            ; CODE XREF: UPX1
UPX1:0040C69E popa
UPX1:0040C69F jmp     start
UPX1:0040C69F ; ---------------------------------------------
```

Step tracing is by far the easiest way to execute instructions until we reach an address in the previously defined range. So let's enable step tracing as soon as we reach our second breakpoint:

```
del_bpt(ea);
if ( !is_library_entry(ea) )
{
  deb(IDA_DEBUG_PLUGIN, "%a: reached unpacker code, switching to trace mode\n",
ea);
  enable_step_trace(true);
  ...
  set_wait_box("Waiting for the unpacker to finish");
}
else
{
  warning("%a: bpt in library code", ea); // how can it be?
  add_bpt(gpa);
}
```

At each instruction step, we now check if the address matches the previously defined range. If we reached this range, we stop to trace, reanalyze the unpacked code, adjust the entry point, rebuild the import table, save resources, and... finally take a snapshot!

```
case dbg_trace:    // A step occured (one instruction was executed). This event
                   // notification is only generated if step tracing is enabled.
                   // Parameter:  none
  ..
  /*tid_t tid =*/ va_arg(va, tid_t);
  ea_t ip   = va_arg(va, ea_t);
  if ( oep_area.contains(ip) )
  {
    // stop the trace mode
    enable_step_trace(false);
    // reanalyze the unpacked code
    set_wait_box("Reanalyzing the unpacked code");
    do_unknown_range(oep_area.startEA, oep_area.endEA, false);
    auto_make_code(ip);
    noUsed(oep_area.startEA, oep_area.endEA);
    auto_mark_range(oep_area.startEA, oep_area.endEA, AU_FINAL);
    // mark the program's entry point
    move_entry(ip);
    set_wait_box();
    ...
    set_wait_box("Recreating the import table");
    invalidate_dbgmem_config();
    ...
    create_impdir();
    set_wait_box("Storing resources to 'resource.res'");
    if ( resfile[0] != '\0' )
      extract_resource(resfile);
    set_wait_box();
    if ( take_memory_snapshot(true) )
      goto FORCE_STOP;
```

The user has obtained a memory dump of the process in his IDA database, allowing him to start analyzing the unpacked code as usual.

Don't hesitate to further look at the source code in the SDK, for all implementation details!

Revision 1.1

DataRescue SA/NV
40 Bld Piercot
4000 Liège, Belgium
T: +32-4-3446510 F: +32-4-3446514