Hacking the Packer

Introduction

I was checking out a packed file and came across a cool way to resolve Kernel32 exports. This led to an interest in detecting packer-specific signatures, which in turn led to a fun registration-process hacking exercise.

Observing Code Obscurities

The code below is from the entry point of the packed file. It derives the base address of its own image with formula:

```
ImageBase = AddrLastInstruction - (AddrEntryPoint +
OffsetToLastInstruction)
```

For example, with values filled in:

```
0x00400000 = 0x004FC01D - (0x000FC000 + 0x1D)
```

The formula may not work for executables packed or compiled differently...but lucky for this piece of code, it doesn't have to worry about any others besides its own.

```
.crt:004FC017 code_begin:
.crt:004FC017                 push    ebp
.crt:004FC018                 call    $+5
.crt:004FC01D                 pop     ebp
.crt:004FC01E                 sub     ebp, 1Dh
.crt:004FC024                 mov     eax, ebp
.crt:004FC026                 push    ebp
.crt:004FC027                 pusha
.crt:004FC028                 pushf
.crt:004FC029                 sub     eax, [ebp+7FCh]
.crt:004FC02F                 mov     [ebp+7E8h], eax
```

Starting at the ImageBase in memory, it starts to scan for values at known offsets, based on the PE header format. In particular, it locates the array of IMAGE_IMPORT_DESCRIPTOR structures and loops through reading the value of the Name1 member (pointer to ASCII name of DLL to import). For each result, it compares that ASCII string with "kernel32.dll" (non-case sensitive). If it encounters a match, it saves the base address of the IMAGE_IMPORT_DESCRIPTOR structure that referenced the string and re-uses it to access the FirstThunk member.

After the PE loading process, the FirstThunk member becomes a component Import Address Table and the program works with this information to resolve the rest of the required functions.

Stripping Down Shell Code

The code appears to be written in straight-up assembly and also has a strange function convention that returns true/false by setting or clearing the CPU's carry flag. Return values are checked with jb instructions, which jump if the flag is set. This is a cool way to lighten up the size of code. For example, rather than what I've recently been seeing, which takes at least 18 bytes...

```
call function
cmp eax,0
jn/z short condition_false
function:
  // do something
  mov eax,0
  retn

char sc[] =
“\xE8\x07\x00\x00\x00\x3D\x00\x00\x00\x00\x75\x06\xB8\x00\x00\x00\x00\x
C3”
```

...the observed method from this packed file only takes 9 bytes:

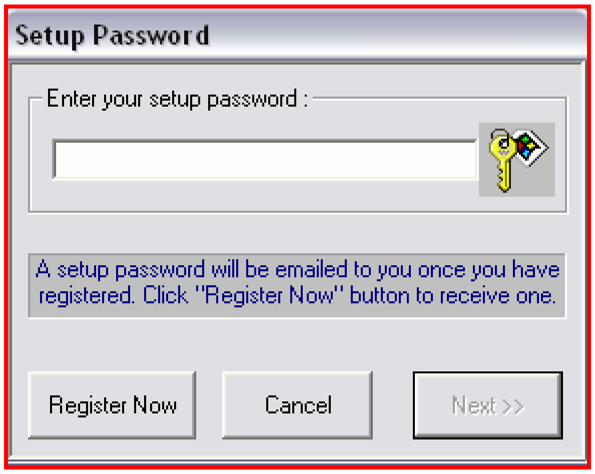
```
call_function
jb short condition_false
function:
  // do something
  clc
  retn

char sc = “\xE8\x02\x00\x00\x00\x72\x02\xF8\xC3”
```

Although the code might not work well as shell code (because of the NULL bytes), it reduces the size in bytes by 50% while maintaining the same type of functionality. This was pretty interesting and made me want to figure out if this procedure is custom or the result of a commercial file packer.

Hacking the Packer

A subtle string from the packed executable identified a suspect product, however the company's website is offline and their utility is pretty rare. To be sure it wasn't just a meaningless string; I wanted to pack a file using the identified product and compare results.

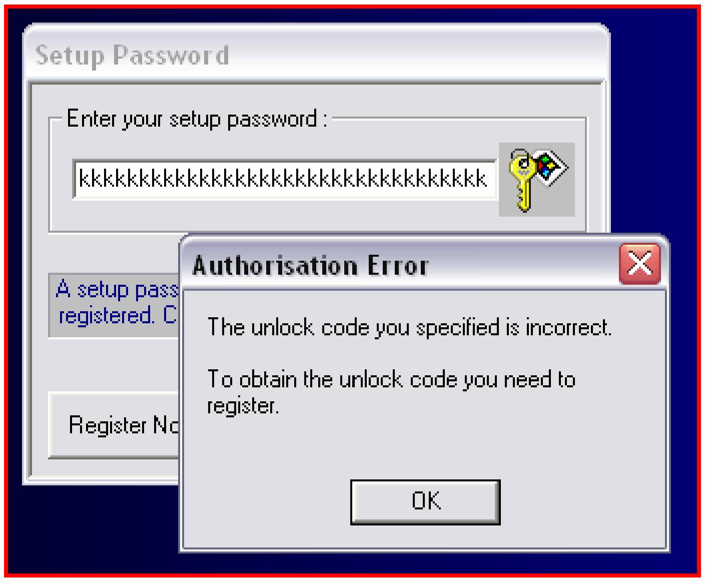
A copy turned up archived on a Polish FTP server, as located by Google. The only problem was that the installation required a password to even unlock the 30 day trial. Since passwords only come via email after registering, and registering cannot be done since the website is offline, the only chance at using this program in the next few minutes is to hack it.


Setup Password

Enter your setup password :

A setup password will be emailed to you once you have registered. Click "Register Now" button to receive one.

The first thing to do is find out where the user's input ends up after entering a password. I typed 255 'k' characters and let the program reach its denial stage, at which point the process was still running.


Setup Password

Enter your setup password :

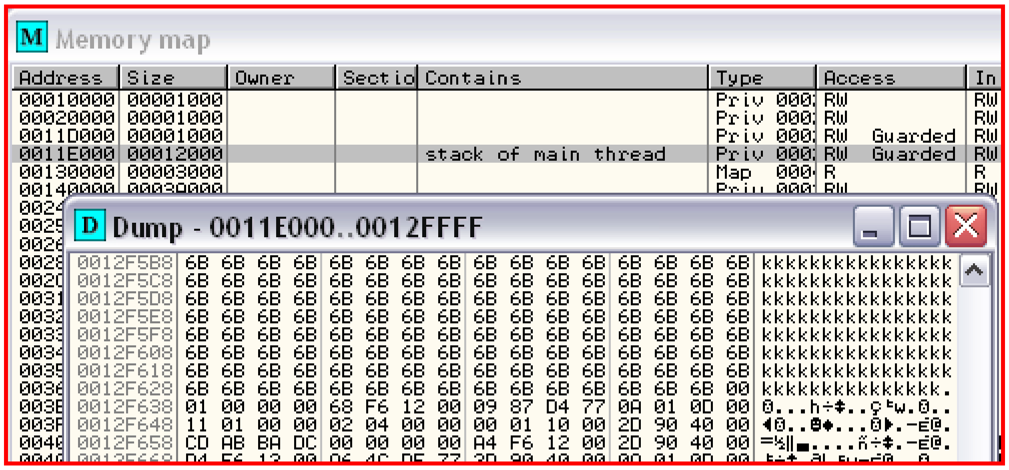
kkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkk

Authorisation Error

The unlock code you specified is incorrect.

To obtain the unlock code you need to register.

Then I attached to it with OllyDbg, brought up the Memory map and searched for a long string of k's - one showed up on the stack of the main thread:



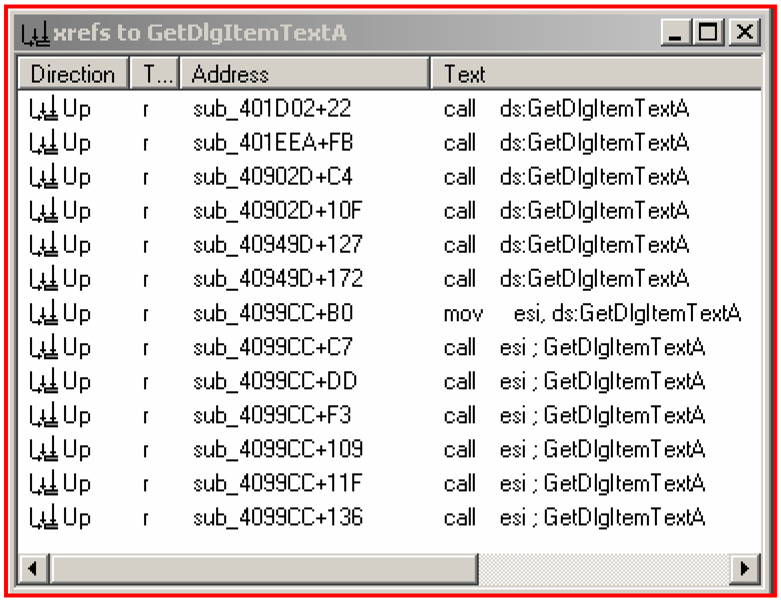
Memory map

```
Address  Size     Owner    Sectio Contains               Type      Access       In
00010000 00001000                                        Priv 000: RW           RW
00020000 00001000                                        Priv 000: RW           RW
0011D000 00001000                                        Priv 000: RW  Guarded  RW
0011E000 00012000                 stack of main thread   Priv 000: RW  Guarded  RW
00130000 00003000                                        Map  000: R            R
00140000 00030000                                        Priv 000: RW           RW
```

Dump - 0011E000..0012FFFF

```
0012F5B8 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B kkkkkkkkkkkkkkkk
0012F5C8 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B kkkkkkkkkkkkkkkk
0012F5D8 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B kkkkkkkkkkkkkkkk
0012F5E8 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B kkkkkkkkkkkkkkkk
0012F5F8 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B kkkkkkkkkkkkkkkk
0012F608 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B kkkkkkkkkkkkkkkk
0012F618 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B kkkkkkkkkkkkkkkk
0012F628 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 6B 00 kkkkkkkkkkkkkkk.
0012F638 01 00 00 00 68 F6 12 00 09 87 D4 77 0A 01 0D 00 0...h÷‡..ç╘w.0..
0012F648 11 01 00 00 02 04 00 00 00 01 10 00 2D 90 40 00 ◄0..☻♦...0►.-É@.
0012F658 CD AB BA DC 00 00 00 00 A4 F6 12 00 2D 90 40 00 =½║▄....ñ÷‡.-É@.
```

Note that only 127 of my 'k' characters made it into the buffer, with a trailing NULL byte. This must mean that there is a 128-byte local buffer declared. Since there are a relatively small amount of Windows API calls that process input from dialog items, I checked the installer's input table and sure enough there are about 15 calls to User32's GetDlgItemTextA().



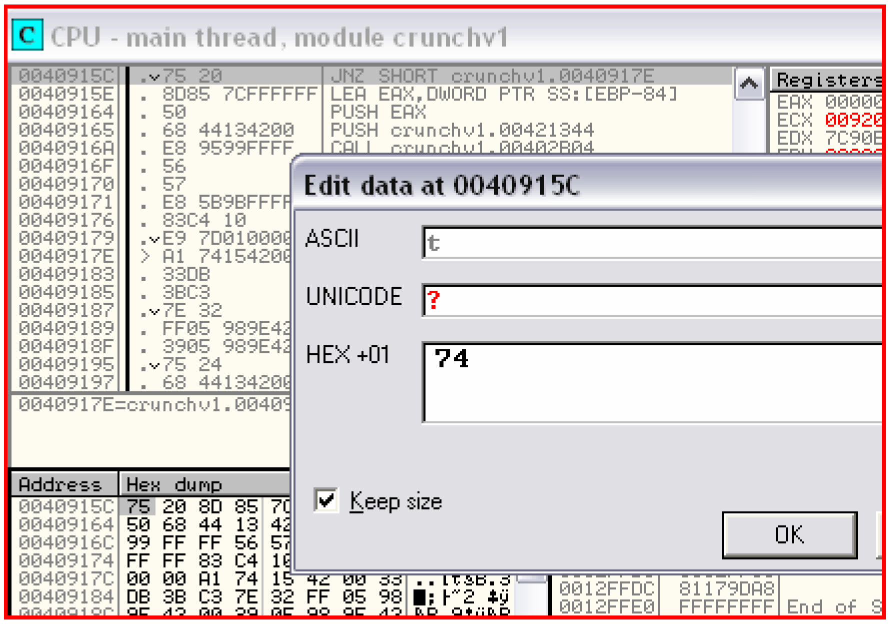
xrefs to GetDlgItemTextA

Direction	T...	Address	Text
Up	r	sub_401D02+22	call ds:GetDlgItemTextA
Up	r	sub_401EEA+FB	call ds:GetDlgItemTextA
Up	r	sub_40902D+C4	call ds:GetDlgItemTextA
Up	r	sub_40902D+10F	call ds:GetDlgItemTextA
Up	r	sub_40949D+127	call ds:GetDlgItemTextA
Up	r	sub_40949D+172	call ds:GetDlgItemTextA
Up	r	sub_4099CC+B0	mov esi, ds:GetDlgItemTextA
Up	r	sub_4099CC+C7	call esi ; GetDlgItemTextA
Up	r	sub_4099CC+DD	call esi ; GetDlgItemTextA
Up	r	sub_4099CC+F3	call esi ; GetDlgItemTextA
Up	r	sub_4099CC+109	call esi ; GetDlgItemTextA
Up	r	sub_4099CC+11F	call esi ; GetDlgItemTextA
Up	r	sub_4099CC+136	call esi ; GetDlgItemTextA

It was easy to scan through these calls and determine which one operated with a buffer size of 80h (128d). Luckily, there was only one possibility and the flow of execution was pretty clear. Right after GetDlgItemTextA(), a jump is taken if the return value is false or NULL. Therefore, the target of that jump can be labeled as an error condition. Now we know the jnz at 0040915Ch needs to fail or be switched to a jz in order to circumvent the checks. Here is the code:

```
.text:0040912A GetPasswordFromUser:
.text:0040912A     lea     eax, [ebp+Data]
.text:00409130     push    80h              ; nMaxCount
.text:00409135     push    eax
.text:00409136     push    410h
.text:0040913B     push    edi
.text:0040913C     call    ds:GetDlgItemTextA
.text:00409142     test    eax, eax
.text:00409144     jz      short error_condition
.text:00409146     lea     eax, [ebp+Data]
.text:0040914C     push    eax
.text:0040914D     push    dword_4290B8
.text:00409153     call    process_password()
.text:00409158     pop     ecx
.text:00409159     test    eax, eax
.text:0040915B     pop     ecx
.text:0040915C     jnz     short error_condition  ; want to bypass
.text:0040915E     lea     eax, [ebp+Data]
.text:00409164     push    eax
.text:00409165     push    offset aUnlock   ; "unlock"
.text:0040916A     call    sub_402B04
```

Here is a screen shot of modifying the jnz to a jz. Once this is done, regardless of the password entered, installation is allowed to proceed.



```
0040915C  .v75 20          JNZ SHORT crunchv1.0040917E
0040915E  . 8D85 7CFFFFFF  LEA EAX,DWORD PTR SS:[EBP-84]
00409164  . 50             PUSH EAX
00409165  . 68 44134200    PUSH crunchv1.00421344
0040916A  . E8 9599FFFF    CALL crunchv1.00402B04
0040916F  . 56
00409170  . 57
00409171  . E8 5B9BFFFF
00409176  . 83C4 10
00409179  .vE9 7D010000
0040917E  > A1 74154200
00409183  . 33DB
00409185  . 3BC3
00409187  .v7E 32
00409189  . FF05 989E42
0040918F  . 3905 989E42
00409195  .v75 24
00409197  . 68 44134200
0040917E=crunchv1.00409

EAX 00000
ECX 00920
EDX 7C90E

ASCII    t
UNICODE  ?
HEX +01  74

Address   Hex dump
0040915C  75 20 8D 85 70
00409164  50 68 44 13 42
0040916C  99 FF FF 56 57
00409174  FF FF 83 C4 10
0040917C  00 00 A1 74 15 42 00 33  ..¡tSB.3
00409184  DB 3B C3 7E 32 FF 05 98  █;├~2 ♣ÿ

0012FFDC  81179DA8
0012FFE0  FFFFFFFF  End of S
```

So I was considering ways to permanently patch the installer. First, I could find the raw offset to the jnz and overwrite it with a jz using a hex editor. That seemed easiest, except the program has a built-in CRC check and won't run if its own binary has been modified. That would be just as easy to crack, but I'd rather finally use the tool and produce a packed executable for analysis since that was my whole purpose in the first place.

Producing Packed Samples

It turns out that even after getting the 30-day demo installed; the program writes protected versions of packed executables (they don't run) until it is registered. This is OK since I'm not using it to produce any packed executables that I expect to run. All I want to see are signatures around the packed executable's entry point, and that can be done without running it. Along the way to creating some samples, I noticed that there is a threshold between when the packer is efficient and when it actually increases the output file size. Note the smaller file at 36kb grew to 48kb when packed. However, the 548kb sample was reduced to 269kb.

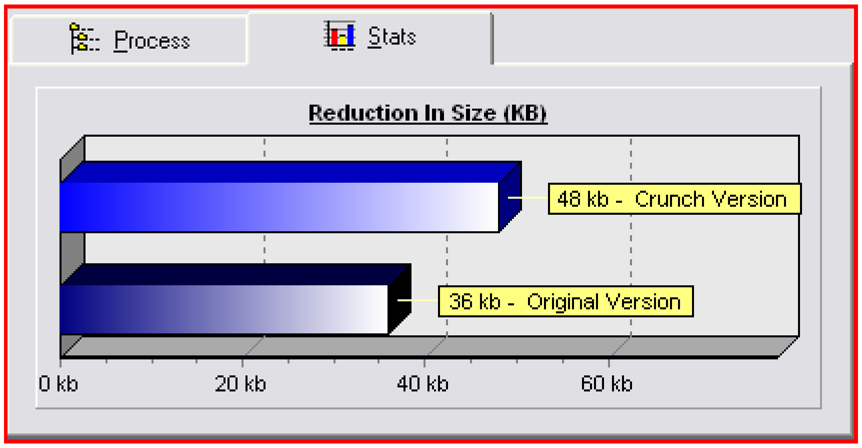


Figure 1: packing a 36kb file.

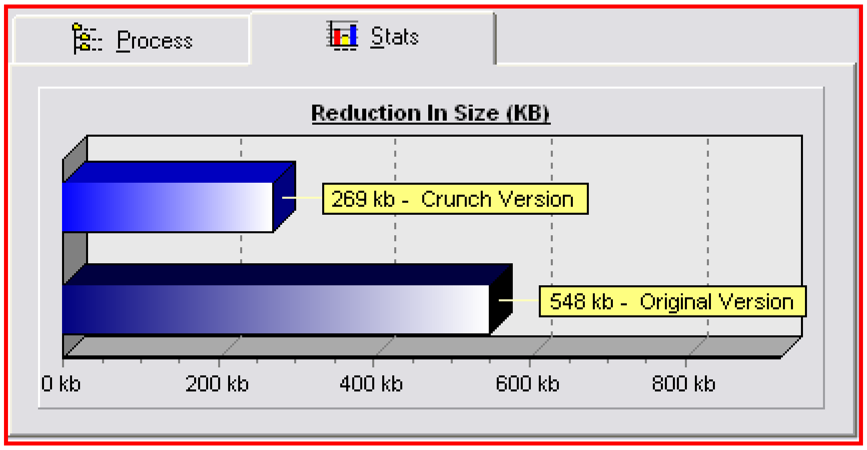


Figure 2: packing a 548kb file.

Possible Signature Set

A (not-necessarily unique) correlation can be made at this point between the new file and the original one. Note the similarity in code at the entry point:

```
Section:004BD000 start          proc near
Section:004BD000                push    ebp
Section:004BD001                call    $+5
Section:004BD006                pop     ebp
Section:004BD007                sub     ebp, 6
Section:004BD00A                mov     eax, ebp
Section:004BD00C                push    ebp
Section:004BD00D                pusha
Section:004BD00E                mov     [ebp+3410h], ebp
Section:004BD014                sub     eax, [ebp+33EBh]
Section:004BD01A                mov     [ebp+249Fh], eax
```

The exact bytes around the entry point of programs packed with this tool will differ, because the section sizes and offsets will change. The instructions and registers would likely remain the same across most versions of the packing tool. It is also likely that a longer, more accurate signature can be derived for this packer.

Extended Kernel32 Resolution

```
.crt:004FC404  ; ¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦¦¦¦¦
.crt:004FC404
.crt:004FC404 find_kernel32_import proc near
.crt:004FC404    mov     eax, [ebp+7E8h] ; ImageBase
.crt:004FC40A    mov     edi, [eax+3Ch]
.crt:004FC40D    add     edi, eax
.crt:004FC40F    mov     edx, [edi+80h]
.crt:004FC415    add     edx, eax
.crt:004FC417
.crt:004FC417 top_loop:
.crt:004FC417    cmp     dword ptr [edx+0Ch], 0
.crt:004FC41B    jz      short loc_4FC47D
.crt:004FC41D    mov     ebx, [edx+0Ch]
.crt:004FC420    add     ebx, eax
.crt:004FC422    cmp     byte ptr [ebx], 'k'
.crt:004FC425    jz      short loc_4FC42E
.crt:004FC427    cmp     byte ptr [ebx], 'K'
.crt:004FC42A    jz      short loc_4FC42E
.crt:004FC42C    jmp     short increment_ptr_and_cont
.crt:004FC42E ; -----------------------------------------------------
.crt:004FC42E
.crt:004FC42E loc_4FC42E:
.crt:004FC42E
.crt:004FC42E    cmp     dword ptr [ebx+1], 'enre'
.crt:004FC435    jz      short loc_4FC442
.crt:004FC437    cmp     dword ptr [ebx+1], 'ENRE'
.crt:004FC43E    jz      short loc_4FC442
.crt:004FC440    jmp     short increment_ptr_and_cont
.crt:004FC442 ; -----------------------------------------------------
.crt:004FC442
.crt:004FC442 loc_4FC442:
.crt:004FC442
.crt:004FC442    cmp     dword ptr [ebx+5], '.23l'
.crt:004FC449    jz      short loc_4FC456
.crt:004FC44B    cmp     dword ptr [ebx+5], '.23L'
.crt:004FC452    jz      short loc_4FC456
.crt:004FC454    jmp     short increment_ptr_and_cont
.crt:004FC456 ; -----------------------------------------------------
.crt:004FC456
.crt:004FC456 loc_4FC456:
.crt:004FC456
.crt:004FC456    cmp     byte ptr [ebx+0Ch], 0
.crt:004FC45A    jz      short found_kernel
.crt:004FC45C
.crt:004FC45C increment_ptr_and_cont:
.crt:004FC45C
.crt:004FC45C    add     edx, 14h
.crt:004FC45F    jmp     short top_loop
.crt:004FC461 ; ------------------------------------------------------
```

C Representation of Resolver

This mimics a function in the original packed sample, except its input source comes from a binary on disk rather than the RAM space of a running image and it is written in C rather than assembly. This code compiled would be huge compared to the straight-up assembly version. Also, this version scans for symbols starting at FILE_BEGIN of the binary, whereas the packed version starts at the ImageBase virtual address. The desired result will appear like this:

```
Examining c:\putty.exe

/* Locate Import Directory Table  */

DOS magic: 0x5a4d @ [raw:0x0000]
PE header offset: 0xf8 @ [raw:0x003C]
PE magic: 0x4550 @ [raw:0x00f8]
Address of entry point: 0x4265f [raw:0x120]
Raw size difference: 0x0
Import Directory offset: 0x60dd8 @ [raw:0x60dd8]

/* Loop through the Name1 member of all existing
   IMAGE_IMPORT_DESCRIPTOR structures and find the one
   which corresponds to kernel32.dll */

Dissing ADVAPI32.dll @ [raw:0x61380]
Dissing COMCTL32.dll @ [raw:0x6138e]
Dissing comdlg32.dll @ [raw:0x613e2]
Dissing GDI32.dll @ [raw:0x616cc]
Dissing IMM32.dll @ [raw:0x6174a]
Dissing SHELL32.dll @ [raw:0x61764]
Dissing USER32.dll @ [raw:0x61e58]
Dissing WINMM.dll @ [raw:0x61e72]
Dissing WINSPOOL.DRV @ [raw:0x61f06]
Found Kernel32 @ [raw:0x62540]
Saving [raw:0x60e8c] as structure base for Kernel32 import

/* Locate the FirstThunk member of Kernel32's
   IMAGE_IMPORT_DESCRIPTOR structure for a pointer
   to the Import Address Table entry */

Kernel32 FirstThunk: [raw:0x60e9c] -> [raw:0x4b100]
Address of imported function: 0x62516 @ [raw:0x4b100]
find_kernel_32_import returned 0x62516
```

It shows the raw (on disk) offset and current value of the 32-bit field that will later be overwritten by a PE loader to contain the virtual address of the first imported function from kernel32.dll. The assembly version would go on to use this information in order to call LoadLibrary(). If the target binary is packed or the entry point RVA is otherwise not aligned with the entry point raw, you will need to set bFixRawSizes to true and manually set the dwRawSizeDiff formula.

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

#define bFixRawSizes false
#define MAX_IMPORTS 0x14

DWORD find_kernel32_import(LPCSTR lpBin);

int main(int argc,char *argv[]) {
      DWORD dwAddress  = 0;
      LPVOID lpLoadLibrary = 0;
      char bin[0x10] = "c:\\putty.exe";
      fprintf(stdout,"\nExamining %s\n",bin);
      dwAddress = find_kernel32_import(bin);
      fprintf(stdout,"function returned 0x%x\n",dwAddress);
      return true;
}

DWORD find_kernel32_import(LPCSTR lpBin) {
      char aImportedDLL[0xD] = "";
      unsigned int nIterations = 0x0;
      WORD wMagic_DOS = 0x00;
      WORD wMagic_PE  = 0x00;
      DWORD dwNumberOfBytesRead       = 0,
              dwOffsetToPEHeader      = 0,
              dwOffsetToImportStructs = 0,
              dwOffsetToDLLName       = 0,
              dwOffsetToFirstThunk    = 0,
              dwAddressOfEntryPoint   = 0,
              dwRawSizeDiff           = 0,
              dwFirstThunkData        = 0,
              dwOriginalFirstThunk    = 0,
              dwVirtualAddress        = 0,
              dwRetValue              = 0,
              dwWhateverValue         = 0;
      HANDLE hFile;
      BOOL bResult;

      hFile = CreateFile(lpBin,GENERIC_READ,\
        FILE_SHARE_READ,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
      if (hFile == INVALID_HANDLE_VALUE)
            return 0x2;

      fprintf(stdout,"\n/* Locate Import Directory Table  */\n\n");

      // query and validate the MS-DOS header
      if (SetFilePointer(hFile,NULL,NULL,FILE_BEGIN) == \
      (DWORD)INVALID_HANDLE_VALUE)
            return 0x3;
      if(!ReadFile(hFile,&wMagic_DOS,0x2,&dwNumberOfBytesRead,NULL) \
      || wMagic_DOS != 0x5a4d)
            return 0x4;

      fprintf(stdout,"DOS magic: 0x%x @ [raw:0x0000]\n",wMagic_DOS);
```

```c
// query for the raw offset to PE header
if (SetFilePointer(hFile,0x3C,NULL,FILE_BEGIN) == \
(DWORD)INVALID_HANDLE_VALUE)
      return 0x5;
if (!ReadFile(hFile,&dwOffsetToPEHeader,0x4, \
&dwNumberOfBytesRead,NULL))
      return 0x6;

fprintf(stdout,"PE header offset: 0x%x @ \
[raw:0x003C]\n",dwOffsetToPEHeader);

// query and validate the PE header
if (SetFilePointer(hFile,dwOffsetToPEHeader,NULL,FILE_BEGIN) \
 == (DWORD)INVALID_HANDLE_VALUE)
      return 0x7;
if (!ReadFile(hFile,&wMagic_PE,0x2,&dwNumberOfBytesRead,NULL) \
|| wMagic_PE != 0x4550)
      return 0x8;

fprintf(stdout,"PE magic: 0x%x @ \
[raw:0x00%x]\n",wMagic_PE,dwOffsetToPEHeader);

if (SetFilePointer(hFile,dwOffsetToPEHeader+0x28,NULL, \
FILE_BEGIN) == (DWORD)INVALID_HANDLE_VALUE)
      return 0x99;
if (!ReadFile(hFile,&dwAddressOfEntryPoint,0x4, \
&dwNumberOfBytesRead,NULL))
      return 0x98;
fprintf(stdout,"Address of entry point: 0x%x \
[raw:0x%x]\n",dwAddressOfEntryPoint,dwOffsetToPEHeader+0x28);

if (bFixRawSizes) { // when rva and raw are not aligned
      dwRawSizeDiff = dwAddressOfEntryPoint - 0x3E00;
}
else dwRawSizeDiff = 0x0; // rva and raw are aligned
fprintf(stdout,"Raw size difference: 0x%0x\n",dwRawSizeDiff);

// query for raw offset of import directory
if (SetFilePointer(hFile,dwOffsetToPEHeader+0x80,NULL, \
FILE_BEGIN) == (DWORD)INVALID_HANDLE_VALUE)
      return 0x9;
if (!ReadFile(hFile,&dwOffsetToImportStructs,0x4, \
&dwNumberOfBytesRead,NULL))
      return 0x10;

fprintf(stdout,"Import Directory offset: 0x%x @ \
[raw:0x%x]\n",dwOffsetToImportStructs, \
dwOffsetToImportStructs-dwRawSizeDiff);

dwOffsetToImportStructs = dwOffsetToImportStructs - \
dwRawSizeDiff;

fprintf(stdout,"\n/* Loop through the Name1 member \
of all existing\n");
fprintf(stdout,"   IMAGE_IMPORT_DESCRIPTOR structures \
and find the one\n");
fprintf(stdout,"   which corresponds to kernel32.dll */\n\n");
```

```
// loop through the Name1 member of
// IMAGE_IMPORT_DESCRIPTOR structures for kernel32.dll
  for(;;) {
        nIterations++;
        if (nIterations > MAX_IMPORTS) {
              fprintf(stdout,"Past max imports…\n");
              break;
        }
        dwOffsetToDLLName = dwOffsetToImportStructs + 0xC;
        if (SetFilePointer(hFile,dwOffsetToDLLName, \
        NULL,FILE_BEGIN) == (DWORD)INVALID_HANDLE_VALUE)
              return 0x11;
        if (!ReadFile(hFile,&dwOffsetToDLLName,0xC, \
        &dwNumberOfBytesRead,NULL))
              return 0x12;
        dwOffsetToDLLName -= dwRawSizeDiff;
        if (SetFilePointer(hFile,dwOffsetToDLLName, \
        NULL,FILE_BEGIN) == (DWORD)INVALID_HANDLE_VALUE)
              return 0x13;
        if (!ReadFile(hFile,aImportedDLL,0xC, \
        &dwNumberOfBytesRead,NULL))
              return 0x14;
        if (stricmp(aImportedDLL,"KERNEL32.dll") == 0) {
              fprintf(stdout,"Found Kernel32 @ \
              [raw:0x%x]\n",dwOffsetToDLLName);
              fprintf(stdout,"Saving [raw:0x%x] as \
              structure base for Kernel32 import\n",
              dwOffsetToImportStructs);
              break;
        }
        fprintf(stdout,"Dissing %s @ [raw:0x%x]\n", \
        aImportedDLL,dwOffsetToDLLName);
        dwOffsetToImportStructs += 0x14;    }

  fprintf(stdout,"\n/* Locate the FirstThunk member of \
  Kernel32's\n");
  fprintf(stdout,"   IMAGE_IMPORT_DESCRIPTOR structure \
  for a pointer\n");
  fprintf(stdout,"   to the Import Address Table entry */\n\n");

  // query for FirstThunk member of IMAGE_IMPORT_DESCRIPTOR
  dwOffsetToFirstThunk = dwOffsetToImportStructs + 0x10;
  if (SetFilePointer(hFile,dwOffsetToFirstThunk,NULL,FILE_BEGIN) \
  == (DWORD)INVALID_HANDLE_VALUE)
        return 0x15;
  if (!ReadFile(hFile,&dwFirstThunkData,0x4, \
  &dwNumberOfBytesRead,NULL))
        return 0x16;

  fprintf(stdout,"Kernel32 FirstThunk: [raw:0x%x] -> [raw:0x%x]\n",
        dwOffsetToFirstThunk,dwFirstThunkData-dwRawSizeDiff);

  /* scan to offset pointed to by the FirstThunk which should
     be the virtual address of the first imported function
     from kernel32.dll as resolved by the PE loader at runtime */
  if (SetFilePointer(hFile,dwFirstThunkData,NULL,FILE_BEGIN) \
        == (DWORD)INVALID_HANDLE_VALUE)
              return 0x17;
        if (!ReadFile(hFile,&dwVirtualAddress,0x4, \
        &dwNumberOfBytesRead,NULL))
              return 0x18;

        fprintf(stdout,"Address of imported function: \
        0x%x @ [raw:0x%x]\n", dwVirtualAddress, \
        dwFirstThunkData-dwRawSizeDiff);

    return dwVirtualAddress;
}
```